Software maintenance by program transformation in a wide spectrum language

Bull, Tim

SOFTWARE MAINTENANCE BY PROGRAM TRANSFORMATION IN A WIDE SPECTRUM LANGUAGE

Ph.D. Thesis

University of Durham

(School of Engineering and Computer Science)

Tim Bull

1994

Abstract

This thesis addresses the software maintenance problem of extracting high-level designs from code. The investigated solution is to use a mathematically-based formal program transformation system. The resulting tool, the Maintainer's Assistant, is based on Ward's [177] WSL (wide spectrum language) and method of proving program equivalence.

The problems addressed include: how to reverse engineer from code alone (the only reliable source of information about a program [158]), how to express program transformations within the system, what kinds of transformations should be incorporated, how to make the tool simple to use, how to perform abstraction and how to create a tool suitable for use with large programs.

Using the Maintainer's Assistant, the program code is automatically translated into WSL and the transformations, although tested for valid applicability by the system, are interactively applied by the user. Notable features include a mathematical simplifier, a large flexible transformation catalogue and, significantly, the use of an extension of WSL, $\mathcal{METAWSL}$, for representing the transformations.

$\mathcal{META}$WSL expands WSL by incorporating a variety of extensions, including: program editing statements, pattern matching and template filling functions, symbolic mathematics and logic functions, statements for moving within the program's syntax tree and statements for repeating an operation at each node of the tree. Using $\mathcal{META}$WSL, 80% of the 601 transformations can be expressed in less than 20 program statements.

The Maintainer's Assistant has been used on a wide variety of examples of up to several thousand lines, including commercial software written in IBM 370 assembler. It has been possible to transform initially unstructured programs into a hierarchy of procedures, facilitating subsequent design recovery.

These results show that program transformation is a viable method of renovating old (370 assembler) code in a cost effective way, and that $\mathcal{META}$WSL provides an effective basis for clearly and concisely expressing the required transformations.

Acknowledgements

I wish to thank Prof. Keith Bennett for his invaluable advice and encouragement as my supervisor.

I also wish to thank the other members of the ReForm project — Brendan Hodgson, Nigel Scriven, Hongji Yang and Martin Ward — for their contributions.

This work was funded by IBM Hursley, the Department of Trade and Industry, and the Science and Engineering Research Council. This work was carried out in conjunction with Durham Software Engineering Ltd. (formally Centre for Software Maintenance Ltd.).

This Ph.D. thesis has been produced using LaTeX.

"The true lover of knowledge naturally strives for truth, and is not content with common opinion, but soars with undimmed and unwearied passion till he grasps the essential nature of things." — Plato, The Republic, 490A

Contents

List of Figures

Chapter 1

Software Engineering

1.1 Introduction

Since the invention of the computer, hardware performance per unit cost has increased by as much as 25% per year [142] while the cost of software has fallen by only 7–9% per year [142]. The result is that software has become by far the most expensive part of installing any computer system. In addition, software complexity has led to severe underestimates of the resources required to complete any given programming project. An early example of such a disaster was OS/360 [64]; although it did eventually work. This failure came to light in October 1968 at the NATO-sponsored conference on the newly coined term **Software Engineering** in Garmisch-Partenkirchen and caused the existence of a **Software Crisis** to be first admitted. The symptoms of this crisis, from which most of the software development industry has yet to escape, are that too much software is late, over budget and does not perform as expected. It became clear that a new approach was required to solve these problems; this approach is software engineering.

There have been numerous definitions of software engineering, for example:

> **Software engineering** is the science and art of specifying, designing,

> implementing and evolving — with economy, timeliness and elegance — programs, documentation and operating procedures whereby computers can be made useful to man [130].

This is a good definition in that it stresses the art (i.e. creativity) required in software engineering, all stages of the software life cycle (which will be defined later), the economics of the software engineering process and the fact that software engineering involves the production of more than just program code.

As with traditional engineering, software engineering involves the use of a rigorous **method** for software production.

> A **method** is a set of procedures (guidelines) for selecting and sequencing the use of tools and techniques [34].

This chapter will examine the tools and techniques used in software engineering, and a typical method of sequencing them — the software life cycle. First, however, it is worth emphasising the differences between traditional engineering and software engineering.

1.2 Differences from Traditional Engineering

According to McDermid [130], there are five areas in which software engineering is inherently more difficult than traditional engineering:

Complexity — Software is complex in that it needs to interface to complex mechanical, social or organisational systems. Also, software is complex "as a material" since (unlike other engineering artifacts) it has no regular structure[1]. Finally, software systems are too large to be understood by a single individual.

[1] Dijkstra [66] claims that programs are the most complex artifacts conceived and illustrates this complexity by contrasting programs with Euclidean geometry — both of which he considers to be branches of mathematics [65].

Difficulty of establishing requirements — Users do not know what they want and they over (or under) estimate the ability of the computer. Many of their activities are "second nature" and specifications are difficult to elicit. In addition, there is no method of determining when a specification is "complete". Also requirements are never stable.

Changeability or malleability of software — It is easy to write and change a small program, but *much* harder for a large system due to the interaction between different parts of the system.

Invisibility — While all other engineering disciplines produce physical artifacts which can be seen and examined, software is much more nebulous. The original design decisions are not manifest in the programs themselves and although there are many points of view from which to look at software (scope of variables, control flow, module hierarchy etc.) all of them are limited.

Development of a theory of the problem domain — In most engineering disciplines, the development of a new system involves the application of existing theories (for example in bridge building). However, with software, every time a system (for air traffic control or payroll or whatever) is developed, a new theory needs to be developed. These theories may even change with time as the engineered systems or social structures, to which the software relates, change too.

1.3 Tools and Techniques

Blank and Krijger [36] list 24 different technical methods and techniques for use in software engineering. These fall roughly into four categories: **structured programming** techniques, methods for clear program and data structure **presentation**, **computer aided software engineering**, and methods for **deriving programs** from their specifications.

1.3.1 Structured Programming

The aim of structured programming is to produce programs which have more **structure.**[2] Such programs are easier to understand, maintain and modify since, for any given statement in the program, it is clear which branches must have been taken, and which procedures called, in order to reach this point; i.e. their dynamic behaviour is clear from their static structure.

> **Software structure** is defined to be the arrangement of, and interrelation between, the components that make up the software system.

Although there is no general agreement about what is "good" structure (such as programming without using Goto statements), there are a number of attributes which are thought to be important [28]:

- The software is partitioned into components (modules) with identifiable and simple boundaries;
- There should be a high dependence within the components (modules) of the system; and
- The relationships between the components (modules) form a hierarchy.

Structured programming has had a very beneficial effect on the quality of software systems. Numerous case studies have documented impressive gains in productivity, reliability and maintainability of these new systems [197].

1.3.2 Modular Programming

A more rigorous version of structured programming is **modular programming** which divides a program into (small) chunks known as modules. A module allows

[2] In fact *all* programs necessarily have structure to them, otherwise they would not execute; so here, software structure has a more specialised meaning.

the programmer to control the visibility of a component within a program [63]. There are many ways of making these divisions, some considerably better than others. Good modular programming has been characterised by Bergland [32] as a decomposition such that each module:

- Implements a single independent function;
- Performs a single logical task;
- Has a single entry and exit point;
- Is separately testable; and
- Is entirely constructed of modules.

1.3.3 Object Oriented Systems

A further enhancement to the idea of modular programming is **object oriented programming.**[3] An object oriented system is one which is composed of independent objects each of which provides a behaviour. This behaviour is a set of operations that the object may be requested to perform; for example, to return or modify, some internal value that it holds.

A good object oriented system is one that contains objects with the following properties [10]:

Encapsulation — An object's state is only accessible using its nominated operations;

Dynamic lifetimes — Objects can be created as the system executes;

Identity — Each object has a name, which can be used to refer to it; and

[3] The term "object oriented" has also been used to cover design methods, user interfaces, databases as so on.

Substitution — Objects that provide compatible operations can be used interchangeably.

Objects which differ only in name and state are said to be **instances** of a **class**. When a class is defined, it may (in some systems) be provided with the behaviour of one or more other classes. This is termed **inheritance**. If used badly, inheritance can lead to programs that are poorly structured.

1.3.4 Presentation Techniques

Software engineering, as well as producing methods for constructing "better" programs, has also spawned methods of representing the structure of these programs diagrammatically. This significantly improves the user's understanding of how the software works, especially when several methods are used together. These presentation methods include flowcharts, HIPO (Hierarchy plus Input, Process, Output) charts and data structure charts, for example, Warnier-Orr diagrams [186] and Jackson diagrams [101]. Other presentation methods which have been used are decision tables and very-high-level pseudocode.

1.3.5 Computer Aided Software Engineering

> **Computer Aided Software Engineering** or **CASE** is the automation of existing software engineering methods and practices with the goal of improving the quality of the product and the efficiency of the software developers [171].
>
> **CASE tools**, in the boardest sense, encompass any facility utilising the computer to assist in the production of software [102].

CASE tools can be subdivided into those that are directed primarily at the front-end activities of design and analysis, and those that are focused on the back-end implementation functions of code generation, testing and maintenance [102].

Those that generate code have improved programmers' productivity, by enabling them to work at a "higher level" (i.e. closer to the level of the application rather than the computer). The ultimate goal of CASE technology is to separate the application program's design from the program's implementation as code [76].

Major issues facing CASE are (a) the integration of the many tools that are required in order to cover the spectrum of software activities from specification, through development, to maintenance; and (b) cost of discarding existing software in order to move to a CASE tool environment.

1.3.6 Deriving Programs from Specifications

Another way of producing "better" programs is to start with the specification of the program and at each stage to develop incrementally by **stepwise refinement**. In other words, the program is first coded at a very high level, and each high-level construct is represented in turn by its components, with more detail and at a lower level, until the program is expressed in terms of the target language. This method was proposed by Wirth [191].

Although stepwise refinement appears good in principle, it is very rarely used in practice since designers actually work, not only from from the top down, but also from the bottom up and from the middle out (see Section 8.2). The problem is that any expression of the high-level design as low-level code may not be practicably implementable on the target machine. Thus, to at least some degree, the designer needs, at an early stage, an idea of the final implementation.

1.4 The Software Life Cycle

A key part of software engineering has been the decomposition of the software process into a number of stages. Such a decomposition has been named a **life cycle**, and a number of different life cycle models have been suggested [131] [38].

Such models generally divide the project into small steps based on the ways in which they are planned and performed as follows:

Requirements Analysis and Definition — Analysis is performed, through observation of existing systems, discussion with potential users and so on, to discover the purpose of the software and to set the overall goals.

Functional Specification — A program specification is produced.

> A **program specification** is a statement of the precise functions which are to be carried out by a computer program, including descriptions of the input to be processed by the program, the processing needed and the output from the program [150].

A good functional specification fulfills the needs of the original requirements definition and leads naturally on to a design for the system in such a way that it rules out any implementation that is unacceptable and such that it is general enough to ensure that few, if any, acceptable programs are precluded.

Non-Functional Specification — Constraints on efficiency, performance, compatibility and reliability are added to the definition of requirements.

System and Software Design — The various tasks needed to be performed in order to fulfill that aim are recognised and specified so as to define the **architecture** of the system.

Implementation — Actual program code is written.

Testing — The program is tested to ensure that it meets its specification.

Release — The software is released for use. As maintenance is performed and the changes are tested, the new software will be "re-released", possibly using **configuration management** [4].

Operation and Maintenance — The software is used and maintained.

The areas of testing and maintenance are of particular interest in this thesis and so will be addressed in more detail.

1.4.1 Testing

The process of checking that a system is correct is described by the collective term **validation and verification** [100].

> **Validation** is the process of ensuring that the developing system matches the user requirements [100].

> **Verification** is the process of checking that the output of a phase of the software life cycle matches the input to that phase [100].

These are paraphrased by Boehm [37]. **Validation**: "Are we building the right product?"; **Verification**: "Are we building the product right?"

Program testing can take several forms (which will be discussed in Chapter 2), but the most common involves exercising the program using data similar to the actual data that the program is designed to execute, observing the program outputs and inferring the existence of program errors or inadequacies from anomalies in that output [169]. The lowest-level program components are tested first. These are then assembled and the program is tested as a whole. During this stage, potential users can identify gaps in the original requirements and the programmers can identify faults in the program code that could cause failures.

> A **fault** in a system is a feature of the system with the potential for causing a failure.

> A **failure** of a system is said to occur when the behaviour of the system first deviates from that required by the specification [165].

In terms of software engineering, a system is correct when it fully reflects the user requirements detailed in the specification and, optionally, satisfies other measures such as the quality of the program code. Implementation and testing are linked in that testing has to determine the correctness, or otherwise, of the implementation. Any faults feed back into the implementation stage causing changes to be made.

Correctness cannot be verified by testing [32] since a successful test can really only be considered to be one which establishes the presence of one or more errors in the software being tested [141]. Whereas in traditional engineering it is usually sufficient to test to within a certain margin of safety, software should ideally be 100% correct. This is termed the **correctness problem** [40].

A more stringent version of program correctness is program reliability [2] [165] [169] in which not only must the program meet its specification, but it must also take meaningful action in unexpected situations. Program reliability is a very important issue, not least in safety-critical systems and in areas where computers are put "in control" of large amounts of money. In all practical cases it is virtually impossible to guarantee 100% program correctness, let alone 100% reliability, since there may be faults that testing has failed to reveal.

1.4.2 Operation and Maintenance

> **Software maintenance** is the modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [1].

Software maintenance can be divided into four areas: **corrective**, **adaptive**, **perfective** and **preventive** maintenance [29].

Corrective maintenance is concerned with the location and removal of faults in the program. These are errors in what the program actually does according to the current specification; it is not concerned with erroneous output caused due to a change in the specification.

Adaptive maintenance involves the updating of the program due to a change in the environment in which it has to run. This may be a minor change which does not involve much change in the structure of the program, for example a change in printed output from English to American spelling, or may be a major change, such as rewriting the program to run in a distributed fashion on a network.

Perfective maintenance is maintenance resulting from a change in a program's specification. This might be as simple as a change in the format in which a report is required, or as complex as the addition of a different kind of account to a financial banking program. Perfective maintenance takes up as much as half of a maintenance programmer's time [29].

Finally, **preventive maintenance** is the modification to software undertaken to improve some attribute of that software, such as its "quality" or "maintainability" without altering its functional specification.

1.5 Legacy Code

Maintenance is a fundamental part of the life cycle of any software and can account for 60–80% [120] of the total software costs. Hence most software is old "legacy" code which has been heavily maintained. Such software usually represents a large financial investment and so cannot just simply be discarded and rewritten due to the arrival of new technology or a change in the specification.

Lehman and Belady [119] characterised the specific problems of maintaining old code in their five laws of software maintenance.

Continuing Change — A program that is used in a real-world environment must change or become less and less useful in that environment.

Increasing Complexity — As an evolving program changes, its structure becomes progressively more complex unless active efforts are made to avoid this phenomenon.

Program Evolution — Program evolution is a self-regulatory process and measurements of systems attributes (such as size, time between releases and number of errors) reveals statistically determinable trends and invariances.

Conservation of Organisational Stability — Over the lifetime of a program, the rate of development of that program is approximately constant and independent of the resources devoted to system development.

Conservation of Familiarity — Over the lifetime of a system, the incremental system changes in each release is approximately constant.

The first two laws are technical hypotheses which have been found to be empirically true for all software, whereas the other three are non-technical and depend on the organisations developing and using the software.

The first law says that as soon as a program has been written it is out of date. The reasons for this are several. Users perceive new features which should be added to the software; new features are added to the hardware which can be used to enhance the software; faults are found in the software and these need to be corrected; the software needs to be moved to another operating system or machine; or the software needs to be made more efficient. The second the law of software evolution says that as changes (excluding preventive maintenance) are made, the structure of the program becomes more complex due to the fact that programmers are unable or even unwilling to use software engineering techniques. However, the problems go beyond this.

First, maintenance is often performed rapidly in an *ad hoc* fashion for several reasons, not least of which is because there is often little time to produce a carefully designed modification. With continued change, programs tend to become less structured. This is manifest by out of date documentation [158], code which does not conform to standards, increased time for programmers to understand code, increased ripple effect of changes and so on. These characteristics can, and usually do, imply higher software maintenance costs [7].

Systems undergoing maintenance often become progressively more difficult to change [118]. This is due to the fact that as maintenance is performed it becomes increasingly difficult to understand the program's function, and it may not even have been possible in the first place since programming involves informal, undocumented decisions which are not available to the maintenance programmer

[16]. Hence, it is often the case that the software cannot be changed without introducing unforeseen side effects, due to interdependencies between variables and procedures which the maintainer did not detect.

In addition to the technical problems there is often also a management problem. Most programmers consider work spent on maintenance to be an inferior activity to development (a view which is often reinforced by the working conditions and salary scales) since it distracts them from the more "exciting" work of software development. This creates low morale. As a result, when it becomes necessary to perform maintenance, rather than employing a systematic maintenance strategy, corrections tend to be rushed, not thought through, undocumented and poorly integrated with the existing code. It is not uncommon for such maintenance itself to introduce further errors and inefficiencies.

From these problems it can be seen that software maintenance would benefit greatly from a rigorous[4] engineering approach but, unlike development, few widely established methods yet exist.

1.6 Summary

Early *ad hoc* methods of producing software were unsuccessful in coping with the trends of increasing size and complexity so software engineering was suggested as a solution. Zelkowitz [198] gives the goals of software engineering as to:

1. Use techniques that manage software complexity;
2. Increase system reliability and correctness; and
3. Develop techniques to predict software costs more accurately.

[4] There are cases in which a rigorous approach is *not* used but which nevertheless have well controlled maintenance through informal methods such as design reviews and testing. The Space Shuttle software is such an example [108].

Software engineering has made a reasonable attempt at meeting all of these goals and has certainly improved the situation that preceded it. However, looking more closely, there is still some way to go before software engineering can truly be ranked alongside other engineering disciplines. The third of these goals is outside the scope of this thesis. Taking the other two in reverse order:

Although (as shall be seen later) there are theoretical solutions to the problems of reliability and correctness, they are not cost-effective enough, due to the extra work that they impose on the software engineer, to be widely accepted.

Software complexity has been partially dealt with through the use of improved programming methods such as modular programming. However, these methods do not readily apply to legacy code which was originally developed, or has been changed, in an *ad hoc* fashion and, thus, are not usually of benefit to maintenance. In particular, once a program undergoes maintenance it will become ever more complex.

Thus, the situation remains that while software engineering has been successful up to a point, several major failings still remain. Two of these will be considered further: the correctness problem and the maintenance problem.

> The **Correctness Problem** is the problem of producing software that performs according to some predefined specification.

> The **Maintenance Problem** is the problem of performing software maintenance rigorously and in such a way that software quality does not deteriorate as a result of this process.

These can be summarised as (a) how should software be developed so as to ensure that it has a given reliability, and (b) how should the software be subsequently maintained so as to preserve this reliability.

1.7 The Contributions of this Thesis

This thesis will approach the maintenance problem by adopting a correctness-preserving reverse engineering approach. Inspiration for this will be taken from a proposed solution to the correctness problem: formally-proved program transformations. It will make a four important contributions to the area of maintaining legacy systems:

1. The system will be shown to work. Unlike existing transformation systems, the system described here will be demonstrated to work at reasonable speed on medium-to-large sized programs taken from an industrial environment.

2. The system. will have an innovative architecture. It will be based on a middle-out design with certain distinct sub-systems such as a pattern matcher and a system for performing symbolic mathematical and logical evaluation and simplification.

3. Efficient representations will be adopted for programs and for transformations.

4. A language, $\mathcal{MET´A}$WSL, will be used to express program transformations. This language will embody knowledge about writing transformations; it will be transferable to other implementations; it will allow transformations to be expressed clearly and concisely; and it will enable arbitrarily complex algorithms to be incorporated, facilitating sophisticated automation.

Chapter 2

A Survey of Solutions to the Correctness and Maintenance Problems

2.1 Introduction

The previous chapter introduced software engineering and considered its successes and failures. While its successes have been considerable, there have also been two important drawbacks relating to correctness (an attribute of the software product) and maintenance (an attribute of the software process). This chapter examines the ways in which the two problems have been addressed, and then proposes a single method by which both problems will be tackled.

2.2 Solutions to the Correctness Problem

Verification can be approached in any of three ways [33]:

- Look and see;
- Test exhaustively; and
- Express the program formally and apply a proof.

Corresponding to each of these three approaches, a number of different solutions have been proposed and, within each approach, a number of tools have been developed to aid the software engineer. The approaches will be dealt with in turn; each will be judged according to its ease of application, its rigour and its effectiveness.

2.2.1 Look and See

Program Inspections

The most simplistic solution to the correctness problem is the "look and see" method which relies on the fact that any faults in the software will be easy to identify, if not by the original programmer, then by disinterested programmers with suitable experience. When confronted with a coding error, the original programmer would be less likely to consider it as such than would a disinterested party since, if he[1] had made a logical mistake once, there is no reason to suppose that he would not be likely to make it again. Also, professional pride would bias him against finding faults that would seem to devalue his work.

In a formal inspection, as originally described by Fagan [69], a small group of people would examine the code in a number of different stages looking for likely errors. The team would be chaired by a moderator whose job it would be to motivate the other team members; the other team members being the program's author, a tester (who would consider the code from a testing point of view) and a reader whose job it would be to present the code to the team.

[1] For stylistic reasons, pronouns are given in the male gender. It is not the author's intention to exclude the possibility of their referring to a female person.

The stages in program inspection would be:

Planning — which involves arranging the inspection and organising the team;

Overview — in which a general description of the program to be considered is presented;

Individual preparation — in which each team member considers the program and its specification;

Program inspection — in which errors are identified, but not corrected;

Rework — in which the program is modified by its author in the light of the inspection; and

Re-inspection — in which the process is repeated.

Program inspections are cheap, straightforward and, although managerial and technical skills are advantageous, do not require any special techniques to be learned by the programmer or the specifier. Inspections may be able efficiently to identify potential faults in the software but by no means guarantee to find them all. Thus, while they are useful and have proved efficacious in practice when properly applied [160] [111], they do not guarantee the correctness of the software.

Static Program Analysers

Another, similar but more limited, "look and see" method uses the computer itself to perform the inspection. This is known as static analysis and tools which perform this task are known as static analysers.

> **Static program analysers** are tools which examine the source code of a program and identify possible faults and anomalies.

Static analysers can check for potential errors, which are purely syntactic, such as uninitialised variables, but are unable to detect semantic errors, i.e. errors where the program is written "correctly", but is performing the wrong function. The list below illustrates potential problems that can be identified.

- Undeclared variables;
- Variables used before initialisation;
- Parameter type or number mismatches;
- Unreachable code;
- Non-terminating loops;
- Uncalled functions or procedures;
- Unused function results; and
- Incorrect array references.

Static program analysers are very cheap and quick to use, as they are automatic, and can be used by the programmer without requiring outside assistance. However, while they help to eliminate frequently occurring sources of faulty behaviour, they are not generally able to identify all faults in the code (let alone design, architecture and so on) and hence do not solve the correctness problem.

2.2.2 Test Exhaustively

The exhaustive testing of software has been an area of considerable research, reflecting the fact that it has grown from an after-programming evaluation process to a concept that is an integral part of each phase of the system development life cycle [156]. Consequently, this has led to the production of a number of different methods and tools.

There are two main types of testing: black box testing in which the internal structure and behaviour of the system being tested is not considered, and white-box testing in which it is. Black-box testing is typified by acceptance testing to ensure that software meets its specification or user requirements, while white-box testing is typified by unit testing in which it is necessary to examine the structure of the code unit (module, procedure etc.) in order to ensure that the tests exercise as many of its statements and paths as possible.

Two of the more important testing tools are the test case generator, which generates typical data on which the program would be run, and the symbolic evaluator which takes (part of) a program and executes it using symbolic, as opposed to numeric, data.

Despite the effort that has been put into testing, the fact remains that, unless every path through the program can be tested with every combination of inputs, testing cannot demonstrate that a program is correct. For small programs, or for programs with a very simple structure, it may be possible to test all the paths through the program but, as the size of the program increases, so the number of paths through it increases. This increase may be linear, but it could also be exponential, making exhaustive testing infeasible. (In general it can be difficult to determine whether testing is cost-effective. Perry [156] gives thirty metrics to measure the effectiveness of testing.)

Despite these apparent problems, testing has proved to be a powerful and useful technique. Shooman [164] gives some statistics as to the success of testing; in particular: test hours versus the number of new instructions, discovery and correction times, and the difficulty of correction and detection against time. A notable result was that 80% of errors were identified after one execution of the software, while the average number of executions to find each error was 1.35.

Different kinds of testing (module test, integration test, code reading and design reviews) are each suited to the identification of different kinds of error (logic, documentation, timing, specification etc.). Thus, although no single test technique is uniformly good over the spectrum of error types [164] there does appear to be a test method suitable for finding each kind of error. However, testing alone cannot solve the correctness problem, i.e. *prove* that a program is correct, in any but the simplest examples.

2.2.3 Formal Solutions

The success of engineering in general can, to a large extent, be attributed to the discovery and deployment of the theories that lie behind the work of the engineer. Currently not much of this theory is in place for software engineering, and that theory which is in place is rarely understood or used. However one of the most promising solutions to the correctness problem does use an underlying theory. In this method, the specification and program are expressed mathematically and a proof that they are equivalent is found. The methods of software development which are based on the underlying mathematical theories of programming are known as **formal methods**. Before looking at the advantages and disadvantages of formal methods, it is necessary to give some definitions.

> **Formal methods** of program construction are methods which are carried out in a language whose vocabulary, syntax and semantics are formally defined.
>
> A **formal software specification** is a specification expressed in a language whose vocabulary, syntax and semantics are formally defined [169].
>
> The **vocabulary** of a (specification) language is the the collection of "meaningful" symbols that it possesses.
>
> The **syntax** of a (specification) language is a definition of the way in which the symbols are allowed to be combined.
>
> The **semantics** of a (specification) language is a definition of the "meaning" attributed to given symbols and combinations of those symbols.

Program specifications can have various degrees of formality. At the informal end of the spectrum, the specifications can be expressed in some convenient combination of English, diagrams[2] and mathematical notation. In contrast, formal specifications are written in a language with explicitly defined syntax and semantics [121]. Thus, formal methods which start with a "formal specification" are able to produce a program which can be mathematically shown to meet that specification[3] by a series of steps akin to a proof. The need for a formal semantic definition of the specification language and the ability mathematically to manipulate this language mean that it must be based on mathematics and not "natural language".

Mathematical formulation allows the specifier to remain much further from the computer than would otherwise be the case and, in this context, any programming language is already too near [91]. In addition, the proper use of mathematical abstractions in the development process helps to create software systems with a coherent and suitable structure. Also a rigorous[4] development method where correctness of developments steps can be justified in a mathematical sense, strongly diminishes the risk of introducing errors and inconveniencies to the system during development [68]. Further advantages of using formal specifications are given by Sommerville [169] as follows:

- The development of a formal specification provides insights into, and an understanding of, the software requirements and software design;
- Given a formal system specification and a complete formal programming language, it *may* be possible to prove that a program conforms to its specification;
- Formal specifications may be automatically processed, for example by the computer; and
- Formal software specifications are mathematical entities and may be studied and analysed using mathematical methods.

[2] Diagrams may be either formal or informal. The formal use of diagrams was mentioned in Section 1.3.4.

[3] By definition, it is not possible to prove *formally* that a program meets an *informal* specification.

[4] A rigorous method is one in which reasoned justification is provided for the approach adopted, and decisions taken, at each stage.

The process of producing a proof that a program meets its specification can be as hard (or harder) than producing the program in the first place; thus mathematically demonstrating a program's correctness can be accomplished in one or more ways [179], such as:

1. Write the program and then attempt formally to verify its correctness against the specification;
2. Develop the program and its correctness proof concurrently;
3. Starting with the program, successively transform it into an executable program by means of a series of transformations which have already been proven to preserve the correctness of any given program.

Thus, formal specifications and associated formal methods can potentially be of great benefit in determining program correctness since a mathematical link can (usually) be established between the specification and the program. There are nevertheless some disadvantages to formal methods, or at least to certain of these methods, the most notable being the time and effort required in the construction of the proofs, and the need to understand these proofs once they have been constructed. Dijkstra [64] argues that formal proofs are shorter and easier to understand than informal ones, but goes on to say: "I have seen a number of proofs that have been produced by (semi) mechanised systems, and, indeed, these proofs were appalling!" Thus, while formal methods could solve the correctness problem, they do require a great deal of extra work by the software developers.

2.2.4 Automatic Program Verification

A method is needed of using formal methods which avoids the problem of producing proofs. Griffiths sums this up as follows [91]: "A specification should be largely mathematical and less computer oriented. From such a specification, we should, however, be capable of producing a program acceptable to some compiler."

As a result a number of automatic and semi-automatic tools have been produced to attempt to solve this problem.

It is not generally possible (because of the halting problem) for a machine to prove program correctness, and it is often easier to prove a weaker version of correctness; that is, **partial correctness**, in which the program is known to meet its specifications *provided it terminates.*

> **Partial program correctness** proof methods show that a program meets its specifications, as given by entry and exit assertions, provided that it terminates [109].
>
> Here an **assertion** is a true predicate about the program's state space (i.e. the values contained in the variables) at some stage before, during or after execution.

Partial program correctness proofs are composed of various steps: summarising the semantic content of a program in a mathematical representation, generating formulas (or assertions) called "verification conditions", and devising inductive statements which allow one to conclude the program's correctness by proving that the verification conditions are theorems in some appropriate mathematical logic [109].

There are a number of different invariant conditions on code (for example "data type invariants") which can be used to prove correctness by means of mathematical induction. Loop invariants are important since, using some formal methods, they are required in order to prove that a loop conforms to some specification.

> A **loop invariant** is a logical formula which is true before and after the execution of each iteration of a loop.

While loop invariants are important, they are hard both to identify and to prove. Methods of generating loop invariants in programs with many nested loops and procedure calls, or complex data types are still very primitive. In fact, the problem of finding the inductive assertion for any given program is theoretically unsolvable [172], although it may be possible in specific simple examples. This has not prevented work on semi-automatic derivation of loop invariants, or at least on systems which suggest loop invariants. One such system is ADI. ADI [172] will find as many loop invariants as it can, and it is hoped that the conjunction of these invariants will be strong enough to be an inductive assertion.

Another way of aiding in the development of correctness proofs is the use of **theorem provers** such as the B-Tool [11]. Theorem provers mechanise, or help to mechanise, the production of a formal proof. These have proved difficult to implement for the general case, so **proof checkers** have been more commonly used. The simplest form of proof checker takes, as its input, a series of inferences in some logical theory, such as first-order predicate calculus, along with the rule of inference to be used (for example, *modus ponens*). Using these, it would determine whether the logical formula obtained does indeed result from the designated rule of inference [31].

One of the best known proof systems is the Boyer-Moore theorem prover [39]. This tools is primarily an *induction* machine which mechanises proofs in a logical theory developed by Boyer and Moore. In doing this it uses various *ad hoc* proof strategies and expression simplifiers. Although the tool can operate without any user intervention, it can be given lemmas (as subgoals) by the user to aid it in its proof. The system is fairly powerful, but does require a large number of user-supplied lemmas before it can prove anything requiring more complex objects (for example, real numbers, logical formulae etc.) than those provided. Thus, its usefulness lies more in proof checking than theorem proving. For proving the correctness of programs, it would be necessary to give a system such as Boyer-Moore's so large a number of guiding lemmas that there would only provide a marginal advantage over doing the proof manually. There is also the problem of verifying the correctness of the theorem prover itself. Nevertheless the Boyer-Moore theorem prover has been used in the proof of correctness for the implementation of some small examples [161] [187].

Thus automatic program verification can usually only be an effective means of proving the correctness of simple programs. As Gries says [90]: "One cannot expect to produce a whole program and then prove it correct. Instead, at each stage of development, the programmer must know that what he has done is correct." This does not mean that computer-aided formal methods should be discarded completely, since there is a third path to constructing programs formally, that of successively transforming a specification into an executable program. The merits of this strategy will be considered later.

2.3 Solutions to the Maintenance Problem

The various solutions to the maintenance problem which have been proposed can be broken down into two categories: those that address the management issues and those that address the technical issues. The former are of less interest to this thesis and will only be considered briefly.

2.3.1 Management Solutions

In financial terms, software maintenance is seen as a continuing consumer of resource with a nebulous and unquantified benefit to the organisation [158]. Thus, there needs to be more organised management support of software maintenance, and this can come about through:

1. Senior management becoming aware of the importance of information technology to the organisation; and

2. Senior management viewing software as a corporate asset which can provide a competitive edge [158].

Thus, in order for the situation to change, management must become dissatisfied with the *status quo*, and therefore able to make a visible and personal commitment to any proposed solutions. Such solutions could, broadly speaking, take one of two forms: resources and quality.

Resources

The key resource in software production and maintenance is people, so a possibly effective way of improving software maintenance could be to have a separate group of programmers employed just to maintain old code.[5] However, because of the unglamourous nature of the work, it is usually the new recruits who are assigned this work. These inexperienced programmers, while they may be able to understand the logical design of the system, are usually unable to understand the conceptual model of the software since they lack experience of both software engineering techniques and domain knowledge of what the program is supposed to do. Thus, they rarely know how to find and fix faults, or make modifications.

Increasing manpower and funding for software maintenance might provide a short-term solution but, for a long-term solution, it would be necessary to adopt an approach which would improve the overall quality of the process.

Quality

Improving the quality of both the software product and the software process follows the trend of increasing concern for quality issues in industry as a whole. Better software quality management techniques include [58]:

- Standard techniques for decomposing software into functional entities;
- Strict software documentation standards;
- Design walk-throughs at each level of software decomposition;

[5] Alternative methods of strategically employing manpower, such as having the same team perform both the development and the maintenance, have also been suggested [166]. This approach mirrors other engineering disciplines in which it would be very unusual to separate the tasks of development and maintenance.

- Use of structured code; and
- Definition of all major software interfaces and data structures before detail design begins.

In addition, metrics could be employed (to measure not only attributes of the product but also attributes of the process) and better tools could be used (for example, integrating an editor, compiler and debugger into a single tool).

2.3.2 Technical Solutions

It is convenient to divide the technical solutions to the software maintenance problem into two: tools and methods.[6] The tools are designed to help the software maintainer understand the program and to test its modification to ensure that no errors have been introduced. As such, many of the tools are the same as, or similar to, those used in software testing. A selection of the tools which have been produced in order to aid the maintenance programmer is listed by Miller[7] [135]. These include the formatter, static analyser, structurer, documenter, interactive debugger, test data generator and comparator.

Software maintenance methods consist of re-engineering and reverse engineering.

Software Re-Engineering and Reverse Engineering

> **Re-Engineering** is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [30].

> **Reverse Engineering** is the process of analysing a subject system to identify the system's components and their inter-relationships, and

[6] Tools are used most effectively to support methods.

[7] Some of these tools (such as the formatter and test data generator) are already in use, whereas others (such as the structurer) are not yet mature enough to be used generally.

to create representations of the system in another form or at higher levels of abstraction [55].

Software Restructuring is the modification of software to make the software (1) easier to understand and to change or (2) less susceptible to errors when later changes are made [7].

Software re-engineering and reverse engineering usually involve taking existing program code and restructuring it (for example, splitting it into modules and procedures). Restructuring measures are not equivalent to reverse engineering and reverse engineering does not necessarily imply restructuring. Programs can be restructured, and even modularised, without reverting back to the logical design level. Conversely, programs can be transformed into a higher semantic level without being restructured [167]; however, as will be seen in Chapters 4 and 9, abstraction to a higher semantic level is simpler if the software is first restructured.

Bennett [30] lists 26 purposes for reverse engineering code. The most important ones are as follows:

- To simplify complex software, or software which has become complex due to maintenance activities;
- To improve the quality of software which contains errors, by identifying and then removing those errors;
- To remove any side effects from an implementation (i.e. unplanned state changes);
- To improve the coding quality and understandability, for example to reduce the number of control transfers, remove dead code, adopt data naming standards, improve control structure, and adopt standards of layout and commenting;
- To undertake a major design repair activity, because the original design (and, therefore, implementation) of the software may be erroneous;

- To allow major changes to be implemented (the structure, documentation and quality of the software may be so poor that it is infeasible to implement a major change without reverse engineering the software to a higher level of abstraction first);
- To help establish and support a reuse policy (for code, designs, specifications, processes etc.);
- To enable better software maintenance management techniques (in terms of planning, monitoring and control) to be introduced. This will provide better visibility of the software maintenance to be achieved and, therefore, better control;
- To bring the existing software into a more modern software engineering development environment consistent with other practices within the organisation. This should provide a higher quality of software, with lower costs for subsequent software maintenance;
- To rediscover and record the design of the system;
- To rediscover and record the requirement specification of the system; and
- To recover and record high-level information about the system including: the system structure, functionality, dynamic behaviour, rationale and construction.

These purposes can be broken down into three categories: fixing the program, changing the program and understanding the program. This last category, which includes recapturing information about the software, allows it to be worked on using modern software engineering techniques, thus putting it back "under control". This is important, as it means that rather than requiring *ad hoc* methods for performing maintenance, methods such as "structured retrofit" can be used.

> Miller and Lyons define **structured retrofit** as the application of today's structured program techniques to yesterday's systems in order to meet tomorrow's demands [134] [124].

Both re-engineering and reverse engineering are activities which can be supported by the computer.

In the process of reverse engineering, there are many different sources of information about the program undergoing maintenance. This information comes in the form of the documentation and manuals, comments in the code, the original specification (if there ever was one!), and the code itself. If the program has undergone many changes or is very old, then the documentation, whatever its form, is unlikely to provide an accurate description of the program's function. Likewise, if a specification of the program is available and *up to date*, then understanding the program is made considerably easier by first understanding the specification. This is because of the complementary nature of the specification, which expresses the purpose of the software without the impediment of the implementation details. However, very often, the specification may not have been updated as the program was changed and it is, therefore, as unreliable a source of information as the informal documentation.

This leaves the source code as being the only reliable source of information, and it is *necessarily* reliable since it is the program code which the system is running. Thus, any reverse engineering tool which is to produce an *accurate* description of a program should work primarily on source code.

There has been much of work on re-engineering and reverse engineering, and there is good reason for this. Both offer to bring old software up to date — re-engineering by directly reimplementing the existing program, and reverse engineering by first producing a high-level specification or description of the code. However, a problem still exists; that of ensuring that the new program (in the case of re-engineering) or the high-level description or specification (in the case of reverse engineering) is an accurate description of the software system in question. Essentially, this is the correctness problem in reverse. Thus, it would seem realistic to ask whether solving the correctness problem could provide a solution to the maintenance problem via reverse engineering.[8] This is the topic of the next

[8] It could also, potentially, offer a solution by means of re-engineering. However, if, as would be necessary, there is a way of formally linking a program with its specification, then re-engineering can be accomplished by means of a combination of formal reverse engineering and of formal re-

section.

2.4 Can One Solution Solve Both Problems?

Due to the necessarily mathematical nature of software, it would seem reasonable to suppose that formal methods could be applied to reverse engineering by proving that a given specification describes an existing program, or by deriving a specification[9] from an existing program. This is indeed the case, provided that a general method exists for determining this correspondence, and this in turn is the correctness problem. Hence, a solution to the correctness problem, when placed in a suitable reverse engineering environment, would also provide benefits in solving the technical side of the maintenance problem. That just leaves the problem of finding simple means of providing a formal link between a specification and a program.

2.4.1 Program Transformations

The third path to the formal construction of programs is that of transformation. This method starts with the program specification, and successively transforms it into an executable program by means of a series of transformations which have already been proven to preserve the correctness of any given program. The precise meaning of the terms "program transformation", "transformation system" and so on will be given in the next chapter. For now, it is enough to comment that a program transformation is a change made to the text of a program in such a way that the program's semantics remains unchanged.[10]

implementation.

[9] All programs, even those that have been heavily modified, necessarily have a specification in that each statement can be specified separately, so that the specification of the whole program would be a combination of these. However, the question of whether such a program has a *concise* and *useful* specification is an open question.

[10] In some approaches, the semantics may be refined in order to reduce non-determinism or increase program definedness.

Of the three different methods of formally showing that executable programs meet their abstract specifications, this seems to be the most promising for two reasons.

First, it seems to be the only one which is capable of scaling up to large programs; this is because a single proof of a large program would be almost impossible to understand, let alone develop [179], while transformations will be seen to apply to programs of any size.

Second, it can be shown that program transformations have inverse operations. That is, if a program has been transformed, then there will be a transformation which has the opposite effect and takes it back to the original version. This phenomenon can be used in program maintenance, by performing transformations that are the "inverse" of those used in development, to derive specifications from existing programs. (As yet, it is only a hypothesis that it is easier to modify and evolve a transformational development than it is to do so for a conventional method with a corresponding proof, but it seems a reasonable claim since, in most cases, the development will proceed along similar lines with the same transformations being applied.)

The transformational approach to programming has some roots in the sixties, when it was shown [115] [116] that certain well-known programming constructs (such as conditions and loops) of ALGOL-like languages were nothing but notational variants on the Lambda-Calculus developed by Alonzo Church [57] in the 1930s. It was also found that certain complex linear recursion schemes could be transformed into simpler recursion schemes such as tail-recursion or iterative schemes. This led to the discovery of increasingly many transformations.

Burstall [48] stated that motivation for the transformational approach to program development is "that programs are complicated, hard to understand and prone to errors because we want them to be efficient... So the idea is to start with a program which does the right job but entirely sacrifices efficiency in favour of simplicity and modularity. We then transform it by correctness-preserving transformations until a tolerably efficient, though less perspicuous, program is obtained." The aim, then, is to extend the scope of transformation systems in two ways: (a) so that it is only necessary to produce a specification since that can be transformed into an efficient program, and (b) so that it is possible to transform an existing program into a specification.

2.5 Summary and Conclusions

The problem of ensuring program correctness can be approached from several directions, most notably through program inspections and other "look and see" methods, testing and formal methods. This last is the only method that can *guarantee* the correctness of a program, but the overhead caused by the need to construct proofs is considerable. Automatic proof generation would alleviate this difficulty, but is currently impractical. However transformational development would provide a formal development method without the need to construct proofs.

Software maintenance needs to be addressed as a management issue in order to impose quality onto the process by means of using more rigorous approaches. Such approaches mean the use of improved tools, improved methods and probably both. An effective tool would be one that permitted the use of formal reverse engineering, since designs could then be recovered more easily. A tool based on program transformations would offer the required functionality.

In conclusion, an effective transformation-based tool would provide a method of achieving enhanced quality in software development, but more importantly, in the often-overlooked phase of software maintenance.

2.6 Topics Addressed in this Thesis

The next chapter introduces transformation systems and surveys existing systems, considering their benefits and shortfalls. It then addresses the question: "If transformation systems are so good, why are they not used more widely?".

Chapter 4 describes Ward's work which forms the background to the transformation system described in this thesis and the ReForm project of which it is a major part. The specific problems to be addressed are also defined in this chapter, together with the criteria for success against which they will be judged.

Chapters 5–8 present the transformation system that has been created, concentrating on the new language *MetaWSL* in which the transformations are written. This language is assessed to determine its suitability for representing various kinds of transformation, concentrating on the transformations needed to perform reverse engineering.

Finally, the results that have been obtained with the resulting transformation system are examined in order to identify its strengths and weaknesses as a tool which purports to aid the maintenance programmer.

Chapter 3

Program Transformation Systems

3.1 Introduction

This chapter introduces transformation systems, giving some important definitions and explaining how these systems can be categorised. Existing transformation systems are analysed in terms of their benefits and shortfalls. The question why, if transformation systems have so many alleged advantages, they are not used more widely, is addressed by identifying specific problem areas. Finally, a summary is presented of what can be learned from these systems and an explanation is given of how these features could be incorporated into a new transformation system. This, therefore, enables the definition of objectives and criteria for the success of the research described here to be presented.

3.2 Important Definitions

Program transformation is a process which treats a program as an object in itself. Transforming a program will preserve some of its properties and alter others. While many choices can be made about what properties to preserve and alter, the workers in the transformation field have focused on altering the performance characteristics of programs while preserving their semantics [17]. From this perspective, a program transformation system and a compiler are essentially equivalent; both translate a high-level program (specification) to a low-level, semantically-equivalent implementation [80]. Transformation is more powerful, however, as it gives more flexibility, for example in the choice of the implementation of sets as lists, arrays, hash tables etc. The transformation system discussed in this thesis will be slightly different from conventional transformation systems in that, while the characteristic that is preserved is still the program's semantics, the aim is that the characteristic that changes is the program's comprehensibility, thus facilitating reverse engineering.

There have been numerous different attempts at defining the terms used in transformational programming. Partsch [152] gives some useful definitions:

> A **program scheme** is a representation of a class of related programs.
>
> **Transformation rules** are partial mappings from one program scheme to another, such that an element of the domain and its image under mapping constitute a correct transformation.
>
> **Transformational programming** is a methodology of program construction by successive applications of transformation rules. Transformational programming is a "constructive approach". This contrasts, for example, with pure verification approaches, where the question of how to obtain the program to be verified is ignored.
>
> A **transformation system** is an implemented system for supporting transformational programming.

Some alternative definitions for these terms have been given by Bauer [26] as follows:

> A **transformation** is the generation of a new piece of program from a given one.
>
> A transformation is said to be **"correct"** if the programs are semantically equivalent.
>
> A **transformation rule** is a mapping between sets of programs. In general such a mapping is a partial one, as it is only defined for particular kinds of programs.

In this thesis the terms "program transformation" or just "transformation" will be used in place of the phrase "transformation rule" and will be defined thus:

> A **program transformation** is a change made to the text of a program in such a way that the program's semantics remains unchanged.[1]

There has already been much research into transformational programming and this has resulted in a large number of experimental systems — see [152], [73] [192] for surveys of these systems.

In general, transformation systems can be classified in a number of ways [152] [151] as follows:

- By their purpose;
- By their transformation catalogue type;
- By their method of expressing transformations;

[1] It will be seen later that in extracting a specification, it may be necessary to change a program so that its semantics become less defined in a precise sense.

- By their level of automation;
- By the language they transform; and
- By their level of formality.

3.3 Purposes of Transformation Systems

Transformation systems fall roughly into two categories: those that use transformations for some specific, limited, purpose and those that are general-purpose program manipulation tools. The former type is typified by the *supercompiler*.

> **Supercompilers** are defined as highly automated transformational programming systems that can translate high-level, mathematical, problem specifications into machine code for a variety of target computers [148].

Specific examples of special-purpose transformation systems include SETL (see Appendix A.1) [62], RAPTS (see Appendix A.2) [148], TAMPR (see Appendix A.3) [42] and the Restructurizer (see Appendix A.4) [7] [168]. While all these systems have proved successful to varying degrees in their own fields, they are not suitable, due to their specificity, either to general program development by transformation or to software maintenance, in particular to reverse engineering by transformation.

In contrast to limited-domain transformation systems, a number of general-purpose transformation systems have been devised, and some of these have been implemented. Notable general-purpose transformation systems include Burstall and Darlington's Work (see Appendix A.5) [49], TI (see Appendix A.7) [18] and CIP (see Appendix A.10) [27]. The aim of these transformation systems is to allow the user to construct programs by transformation while the computer performs the clerical work of constructing the intermediate program versions.

Currently, some of these systems have been used in software maintenance, but mostly just to the extent that transformational developments can be replayed from slightly different starting conditions to produce alternative program versions. Systems that have been used in this way include the ZAP system (see Appendix A.6) [70] [71] [74] and DEDALUS (see Appendix A.11) [127]. Unfortunately, no work has been undertaken on transforming existing code with these systems, rendering them of little use in the realm of legacy code. However, two pieces of work do seem applicable to reverse engineering: Kozaczynski's program transformation system (see Appendix A.13) [113] and Ward's work (see Appendix A.14) [177]. There are problems, though. Kozaczynski's system lacks a formal basis — a necessary requirement for a system which is to be used on code before its purpose is understood — and Ward's work only exists in the form of a number of theorems on program equivalence; no transformation system has been built on this work.

3.4 Transformation Catalogues

Most transformation systems rely on a predefined collection of rules (which may be expressed in various ways) describing how the program may be changed. There are two (not necessarily opposing) ways of constructing the collections of rules. The first method is the **catalogue approach**. Examples of this manner of working include TI (see Appendix A.7) [18] and PSI (see Appendix A.9) [87] [85] [86] [88].

> A **catalogue** of transformation rules is a structured collection of transformation rules relevant for a particular aspect of the development process [152].

In this approach there is a large set of transformations covering all aspects of program development. For example, there could be rules containing programming knowledge (such as how best to search an ordered tree), there could be rules about features of the language (such as how to remove recursion), there could be rules relating to the programming domain (for example, the rules of arithmetic), and finally there could be rules about efficiency of implementation and choice of data structure.

This approach, although powerful, has two drawbacks. First, the rules are fixed so if the system is used outside its perceived domain it becomes less suitable and, second, with such a large catalogue, finding the "best" transformation to apply at a particular point is relatively time consuming, especially if the programmer is unfamiliar with all the options at his disposal.

The other method is the **generative set approach**. Examples of this catalogue style include Burstall and Darlington's work (see Appendix A.5) [49] and RAPTS (see Appendix A.2) [148]. In this approach there is a small set of powerful, possibly language-independent, elementary transformations from which others can be produced by combination.

Compared with the catalogue approach, this method is much more flexible since transformations appropriate to the situation can be constructed by the programmer (who knows they will be "correct", in the sense that they preserve the semantics, since the elementary transformations are correct). This advantage is also the approach's drawback since the programmer's effort is shifted on to trying to work out what sequence of very minor changes he needs to make in order to produce some desired large-scale effect.

3.5 Methods of Expressing Transformations

A transformation rule can be described in one of two ways: as an ordered pair of program schemes, the "input template" and the "output template" [26], or in the form of an algorithm, which takes a given program as input and produces an equivalent one as output — provided that the input program is in the domain of the rule (compilers behave this way). Of the former type, examples are TAMPR (see Appendix A.3) [41] and DEDALUS (see Appendix A.11) [127] [125]. Of the latter type, examples are the ZAP system (see Appendix A.6) [70] [71] [74] and

Hildum and Cohen's work (see Appendix A.12) [95].

Representing transformations in terms of input and output templates enables the correctness of the transformations to be checked easily. It also makes their purpose clear and reduces the amount of storage that each requires. However, it is clear that certain information, such as whether a variable is assigned in a particular section of code, cannot be represented in terms of patterns; or at least cannot be represented without adding greatly to the patterns' complexity. One means of tackling this problem, which has been adopted by CIP (see Appendix A.10) [27], is to represent the transformations as input and output templates *plus* additional "semantic" predicates on the code being transformed. This method works well for simple transformations, but for more complex transformations, in particular those that require information about a part of the program other than the part that the transformation changes such as replacing a variable by its value, it becomes rather clumsy [155]. Thus, a system which represents transformations as algorithms seems more flexible, even if the expression of each transformation might be more difficult to construct, read and verify.

3.6 Automation Level

There are also different approaches to applying transformations.

User responsible systems, such as TI (see Appendix A.7) [18] and CIP (see Appendix A.10) [27], make the user responsible for the selection of each and every transformation. In order to make these systems viable, it is necessary to have sufficiently high-level transformations such as "remove recursion" so that the programmer does not get unnecessarily involved with minor details. Hence this method is best suited to working with large-catalogue systems.

Fully automatic systems, such as RAPTS (see Appendix A.2) [148] and TAMPR (see Appendix A.3) [42], are similar to very-high-level optimising compilers in that they run unaided. These systems use heuristics, machine evaluation of different possibilities, back-tracking and other strategic devices to select transformations. However, any given system is only really applicable in a small domain. Another problem is exemplified by the SETL system (see Appendix A.1) [163] [61]; although it has been used to deal with some complex problems it has required a great deal of informal reasoning which would be difficult to treat automatically in this, or indeed any, system.

Semi-automatic systems, such as ZAP (see Appendix A.6) [70] [71] [74] and GLITTER (see Appendix A.8) [75], let the user set a medium range goal for the computer to perform automatically. A typical goal might be the removal of a loop or a change in data structure. These systems have the advantage that the user can make intuitive guesses (based on his programming knowledge) as to the best way forward, but leave the computer to do the actual mechanical manipulation of the program.

3.7 Language to Transform

There is also a choice in the language used for the systems. Some systems, such as SAFE (see Appendix A.7) [190], work only with specification languages which allow formal statements of problems but not their implementation. Others work with (sometimes specially designed) programming languages in which solutions can be formulated; examples include the Restructurizer (see Appendix A.4) [7] [168] which uses COBOL, and TAMPR (see Appendix A.3) [42] which uses 20 language levels ranging from pure applicative LISP to FORTRAN. Still others systems are used to move between specification and programming languages and work in wide spectrum languages in which both specifications and programs may be expressed; examples include GIST (see Appendixes A.7 and A.8) [81] [19], CHI which uses a language called V (see Appendix A.9) [89] and the CIP project (see Appendix A.10) [27] which uses a language called CIP-L.

3.8 Formality

For transformation systems to contain proven transformations it is necessary to have a mathematical formulation of the language that is being used. While some transformation systems, for example CIP (see Appendix A.10) [153] [26] and Ward's work (see Appendix A.14) [177], do have a formal basis, most do not. Only those that have a formal basis are suitable for manipulating legacy code since even if the user has no understanding of the code that he is transforming, he would still want to be able manipulate it so as to be sure that his changes preserve the code's meaning.

There are several ways to define the semantics of specification and programming languages, and hence to prove transformations. See [140] and [144] for surveys of the main methods for describing semantics.

3.8.1 Semantics of Specification Languages

There are three main approaches to the semantics of specifications: the state-machine approach, the algebraic approach and the modelling approach. Each of them defines results in terms of underlying abstractions, usually associated with some known mathematical entity about which it is possible to reason with rigour. With the algebraic and state-machine approaches, the underlying abstraction is part of the approach; with the abstract model technique, it is chosen by the specifier [31]. The three approaches will be illustrated by means of a simple "push" operation onto a stack.

The **state-machine**[2] technique was first developed on an *ad hoc* basis and was subsequently formalised. The underlying abstractions of state machines are integers and Booleans, but these have been extended to include real numbers and character strings. A specification is a set of functions that specify transformations on inputs, and this set may be viewed either as defining the nature of an

[2] A state is a mapping from a given set of components (i.e. variables) onto values.

abstract data type or as describing the behaviour of an abstract machine. A state-machine specification is given in terms of states and transitions between states. Its functions are divided into two classes [31]:

V-Functions — allow an element of a state to be observed but do not define any aspect of transitions.

O-Functions — define transitions by means of effects. The effect of an O-function is to change a state; this is done by denoting a V-function and altering the value it will return.

V-functions and O-functions correspond to array variables and operations in a programming language, respectively, in that V-functions can be treated as mapping symbols (i.e. variables) onto values, and O-functions modify the values to which V-functions map. V-functions can, therefore, readily describe any structure that resembles an array, list, tree, or constructs obtainable by the structuring facilities of languages such as PL/1 or COBOL [31].

The definition of a stack push would be:

```
OFUN  push(item)
      EFFECTS  'stack(depth) = item
               'depth = depth + 1
```

While the state-machine approach has not been widely described in the literature, it is frequently used in practice as a result of its suitability for validation of security as is described by Berg *et all* [31]. There are, however, two severe drawbacks with this approach. The first is that specifications of non-array-based structures, for example algebraic formulae, rapidly become very complex. Also, the specification of exception definitions — that is error conditions — is very weak, since there is only really scope for a function to return an "exception number".

SPECIAL [92] is an example of a state-machine specification language.

Algebraic specifications are based on a concept of defining abstract data types which is called algebraic. A data type is characterised by one or more sets of values *and* the operations that are allowed on the values [79]. As an example, while stacks and queues of integers both correspond to sets of integers, they are different data types since different operations are allowed on them. The technique is called "algebraic" because the values and functions of a specification can be viewed as forming an abstract algebra.[3] Algebraic specification languages also assume built-in functions such as If-Then-Else and boolean operators. Functions, which are mathematical functions in that they may not have side effects, are defined in algebraic specifications by stating their relation to one another.

It is not possible to define a stack push in isolation from the other stack operations. The definition of a stack would include algebraic equations such as:

$$\text{pop (push (stack, item))} = \text{stack}$$

together with typing information, which in this case would indicate that "pop" is function which maps stacks to stacks.

While they express simple objects well, algebraic specifications share the problem of exception definitions. Another problem with algebraic specifications is the number of hidden or auxiliary functions (i.e. functions which preserve certain internal values between function calls) required to specify even simple objects. Algebraic specifications are also difficult to read; Berg *et al* give an example of two specifications, one of a bag and one of a set, which differ only very slightly in one line (of nineteen) [31] making it difficult to distinguish between them. Finally, limiting functions to those which cannot have side effects makes it impossible to specify a stack "top" statement which returns the top value of a stack and has the *side effect* of popping a value off the stack.

CLEAR [50] and OBJ [84] are examples of algebraic specification languages.

Model-based specifications were developed by Hoare [96] as part of a unified technique for the specification and verification of abstract data types. It is also known as the predicate transform method.

[3] Abstract algebra is concerned with general mathematical structures which have analogues of the arithmetical operations; for example, Booleans, groups, matrices and vectors.

A model-based specification is a description of a software system presented in terms of a particular state space, together with a collection of operations and functions which act upon it [136]. *Preconditions* and *postconditions* are used to indicate under what conditions given functions are valid, and what results they give under those conditions. Functions are defined in terms of an underlying abstraction (or model) that is defined by the specifier. An abstract model has no intrinsic meaning, but rather its meaning depends on the selected underlying abstraction, and so for a model-based specification to be useful an appropriate underlying abstraction must be chosen. Thus, this abstraction can be anything about which it is possible to reason formally but is generally carefully chosen to be an appropriate abstraction of some commonly-arising, well-defined computer-oriented concept (such as sets, sequences, Cartesian products and various forms of relation).

As an example, a bounded integer stack would be defined by using, as the underlying abstraction, a sequence, which in turn would be defined algebraically. (Algebraic specifications express the behaviour of simple objects very well.) The stack definition would include the following lines:

```
FUNCTIONS   push(item: integer)
            PRE          $0 \leq \text{length(stack)} \leq (\text{max\_length} - 1)$
            POST         $'\text{stack} = \text{append(stack, item)}$
```

Two factors affect the appropriateness of a given abstract model: whether the function to be specified can be expressed in the precondition/postcondition format, and whether the chosen underlying abstraction permits a "clean" specification of the desired functions. Assuming that a good underlying abstraction is chosen, the only drawback to this approach is that the implementor of the specification may be swayed in his choice of data representation by the underlying abstraction used in the specification.

Z [170] and VDM [103] [94] are examples of model-based specification languages.

Berg *et al* [31] examine these different technique in detail, giving certain basic requirements that an adequate specification language should satisfy. Model-based specifications have the fewest disadvantages (provided suitable abstractions are chosen) and appear to have the greatest potential for writing clear and concise specifications. It should be noted, however, that these different methods have been shown to be equivalent.

3.8.2 Semantics of Programming Languages

Just as there are a number of ways of specifying the semantics of a specification, so there are different ways to specify the semantics of a program; however, not all of these are conducive to the production of a useful transformation system. The most important methods of expressing the semantics of a programming language are operational semantics, axiomatic semantics and denotational semantics.

The semantics of a programming language can be defined via a hypothetical, or abstract, machine[4] which interprets the programs of that language; such methods have been called **operational semantics** [35]. The semantics of a construct is specified by the computation it induces when it is executed on such a machine. In particular, it is of interest *how* the effect of a computation is produced [144]. The machine performs a mapping of initial internal states to final states by carrying out a sequence of these primitive operations (each producing a new internal state). The whole sequence of such states corresponds to the execution of the program. It is assumed that the state space and operations defining the primitive abstract machine are so simple that their meaning or effect cannot possibly be misunderstood. Nevertheless, the operations of the abstract machine must still be defined formally, leading to the potential for infinite regress unless the abstract machine is defined in some other way.

Two programs are equivalent according to their operational semantics if they lead to the same sequence of operations performed by the primitive abstract machine. Thus, the operational approach characterises the actual effect of program execution by relating it to executions at a separate, more primitive, level. More

[4] An abstract machine is defined by a pair consisting of a state and a set of operations for effecting state changes.

importantly, however, the operational approach defines the semantics of a program for each specific computation of that program, rather than for the class of all computations that it can perform [31]. Thus, proving that two programs have the same operational semantics necessitates considering every possible execution of the program. Linked to this is a more fundamental drawback to operational semantics when working with program transformations. While it can be easily seen, for example, that recursive and iterative versions of the same program give identical results with identical input, they lead to completely different sequences of internal operations. Thus they have different operational semantics and a transformation from one to the other would not be semantic-preserving in the operational sense.

The **axiomatic** method views the definition of programming languages[5] from another perspective: a language's semantics as a *theory* of the programs written in that language [140]. It does not try to ascertain what a program means, but only what may be proved about it. This is achieved by associating the semantics of programming language constructs (and, hence, programs) with logical assertions of two kinds. The first assertion is assumed true prior to execution of a programming language construct. From it, and from the nature of the language construct, a second assertion that is true after the execution of the construct is derived. The pair of assertions thus characterises legitimate input and output states of the construct and, thus, it is possible to define implicitly the semantics of a programming language by a collection of *axioms* (derived from the assertions) and rules of inference (which are usually taken from mathematical logic) [35]. These axioms and rules of inference permit the proof of properties of programs, in particular that a given program is correct and that it realises a given input/output relation. To do this a notation is introduced as follows:

$$\{P\}\ S\ \{Q\}$$

where P and Q are logical propositions relating to the variables of the program and S is a statement. This has the meaning: if P is true before execution of S and

[5] The most notable example of this kind of language definition is Hoare and Wirth's definition of PASCAL [99].

S terminates, then Q will be true afterwards. Hence, P is called the *precondition* and Q is called the *postcondition.*

To prove the correctness of a program given an initial condition and a final condition, for example,

$$\{P\}\ S_1;\ S_2;\ \ldots ;S_n\ \{Q\}$$

it is necessary to introduce suitable propositional formulae between all the statements. This is done by finding a condition P_n which, if it is true before the execution of S_n, will yield Q. From this it is possible to find P_{n-1} in a similar way, and so on, until the program

$$\{P\}\ \{P_1\}\ S_1\ \{P_2\}\ S_2 \ldots \{P_n\}\ S_n\ \{Q\}$$

is obtained. This is known as a **proof tableau** [14]. All that then remains to be done is to prove

$$\{P\}\ \Rightarrow\ \{P_1\}.$$

Using this notation, axioms can be introduced by using axiom schemas [35] [133] and these axioms allow facts about program statements, and indeed whole programs, to be proved. However, the axiom schemas soon become very complex. For example, a section of code with a single label, L, in it preceded by some statements, S_1, followed by some more statements, S_2 and with a jump (Goto) to the label would have the axiom

$$\frac{\{Q\}\ \text{Goto}\ L\ \{\text{false}\}\ \vdash\ \{P\}\ S_1\ \{Q\}\ ,\ \{Q\}\ \text{Goto}\ L\ \{\text{false}\}\ \vdash\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ L;\ S_2\ \{R\}}$$

Already, the formulae are getting rather complex, and the formula for something such as

$$\text{If } B \text{ Then Goto } L_1 \text{ Else Goto } L_2 \text{ Fi}$$

although it could be written and proved, would be too hard to deal with in practice.

Program derivations (and transformations) within the axiomatic theoretical framework give rise to a very large number of proof obligations, since at each transformation stage the correctness of the new implementation must be proved against the previous implementation by proving all the required properties [179]. Large programs may require over one hundred proofs [179] and in general few, if any, of these proofs are likely to be rigorously carried out. So, what this amounts to is a formal method of program specification and an *informal* development method.

Another disadvantage of axiomatic semantics is the fact that while assertions about programs can be derived, it is not always clear that these are the most *useful* such assertions. An example of this is the requirement that loop invariants be determined in order to prove facts about loops; it is difficult to ascertain whether these invariants are the most efficacious. In general, the axiomatic approach can be used to reason about certain aspects of programs but not to express their *meaning*.

Denotational semantics is concerned with the *effect* of executing a program, where the effect is an association between initial and final states [144]. Thus, the aim in defining the denotational semantics of a given language is to associate a suitable mathematical object (number, set, function etc.) with each construct of the language. The semantics of the constructs are defined by so-called *semantic valuation functions* which map the constructs to suitable objects, or *denotations*, that they denote. Although *any* object can be associated with each programming language construct the language, it is most convenient to choose representations that use objects of standard mathematical domains since these can be reasoned about formally. Careful choice of representation can greatly simplify the specification of operations [121].

The semantics of the statements in a simple language containing assignments, Skip statements, If statements and While loops could be denoted by the semantic valuation functions shown in Figure 3.1 [144]. Here $\mathcal{S}_{ds}$, $\mathcal{A}$ and $\mathcal{B}$ are the semantic valuation functions of statements, expressions and Booleans, respectively. $s[x \mapsto \mathcal{A}[\![a]\!]s]$ is the state s with the value for x replaced by the semantic valuation of a with the values of variables in a taken from s. cond is an auxiliary function which

$$
\begin{aligned}
\mathcal{S}_{ds}[\![\mathsf{x := a}]\!]s &= s[x \mapsto \mathcal{A}[\![a]\!]s] \\
\mathcal{S}_{ds}[\![\mathsf{Skip}]\!] &= \mathrm{id} \\
\mathcal{S}_{ds}[\![S_1;\ S_2]\!] &= \mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!] \\
\mathcal{S}_{ds}[\![\mathsf{If}\ B\ \mathsf{Then}\ S_1\ \mathsf{Else}\ S_2\ \mathsf{Fi}]\!] &= \mathrm{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{ds}[\![S_1]\!],\ \mathcal{S}_{ds}[\![S_2]\!]) \\
\mathcal{S}_{ds}[\![\mathsf{While}\ B\ \mathsf{Do}\ S\ \mathsf{Od}]\!] &= \mathrm{FIX}\ F \\
&\ \text{where } F\ g = \mathrm{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}_{ds}[\![S]\!],\ \mathrm{id})
\end{aligned}
$$

Figure 3.1: The Denotational Semantics of a Simple Language

has the definition:

$$
\mathrm{cond}(p,\ g_1,\ g_2) = \begin{cases} g_1\ s & \text{if } p\ s = \text{true} \\ g_2\ s & \text{if } p\ s = \text{false} \end{cases}
$$

Defining the semantics of the While loop is a more major task since the loop could execute any number of times. The definition makes use of a *fixed point* of the functional F.

While the semantic definition in this example if quite straight-forward, increasing the complexity of the language causes a corresponding increase in the denotational semantics. For example, the introduction of local variables necessitates the use of an "environment" in place of a state, and the introduction of Goto statements requires the use of "continuations" [144]. In each case, these complications must be added to every language construct, making the definition of the simplest as complicated as the most complex [177].

The denotational approach to semantic definition makes it possible to talk about programming language constructs and program equality in the sense that two constructs or programs are equal if they both have the same denotation. With the notation of Figure 3.1, S_1 and S_2 are semantically equivalent if and only if [144]:

$$\mathcal{S}_{ds}[\![S_1]\!] = \mathcal{S}_{ds}[\![S_2]\!]$$

Proving that two programs are equal is equivalent to a mathematical proof that two given mathematical objects are the same. For this the full battery of proven mathematical techniques can be used; however, even then it is possible that the equivalence may be "undecidable" in that it cannot be proved to be either true or false.

These three approaches are similar in that they all map a program into a state space for that program. With operational semantics that state space is a more primitive abstract machine and the mapping is by way of implicit definition. This has few advantages and several disadvantages which relate to the fact that the semantics really only define a particular *execution* of the program.

With axiomatic semantics the state space is the set of formulae of mathematical logic and logical deduction is used to show program equivalence. While this approach has many advantages for program verification — notably that there is a consistent method of constructing proof obligations and then demonstrating their correctness — these proof obligations quickly become very complex and unwieldy, making the approach less straight-forward in practice.

With denotational semantics the state space consists of the mathematical objects associated with the programming language constructs, and the mapping is by way of semantic evaluation. This approach has not been widely used for program verification, since manipulating general mathematical objects is harder than manipulating formulae of logic. However, for defining a language suitable for program transformation, the ability to demonstrate program equivalence is the central concern, and denotational semantics provides this in a straightforward manner. Moreover, the use of denotational semantics would be greatly simplified if it could be defined without recourse to "tricks" such as the use of "continuations".

3.9 Summary

The transformation systems considered in this thesis (see Appendix A for specific details) are summarised in Table 3.1.

Name of System	Level of Applicability	Catalogue	Automatic or User-Driven	Language	Formal?
SETL	Spec. → Code	N/A	Automatic	Very-High-Level	No
RAPTS	Spec. → Code	Small	Automatic	Specification	Yes
TAMPR	Code → Code	Small	Automatic	LISP → FORTRAN + Intermediates	Yes
Restructurizer	Code → Code	Small	Automatic	COBOL + Intermediate	No
ZAP	Code → Code	Small	Mostly Automatic	LISP	Yes
SAFE	Informal Spec. → Formal Spec.	N/A	Automatic	Specification	No
TI	Spec. → Code	User-Extensible	User Driven	Wide Spectrum	No
GLITTER	Spec. → Code	User-Extensible	Semi-Automatic	Wide Spectrum	No
PSI	Dialogue → Code	Large	User Driven	Standard Languages	No
CHI	Dialogue → Code	Large	User Driven	Wide Spectrum	No
CIP	Spec. → Code	Large	User Driven	Wide Spectrum	Yes
DEDALUS	Spec. → Code	Large	User Driven	LISP	No
Hildum and Cohen'S Work	Code → Code	User-Constructed	N/A	User-Dependent	No
Kozaczynski's Work	Code → Spec.	User-Constructed	Automatic	COBOL	No
Ward's work	Spec. ↔ Code	Large	User Driven	Wide Spectrum	Yes

Figure 3.2: A Summary of Transformation Systems

3.10 In what ways have existing transformation systems been successful?

There are several distinct advantages to software development by formal[6] transformation [27]:

- The final program can be relied on to be correct (according to the initial specification) *by construction* (since each stage in the development follows from the previous one in a provably correct manner), provided that the transformation system being used has a sound theoretical foundation;
- Transformations can be described by *semantic rules* and can thus be 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 almost invariably 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 and the programmer to concentrate on the actual design decisions.); and
- The overall program structure is no longer fixed throughout the development, so the approach is quite flexible.

There are potential advantages, too, in the use of program transformations for software maintenance. These are mainly related to the fact that, just as it is possible formally to produce a program from a specification, so a specification produced by transformation from existing code would provably be correct. However, in general it remains an open question as to whether an existing program

[6] Transformation systems whose transformations are not proven to be semantic preserving may, by chance, produce correct programs. Goldberg [80] argues that a transformation need not be formally proven since extensive use of that transformation improves the confidence in its correctness. This, however, is still no substitute for a proof.

which either had no formal specification originally, or which has been heavily modified, has a concise specification. The advantages are summarised by Yang [193] as follows:

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

3.11 In what ways have existing transformation systems failed?

The disadvantages to using transformation systems are as follows:

- Users of transformation systems need a considerable amount of training to become proficient at using the system. A novice might be able immediately to apply transformations to the code, but it would not always be clear whether the transformations being applied were improving the code (according to some measure such as efficiency or comprehensibility);
- Although transformations can be used to restructure unstructured code with little difficulty, it is not clear how they could be used to cross levels of abstraction in the most "meaningful" way when used for maintenance. For

example, an array could have been used to implement any of a variety of abstract data types such as sets, trees or tuples;

- Most transformation systems rely on simplification "rules" such as how to rearrange a mathematical formula. The implementation of such rules is an area which needs addressing in more depth;
- Few systems have been shown to work with industrial-scale programs;
- For an effective tool, the correctness of the implementation of the transformations themselves needs to be verified. This is a practical, as opposed to a theoretical, limitation and in the long term it could even be an advantage since the transformations' correctness only needs to be checked once; and
- From a technical point of view, there is still no satisfactory way of obtaining non-local information about a program being transformed — i.e. information outside the transformation's pattern — such as the definition of a procedure. This reflects the limitations of a purely pattern-based transformation system.

The research described in this thesis will involve the construction of a transformation system which will act as a framework in which solutions to these problems can be investigated.

3.12 Why are transformation systems not more widely used?

Although program transformations are widely used in limited domains, such as in optimising compilers, there has been no wide-spread adoption of transformation systems for either software development or maintenance. In addition to the above failures, there are also additional factors which have prevented the widespread adoption of transformational programming.

3.12.1 Applicability

The first set of factors relate to the applicability of program transformations:

- Formal specification is not widely used so there is little call for a system which can produce code from specifications, or *vice versa*;
- Existing systems usually only work on toy problems and have not generally been shown to scale up;
- Existing systems are only of use in a limited domain or with particular programming languages which are not used in traditional applications; and
- Existing systems do not address maintenance.

3.12.2 Usability

The second set of factors relate to the usability of program transformation systems:

- There is usually a need to become familiar with mathematical methods in order to use the tools;
- It is possible that there are no formal proofs of the transformations, so little confidence may be placed in them;
- There are few simple user interfaces; and
- Change is required in the software process and programmer mentality in order to make use of the systems.

These points are summarised by Ould: the challenge for tool developers is to find better ways of disguising the formality, so that the user need not have impractical amounts of formal methods skills [146].

3.13 What can be learned from existing transformation systems?

The transformation systems surveyed vary widely in their aims — at one end of the spectrum are systems such as SETL and TAMPR which use transformations to achieve some other goal, and at the other are systems like CIP whose purpose is to allow programs to be developed solely using transformations. In terms of success, the systems also vary greatly from DEDALUS which has only been used on extremely simple examples through to ZAP, for example, which has been used in the development of a small compiler. Nevertheless, all the examples tend to be either small and algorithmically based with little emphasis on data complexity, or large programs with a simple structure.

Most existing transformation systems are designed to transform from specifications to programs and do not attempt to address the issues of maintenance. Those that do either require that the program being maintained was developed with the same system, or else do little more than simple restructuring. Only ZAP, DEDALUS, Kozaczynski's work and Ward's work showing any promise here. The systems do show that there have been enough small-scale successes to make this a path which is worth following further.

3.14 Conclusions

From the survey of the existing transformation systems it seems as if the most valuable system would be one which works on a wide spectrum language, uses formally proven transformations based on denotational semantics, has a large but easily accessed catalogue of rules and which provides a high degree of automation, but which nevertheless works interactively to allow for intelligent guidance by the user. Although CIP comes closest to the above requirements, it is hampered by being based on algebraic semantics (which makes it necessary to construct *additional* mental models in order to understand the abstract types in the program), by having an applicative kernel (which makes it poorly suited to modeling real-world programs) and by the problem with non-local information (which necessitates storing the definitions of procedures at every point in the program). Provided that the transformations were expressed in a suitable form, Ward's approach could overcome all these drawbacks.

The rest of this thesis will, therefore, focus on the design of a transformation system to exploit Ward's work. In order to address the weaknesses of existing systems, the new system must:

- Be easy to use within some clearly defined method;
- Be able to perform large restructuring changes;
- Be applicable to all aspects of software maintenance and reverse engineering, including the task of crossing levels of abstraction;
- Be applicable to real programs;
- Incorporate a sufficiently powerful subsystem for performing mathematical manipulations;
- Be able to obtain non-local information about the program being transformed; and
- Be correctly implemented.

Chapter 4

The Area of Research

4.1 Introduction

As was put forward at the end of the previous chapter, Ward's methods of proving refinements and transformations of programs [177] is a comprehensive theory. However, it also has the potential advantage that it is applicable to all stages of the software life cycle including that area in which transformation systems have traditionally been weak: maintenance. Thus, a computer-based, semi-automated transformation system founded on Ward's theorems would not only be a powerful system, but would also be widely applicable.

The area of research that will be addressed in this thesis, therefore, is concerned with some of the specific problems which relate to the construction of such a transformation system. This system will be applicable primarily to software maintenance, in particular reverse engineering.

4.2 A More Detailed Look at Ward's Work

The Wide Spectrum Language (**WSL**) which is used in Ward's[1] transformation system was originally designed to simplify proofs of program equivalence. It is based on a core kernel language with denotational semantics and a model-based theory of semantic equivalence. Extensions to this language are defined in terms of the basic constructs. The kernel is not purely applicative, but includes the concept of a state, unlike CIP [27], so that imperative programs can be operated on, by means of Back's [12] atomic description construct $(x/y \cdot Q)$ which will be explained in the next section. In order for this approach to be applicable to maintenance, WSL must be able to represent existing programs which are generally[2] written in imperative languages such as COBOL, FORTRAN and C. Thus, the use of an imperative, as opposed to a functional, kernel means that WSL and its associated transformations are potentially suitable for this purpose. In addition, specifications, expressed in terms of first order infinitary[3] logic, may be included in WSL, making it genuinely "wide spectrum".

4.2.1 The Kernel Language

The kernel language has two primitive statements: the atomic specification and the guard statement. The atomic specification is based on Back's atomic description [12]; it is written $x/y \cdot Q$, where Q is a formula of first order infinitary logic (with equality) and x and y are sets of variables. Its effect is to add the variables in x to the state space, assign new values to them such that Q is satisfied, remove the variables in y from the state[4] and terminate. For example, the statement $\langle x, y\rangle/\langle\rangle \cdot x + y = 10$ sets the variables x and y to arbitrary values such that their sum is ten.

[1] Much of this section is taken from Ward's thesis [177] and [180].

[2] Pure LISP is the major exception, but most LISP programs use impure extensions.

[3] Infinitary logic allows formulae which consist of the conjunction or disjunction of (countably) infinitely many terms. The reason for using infinitary logic will be explained later.

[4] In his new language [13], Back does not include the concept of removing variables from the state space.

The guard statement is written $[P]$, where P is a formula of first order infinitary logic. The statement $[P]$ always terminates and it forces P to be true at this point in the program *without changing the values of any variables.* In effect it restricts previous nondeterminism to those cases which leave P true at this point. Guard statements provide a useful means for defining extensions to the kernel and a useful theoretical tool for reasoning about programs, but they cannot be directly implemented[5]. (More examples of the use of the kernel language are given in the section on defining WSL by means of transformational extensions.)

There are three ways of combining statements in the kernel language:

1. **Sequential Composition**: $(S_1;\ S_2)$

   First S_1 is executed followed by S_2.

2. **Choice**: $(S_1 \sqcap S_2)$

   One of the statements S_1 or S_2 is chosen, nondeterministically, for execution.

3. **Recursive Procedure** $(\mu X \cdot S)$

   Within the body S, occurrences of the statement variable X represent recursive calls to the procedure.

Although the kernel language is elegant and tractable, it is too primitive to form a

[5] Later work by Ward [180] [182] uses a modified kernel language with four primitive statements. Let P be any formulae and x and y be any non-empty sequences of variables. The four primitive statements are:

1. $\{P\}$ is an assertion statement which acts as a partial Skip. If the formula P is true, then then statement terminates immediately without changing any variables;
2. $[P]$ is a guard statement, as before;
3. *add*(x) add the variables x to the state space (if they are not already present) and assigns arbitrary values to them;
4. *remove*(y) removes the variables y from the state space (if they are present).

There is an elegant duality between the assertion and guard, and between the *add* and *remove* statements.

This formulation of the kernel language is equivalent to Ward's original formulation. In particular, Back's atomic description is equivalent to the sequence:

$$\exists x \cdot Q;\ add(x);\ [Q];\ remove(y)$$

useful language for constructing "real" programs and needs to be extended. This is achieved by defining new constructs in terms of the existing ones by means of **definitional transformations**.

4.2.2 Extending WSL by means of Definitional Transformations

WSL is built up from the kernel in stages, or levels, so as to provide similar constructs to conventional languages (i.e. conditions, loops, local variable structures, expressions and conditions with side effects, and so on). Each level is defined in terms of the previous level. In this way, each new level inherits all the transformations of the previous levels and transformations are proved by appealing to the definitional transformation of the construct and carrying out the actual manipulations in the previous level of language.

Before describing the language extensions, it is worth pointing out that all levels of WSL can use expressions and logical formulae. These include variable names (which are composed of alphanumeric characters together with a few extra symbols such as "_"), numbers (which are treated in the mathematical sense, i.e. they have arbitrary size and precision), strings (which are arbitrarily long sequences of ASCII characters), sequences (which may contain elements of any type), Boolean constants and the full complement of mathematical operations for manipulating them.

The first-level language extension definitions are given in the following sections. (The symbol "$=_{\text{DF}}$" is read as "is defined as".)

Sequential Composition

The sequencing operator is associative, so the brackets can be removed:

$$S_1;\ S_2;\ S_3;\ \ldots;\ S_n\ =_{\text{DF}}\ (\ldots((S_1;\ S_2);\ S_3);\ \ldots;\ S_n)$$

Assertion

An assertion statement is a partial Skip statement; i.e. it takes a condition and does nothing if the the condition is true, and aborts if it is false. Thus, an Assert statement gives information about the context in which it occurs, making it easier to transformation that part of the program. An Assert statement is defined as follows:

$$\{B\} =_{\mathrm{DF}} \langle\rangle/\langle\rangle \cdot B$$

Deterministic Choice

Guards can be used to turn nondeterministic choice into deterministic choice[6]:

$$\mathsf{If}\ B\ \mathsf{Then}\ S_1\ \mathsf{Else}\ S_2\ \mathsf{Fi}\ =_{\mathrm{DF}} (([B];\ S_1) \sqcap ([\neg B];\ S_2))$$

Assignment

A general assignment can be expressed as follows:

$$x := x' \cdot Q \ =_{\mathrm{DF}} (x'/\langle\rangle \cdot Q);\ (x/x' \cdot (x = x'))$$

Here x is a sequence of variables and x' is a sequence of new variables. The formula Q expresses the relation between the initial values of x and the final values. For example, $\langle n \rangle := \langle n' \rangle \cdot (n' = n + 1)$ increments the value of the variable n and is defined as:

$$(\langle n' \rangle/\langle\rangle \cdot (n' = n' + 1));\ (\langle n \rangle/\langle n' \rangle \cdot (n = n'))$$

[6] In the LISP-like form of WSL, which will be described later in this thesis, deterministic choice statements have the type "Cond".

Simple Assignment

If e is a list of expressions, x is a list of variables and x' a list of new variable, then

$$x := e \ =_{\text{DF}} x := x' \cdot (x' = e)$$

With this notation, the statement to increment n can be written: n:=n+1.

Nondeterministic Choice

WSL includes a version of Dijkstra's guarded command [64]:

$$\text{If } B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n \text{ Fi}$$
$$=_{\text{DF}}$$
$$(((\ldots(([B_1];\ S_1) \sqcap ([B_2];\ S_2)) \sqcap \ldots) \sqcap ([B_n];\ S_n)) \sqcap$$
$$([\neg(B_1 \vee B_2 \vee \ldots \vee B_n)];\ \text{Abort}))$$

Deterministic Iteration

A While loop is defined using a new recursive procedure X which does not occur free in S:

$$\text{While } B \text{ Do } S_1 \text{ Od } =_{\text{DF}} (\mu X \cdot (([B];\ S;\ X) \sqcap [\neg B]))$$

Nondeterministic Iteration

Nondeterministic iteration is similar to nondeterministic choice:

$$\text{Do } B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n \text{ Od}$$
$$=_{\text{DF}}$$
$$\text{While } (B_1 \vee B_2 \vee \ldots \vee B_n) \text{ Do}$$
$$\text{If } B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n \text{ Fi Od}$$

which can be expanded using the rules above.

Initialised Local Variables

Initialised local variables are introduced with the WSL Var construct which is defined as follows:

$$\mathsf{Var}\ x := t\ \ :\ \ S\ \mathsf{End}$$

$$=_{\mathrm{DF}}$$

$$\langle x\rangle/\langle\rangle \cdot (x = t));\ S;\ (\langle\rangle/\langle x\rangle \cdot \mathsf{True})$$

Unbounded Loops with Exit Statements

One of the most powerful programming language statements in WSL is the Exit statement, which takes the form Exit(n) where n is an integer (*not* a variable), and which can only occur within loops of the form Do S Od where S is a statement.[7] These were described in [110]. They are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form Exit(n) which causes the termination of n enclosing loops. These statements are disallowed from terminating blocks and loops other than unbounded loops.

Previously the only formal treatments of Exit statements have dealt with them in the same way as unstructured Goto statements by adding "continuations" to the denotational semantics of all the other statements. This adds greatly to the complexity of the semantics and also means that all the results of program equivalence prior to this modification have to be re-proved with respect to the new semantics. The approach taken by Ward [177] is to express every program which uses Exit statements in terms of the kernel language. This means that the new statements do not change the denotational semantics of the kernel so all the transformations developed without reference to Exit statements still apply. The interpretation of these statements in terms of the kernel language is as follows:

[7] In the LISP-like form of WSL, which will be described later in this thesis, unbounded loops have the type "Loop".

An integer variable depth records the current depth of nesting of loops. At the beginning of the program there is the assignment depth:=0. Each statement Exit(k) is translated as depth:=depth−k since it changes the depth of the "current execution" by moving out of k enclosing loops. To prevent any more statements at the current depth being executed after an Exit statement has been executed all statements are surrounded by "guards" which are If statements which test depth and only allow the statement to be executed if depth has the correct value. Each unbounded loop Do S Od is translated as:

$$\mathsf{depth{:=}n;\ While\ depth{=}n\ Do}\ guard_n(S)\ \mathsf{Od}$$

where n is an integer constant representing the depth of the loop and $guard_n(S)$ is the statement S with each component statement guarded so that if the depth is changed by an Exit statement then no more statements in the loop are executed and the loop terminates. Formally, $guard_n(S)$ is defined by induction on the structure of S. For example:

$$\begin{aligned}
guard_n(S_1;\ S_2) &=_{\text{DF}} guard_n(S_1);\ guard_n(S_2)\\
guard_n(S_1 \sqcap S_2) &=_{\text{DF}} guard_n(S_1) \sqcap guard_n(S_2)\\
guard_n(\mathsf{Exit(k)}) &=_{\text{DF}} \mathsf{If\ depth{=}n\ Then\ depth{:=}depth{-}k\ Fi}
\end{aligned}$$

The rest of the definitions are given in [177]. The important property of a guarded statement is that it will only be executed if depth has the correct value. Thus $\{\mathsf{depth}{\neq}\mathsf{n}\};\ guard_n(S)$ is equivalent to a Skip statement.

Action Systems

An action system is a set of parameterless mutually recursive procedures [9] [8]. A program written using labels and jumps translates directly into an action system, with the statements following each label forming one action. If the end of the body of an action is reached, then control is passed back to the action which called it (or to the statement following the action system) rather than "falling through" to the next label. The exception is a special action referred to as the terminating action, Z, which when called results in the termination of the whole action system.

Action systems are defined in a similar manner to unbounded loops with Exit statements in that they are expressed using the kernel language.

An action is **regular** if every execution of the action leads to an action call, and an action system is regular if every action is regular. Any algorithm defined by a flowchart or program which contains labels and Gotos but no procedure calls in terminal positions, can be expressed as a regular action system.

Other WSL Constructs

WSL also has other constructs which are listed in the table in Appendix B. Among them are the following:

- Counted iteration;
- Procedure calls;
- Blocks with local procedures; and
- Expressions and conditions with side effects.

4.2.3 Proving Transformations

A program S is a piece of formal text, i.e. a sequence of formal symbols. There are two ways in which Ward gives meaning to these texts (see for example [177] and [184]):

1. Given a structure M for the logical language $\mathcal{L}$ from which the programs are constructed, and a final state space (from which a suitable initial space can be constructed), the program can be interpreted as a function f (i.e. a

state *transformation*) which maps each initial state s to the set of possible final states for s. Thus, a program can be interpreted as a function from structures to state transformations;

2. Given any formula of logic R (which represents a condition on the final state), it is possible to construct a formula of first order infinitary[8] logic WP(S,R). This is known as the *weakest precondition* of S on R and is the weakest condition on the initial state such that the program S is guaranteed to terminate in a state satisfying R, provided it started in a state satisfying WP(S,R).

From these two methods of the interpretation of programs, two corresponding methods of refinement arise: *semantic refinement* and *proof-theoretic refinement.*

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 non-termination or error conditions. A state transformation f maps each initial state s in one state space, to a set of possible final states $f(s)$ which may be a different state space.[9] 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. If $\perp \in g(s)$ for some s, then $f(s)$ can be anything at all. Thus, an "undefined" program can be refined to do anything at all. If f is a refinement of g (g is refined by f) then

[8] Using infinitary logic permits a simple definition of the weakest precondition of *any* statement, including an arbitrary loop, for any postcondition.

[9] If $f(s)$ is empty then the state transformation is *null* on s; the program still terminates even though the set of possible final states is empty. Such a program is known as a *miracle* since every final state satisfies any predicate, including *false*. It is the exact opposite of Abort in that it refines everything and is only refined by itself. Thus, *null* satisfies every specification.

this is written: $g \leq f$.

A *structure* for a logical language $\mathcal{L}$ consists of a set of values, plus a mapping between constant symbols, functions and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values. A *model* for a set of sentences (i.e. logical 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 S_1 under the structure M is refined by the interpretation of statement S_2 under the same structure, then this is written as $S_1 \leq_M S_2$. If this is true for every model of a countable set Δ of sentences of $\mathcal{L}$ then this is denoted by $\Delta \models S_1 \leq S_2$.

Proof-Theoretic Refinement

Given two statements S_1 and S_2 and a formula R, then it is possible to construct two formulae WP(S_1,R) and WP(S_2,R). If there exists a proof of the formula WP(S_1,R) $\Rightarrow$ WP(S_2,R) using the set Δ of assumptions, then this is written $\Delta \vdash$ WP(S_1,R) $\Rightarrow$ WP(S_2,R). For S_2 to be a refinement of S_1, this result has to hold for every formula R. This involves quantification over R (and is thus a statement of second order logic), but the quantification over R can be avoided by extending the language $\mathcal{L}$ by adding a new relation symbol $G(w)$, where w is a list of all free variables in S_1 and S_2, to give a new language $\mathcal{L}'$. If it can be proved that $\Delta \vdash$ WP(S_1,$G(w)$) $\Rightarrow$ WP(S_2,$G(w)$) then the proof makes no assumptions about $G(w)$ and, therefore, remains valid when $G(w)$ is replaced by any other formula. This is written: $\Delta \vdash S_1 \leq S_2$.

The weakest preconditions of the kernel language is given by the set of formulae in Figure 4.1. Note that the weakest precondition of the atomic specification statement makes no reference to the removed variables, y, since these can have any values whatsoever before they are removed.

$$\begin{aligned}
\mathrm{WP}(x/y \cdot Q, R) &= (\exists x \cdot Q \land \forall x \cdot (Q \Rightarrow R)) \\
\mathrm{WP}([P], R) &= P \Rightarrow R \\
\mathrm{WP}((S_1;\ S_2), R) &= \mathrm{WP}(S_1, \mathrm{WP}(S_2, R)) \\
\mathrm{WP}((S_1 \sqcap S_2), R) &= \mathrm{WP}(S_1, R) \land \mathrm{WP}(S_2, R) \\
\mathrm{WP}((\mu X \cdot S), R) &= \bigvee_{n<\omega} \mathrm{WP}((\mu X \cdot S)^n, R)
\end{aligned}$$

Figure 4.1: The Weakest Preconditions of WSL's Kernel Language

Equivalence of Refinement Methods

A fundamental result proved by Ward [176] [182] is that these two methods of defining refinement are equivalent[10], i.e. for any statements S_1 and S_2, and any countable set Δ of sentences of $\mathcal{L}$:

$$\Delta \models S_1 \leq S_2 \text{ if and only if } \Delta \vdash S_1 \leq S_2$$

Thus, it is possible to write, without loss of generality, $S_1 \leq S_2$ to mean that S_1 refines S_2.

The proof makes use of two formulations of Dijkstra's weakest precondition [64]: the first is a function which maps the semantics of a program and a condition on the final state space to a condition on the initial state space. (A condition on a state space is simply a set of states: those that satisfy the condition.) The second is a function which maps the syntax of a program and a formula of first order logic to another formula of first order logic. The two definitions are proved to be equivalent [176] [182] given a suitable interpretation of formulae as state conditions.

[10] First order infinitary logic is *complete*, so Ward [182] demonstrates that if there is a refinement then there is also guaranteed to be a proof of the corresponding formula (although the proof may be infinitely long). Conversely, by the *soundness* of first order infinitary logic, if there is a proof of the formula then there is guaranteed to be a proof of the refinement.

Refinement and Transformation

If S_1 refines S_2 and also S_2 refines S_1, then there is a program transformation from S_1 to S_2 and *vice versa*. This is written:

$$S_1 \approx S_2$$

An Example: Forward Expansion

The following demonstrates the proof-theoretic technique for deriving a transformation that is often used to replace two copies of a statement by a single copy.

$$\Delta \vdash \textsf{If } B \textsf{ Then } S_1 \textsf{ Else } S_2 \textsf{ Fi; } S \approx \textsf{If } B \textsf{ Then } S_1\textsf{; } S \textsf{ Else } S_2\textsf{; } S \textsf{ Fi}$$

Proof:	$\mathrm{WP}(\textsf{If } B \textsf{ Then } S_1 \textsf{ Else } S_2 \textsf{ Fi; } S, R)$	
$\Leftrightarrow$	$\mathrm{WP}(\textsf{If } B \textsf{ Then } S_1 \textsf{ Else } S_2 \textsf{ Fi}, \mathrm{WP}(S, R))$	(Defn of ;)
$\Leftrightarrow$	$(B \Rightarrow \mathrm{WP}(S_1, \mathrm{WP}(S, R))) \wedge (\neg B \Rightarrow \mathrm{WP}(S_2, \mathrm{WP}(S, R)))$	(Defn of If)
$\Leftrightarrow$	$(B \Rightarrow \mathrm{WP}(S_1;\ S, R)) \wedge (\neg B \Rightarrow \mathrm{WP}(S_2;\ S, R))$	(Defn of ;)
$\Leftrightarrow$	$\textsf{If } B \textsf{ Then } S_1\textsf{; } S \textsf{ Else } S_2\textsf{; } S \textsf{ Fi}$	(Defn of If)

4.2.4 Ward's Catalogue of Program Transformations

The flexibility of having two methods of proving program transformations has resulted in there being a great many proven transformations in Ward's thesis [177]. Thus, only a flavour of them will be given here. The classification of transformations given below is the same as Ward's classification [174].

Assertions

Among the simplest transformations are those for introducing, removing and manipulating assertions. It can be shown [174] that these transformations are all that is required to include all the results of Hoare's axiomatic basis for programming [96].

Simplifications

There are several important basic transformations that are used extensively both in proving more complex transformations and in putting a program into the correct form for applying more complex transformations. These include:

- Reordering conditionals;
- Removing conditionals which follow an assertion that indicates which branch will be taken;
- Merging and splitting assignments; and
- Inserting and eliminating assignments after assertions.

Manipulations

Unlike the simplifications in the last section, which do not significantly alter the structure the program, these manipulations provide the means by which code can be restructured. Such transformations include:

- Removing unused local variables;
- Expanding and factoring (i.e. moving statements into or out of) conditionals;
- Unrolling and rolling loops; and
- Merging loops.

Exit Statements

Ward's theory of Exit statements is very comprehensive and gives rise to a great many transformations, including the following:

- Replacing non-terminating loops by Abort statements;
- Removing statements which occur after Exit statements;
- Various kinds of loops unrolling, rolling and inversion;
- Replacing double by single loops (in the right circumstances); and
- Removing redundant loops.

Action Systems

The transformations on action systems are particularly important since they can be used to restructure unstructured code. They include:

- Replacing an action call by the body of the called action;
- Removing an action which is never called;
- Within a regular action system, removing statements which occur after Call statements; and
- Replacing an action body (with recursive calls) by a double loop, or in certain circumstances, by a single loop.

Recursion Removal

Ward's thesis [177] includes proofs of all the common techniques of linear recursion removal and also several methods of replacing non-linear recursion by iteration. These give rise to a number of general transformations and a larger number of transformations applicable in specific circumstances.

4.3 The Advantages of A Practical System Based on Ward's Approach

Ward's approach to program transformation has a number of advantages and *potential* advantages over other transformation systems. Among the former are the benefits gained by using Ward's formal theory:

- The theory is well founded on a sound mathematical basis of set theory and first order infinitary logic and and every transformation has been rigorously proved in Ward's thesis [177];
- The use of infinitary logic eliminates the need to determine loop invariants or fixed points of functionals when transforming loops;
- Since the kernel language is imperative, as opposed to functional, it is suitable for working with existing programs, as required by reverse engineering;
- WSL is defined in terms of definitional transformations of the kernel language, so "continuations" do not need to be introduced in order to handle control transfers. Similarly, "environments" do not need to be added to handle local variables;
- Specifications can be included within programs and non-determinism allows flexibility and generality at higher levels of abstraction; and
- There are a large number of theorems which cover all aspects of programming in WSL, including badly written programs with jumps (such as Exit statements and action systems) and side effects (but not exceptions, higher-order functions and concurrency).

Among the potential advantages of the approach are the following:

- The large number of theorems in Ward's thesis means that it might be possible to construct a transformation system that is applicable to a wide variety of programs;

- All Ward's transformations appear to be automatically checkable, which would remove any need for the user to justify, from a mathematical perspective, the transformations he applied;
- It would seem as if the techniques can be used either to increase the efficiency of a program, or its clarity, or both;
- Since the theorems have been shown to work in difficult examples [176] [175] [179], a tool based on this approach should also be suitable for transforming such programs; and
- The approach appears to be applicable to maintenance as well as development.

4.4 A Method for Reverse Engineering using Transformation

In order for any transformation tool to be useful, it must be used within the framework of a method. The proposed method for using the tool described in this thesis has four stages:

1. Automatic translation of the source code into WSL;
2. Automatic removal of idiosyncrasies introduced by translation;
3. Manual selection, combined with automatic application, of transformations to produce a structured form; and
4. Abstraction to a specification.

4.4.1 Translation

Since the tool described in this thesis will be designed to transform only WSL, there has to be a method of translating other languages into WSL. This can be accomplished using translators based on existing (compiler) technology. These would not necessarily have to produce optimal translations since, once the code was been translated into WSL, it would then be possible to use the tool to remove any anomalies that translation might have introduced and to do some simple tidying of the code.

Details of the approach to translation which is used in practice is given in Chapter 9.

4.4.2 Automatic Removal of Idiosyncrasies

It is anticipated that translation into WSL will introduce additional code caused by the fact that WSL has to model a source language which is not, in general, equivalent; for example, where assembler uses "branch to register" instructions, WSL might use procedures. It is proposed that these idiosyncrasies be removed by an automatic transformation process that occurs after translation, as opposed to introducing complexity into the translator.

4.4.3 Manual Transformation

This thesis is based on the premise that Ward's transformations are sufficient for performing code restructuring. Thus, this stage of the method would require the user to select which transformations for the system to apply to which sections of code, and to repeat this process until the code is in a form that is considered in some sense the "best" that can be attained.

4.4.4 The Reason for Functional Abstraction

The ultimate objective of the transformation tool is to facilitate the transformation of existing, large-scale source programs to high-level requirement specifications. It is envisaged that such specifications will, in general, be represented in non-executable form, using a language such as Z [170] or VDM [103]. The major attractions of this approach are that the specification will be semantically equivalent to the original code or the latter will be a refinement of the specification. Thus the user can be confident that the specification can provide a representation which can be maintained in place of the original source code. Maintaining a high-level, more abstract representation has a number of important advantages [80]:

- Designs may appear implicitly in the implementation and not be properly documented. Hence a modification may inadvertently violate the implicit constraints on the code which are implied by design;
- The process of implementation is one of *information spreading*, that is, assertions describing the problem which are simply expressed in the specification may be reflected diffusely throughout the implementation; and
- The implementation is cluttered with information about the efficiency and performance of the target architecture.

In addition, a specification is more compact than the source code, it is expressed in a more problem-oriented notation and executable code can potentially be generated from it automatically or semi-automatically.

4.4.5 Problems in Functional Abstraction

In performing source-to-source transformations, the user can apply the transformations in the knowledge that the semantics of the program stays the same (by the definition of a program transformation). However, in performing abstraction the aim is to *remove* information from the program in order to make it more **abstract**. Thus, this cannot be accomplished with transformations *per se*, since these preserve the semantics, but must be done with **abstractions**.

> **Abstract specifications** say *what* a program does without necessarily saying *how* it does it [181].

Abstraction is a process of generalisation, removing restrictions, eliminating detail and removing inessential information (such as algorithmic details) [181]. Thus the abstractions cannot be applied without a clear idea of which information contained in the program refers simply to the implementation, and not to the the function of the program. In the general case[11], this information cannot be determined automatically within the system, so user guidance is needed at this stage. As a trivial example, the program ⟨X:=2, Y:=3⟩ may be an implementation of any of the following specifications:

- "Assign X the value 2, and Y the value 3";
- "Assign X and Y values such that $X + Y = 5$"; or
- "Assign X and Y values such that $X < Y$".

Indeed, this statement is actually an implementation of infinitely many specifications of the form: "Assign X and Y values such that $X + Y \neq K$" where K is any constant not equal to 5.

Abstraction can be performed by applying refinements [138] [137] "in reverse". However, in order to perform abstraction, it may be necessary first to perform particular restructuring operations on the program. These are detailed in the next section.

4.4.6 Steps in Functional Abstraction

There are a number of steps that have been identified as being of key importance in the process of crossing levels of abstraction in order to acquire a specification.

[11] *Some* information, such as the usage of local variables, is clearly part of the implementation since it is not "visible" outside the program.

Procedurisation and Parameterisation

Two factors that make a program difficult to understand are a lack of localisation and a lack of information hiding.

> **Localisation** is the process of collecting logically related computational resources into one physical module [159].

> **Information hiding** suppresses the implementation of an object or operation, thereby focusing attention on its definition and interface [159].

Among the characteristics that make a program difficult to understand are the following [56]:

- No, or very little, use of design abstraction;
- Local functions and variables may be represented globally, and globally defined functions and variables may be used only locally; and
- A lack of information hiding increases the difficulty of program understanding by expressing irrelevant information.

While the original program may have been composed of procedures, these may no longer reflect the functional division of the code because of substantial changes made to it and may, in fact, be little more than historical boundaries. So the first step in abstraction is to expand these procedures in line and to create a new procedural decomposition.

In creating the new procedurisation it is desirable to divide the program into procedures such that each procedure fulfills Bergland's criteria (see Section 1.3.2) [32] that the procedures implement a single independent function, perform a single logical task and have a single entry and exit point. There is no purely *automatic* way that the system is able to identify the *best* procedural decomposition, but the system could provide help with this process.

There should be transformations which replace a section of code by a procedure and which will then search for all identical occurrences of the body of that procedure and replace them by a procedure call. Also, the system should identify which variables are used and assigned in any section of code. This information can be used to identify potential procedures, since a procedure should have as few inputs and outputs as possible. Thus, if a number of variables are only used in a particular section of code, they are probably local to some logical (but possibly not yet created) procedure.

Having created a procedure from a section of code, the next stage in the abstraction process is to parameterise it. In the previous stage, the variables of the procedure were identified, and it is often valid to assume that any variables which are not local to the procedure can be made into parameters.

The system should be able to provide transformations which assist the user in this process. First, there should be transformations which replace a variable in a procedure by a parameter and modify the calls to that procedure to incorporate the extra value that must be passed, and transformations which reverse this operation. Second, there should be transformations which search for all other occurrences of the body of that procedure and replace them by procedure calls and these can be applied again, since a parameterised procedure is more likely to match other sections of code.

As with procedurisation, the system could not select the parameterisation *automatically* since the heuristics to determine which variables to replace with parameters are not completely determined. However, the system could attempt to parameterise the procedure in different ways and to choose the parameterisation which caused the greatest number of matches with other sections of code.

Since Ward's thesis includes all the theorems which prove the feasibility of these operations, then provided the system being built is capable of including any possible transformation, it is reasonable to conclude that they would be possible in practice.

Recursion Introduction

Ward [180] claims that the derivation of algorithms from specifications by formal refinement can be broken down into the following stages:

1. Nonexecutable specification;
2. Recursively defined specification;
3. Recursive procedure; and
4. Iterative algorithm.

In abstraction, which is the opposite of refinement, it seems reasonable to assume that these steps might be taken in reverse. Thus, one of the most important steps in many instances of crossing levels of abstraction is the introduction of recursion. This can be done using the inverse of the transformations for recursion removal, including Ward's general recursion removal theorem [180].

Having introduced recursion, it might then be possible to identify invariants over the body of the procedure, which it would not have been possible to identify in the iterative version.

Using the general recursion removal theorem necessitates the code being rewritten as an action system of a particular form (although there is a choice of the precise form). This cannot, in general, be undertaken automatically since the user needs to identify which variable is being used as the control stack (so that the tests of variable can be put into a separate action). It is not possible to identify this variable automatically; however, having put the code into the correct form of action system, the system could apply the theorem without further user intervention.

Invariants

As was seen in Chapter 2, invariants form an important part of proving program correctness, and in abstraction, they perform a similar rôle: the identification of the function of a section of code (typically a loop).

The problem of finding the inductive assertion for a given program is theoretically unsolvable [172], so it is the responsibility of the user to introduces a true assertion before the beginning of the loop, say. The system would then employ its symbolic mathematical and logic routines to determine whether or not this assertion is invariant over the loop, i.e. that it is true at the beginning of the loop body, at the end of the loop body and, therefore, after the end of the loop. (The system is not able to *calculate* or *deduce* these invariants, although it is able to calculate terminating conditions based on the loop condition or exit test.)

There are two ways in which the user can introduce the initial assertion. The first is to use the transformations which add or insert Assert statements after existing statements, such as adding the assertion $x = y$ after assigning y to x. The second is to enter an assertion and to have the system prove that this is true either by determining if it is one of the assertions that could have been introduced by the first method, or else by replacing the variables of the assertion by their values and simplifying the resulting expression to **true**. Some assertions, such as $x < (x + 1)$, trivially make loop invariants, so the user must choose the condition carefully.

Specification Statements

A key part of crossing levels of abstraction is the introduction of specification statements which indicate what the program does, without saying how it does it. Thus WSL includes a specification statement which indicates which variables are changed and the result of changing those variables (as a condition relating the old and new values), without saying anything about how the values of the variables might be determined. The statement is of the form Assign X Such That C where X is a set of variables and C is a condition.

Any references in the condition to variables which are being changed refer to the *new* values of the variables, unless there is a specific indicator otherwise (as shown in the next example). The specification statement can be mixed freely with other statements because of the wide spectrum nature of WSL. Many of the simpler statements may be interchanged directly with these specification statements — operations which are actually transformations. For example, the parallel assignment statement ⟨X:=2, Y:=(X+1)⟩ could be replaced by the specification statement Assign (X Y) Such That (X=2) ∧ (Y=Old(X)+1). These specification statements can be combined in various ways, often by introducing an existential quantifier (as in the example in Chapter 9).

By introducing and combining specification statements in this way, it is possible to build up a specification of a simple section of code, which is actually equivalent in function to that code. For more complex program constructs (notably loops), additional techniques are required.

Specifications from Assertions

The first step of abstracting a loop to a specification would be to add assertions which are loop invariants. The second step would replace the loop and its associated assertions by a specification statement. In this stage abstraction is necessary since the change made to the program may change (by weakening) its semantics. The process could be as follows:

For any section of code which finishes with an assertion, if the variables assigned to in the section of code are a subset of the variables referred to in the assertion, then the code can be removed and a specification statement added whose condition is the condition of the assertion and whose list of assigned variables contains precisely those variables which are assigned to in the code.

When performing this kind of change to the program, since the result is not necessary equivalent to the starting code, the system would prompt the user for some textual justification for this abstraction. This information could be stored as part of the history of the transformation process and in this way a trail of all the abstractions that have been used would be kept and could be audited for "correctness".

Condition Weakening

Another method of abstracting a program would be to replace the condition of a specification statement by a weaker condition. For example, the specification statement Assign (X Y) Such That (X=2) $\wedge$ (Y=7) can be weakened to the the specification Assign (X Y) Such That (X+Y)=9.

As with replacing some code and an assertion by a specification statement, this change does not precisely preserve the semantics of the program so the system would again prompt the user for some textual justification for this abstraction.

The reason for performing such an abstraction would be if the program were a *specific* implementation of some *general* specification, and the general form was the one required.

Data Abstraction

The data types that are used within a program are *concrete* representations of some *abstract* data type [97]. Every value of the abstract type must be representable by one or more[12] values in the concrete type. Thus, the abstract and concrete data types are related by means of a *refine* function and a *retrieve* function [188].

Data abstraction involves recovering the abstract data representation from the concrete data representation. This can be done by means of introducing **ghost variables** [137] [178] in such a way that each operation on the concrete data is paralleled by an operation on the ghost variables. The system would then find (or more realistically the user would provide) a function that maps from the concrete data into the ghost variables, and assertions would be added to the code

[12] There may be more than one. For example, if an abstract set is represented by a concrete list, then there are many orderings of elements of the list for each set.

to indicate this invariant link specified by this function. The result of this would be that the concrete variables could be removed and the ghost variables would take their place as an abstraction.

The work in this area is still in its early stages, but this method seems promising.

Another approach currently being considered by Yang [194] is to represent directly the data definitions of the source language using additional WSL constructs. From these the aim is to construct Entity-Relationship-Attribute (ERA) diagrams. A similar method was adopted by Sneed and Jandrasics [167].

4.4.7 Summary of the Method

The method for using the tool would involve translating the source code into WSL and applying Ward's transformations to it until it is in the desired form. Functional abstraction could then be carried out using (not necessarily all) the following stages:

- Procedurisation and parameterisation;
- Recursion introduction;
- Invariant introduction;
- Introduction of simple specification statements;
- Creation of specifications from assertions;
- Condition weakening; and
- Data abstraction.

Only the last three actually involve removing information from the program (i.e. abstraction); the others may *introduce* information from the application domain in the form of, say, procedure names.

For the the system to provide the ability to perform functional abstraction a few extensions would be required. Most of these would fall into the category of adding new "transformations" (which are actually *not* transformations in the strict sense); however, certain extensions to the symbolic mathematics functions would also be required.

Data abstraction has yet to be addressed in detail.

4.5 An Outline of the Programme of Work and Problem Definition

At this point the problem that this thesis is tackling will be summarised by outlining the programme of work. The work for this thesis can be divided into eight sections which, although they can be regarded as distinct, will necessarily overlap in certain areas.

4.5.1 Review the field

The first part of the work — that of reviewing software engineering in general, and transformation systems in particular — has already been covered. It looked at the problem of producing correct programs from specifications, at the problem of deriving correct specifications from existing code, and at possible solutions to these problems. Transformation systems offer several important potential advantages over other informal and formal methods. The success of existing transformation systems was assessed. This led to the conclusion that while none of those in existence is suitable for both development and maintenance, a system based on Ward's approach might provide the required functionality.

4.5.2 Examine the Shortcomings

The transformation system that forms the subject of this thesis is used as a vehicle for investigating how to overcome the shortcomings of existing transformation systems — applicability and usability — that have been identified in the review.

The system should be applicable to real programs and not just toy examples, it should work on a wide range of programming languages using WSL as an internal representation of the code) and it should be of use in cases in which formal specification has not been used. Also, the system should be usable by means of a simple user interface and a method that does not require the user to become familiar with mathematical methods. Finally, it should be possible to incorporate the use of the system into existing software processes.

These questions can best be answered by actually building a transformation system. To do this, a number of design decisions must be made.

4.5.3 Develop the Basis for Building a Transformation System

The next part of the research will look at the design issues, foremost among which are the following:

- How should programs undergoing transformations be represented?
- How should the transformations, and their point of application, be selected?
- How should the applicability of the transformations be tested?
- How should the transformations be represented?
- How should the transformations be stored in the system?
- How can transformations be combined so as to provide transformations which have greater effect?

- What other facilities should be included in a usable transformation system?

Design solutions are presented, and a prototype system and transformation catalogue are built, in order to put these ideas into practice.

4.5.4 Develop *MetA*WSL

Transformations will be represented in the system by means of a specially designed language — *MetA*WSL. A description of this language will form the core of the thesis. It must be a language within which transformations can clearly and concisely be expressed. In particular, it must be assessed in the light of the criteria for success given below.

4.5.5 Code Ward's Transformations

In order for *MetA*WSL to be an effective language for expressing program transformations it must certainly be capable of expressing Ward's transformations. Thus, this stage of the work will involve coding Ward's transformations using *MetA*WSL. Not only will this help to assess the capabilities of *MetA*WSL, but it will also provide a useful working system.

4.5.6 Create Additional Transformations

Since many of Ward's transformations are elementary in their nature — making only small changes to programs — it will probably be necessary to combine these in various different ways; for example, to create transformation strategies. Such transformations will combine the effects of several smaller transformations, choosing the correct ones as appropriate, in order to perform some large-scale change. *MetA*WSL's suitability in this area will be assessed by writing some transformations of this kind.

4.5.7 Create some Transformations For Abstraction And Design Recovery

In order to perform effective reverse engineering by means of program transformation, it will be necessary to have some transformations for functional abstraction and design recovery. In order to assess $\mathcal{M}\mathcal{ETA}$WSL's strengths and weaknesses in this area, a number of such transformations will be created. This area will also consider how further transformations for functional abstraction could be written.

4.5.8 Assessment of Success with Real Programs

The final stage of this work will involve using the resulting system on some real code in order to assess its usefulness. Since this work is partially funded by IBM Hursley, the project has access to a quantity of commercial IBM 370 Assembler code, which will be used as a testbed for the system.

4.6 Criteria for Success

The success of the work will be judged according to how the following questions, which have been divided into three groups, are answered.

4.6.1 Preliminary Questions — Maintenance by Transformation

- Is software maintenance made simpler by using transformation-based reverse engineering?
- Is WSL a good language for this purpose; i.e. can existing programs be expressed in WSL and is there a suitable range of WSL transformations?

- Crossing levels of abstraction, for reverse engineering, involves removing details of the program's implementation while retaining details of its function. How can one do this in a transformation-based system?

4.6.2 Central Questions — The Assessment of *META*WSL

- What constructs should *META*WSL include so as to be flexible enough to express program transformations without becoming overburdened with little-used constructs? i.e. what constructs should *META*WSL include so as to be simple yet complete?
- Can *META*WSL clearly and concisely represent Ward's transformations?
- What other transformations are required?
- Can *META*WSL be used to express clearly and concisely these transformations?

4.6.3 Questions on the Effectiveness of the Tool

- Does the approach result in a usable tool? In particular, what training is required?
- Is the implementation of the transformation catalogue efficient, reliable, correct and complete?
- Does the method scale up to larger programs?
- How well does the system work on real programs in an industrial environment?
- What weaknesses does the system have?
- How does the use of the tool fit into the software process?
- In what ways does this system add to the study of transformation systems in general?

- Can the system be used to maintain itself?

4.7 Summary and Conclusions

This chapter has laid the foundations for the rest of this thesis. It has introduced WSL, showing how it is defined in terms of a kernel language. It has outlined the proof principles used by Ward to prove his transformations, and has given examples of the transformations that have been proved. Finally, it has explained the advantages of a transformation system based on these principles and has described the nature of the rest of this work, giving criteria for its success. The rest of the thesis will answer the questions raised in this chapter and give details of both $\mathcal{META}$WSL and the underlying structures needed in creating a transformation system with the desired attributes.

Chapter 5

Fundamental Design Decisions

5.1 Introduction

This chapter first describes the overall tool of which the transformation system described in this thesis is a part. The chapter then discusses the underlying structures and foundations on which the transformation system is based. These cover design decisions such as how WSL programs should be represented, how the point of application of the transformations and the transformations to apply should be selected, how the applicability of transformations should be tested, and how transformations should be represented and stored. Finally, certain components which form the core of the transformation system, notably the pattern matcher, the database and query functions and the symbolic mathematics functions, will also be described.

5.2 The ReForm Project

This work in constructing a transformation system forms part of a larger project — the **ReForm** project [78] — sponsored by *IBM Hursley* and the *DTI*, and carried out by the *University of Durham*, *Durham Software Engineering Ltd* (formerly *Centre for Software Maintenance Ltd*) and *IBM Hursley*. The other members of the project team are Keith Bennett, Martin Ward, Hongji Yang, Nigel Scriven and Brendan Hodgson.

The aim of the ReForm project is to create a code analysis tool — the **Maintainer's Assistant** [53] [185] [46] [193] [47] — aimed at helping the maintenance programmer to understand and modify a given program. Program transformation techniques are employed by the Maintainer's Assistant both to derive a specification from a section of code, and to transform a section of code into a logically equivalent form. The aim is to provide a tool with features such that:

- It acts, initially, on existing program code as a tool to aid comprehension (possibly by producing specifications);
- Only the program code is required;
- The system can work with any language by first translating — with a standalone translator — into WSL;
- Changes are made to the WSL by means of transformation;
- Transformations are represented in an extension of WSL — *MetaWSL*;
- The system incorporates a large, flexible catalogue of transformations;
- The applicability of each transformation is tested before it can be applied;
- A history/future structure is built-in to provide back-tracking and forward-tracking allowing the programmer to change his mind;
- The system is interactive and incorporates an X-Windows front end and pretty-printer called the **Browser** [193];
- The system includes a database structure to store information about the program being transformed, such as the variables assigned to within a given piece of code [193];

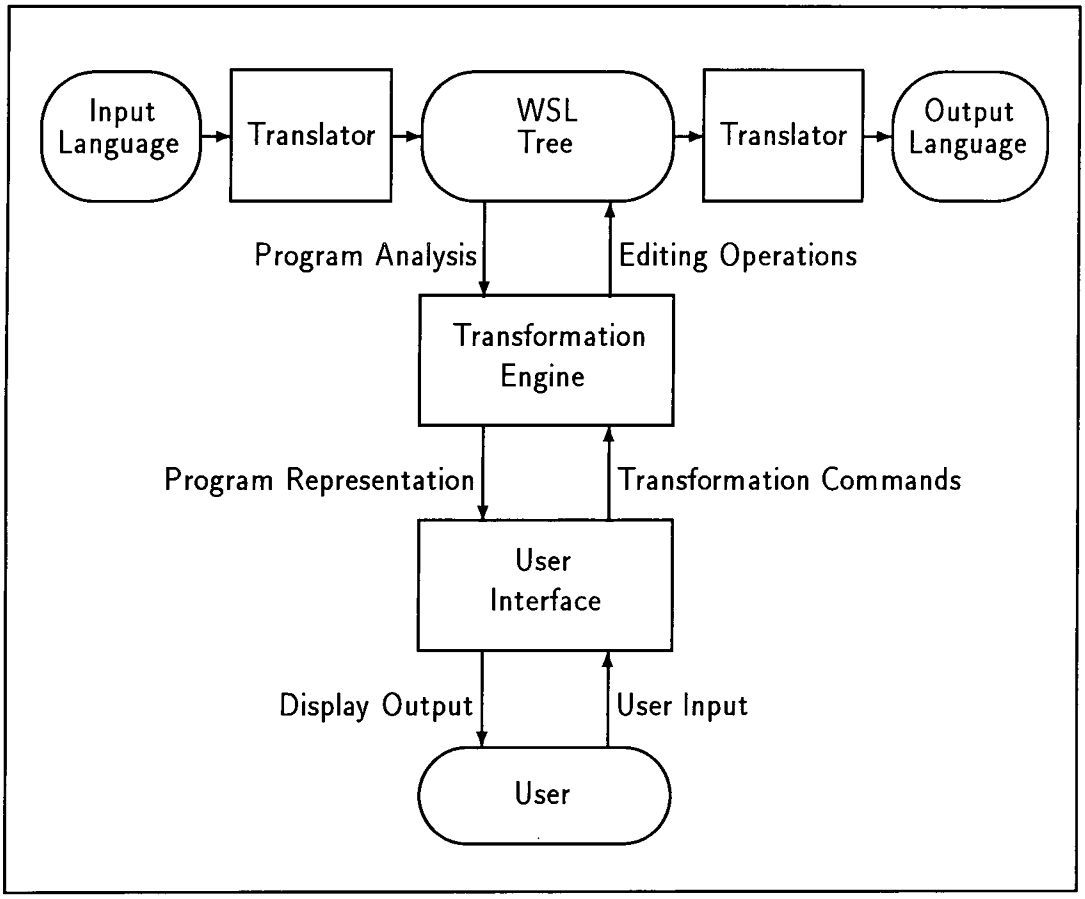


Figure 5.1: The Structure of the Maintainer's Assistant

- The system includes a simplifier for mathematical and logical expressions; and
- The system includes a facility to calculate metrics for the code being transformed.

While the core of the tool is the program transformation system, there are other parts to the tool. All these are shown in Figure 5.1 and will be described briefly in the following sections. (The work described in Sections 5.2.1 to 5.2.4 has been carried out by other members of the research group.)

5.2.1 Translating Other Languages to WSL

Part of the ReForm project has involved taking IBM 370 assembler and using the system on it. To do this it has been necessary first to translate the assembler into WSL, as described in Section 4.4.1.

5.2.2 Metrics and Supporting Tools

A number of tools have been incorporated into the system, in addition to the transformation system. Among these is the ability to obtain metrics relating to the code being manipulated.

During software development, software metrics are often used for assessing, measuring and predicting attributes of the code. Among the many software metrics available, complexity measures are the most well-accepted method of determining the intrinsic quality of software. It has been recognised [132] that like software development, software maintenance also benefits from measurement, even though the best metrics are the subject of debate. Thus, a generic system has been built that enables a variety of measures to be made. The objectives of using metrics in ReForm are to help the user to select transformations (to help develop heuristics), to measure the progress made in optimising the program code and to measure the resulting quality of the program being transformed.

Other tools which are in a less well-developed stage include a program slicer, a static analyser and a call-graph generator.

5.2.3 The User Interface

The Maintainer's Assistant is an interactive tool and as such requires an intuitive and fast user interface to allow the user to try out, quickly and simply, different options (such as different procedural decompositions) when working on a program. A suitable interface has been constructed using the *Motif* toolkit in a *Unix* and

X-Windows environment.

The interface allows the user to select a piece of code by pointing to it with the mouse and clicking a mouse button. Transformations can be selected from pull-down menus. Before a transformation appears on a menu, however, the transformation system needs to checks it applicability conditions so that the user may not attempt to perform an invalid transformation.

In addition, the interface includes facilities for editing the program (to correct faults), calculating metrics and saving and loading versions of the program to and from file.

5.2.4 Translating from WSL to Other Languages

Just as there is a translator for producing WSL from assembler, so it would be possible to create a translator which would take WSL as its input and produce equivalent code in some other language. Such a translator remains to be built. One method of simplifying the translation would be to have a set of transformations for producing WSL that is similar to the target language; for example, WSL which does not use recursive procedures.

5.2.5 Transformation Engine

The transformation engine which forms the core part of the Maintainer's Assistant is the subject of this thesis.

5.3 Storing WSL Programs

In many transformation systems, particularly the earliest ones, the program undergoing transformation is stored as a simple text file and the program transformer is essentially a text editor that performs a series of commands (in this case specified by the program transformation) and produces an altered version of the original text (a sequence of code). Hildum's transformation system is a more recent example of such a system [95]. However, this is neither the simplest nor the most convenient way to store and transform the code.

Although Hildum's specification language was designed to work on code which is essentially linear, he points out [95] that the transformation of tree structures is an area of work which deserves further attention. In the Maintainer's Assistant, internally WSL is represented as a syntax tree and is expressed, in a LISP style, as a series of nested lists, as is described in Chapter 8. However, to the user WSL is presented by the interface in a much easier-to-read Algol-style text form. As an example the parallel assignment, which in text form would be written as ⟨X:=A+B+C, Y:=0⟩, would be represented by the tree shown in Figure 5.2 which in turn would be expressed in LISP form as:

```
(Assign (X (+ A B C)) (Y 0)).
```

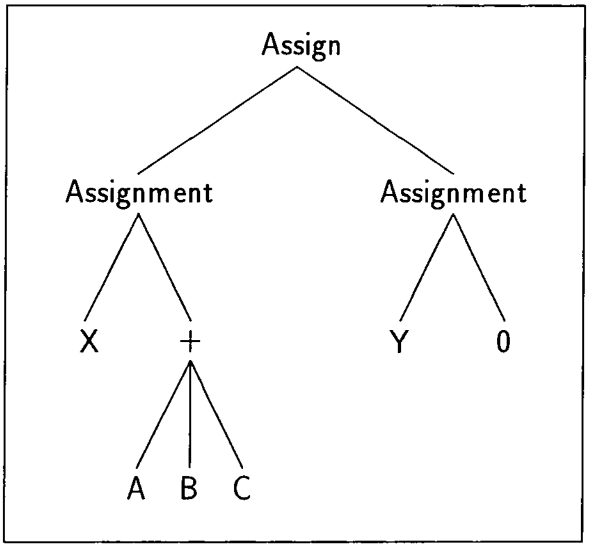


Figure 5.2: Tree Form of a WSL Assign Statement

The exact details of the tree form of WSL are given in Chapter 8.

5.4 Interaction

It is recognised that the acquisition of a specification at a high level of abstraction cannot be an automatic task; this problem is undecidable since a program satisfies infinitely many specifications. Thus, the Maintainer's Assistant must necessarily require human interaction. In fact, this is one of the system's advantages since it allows human expertise of both software engineering and maintenance, and also of the application domain, to influence the direction of the transformation process. However, being computer-based enables the utilisation of the computer's ability to reduce errors, which might be otherwise introduced by clerical work.

The interactive nature of the system also means that the user is able, through performing the transformations, to gain an understanding of the program on which he is working. The system is unable to understand the program and, even if it were, this is not particularly desirable since the user would still not have an improved understanding of it.

5.5 Selecting the Point of Application of Transformations

In an interactive system, it is necessary to indicate explicitly which part of the program should be transformed. For example, with the code

```
(Assign (X (+ A B)) (Y 0))
```

it would be possible to apply the transformation "swap the order of the two components of an item" in two different places: first to the addition, since numeric addition is commutative, and second to the Assign statement, since the two assignments (X:=A+B and Y:=0) are performed in parallel. Using the tree method of representing the program the user can indicate the point of application of a transformation by selecting a branch or leaf in the tree. The selection can be made in several different ways, two of which will be incorporated into the system.

5.5.1 Selecting Program Positions Directly

With the first method of selection, the program would be presented to the user (in a pretty-printed format) on a graphics screen. The user would point to the required program item with the mouse and press a the mouse button. The system would then select the smallest syntactic object (tree branch or leaf) that contained the location of the mouse click. For example, if the user clicked on the symbol for assignment (:=), then the whole assignment (a branch) would be selected, while if he clicked on a variable name, then just that variable (a leaf) would be selected. The graphics interface would convert the position clicked on to a suitable representation of the position in the tree and issues a command to the transformation engine to make that position the selected one.

5.5.2 Selecting Program Positions Relatively

With the second method, the user would be furnished with a series of commands for moving within the program tree. These would include LEFT and RIGHT for selecting branches (or leaves) on either side of the current branch or leaf (if they exist); a DOWN command for selecting the first branch (or leaf) of the current item; and an UP command for selecting the object of which the current item is a component. Other commands could be added for moving to the first or last item in the current component, or to the *nth* item, but these are not strictly necessary.[1] However, in the interests of simplicity, a number of these commands would be incorporated into the system.

Although this method of identifying program items is more cumbersome from the user's point of view, it is the method used *within* the representation of program transformations.

[1] In fact, the command to move LEFT is not strictly necessary, either, as it can be effected by performing UP, followed by a DOWN to select the first component. This would then be followed by a suitable number of RIGHT moves to select the item which was actually required.

5.6 Selecting Several Items

Some transformations allow an operation to be performed on a sequence of program items, rather than on just a single item. For example, a transformation to make a procedure out of a single statement may not be particularly useful. However, a transformation to make a procedure out of several consecutive statements would be much more useful. A sequence of items is referred to as a **span**.

As with the selection of positions in the program, spans could be selected in one of two different ways. First, the user could use the mouse to point at the opposite end of the span of items from the selected item, and click on another of the mouse buttons. Second, a series of commands could be provided for increasing, decreasing and setting the number of items in the span directly. Both of these methods are included in the system. The first is quick and simple for the user, while the second is more efficient, but cumbersome, and is used within the transformations.

5.7 Selecting Transformations

Once the point of application of a transformation has been selected, it is necessary to select the transformation that is actually required, as in most cases more than one will be applicable. The simplest methods of doing this are to allow the user to type in the name of the desired transformation or else to select it from a menu of all the transformations in the system. (Hildum and Cohen suggest this approach [95].) These methods both have drawbacks, however. In the first case, the user would need to do a lot of unnecessary typing (with all the problems of inaccurately typing names), and in the second case, since the number of transformations in the system would probably be very large (several hundred), identifying the transformation which is actually required would be a problem.

In this project, the second method has been selected, but has been modified in four ways in order to make it more usable:

- There are several menus[2] which contain transformations that have similar effects; for example, those which join two program items, those which move a program item and those which delete redundant items;
- Only applicable transformations appear in the menus;
- The transformations which finally do appear in a menu are sorted alphabetically; and
- The user can obtain a description of each transformation which appears in a transformation menu so that, if an unfamiliar transformation appears, he can identify what it does.

The adoption of the approach whereby only *valid* transformations are included in the menus, necessitates that each transformation be in two parts. The first part tests its applicability and the second effects the changes to the program.

5.8 Testing Transformation Applicability

By its nature, a transformation system is one in which the user can make significant changes to the program on which he is working, and he therefore needs to have confidence that the result is always semantically equivalent to the program with which he started. For this to be the case, it is advisable to have a transformation system which checks the applicability (or validity) of the transformations before applying them. If the system is to do this checking, then each transformation needs to coded along with its "applicability condition".

> A transformation's **applicability condition** is the test which determines whether that particular transformation can be legitimately applied (i.e. applied without changing the program's semantics) at the currently selected point in the program.

[2] The menu classes are "Move", "Join", "Use/Apply", "Reorder", "Rewrite", "Insert", "Simplify/Delete", "Multiple" and "Complex".

In the simplest cases, this would be a simple pattern that the particular part of the program has to match with, but in more complex transformations, it might include tests on variable usage, the number of possible exits that a loop has, and so on. Many transformation systems, for example CIP [153] [155] [26] [27], use this method, and encode the tests as a pattern (which contains the former, syntactic information) and as a formula of logic (which contains the latter, so called "semantic", information).

There is a drawback to this simplistic approach; it is not uncommon for the information required by the tests to lie outside the syntactic scope of that section being changed. The transformations for expanding a procedure or function call, or for replacing a variable by its value are cases where this is so. This suggests that a method of selecting ("moving" to) other parts of the program is required. Hence, it seems reasonable to code the transformations' applicability conditions in a *language* designed for that very purpose, i.e. one which includes not only facilities for pattern matching and so on but also "movement" commands. This is the method that has been chosen.

It is necessary to store the applicability conditions separately from the information on how to *perform* the transformation since the tests for each transformation need to be performed without actually modifying the program and are carried out much more frequently. (In many instances, testing a transformation is a much quicker operation than performing the transformation. Thus coding the two parts separately also increases the system's efficiency.)

5.9 Coding the Transformations

At the simplest level, a transformation can be expressed as two patterns: the first a series of elements to be found and some actions to be performed while finding these elements; and second, a new ordering of the elements that describes the result of applying the transformation [95]. Thus, the most obvious way to store transformations is simply as pairs of patterns, but this excludes the possibility of recording more "algorithmic" transformations.

As Hildum pointed out [95], the transformations may require actions to be performed on the elements that are found by the initial pattern match, and, in fact, these actions can become complex as will be seen in Chapter 7. Examples are a transformation to replace a variable with its value, or a transformation to reduce a piece of code with labels and jumps to one with nested loops and If statements. If these are to be accomplished efficiently, sophisticated algorithms need to be built into the transformations. Also, the former requires the transformation to examine sections of code outside the syntactic scope of the section being changed and to perform analysis of this information. Hence, it would seem reasonable to code the transformations in a *language* (to express the algorithms) which includes not only facilities for pattern matching but also "movement" commands. Thus, the language for writing transformations is very similar to the language for writing applicability tests. In fact, in this project the same language is used.

Rather than write a completely new language for this purpose, WSL has been extended so as to be suitable for this purpose. This language is called $\mathcal{META}$WSL, reflecting the fact that it is both an extension of WSL, and designed to manipulate WSL. This contrasts with the CIP project which uses several languages [25] for formulating program schemes, transformation algorithms and applicability tests.

In theory it is not necessary to make any extensions to WSL at all in order to create a means of writing program transformations. If it is assumed that the program being transformed is represented (as a tree) in some global variable, then all that a transformation need do is modify the contents of this variable. Since WSL is a completely general language it would be possible to write a pattern matcher, for example, using WSL and incorporate it into each transformation. However, that would be a very inefficient way to write transformations since each one would duplicate a great deal of code.

There are several advantages of using $\mathcal{META}$WSL:

- $\mathcal{META}$WSL embodies knowledge about program transformations (see Section 6.3);

- Duplication of code by different transformations is avoided;
- No new language needs to be learned in order to write transformations;
- There is a compiler for executable WSL and thus the transformations' code can be compiled for efficiency;
- The system incorporates a pretty-printer which can be used to display the transformations' code (although this must be extended to deal with the statements specific to *META*WSL); and
- Transformations can be applied to themselves, allowing the system to be used in the maintenance of itself.

*META*WSL incorporates a number of statements and functions over and above those provided by WSL. These fall into eight categories:

Program Editing Statements — The most obvious and most important operation on programs by transformations is to edit them. Thus, there need to be some program editing statements.

Pattern Matching and Template Filling — A key part of transformations involves replacing one pattern with another. Thus, *META*WSL includes a function which matches a section of program against a given pattern and returns the result in a table. There is also a function which takes a pattern and a table and replaces the tokens in the pattern with values from the table.

Movement Statements — In order to perform a transformation, a user first selects the section of code on which the transformation is to be performed. Having done this, however, it may be necessary for the transformation temporarily to select another section of the program. Thus, *META*WSL includes statements for moving to different parts of the program tree.

Movement Applicability Testing Functions — Since a specific movement within a tree is not always possible — for example, it is not possible to move down from a leaf node — there are functions which test the applicability of a particular direction of movement within a program tree.

Global Context Variables — Transformations often need to have access to information such as the type of the current item. This is obtained from a number of global variables.

Query Functions — Most transformations require additional knowledge about the program and this cannot easily be obtained from the syntactic process of pattern matching. Such knowledge (which is often called "semantic knowledge" [113]) includes information about variable usage, whether an action system is "regular" and so on. This is obtained by way of query functions.

Symbolic Mathematics and Logic Functions — It is often necessary for a transformation to simplify a mathematical or logical expression, or to demonstrate that one condition implies another. In order to do this, $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL includes some symbolic mathematics and logic functions.

Repetition Statements — It is often necessary within a transformation to test a condition or perform some operation at every node within the subtree with represents the selected program item. The repetition statements allow this to be achieved easily.

This last phrase — "allow this to be achieved easily" — is worth emphasising. All the statements and functions above are designed to make transformations easy to write and to read. All of them could be implemented using WSL to manipulate the variable which holds the program tree.

The extensions included in $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL are outlined in Chapter 6 and given in detail in Appendix C.

5.10 Transformations which Require User Input

Some of the transformations in the system require the user to input information. This information could be the name of a new procedure, or a sequence of statements that are to be inserted after an Exit statement. This information is not required by the applicability tests and *should* not be requested by them, otherwise the user would be required to provide a great deal of unnecessary information just to build the contents of the transformation menus. Thus the input is only requested once a transformation has been selected from a menu.

After a transformation has been selected, the system is able to determine what kind of input is required and, before performing that transformation, the interface presents the user with a dialog box requesting that information. The user enters the information, and the system stores it in a global variable, %Data%, before executing the *MetaWSL* code which performs the transformation. This *MetaWSL* code is able to refer to the variable %Data% in order to make use of the user's input.

5.11 Storing the Transformations

Since the whole of Ward's thesis is to be implemented, there will be a large number of transformations in the system and it is necessary to use an efficient way to store and access them. This is accomplished by using another tree structure.

Each transformation is designed to work on a particular type of WSL program item, given by a pair consisting of the item's generic type and its specific type (see Appendix B). Thus, a transformation which works on an Assign statement would correspond to the pair Statement/Assign. Transformations are categorised in this way so that when transformations are being tested for applicability, only those which operate on the correct type need to be tested. The efficiency of this selective transformation testing is enhanced by putting the transformation index in tree form with the generic type as the top-level of branching and the specific type and the second level. Thus, the part of the tree that needs to be inspected can be determined quickly.

Either the specific type or the generic type can be of the type Any. This signifies that the transformation works on any item of that type. For example, a trans-

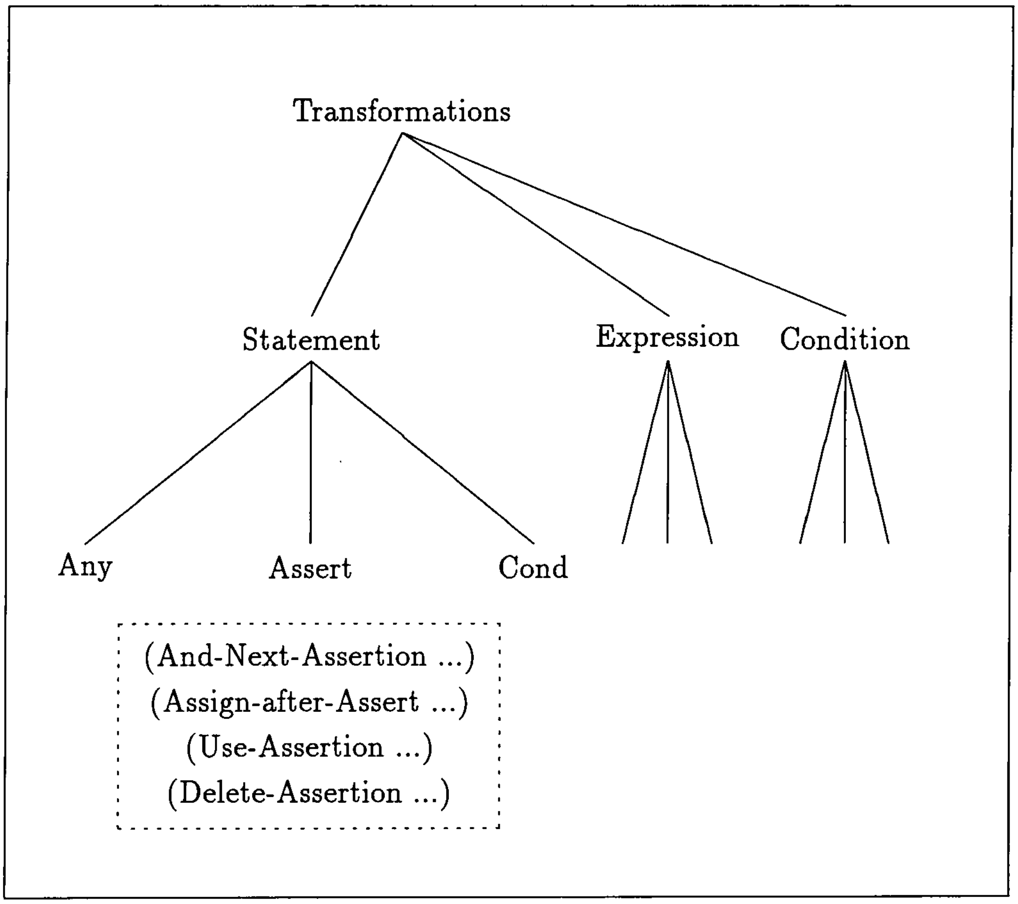


Figure 5.3: Part of the Transformation Catalogue Structure

formation that has the type pair Statement/Any would potentially work on any statement. Such a transformation might move any statement into a loop, for example. If the generic type if Any, the specific type must also, necessarily, be Any. A transformation that has the type pair Any/Any would work on any program item. Such a transformation might simplify all the expressions within the selected program item, for example.

Figure 5.3 shows part of the transformation tree. The ellipses indicate that there is additional data that is stored with each transformation which is not shown in the diagram.

5.12 Constructing the Transformations Catalogue

The Maintainer's Assistant incorporates a large number of transformations, and therefore appears to use the large catalogue approach to building a program transformation library. However, these are divided into four major groups: elementary, compound, generic and high-level transformations. The user can select to work with any or all of these groups of transformations.

5.12.1 Elementary Transformations

The Maintainer's Assistant includes all the elementary transformations (such as inserting assertions) that have been proved by Ward [177]. Selecting to work with just the group of elementary transformations corresponds to the generative set approach to transformation catalogue construction. The maintainer may choose to use sequences of these simple transformations to accomplish some more complex effect, such as removing a local variable.

5.12.2 Compound Transformations

In using the Maintainer's Assistant, a number of common sequences and combinations of transformations have been identified by experience and case studies. Rather than expecting the maintainer to remember such sequences, they are built in as transformations that can be selected in the same way as the elementary transformations. Although there are a large number of these compound transformations, the efficiency gained by learning to use even a few of them seems to outweigh the initial learning time.

5.12.3 Generic Transformations

A great many of the elementary (and some of the compound) transformations are variations on themes. For example, there are separate elementary transformations for taking a statement out of a local variable structure and for taking a statement out of a loop. In addition to the many distinct transformations, these themes have been combined into 22 generic transformations so that the user does not need to know which specific transformation has to be selected.

Although many operations are covered by one of the generic transformations, some operations cannot be incorporated into the generic set since the user has to be more explicit in stating the desired form of the transformed program. For example the user has to be able to choose between changing an unbounded loop into a For loop or into a While loop.

5.12.4 High-Level Transformations

These are the transformations needed to perform the abstractions described in Section 4.4.5.

5.13 Further Facilities for a Usable Transformation System

There are a number of other facilities which are necessary in order to build a usable transformation system. These include the following:

- The ability to undo changes made to the program so that the programmer or maintainer can change his mind and try out various possible options. There is also a corresponding redo facility so that after undoing a sequence of transformations they can be redone without recourse to having actually to perform the transformations again.

- An "audit trail" facility so that as a program is transformed and, if necessary edited, the selection of program items and the operations performed on them, are recorded. This allows the history of the program to be examined to determine whether, for example, any invalid editing operations have been carried out. It would also allow, with a suitable extension of the system, a transformational development to be "replayed" on a modified initial specification, to produce a new version of the program;
- An intuitive and fast user interface to allow the user to try out, quickly and simply, different options (such as which data structure to use) when constructing his program or producing a specification from it; and
- The ability to calculate metrics about the program undergoing transformation so that the user has some feedback as to the progress being made with the restructuring and simplification.

5.14 Other Components of a Working System

It was mentioned earlier that $\mathcal{META}$WSL incorporates certain types of extension over WSL. In particular, these include pattern matching and template filling, query and database functions, and symbolic mathematics and logic functions. Each of these requires an underlying "engine" to provide these facilities. These engines will be described in the next sections of this chapter.

5.14.1 The Pattern Matcher and Template Filler

One of the most important components of the program transformer is the pattern matcher and template filler. Many transformations can be represented as an initial pattern written in terms of literals and general tokens, and final pattern, or template, using the same tokens but possibly different literals. The transformer matches the program against the initial pattern, creating an association table[3] in which the tokens correspond to the actual code which occurred in the program. (If no possible match occurred, then the matcher would return an empty association table of tokens.) The pattern matcher is defined recursively and is essentially similar to the matcher in [173] but ensures that when a token occurs several times, it matches the same item at each occurrence.

The template filler takes the values for these tokens from the table and puts them into the template which represented the transformed version of the program. Even in cases which are not as straightforward as a simple match and fill, pattern matching and template filling often form a crucial part of defining program transformations.

The definition and use of patterns and templates will be explained in detail in Chapter 6, and the mechanism by which pattern matching and filling works will be described briefly in Chapter 8.

5.14.2 Program Query and Database Routines

It is necessary, as part of many of the transformations, to be able to calculate certain information about the program being transformed: the use of variables, whether a program component will always terminate by way of an Exit statement and so on.

For certain queries, the results that are given for any node (program item) are dependent solely on the results for that node's subnodes. Thus, if the information for all the subnodes of a node is known, the value for the node itself can easily be determined. Thus the information for a node is stored in a **database** table linked to that node and rather than recalculating the information each time it is needed, it can be extracted from the database table. Moreover, when the node is moved to another location in the program (as the result of a transformation)

[3] An association table is a table of pairs in which the first element of each pair is a key and the second is some arbitrary data associated with it.

the information at that node remains valid (since the subnodes would have moved with it) and is still stored. If, and only if, one of the components of a node is changed, does the information ever become unreliable and need to be discarded. Thus, when a node changes, all database entries which could depend on it, i.e. those higher in the program tree, must be emptied.

Database tables take the form of a list of pairs. Each pair holds the name of the query and the result, and each list is linked to a node in the program tree. The details and the functions for manipulating the database tables themselves are described briefly in Chapter 8. The $\mathcal{META}$WSL query functions for calculating the information to put in the database are outlined in Chapter 6 and described in detail in Appendix C.

5.14.3 Symbolic Mathematical and Logic Functions

When modifying WSL programs, it is often necessary to rewrite algebraic expressions in simpler forms. This component provides the facilities to do this. In addition it provides similar functions for logical expressions, and is able to test whether one condition logically implies another, for example that a=b+1 implies that a>b — something which is often needed, for example, within the transformations for removing redundant code.

(The symbolic mathematics and logic routines that have been implemented, while incorporating most rewrite rules such as $(a * b) + (a * c) = a * (b + c)$, as yet only provide enough functionality to work on fairly simple examples. For the tool to become more widely applicable, these rules must be supplemented with, for example, the ability to prove relationships inductively. In addition, it is essential that, as with the rest of the system, the correctness of the implementation of this component be proved.)

5.15 Summary and Conclusions

This chapter has addressed the questions raised in Section 4.4.3 and has presented an overview of the design rationale and decisions that have needed to be made in order to construct a practical transformation system from the transformations in Ward's thesis. To summarise:

- WSL programs are represented as abstract syntax trees to obviate the need to parse them;
- The system is interactive to allow the user to employ his experience when transforming a program;
- The point at which transformations are to be applied is selected by the user by pointing and clicking with a mouse, and by the system by "moving" through the program (thereby overcoming the problem of obtaining information about contexts that was encountered in the CIP project);
- It is possible to select and work with several program items in a span;
- Transformations are selected from a number of menus that contain only valid transformations;
- Transformations have their applicability conditions stored separately from the code which performs the changes to the program, so that the applicability can be tested without performing the transformation;
- Both the transformation's instructions for editing a program and the applicability condition are coded using $\mathcal{META}$WSL which is an extension of WSL;
- A tree structure is used to store the transformations since this increases the efficiency of searching the transformation catalogue;
- The transformation catalogue contains a wide variety of transformations covering elementary, compound, generic and high-level transformations;

- An important component of the system is the pattern matcher and template filler;
- The system's efficiency is increased by the inclusion of database tables to hold the result of semantic queries; and
- A subsystem for performing mathematical operations and logical tests symbolically is of great importance.

The implementation based on these design decisions will be described in Chapter 8. However so that it is clear what facilities will be required, it is necessary to give a definition of $\mathcal{META}$WSL and this will form the subject of the next chapter.

Chapter 6

METAWSL

6.1 Introduction

In Chapter 5 it was seen that in order to construct transformations in an optimal way, it was necessary to create a *programming language* — *META*WSL — designed for this purpose. *META*WSL is described in this chapter.

6.2 How Could *META*WSL be Formalised?

Whereas WSL is currently a formally defined language, *META*WSL has yet to be specificed formally. This is an important outstanding area of work. The formal description of *META*WSL could be produced by defining the new constructs in terms of those of WSL by means of **definitional transformations**. This could be achieved by implementing *META*WSL using WSL and a formally-defined abstract data type for program representations. The data type and the implementation in terms of WSL would essentially *be* the definitional transformations of the *META*WSL constructs. A formal definition of *META*WSL would allow (a) the correctness of transformation implementations to be demonstrated; and (b) transformations about *MetA*WSL constructs to be proved and implemented, greatly extending the degree to which the tool can be used in its own maintenance.

6.3 *MetA*WSL as Transformation Knowledge

*MetA*WSL is a domain-oriented language in that it has been specifically designed for use in a particular domain: that of program transformations. Objects from the domain, such as program sections, appear in the language and operations on these objects are readily available as language constructs. Even though the implementations of these objects and operations could be large and complex, they have simple representations in the language and can be combined in such as way that complex program transformations can be written in a few lines of code. The details and special cases are dealt with in the implementation of the constructs. The range of objects and constructs, and the details of any special cases forms the *knowledge* that is represented by the language.

The use of a language for representing domain knowledge should be compared with the IKBS (Intelligent Knowledge-Based System) approach of representing domain knowledge in the form of a rule-based system. Using a rule-based system gives rise to two problems [183]:

1. The *knowledge elicitation problem*: transferring knowledge from the domain expert into a collection of rules suitable for implementing in a rule-based system; and

2. Enabling programmers to extract and make use of the information in the knowledge-base.

Although there are cases, such as medical diagnosis systems, in which the first problem is minimised due to the availability of the knowledge, in the case of program transformation systems there are few, if any, experts with relevant experience. The second problem causes specific difficulties in the case of program transformations systems since it is difficult to see how programmers could make use of a rule-based representation of domain knowledge. However, the knowledge embodied in a very high-level domain-oriented language such as MetaWSL can be employed in the development of program transformations.

6.4 Criteria for Selecting MetaWSL Constructs

MetaWSL was developed through a process of rapid prototyping. A core of language constructs that were thought would be useful, for example those for pattern matching and template filling, were implemented and then used to construct transformations. During the construction of the transformations, it became clear that there were some necessary operations that could not be performed with this basic set of constructs and some sequences of operations which occurred many times. An example of the former case was moving to a different point in the WSL program tree, while an example of the latter was determining the variables that occurred in a section of code. This new *knowledge* was incorporated into the language by adding constructs which performed these operations. Likewise, constructs that were rarely, or never, used were removed.

In addition, the MetaWSL constructs were designed to fulfill Hoare's four "Basic principles of language design" [98]:

1. Security: Every syntactically incorrect program should be rejected by the compiler, interpreter or translator, and executing any syntactically correct program should produce a result or an error message expressed in terms of the source code;

2. Brevity of object code and compactness of run time working data: Despite the reductions in hardware cost, processors are still cheap in comparison with the amount of main store they can address, and backing store is many orders of magnitude slower. Programmers should be able to take advantages of this "spare" capacity to increase a program's quality, simplicity, ruggedness and reliability;

3. Entry and exit code for procedures and functions should be as compact and efficient as for tightly coded machine code subroutines: More generally, there should be no impediments to the use of convenient high-level facilities in the language; and

4. The language should be parsable in a single pass with a simple recursive-descent parser so that the language is easy to read by people and so that it is easier to ensure the correctness of the compiler.

Criteria 2 and 3 are met by the design of *MetA*WSL as can be seen from the remainder of this chapter. Criteria 1 and 4 are functions of the compiler and can be verified from the implementation described in Chapter 8.

6.5 A Survey of *MetA*WSL Constructs

The approach adopted for producing the transformation system and *MetA*WSL was that of rapid prototyping. As a result *MetA*WSL is currently written, and defined, in LISP (see Chapter 8). In order that this chapter not consist of large amounts of LISP code, only informal descriptions of the constructs are given here.

6.5.1 Predefined Variables

*MetA*WSL includes a number of predefined variables which can be referenced (but not assigned to) within transformations. To distinguish them, *MetA*WSL variables begin and end with "%" symbols. Essential among these variables are %Program%, %Posn% and %Item% since these store the program being transformed and the item within the program that is being manipulated at any given time.

%Program% holds the whole program that is currently being transformed. %Item% holds the currently selected syntactic program item. %Posn% holds the

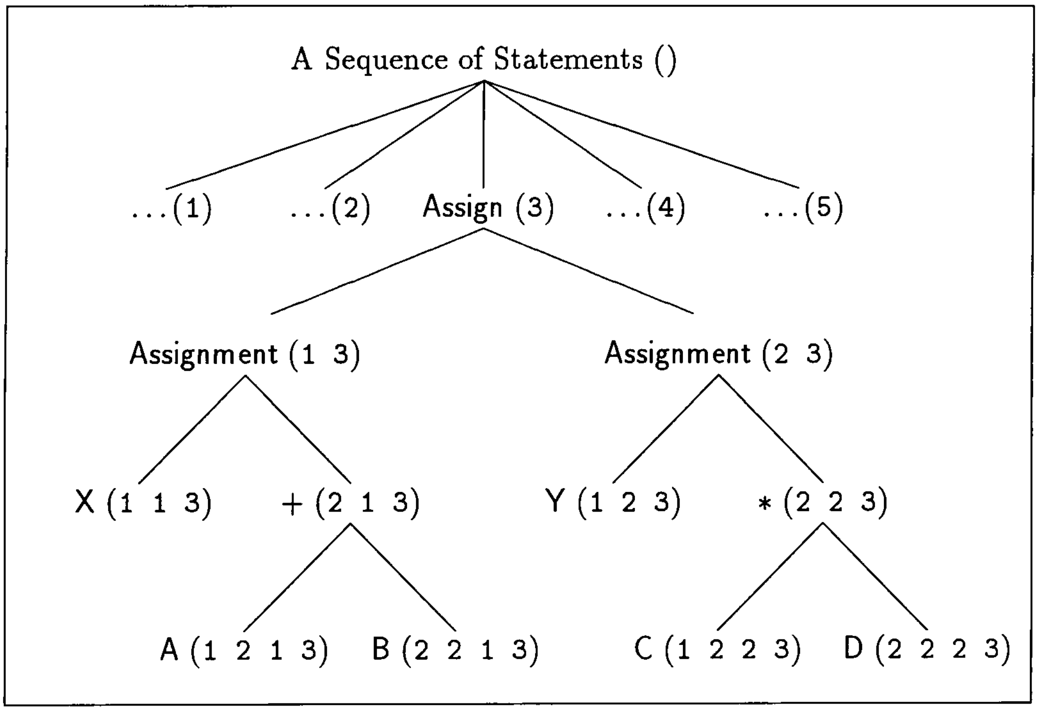


Figure 6.1: Positions in a WSL Program Tree

position, relative to the root node, of the currently selected program item. The position relates to the program components as follows:

- The root of the tree (i.e. the whole program) has its position represented by the empty list; and
- The nth sub-component of each node has the position of its parent, but with an n added (with the function cons) onto the beginning.

Figure 6.1 shows the positions of some of the nodes in the tree for the program

```
(... (Assign (X (+ A B)) (Y (* C D))) ...)
```

in which the Assign statement is the third item in the program. For example, (1 2 3) represents the position of the Y.

The other predefined variables are described in Appendix C.2.

6.5.2 Statements for Moving through the Program

As was seen in Chapter 5, *MetA*WSL needs to include statements for moving within program trees. These statements are: @Up, @Down, @Left, @Right, @To_Last, @Down_Last, @To, @Goto, @Follow and @Return. (*MetA*WSL statements begin and with "@" symbols to distinguish them.) The specific details of each statement are described in Appendix C.3, however, two important characteristics of these statements are worth noting here.

First, these statements, being used internally by transformations, do not record their action in the audit trail of operations performed on the program. This is because the application of a transformation is itself stored, and if the actions that that transformation performed were also stored individually, then the actions of a transformation would effectively be recorded twice. Second, these statements provide **no error checking whatsoever**. Thus, if the current item is a leaf node and the @Down statement is executed, then an error occurs. The reasons for the lack of error checking are twofold:

- The *MetA*WSL movement statements can be executed much more efficiently, which is useful in transformations in which these statements may be executed many thousands of times; and
- The transformations can be written in such a way that no movement to non-existent nodes will ever occur — in fact there are *MetA*WSL conditions which can be used to prevent illegal movement from happening.

6.5.3 Functions for Testing whether Movement is Possible

As mentioned earlier, the *MetA*WSL movement statements do not do any error checking since this is left up to the individual transformations. Thus, it is necessary to incorporate into *MetA*WSL a set of functions to check the validity of movement within the program tree. These return true if and only if it is possible to move in the indicated direction: [_Up?_], [_Down?_], [_Left?_] and [_Right?_]. (MetaWSL functions begin and with "[_" and end with "_]" to distinguish them.) These functions would most likely be used in a MetaWSL program fragment such as:

```
(Cond (([_Up?_]) (@Up))).
```

The specific details of each of these functions are described in Appendix C.4.

6.5.4 Statements for Working with Spans

MetaWSL provides statements for selecting several items in a sequence. These statements neither provide any error checking whatsoever, nor do they record their action in the audit trail of operations performed on the program. The reasons are the same as for the MetaWSL movement statements.

Spans make use of two global variables: %Span% and %Items%. The variable %Span% holds a number one less than the number of items in the current span. (If no sequence has been selected, then this variable has the value zero.) The variable %Items% is a list[1] which contains all the items that make up the span. These variables can be accessed by a MetaWSL program within a transformation and thus provide information about the size and content of spans.

The statements for working on spans are: @Inc_Span, @Dec_Span, @Set_Span and @All_Span. They are described in more detail in Appendix C.5.

[1] Although %Items% contains WSL items, it is not generally a syntactically correct WSL program item. For example, %Items% may contain a list of expressions, but it is not of itself an object of type *Expression*, or any other standard object type.

6.5.5 Statements for Editing the Program

The system would not be able to transform a program at all unless there were some MetaWSL statements to modify the program. As with the statements presented above, the editing statements that MetaWSL includes do not do full error checking (such as checking that the resulting program is syntacticly valid), nor do they update the audit trail of operations performed on the program.

Since these statements can change the program tree in *any* way, they are capable of being used to produce incorrect transformations (i.e. code which changes a program so as not to preserve its semantics). Thus it is dependent on the person who implements the transformations to check that they are used appropriately.[2]

The statements @Del, @Del_Back and @Del_Rest delete items from the program in various ways. Items that have been deleted can be re-inserted elsewhere using the statements @UnDel_Before and @UnDel_After. Any other program items can be inserted using the statements @Ins_Before and @Ins_After, and the the statement @Change_To changes the currently selected program item. These are described in more detail in Appendix C.6.

As was said in Section 6.4, the only constructs that exist in MetaWSL are those that have been found to be useful. A good example is the @Del_Rest statement which deletes all the items in the current branch *after* (but not including) the item at the current position. Thus, it can be used to delete all the unreachable statements after an Exit statement. The corresponding statement, which would delete all the items in the current branch *before* (but not including) the item at the current position, is not included as part of MetaWSL since there are no transformations which permit the deletion of all the statements *before* the current statement.

[2] Once a formal definition of MetaWSLis produced, the implementation of the transformations could potentially be verified formally.

6.5.6 Statements for Repeating an Operation at Different Nodes

Many of the transformations involve manipulating or examining all the nodes within a subtree for which a particular condition is true. For example, if there is a transformation which moves a statement from outside the end of a Loop into the Loop, then it is necessary to be able to identify all the relevant Exit statements within the Loop so that the statement being moved can be inserted in front of all of them. This is a general process, and although it could be coded explicitly on each occasion by a series of movements and tests, there is are general *MetaWSL* statements for performing such tasks. These are the statements @When, @When_Terminal and @When_Terminal_0.

The @When statement performs the specified actions at each item, within a program item, which meets any of a set of given criteria. The criteria could include a test for a "terminal" statement [177], possibly with a particular terminal value. However, since these tests necessitate walking down the tree, which the @When statement does anyway, their functionality can be combined to create the more efficient @When_Terminal and @When_Terminal_0 statements.

With each of these statements, it is possible to prevent the searching of a subtree by using the @No_Deeper statement and to leave prematurely a @When statement and return to the starting item using the @Exit_When statement. The @No_Deeper statement might be used when a @When statement were searching for a particular variable. If it were possible to determine (from the database or by using the functions which return information about variable usage) that the variable did not occur in a particular branch, then the system could avoid searching inside it by executing a @No_Deeper statement. The @Exit_When statement might be used when a @When statement were being used to determine whether a particular condition was true at any point inside a program item. If the condition was fulfilled the transformation would set a flag to indicate the fact and then execute an @Exit_When.

All these statements are described in more detail, with examples, in Appendix

C.7.

6.5.7 Other METAWSL Statements

A transformation's applicability test sets a flag to be true or false in order to indicate whether the current program has passed the test. Two METAWSL statements are used to set this flag to be true or false, and these are @Pass and @Fail.

Some transformations will only be found to fail once they have started to make changes. It may also be that a transformation attempts to make some changes, finds that they do not work and goes on to try some others. In both these cases it is necessary to revert to an earlier version of the program. Therefore METAWSL statements to control this are required. These are: @Wrong, @Mark, @Reposition, @Undo and @Drop.

All these statements are described in more detail, with examples, in Appendix C.8.

6.5.8 Pattern Matching and Template Filling

A transformation can be written using a "pattern match" function that compares an initial program with a pattern, and a "template fill" that replaces the initial program with a final program according to a second pattern.

Defining "Match" Patterns

Patterns used for matching are defined using normal WSL constructs, together with a set of special symbols, some of which include tokens. These special symbols are wildcards that can occur in place of any normal WSL item, and which match any item (or possibly a sequence of items) that occurs at that position. Thus, the *form* of a WSL item can be represented without necessarily specifying its *content*.

The complete set of match pattern symbols is shown in Figure 6.2.

(~?~)	Match one program item.
(~*~)	Match a sequence (possibly empty).
(~?*~)	Match a non-empty sequence.
(~>?~ Token)	Replace Token by the matched program item.
(~>*~ Token)	Replace Token by the matched possibly-empty sequence.
(~>?*~ Token)	Replace Token by the matched non-empty sequence.
(~<?~ Token)	Check that the thing matches the value of Token.
(~<*~ Token)	Check that the list matches the value of Token.
(~OR~ Patterns)	Tests for one of several possible patterns.

Figure 6.2: The Pattern Matching Symbols

The final pattern is a compound pattern. It takes as its arguments a list of possible patterns, and returns the association table with the token values from the first successful match, if there is one.

For example, with the pattern

```
(Cond ((~?~) (~>*~ S))
      ((Else) (~<*~ S)))
```

and the WSL code

```
(Cond ((= A B) (Assign (X 0)) (Skip))
      ((Else) (Assign (X 0)) (Skip)))
```

the matcher would return an association table with S given the value of a list containing two statements (Assign (X 0)) and (Skip).

Defining "Fill" Patterns

Like patterns used for matching, patterns used for filling are defined using normal WSL constructs, together with a set of special symbols. Again, these symbols may occur in place of any normal WSL item. They extract the entry indicated by the token from an association table and put it into the pattern in the place of the symbol. The complete set of fill pattern symbols is shown in Figure 6.3.

(~<?~ Token)	Replace by the single item, Token.
(~<*~ Token)	Replace by the sequence of items, Token.
(~<SE~ Token)	Replace by the simplified form of the expression Token.
(~<SC~ Token)	Replace by the simplified form of the condition Token.

Figure 6.3: The Template Filling Symbols

For example, with the pattern

```
(Assign (X (~<SE~ (- (~<?~ E) (+ A B)))))
```

and the association table containing E with the value (+ A B), the template filler would return the WSL code

```
(Assign (X 0))
```

since it would have simplified the expression (- (+ A B) (+ A B)) to give 0.

The MetaWSL functions which enable pattern matching to be incorporated into transformations are [_Check?_] and [_Match_]. [_Check?_] tests a section of WSL against a pattern and just returns true or false, and [_Match_] returns, in the form of an association table, the values of any tokens in the pattern that have matched. The MetaWSL functions for template filling are [_Fill_In_] and [_Fill_Args_]. All these functions are described in more detail in Appendix C.9.

6.5.9 Functions for Association Tables

The pattern matching functions return association tables as their results, and the template filling functions require association tables to provide the values of the tokens being replaced. Provided it were never necessary to do intermediate work on these tables between performing a match and a fill, then no other operations on the tables would be necessary. However it is often the case that transformations need to be able to access particular values in the table, or to be able to add items to, or change items in, the tables. The functions to do this are [_Put_], [_Get_] and [_Val_] and are described in Appendix C.10.

6.5.10 Functions for Examining the Program undergoing Transformed

Within program transformations, it is necessary for the transformation to determine attributes of the code being transformed. This can often be achieved by means of a pattern match, but sometimes the information required cannot be obtained, or more usually, cannot be obtained efficiently, this way. Thus, the following functions are provided to make the transformations simpler: [_With_Else?_], [_Size_], [_Comps_], [_Contents_], [_All_Contents_], [_Statements_], [_Calls_], [_Total_Size_] and [_Body_]. They are described in more detail in Appendix C.11.

6.5.11 Functions Relating to Variable Usage

The transformations often require information about the use of variables. In particular, transformations need to know, for any section of code, all the variables that are referred to, those that are used (or accessed) and those that are assigned to. The functions to do this are described in detail in Appendix C.12, and are: [_Variables_], [_Used_], [_Assigned_], [_Used_Only_], [_Assd_Only_] and [_Assd_To_Self_].

6.5.12 Functions for Testing Types and Syntax

At certain points when performing transformations (or when testing their applicability) it is necessary to check the type or syntax of particular items. These occasions occur when the action of a transformation is only valid for a particular type of item (or set of types) or when the user has supplied some information as input and the transformation needs to check the validity of its syntax. The functions to perform these operations are the following: [_Number?_], [_Variable?_], [_Syntax?_], [_S_Type?_], [_G_Type?_], [_P_Type?_] and ==. These functions are described in detail in Appendix C.13.

6.5.13 Functions Relating to Loops

There are functions which return the information about Loops which transformations often need to have available. This information cannot be obtained (simply) by any other means. These functions are the following: [_Primitive?_], [_Depth_], [_Terminal_Value_], [_Terminal?_], [_Reducible?_], [_Proper?_], [_Improper?_] and [_Dummy?_]. They are defined in Ward's thesis [177] and are also described in detail in Appendix C.14.

6.5.14 Functions for Testing Action Systems

Just as there are *Meta*WSL functions for determining information about Loops, so there are also functions for returning information about action systems. These functions are the following: [_Regular?_], [_Regular_System?_] and [_Calls_Terminal?_]. They are defined in Ward's thesis [177] and are also described in detail in Appendix C.15.

6.5.15 Functions for Symbolic Mathematics and Logic

While symbolic simplification functions can be built into the templates used for filling, it is often necessary to be able to perform explicit symbolic manipulations such as forming the conjunction of two conditions. The following functions allow certain symbolic manipulations to be performed in *Meta*WSL: [_And_], [_Or_], [_Not_], [_->T?_], [_->F?_], [_Simplify_], [_Simplex?_] and [_Isolate_]. These functions are described in more detail in Appendix C.16.

6.5.16 Other Sundry METAWSL Functions

In addition to the METAWSL functions already described, there are some which do not fit into any particular category. These are [_Replace_] and [_Rplc_All_], which perform search-and-replace operations on the given data, [_Arguments_] and [_Occ_], which return information about each occurrence of named constructs, [_Diff_] which calculates the differences between two program sections using a unification algorithm, and [_Increment_] and [_Decrement_], which increment or decrement a sequence of statements according to Ward's definition [177] [174]. These functions are described in more detail in Appendix C.17.

6.5.17 Calling other Transformations

Transformations can be built up from the METAWSL statements and functions described above, together with ordinary WSL constructs. However, it is often the case that more powerful transformations can be constructed by combining existing transformations — provided that the transformations are legitimately applied. In fact, some more complex transformations can really only practicably be constructed by this method.

The METAWSL [_Trans?_] function returns true if and only if the named transformation is applicable at the current point in the program and the METAWSL @Trans statement performs a named transformation without testing the transformation's applicability. These statements can be combined to ensure that only applicable transformations are performed. More details and an example are given in Appendix C.18.

6.6 Summary and Conclusions

This chapter has introduced the statements and functions of MetaWSL. It is proposed that MetaWSL meets the criteria of success given in Chapter 4 in that it is both simple and complete.[3] The next chapter considers whether this is the case. In particular, it looks at whether MetaWSL is simple, in that transformations can be expressed in such a way that their function is not obscured by implementation details, and complete, in that all the transformations that could potentially be included in the system can be expressed using MetaWSL. This assessment will be performed by representing transformations using MetaWSL.

[3] MetaWSL is not a *minimal* extension of WSL; for example there is a @Change_To statement which could have been omited since it can be replaced by a combination of @Delete with @Ins_After or @Ins_Before.

Chapter 7

The Transformations in the Maintainer's Assistant

7.1 Introduction

The previous chapter introduced $\mathcal{M}\varepsilon\mathcal{TA}$WSL and gave some simple examples of how the different statements and functions could be used. This chapter explains how a catalogue of transformations can be built up using $\mathcal{M}\varepsilon\mathcal{TA}$WSL. The catalogue will be seen to be a complete implementation of Ward's transformations[1], and also usable in the sense that are transformations which combine Ward's transformations in efficacious ways, reducing the need for the user of the system to learn long sequences of elementary transformations. The construction of such a catalogue provides insight into the questions of whether, and how clearly and concisely, $\mathcal{M}\varepsilon\mathcal{TA}$WSL can represent program transformations.

Details will be given in this chapter of the types of transformations that are incorporated into the Maintainer's Assistant together with examples. There are

[1] Every theorem of mathematics gives rise to a transformation or refinement (replacing one assertion by another), so completeness is unattainable. In practical terms, there are only a small number of transformations which had to be proved from first principles [177]; all others are combinations of these or applications of the induction rules for iteration and recursion.

eight examples, illustrating different aspects of transformation construction, as follows:

Assert_After_While uses a single pattern match;

Split_Cond uses several pattern matches to distinguish between cases;

Remove_Unused_Local_Var uses query operations;

Remove_Dummy_Loop tries a sequence of operations and undoes the changes if they prove not to be beneficial;

Fully_Factor_Cond is built from calls to simpler transformations;

Reduce_Exits_in_Loop uses the basic transformation facilities, but incorporates a complex algorithm to select the best course of action;

Take_Out−>> shows the simplicity of constructing generic transformations; and

Insert_Invariant shows how the symbolic maths system is used in a high-level transformation.

7.2 An Overview of the Transformation Catalogue

The transformation catalogue of the Maintainer's Assistant currently contains 601 transformations. This collection has come about through an evolutionary process. A minimal catalogue — in the sense that all other transformations could be produced as sequences of these — containing only Ward's elementary transformations (i.e. those proved in his thesis [177]) were implemented first. This, however, has been extended to include compound, generic and high-level transformations, all of which will be explained below.

Each new transformation was added either as the result of observing a common combination of simpler transformations — an example being the transformation to replace an action system by a series of nested conditions and loops — or as a generalisation of a number of other transformations — an example being the general "Merge->>" which merges an item into the following item.

7.3 Elementary Transformations

The most fundamental transformations in the Maintainer's Assistant are those that have been proved by Ward in his thesis [177]. These provide a core set of about 200 transformations from which others can be constructed.

7.3.1 Method

These transformations are, on the whole, very simple and this simplicity is not obscured by representing them in *MetA*WSL. The transformations can be written using pattern matching, combined with some tests of variable usage, logical simplification and use of the @When statement.

7.3.2 Examples

For each of the transformations in this chapter, the following information will be given:

- A "header" consisting of:
  1. The name of the transformation;
  2. The generic type (such as Statement) on which the transformation works;
  3. The specific type (such as Assign) on which the transformation works; and

- The MetaWSL code for the transformation's applicability test; and
- The MetaWSL code for performing the transformation.

Adding an Assert Statement

The following transformation creates an extra Assert statement after the end of a While loop. The statement asserts that the condition of the loop is false. For example, after the statement

```
(While (> a 1) (...))
```

the assertion

```
(Assert (<= a 1))
```

can be added. This transformation makes use of simple pattern matching, and symbolic simplification when performing the [_Fill_In_]. The header information for this transformation is as follows:

Name	Assert_After_While
Generic Type	Statement
Specific Type	While

The applicability test would just be as follows:

```
((@Pass))
```

since the transformation is valid for *any* While statement at any position in the program. The code for performing the transformation would be as follows:

```
((@Ins_After ([_Fill_In_] Statement
                          (Assert (~<Sc~ (Not (~<?~ B))))
                          ([_Match_] Statement
                                     (While (~>?~ B) (~*~))
                                     Empty)))
(@Right)).
```

The third argument of the [_Fill_In_] statement should be an association table, and this is indeed the case since the function [_Match_] returns such a table. The final @Right statement ensures that the newly inserted Assert statement is the currently selected program item after the transformation has been performed.

Splitting a Cond Statement

The next transformation splits a Cond statement whose condition is the conjunction (i.e. And) or disjunction (i.e. Or) of two conditions, into a combination of Cond statements. For example, the statement

```
(Cond ((Or (= A 1) (= B 1)) (Assign (Q 1)))
      ((Else)               (Assign (Q 2))))
```

can be rewritten (provided that neither condition has any side effects) as

```
(Cond ((= A 1) (Assign (Q 1)))
        ((Else)  (Cond ((= B 1) (Assign Q 1))
                       ((Else)  (Assign Q 2))))).
```

The transformation is written using pattern matching and template filling, and makes use of conditional patterns in order to check for the cases both of an And and of an Or. The header information for this transformation is as follows:

Name	Split_Cond
Generic Type	Statement
Specific Type	Cond

The applicability test would be as follows:

```
((Cond ((([_Check?_] Statement
                     (Cond (((~OR~ And Or) (~?~) (~?~))
                               (~*~))
                             ((Else)
                        (~*~))))

        (Comment "Now test for a general expression or condition.")

        (@Down)
        (@Down)
        (Cond ((And (= ([_Occ_] Gen_Expr %Item%) 0)
                    (= ([_Occ_] Gen_Cond %Item%) 0))
                (@Pass))
              ((Else)
                (@Fail))))
      ((Else)
        (@Fail))))
```

and the *MetA*WSL code for performing the transformation would be as follows:

```
((Var ((Table ([_Match_] Statement
                    (Cond ((And (~>?~ B1) (~>?~ B2))
                            (~>*~ S1))
                          ((Else)
                            (~>*~ S2)))
                    Empty)))

    (Cond ((Non_Empty? Table)

          (Comment "This is the case for an 'And' condition.")

          (@Change_To ([_Fill_In_]
                       Statement
                       (Cond ((~<?~ B1) (Cond ((~<?~ B2) (~<*~ S1))
                                               ((Else) (~<*~ S2))))
                             ((Else) (~<*~ S2)))
                       Table)))

        ((Else)

          (Comment "This is the case for an 'Or' condition.")

          (Assign (Table ([_Match_] Statement
                                    (Cond ((Or (~>?~ B1) (~>?~ B2))
                                            (~>*~ S1))
                                          ((Else)
                                            (~>*~ S2)))
                                    Empty)))

          (@Change_To ([_Fill_In_]
Statement
(Cond ((~<?~ B1) (~<*~ S1))
     ((Else) (Cond ((~<?~ B2) (~<*~ S1))
                   ((Else) (~<*~ S2)))))
Table)))))
```

The transformation has two cases corresponding to the two clauses of the Cond statement. These deal with the cases of the condition being formed with an And and an Or, respectively, and are distinguished by means of pattern matching. If a pattern match fails, then the association table in which the results are stored will be empty, and the predicate function Non_Empty? can be used to test this.

The second call to the function [_Match_] will always return a non-empty association table, since the applicability test ensures that one of the patterns in the two calls to [_Match_] matches the current program item.

Removing a Local Variable

The following transformation removes a local variable structure when there is a single local variable which is never referenced (except possibly in assignments to itself) within the body of the local variable structure.[2] For example, the transformation would change the code

```
(Var ((X 2))
   (While (< Y 10)
         (Assign (X (* X 4)))
         (Assign (Y (+ Y 1)))))
```

into

```
(While (< Y 10)
        (Assign (Y (+ Y 1))))
```

[2] To enhance the clarity of how $\mathcal{M}$ETAWSL is used, this transformation is a simplification of one that is actually implemented in the Maintainer's Assistant.

since the local variable X is only ever used in assigning to itself, and does not affect the value of Y.

The header information for this transformation is as follows:

Name	Remove_Unused_Local_Var
Generic Type	Statement
Specific Type	Var

The transformation's applicability test is written using some of the functions for examining the program being transformed and also the database query functions which test variable usage. The applicability test would be as follows:

```
((@Down)
 (Cond ((> ([_Size_] %Item%) 1)
          (@Fail))
       ((Else)
          (Var ((V (Hd ([_Assigned_] %Item%))))
              (@Up)
              (Cond ((Member? V (Set_Diff ([_Used_]        ([_Body_] %Item%))
                                      ([_Assd_To_Self_] ([_Body_] %Item%))))
                        (@Fail))
                    ((Else)
                        (@Pass))))))).
```

The test works by moving down into the Var thereby selecting the list of initial assignments to the local variables. It then tests whether there is more than one initial assignment, in which case the transformation is not valid. Otherwise, the transformation stores the name of the local variable in the variable V. (Since the function [_Assigned_] returns a list, it is necessary to take the head of the list.) Next, the whole Var is selected again, and there is a test to determine whether the variable named in V is ever used other than in assignments to itself. If it is, then the transformation is not valid, otherwise it is valid.

The MetaWSL code for performing the transformation would be as follows:

```
((@Down)
 (Var ((V ([_Assigned_] %Item%)))
(While ([_Right?_])
        (@Right)
        (@When 0 ((And ([_Type?_] Assignment)
                          (== ([_Assigned_] %Item%) V))
                       (Cond ((= %Length% 1) (@Up)))
                       (@Del))))

(@Up)
(@Ins_After ([_Fill_Args_] Statements
                          ((~<*~ S))
                          ([_Match_] Statement
                                       (Var (~?~) (~>*~ S))
                                       Empty)))
(@Del))).
```

The transformation works by moving down into the Var and storing the name of the local variable in V. Next the transformation moves through every other top-level component of the Var statement (using a While loop) and for each such component it considers all the assignments inside them which assign to the local variable named in V. These are the assignments which need to be removed. If they are the only assignment in an Assign statement, then it is necessary to move up so that it is the Assign statement, and not the assignment, that is deleted. Finally the transformation selects the whole Var once more and does a pattern match and template fill in order to complete the change to the program. The statement @Ins_After is used with the function [_Fill_Args_] since a *list* of statements is inserted in place of a since statement (which is then deleted).

This pattern matching and template filling could also have been achieved using the statement

```
(@Ins_After (Args ([_Body_] %Item%)))
```

— a form which is used in the next example.

Removing a Dummy Loop

The final example transformation in this section removes a dummy Loop, replacing it with the sequence of statements that that loop contained. For example, the Loop

```
(Loop (Assign (X (+ X 1)))
      (Cond ((= X 10) (Assign (Y 0)) (Exit 1) (Assign (Q 0)))
            ((Else)    (Assign (Y 3)) (Exit 1))))
```

can be replaced by the statements

```
(Assign (X (+ X 1)))
(Cond ((= X 10) (Assign (Y 0))
      ((Else)    (Assign (Y 3)))).
```

The transformation uses the *MetA*WSL functions which manipulate loops with Exit statements. The header information for this transformation is as follows:

Name	Remove_Dummy_Loop
Generic Type	Statement
Specific Type	Loop

The transformation's applicability test is as follows:

```
((Cond (([_Dummy?_] %Item%)
          (@Pass))
        ((Else)
          (@Mark)

          (@When 0 ((And ([_G_Type?_] Statement)
                          ([_Right?_])
                          ([_Improper?_] %Item%))
                     (@Del_Rest)))

          (Cond (([_Dummy?_] %Item%)
                  (@Pass))
                ((Else)
                  (@Fail)))
          (@Undo)))).
```

The applicability test first tests whether the loop is a dummy loop. If it is, then the transformation is valid. If it is not, then the applicability test records the current version of the program and modifies it (where possible) to remove all statements which occur after improper statements (since these will never be executed). This may make the loop into a dummy loop and, if it does, then the transformation is valid otherwise it is not valid. In both these last cases, the changes are undone to leave the program as before.

The *MetA*WSL code for performing the transformation would be as follows:

```
((@When 0 ((And ([_G_Type?_] Statement)
                ([_Right?_])
                ([_Improper?_] %Item%))
           (@Del_Rest)))

 (@Change_To ([_Decrement_] %Item% 1))

 (Cond (([_Trans?_] Delete_All_Skips)
        (@Trans Delete_All_Skips)))

 (@Ins_Before (Args ([_Body_] %Item%)))
 (@Del)).
```

The transformation first removes any statements which occur after improper statements. Next it decrements the loop by one, which may leave spurious Skip statements (since Exit 1 statements reduce to Skip statements) so these must be removed by calling the appropriate transformation. Finally, the statements of the loop are inserted after the loop and the loop is deleted, leaving only the decremented loop body.

7.3.3 Summary

The examples in this section demonstrate that is it *possible* to express Ward's transformations using $\mathcal{META}$WSL. It can also be seen that with regard to applicability conditions there are four cases:

1. Transformations that are always valid and thus have trivial applicability conditions; for example "Assert_After_While";
2. Transformations that involve a single pattern match which directly reflects the form of the code on which they work; for example "Split_Cond";
3. Transformations that make use of calls to the database query functions; and
4. Transformations that combine one of the previous cases with a *temporary* change to the program.

It can be seen from the examples given, that all these cases make for clear and concise code.

There are three different varieties of code for performing the transformations:

1. Transformations that consist of a pattern, a template and the $\mathcal{META}$WSL constructs for doing the matching and filling; for example "Assert_After_While".
2. Transformations that are similar, but slightly larger and more complex since they are an implicit combinations of other transformations; for example "Split_Cond".
3. Transformations which use calls to the database query functions, or the @When statement, thus obviating the need to include explicit tree walking algorithms.

From the examples given, and from the tables at the end of this chapter, it can be seen that $\mathcal{META}$WSL provides a good means by which elementary transformations can be expressed. Doing so enables the complete set of Ward's transformations to be implemented within the Maintainer's Assistant.

7.4 Compound Transformations

The compound transformations in the Maintainer's Assistant perform common sequences and combinations of transformations but can be selected in the same way as the elementary transformations. There are about 400 of these transformations.

7.4.1 Method

Coding these transformations uses techniques that are very similar to the techniques used in coding the elementary transformations. However, the additional features of WSL are used in order to guide the selection of patterns and templates, the selection of program items on which to work and so on. A simple choice of this type was used in the example above which removed dummy loops. If the initial test failed, then the transformation attempted to put the code into a form so that the test might then work.

$\mathcal{META}$WSL has been designed to incorporate the features that are needed to construct compound transformations with the minimum of effort. The most important of these features are as follows:

- The heuristics that are used to guide the choice of action in compound transformations are usually based on the program size, number of Call statements in a section of code, the use of variables and so on. All this information is provided through $\mathcal{META}$WSL's functions for examining the program being transformed and also the database query functions;
- Use of the @Mark, @Undo, @Reposition and @Drop statements allow the compound transformations to perform back-tracking should a particular sequence of operations not "improve" the program;

- The @When, @When_Terminal and @When_Terminal_0 statements allow the compound transformations to work on relevant items within the program (i.e. those that meet particular criteria) and not on others; and
- The ability to call other transformations from *MetaWSL*, allows complex transformations to be built up by combining simpler ones. This improves the maintainability and readability of the code.

7.4.2 Examples

Fully Factorising a Cond Statement

The first example transformation in this section takes as many statements as possible out of the beginning and end of a Cond statement. For example, it would replace

```
(Cond ((= A B) (Assign (X 1)) (Assign (Y 2)) (Exit 1))
      ((= C D) (Assign (X 1)) (Assign (Y 7)) (Exit 1))
      ((Else)  (Assign (X 1)) (Exit 1)))
```

by

```
(Assign (X 1))
(Cond ((= A B) (Assign (Y 2)))
      ((= C D) (Assign (Y 7))))
(Exit 1).
```

The way that the transformation is written reflects the transformation's function. In the code for performing the transformation there is an additional test to determine whether the transformation "Remove_Empty_Cases" in valid. This deals with the case in which any of the Cond statement's guards contain no statements, or only a Skip statement, and can be removed.

The header information for this transformation is as follows:

Name	Fully_Factor_Cond
Generic Type	Statement
Specific Type	Cond

The applicability condition is as follows:

```
((Cond ((Or ([_Trans?_] Backward_Factor_Cond)
            ([_Trans?_] Forward_Factor_Cond))
          (@Pass))
       ((Else)
          (@Fail)))).
```

The code for performing the transformation as follows:

```
((While ([_Trans?_] Backward_Factor_Cond)
          (@Trans Backward_Factor_Cond)
          (Cond (([_Trans?_] Remove_Empty_Cases)
                   (@Trans Remove_Empty_Cases))))
 (While ([_Trans?_] Forward_Factor_Cond)
          (@Trans Forward_Factor_Cond)
          (Cond (([_Trans?_] Remove_Empty_Cases)
                   (@Trans Remove_Empty_Cases))))).
```

Removing Exit Statements from a Loop

The following example is a transformation which, when applied to a Loop statement, reduces the number of Exit statements within that loop if this is possible. The header information for this transformation is as follows:

Name	Reduce_Exits_In_Loop
Generic Type	Statement
Specific Type	Loop

The applicability condition is as follows:

```
((Var ((Num 0))
    (@When 0 ((Not (Member? (!L Exit) ([_Statements_] %Item%)))
               (@No_Deeper))
             (([_S_Type?_] Exit)
               (Assign (Num (+ Num 1)))
               (Cond ((> Num 1) (@Exit_When)))))

    (Cond ((> Num 1) (@Pass))
          ((Else) (@Fail)))))
```

The test is quite straightforward in that it counts the number of Exit statements in the selected program item, ignoring branches that contain no such statements and ceasing to search when more than one has been found.

The code for performing the transformation is as follows:

```
((Var ((Num 0) (Temp 0))

    (Comment "Count the number of 'Exit' statements.")

    (@When 0 ((Not (Member? (!L Exit) ([_Statements_] %Item%)))
               (@No_Deeper))
             (([_S_Type?_] Exit)
               (Assign (Num (+ Num 1)))))

    (Comment "Move statements that follow 'Var' statements inside the
               'Var' statements.")

    (@When 1 ((Not (Any_Member? (!L (Var Exit))
                       ([_Statements_] %Item%)))
               (@No_Deeper))
             ((And ([_S_Type?_] Var) ([_Right?_]))
               (While ([_Trans?_] Forward_Absorb_Var)
                       (@Trans Forward_Absorb_Var))))

    (Comment "Search for 'Cond' statements that include 'Exit' statements
               and simplify them by merging common guards.")

    (@When 0 ((Not (Any_Member? (!L (Cond Exit)) ([_Statements_] %Item%)))
               (@No_Deeper))
             (([_S_Type?_] Cond)
               (@Mark)
               (@Trans Super_Expand_And_Factor)
               (While (Not ([_S_Type?_] Loop)) (@Up))
               (Assign (Temp 0))
```

```
           (Comment "Count the new number of 'Exit' statements.")

           (@When 0 ((Not (Member? (!L Exit) ([_Statements_] %Item%)))
                      (@No_Deeper))
                    (([_S_Type?_] Exit)
                      (Assign (Temp (+ Temp 1)))))

           (Comment "If there are no fewer 'Exit' statements then revert
                     to the previous version of the program, otherwise
                     continue with this version.")

           (Cond ((>= Temp Num)
                   (@Undo))
                 ((Else)
                   (@Reposition)
                   (@Drop)
                   (@Exit_When)))))

(Comment "Now move as many statements as possible from each 'Cond'.")

(@When 1 ((Not (Member? (!L Cond) ([_Statements_] %Item%)))
           (@No_Deeper))
         (([_Trans?_] Fully_Factor_Cond)
           (@Trans Fully_Factor_Cond)))

(Comment "Now move as many statements as possible from each 'Var'.")

(@When 1 ((Not (Member? (!L Var) ([_Statements_] %Item%)))
           (@No_Deeper))
         (([_Trans?_] Fully_Factor_Var)
           (@Trans Fully_Factor_Var))))
```

The transformation initially counts the number of Exit statements and stores the result in the variable Num.

Next the transformation searches for all Var statements, within the selected statement, which include Exit statements and absorbs as much as possible into them; i.e. the transformation moves statements that follow a Var inside the Var so that tests are moved "closer" (in the sense that they are on the same level in the program tree) to assignments.

Next the transformation makes a copy of the current version of the program since the next stage may *increase* the number of Exit statements and not *decrease* it. The transformation now performs the transformation "Super_Expand_And_Factor" at each Cond statement within the loop. The transformation "Super_Expand_And_Factor" copies all statements before and after a Cond statement into the body of the Cond statement (where possible), attempts to merge as many guards and statements as possible, and finally takes as many statements as possible out of the beginning and end of the Cond. This often has the effect of reducing the number of Exit (and other) statements, but if it does not, then it is necessary to revert the an earlier version of the program.

Finally, the transformation attempts to simplify the result by taking as many statements as possible out of the beginning and end of all Cond and Var statements.

Restructuring an Action System

By looking at examples of real programs, a number of common scenarios have been identified in which more complex transformation strategies can beneficially be employed. Since transformations are written as programs in $\mathcal{META}$WSL, it is possible to incorporate arbitrarily complex heuristics into transformations. One such transformation is "Collapse_Action_System" which replaces any regular action system (see Section 4.2.2) by a series of nested loops and conditions.

While existing code restructurers that can do this kind of restructuring, none of them is able to do it in the general case without either copying code or introducing flag variables, thus complicating the data flow for the sake of simpler control flow. The "Collapse_Action_System" transformation of the Maintainer's Assistant employs an algorithm by which the labels and jumps (i.e. the actions and Call statements) can be removed with no (or negligible) copying of code and without having to resort to the introduction of new variables.

The transformation for doing this is too large to give as an example in this chapter, but is given in Appendix D.

7.4.3 Summary

Compound transformations provide the user of the Maintainer's Assistant with some powerful tools for restructuring software.

While the tables at the end of this chapter give a summary of the clarity and conciseness of the transformations, it is helpful to consider actual cases. In assessing whether MetaWSL provides a good basis for writing compound transformations it is necessary to consider two different types of transformation as typified by the examples above. The first type, exemplified by "Fully_Factor_Cond", does no processing except by way of calls to other transformations. As a result, this type of transformation is both concise and clear.

The other type of transformation, for example "Reduce_Exits_In_Loop", uses an (often complex) algorithm to determine the changes that must be made to the program. This complexity means that the corresponding coding of the transformation may also be complex, and MetaWSL must be assessed on whether its use causes any undue obscurity to be introduced. As can be seen from the example, the only *changes* to the program that the transformation makes are made by calls to simpler transformations. The algorithm in question arises from determining *which* transformations to use on *which* parts of the program. This determination of which transformation to apply is undertaken in one of several ways:

1. Using a pattern match;

2. Using the database query functions;

3. Moving up, right, or left through the program until a statement of the correct type is reached — this is done using MetaWSL of the form

   ```
   (While (Not ([_S_Type?_] Loop)) (@Up))
   ```

   for example;

4. Moving down through the program to each place where a certain condition holds — this is easily written using the MetaWSL @When statement; or

5. Counting the number of occurrences of a particular form within a section of code — in simple cases this can be done using the [_Occ_] function and in all other cases the @When statement provides this facility.

All these methods can be written as $\mathcal{META}$WSL code which clearly expresses the purpose of the code. In addition the code is fairly concise, especially where is removes the need to include explicit tree walking algorithms. (The use of the @When construct could be made more concise by removing the tests which prevent it walking down unprofitable subtrees. However, removing these tests would also result in a marked reduction in efficiency.)

Overall, $\mathcal{META}$WSL provides a good method of expressing compound transformations, which are an important part of the Maintainer's Assistant.

7.5 Generic Transformations

The generic transformations, by selecting from a number of more specific transformations, enable a user to accomplish much of his work using only a small number of transformations. The complete list of generic transformations is as follows:

Delete	Delete the redundant program item.
Simplify	Simplify the program item or the sequence of program items.
->>	Swap the program item with the one following it; i.e. move the current item to the right.
<<-	Swap the program item with the one preceding it; i.e. move the current item to the left.

Merge–>> Merge the program item into the one following it, making several copies of the merged program item if necessary.

Absorb–>> Absorb into the selected program item the one that follows it, making several copies of the following program item if necessary.

<<–Merge Merge the program item into the one preceding it, making several copies of the merged program item if necessary.

<<–Absorb Absorb into the selected program item the one that precedes it, making several copies of the preceding program item if necessary.

Multi_Move–>> Move the program item as far as possible to the right.

<<–Multi_Move Move the program item as far as possible to the left.

Multi_Absorb–>> Absorb into the selected program item as many program items that follow it as possible.

<<–Multi_Absorb Absorb into the selected program item as many program items that precede it as possible.

Separate–>> Take the currently selected item out of its enclosing structure towards the right.

<<–Separate Take the currently selected item out of its enclosing structure towards the left.

Take_Out–>> Separate a component of the selected program item towards the right.

<<–Take_Out Separate a component of the selected program item towards the left.

<<–Take_Out–>> Separate as many components as possible from the program item, by taking them out in both directions.

Apply->> Use the current program item to simplify some program item that follows it.

<<-Use Simplify the current program item by using the one that precedes it.

Insert_Assert Insert an Assert statement inside the current item.

Add_Assert Add an Assert statement after the current item.

Add_And_Insert_Assert Add an Assert statement after, and insert an Assert statement inside, the current item.

7.5.1 Method

Generic transformations can be written very simply using the @Trans statement and [_Trans?_] function which are provided by $\mathcal{META}$WSL. The example below illustrates this.

7.5.2 An Example

This transformation takes a sub-component out of the selected item, towards the right. For example, it would transform

```
(Cond ((= A B) (Assign (X 0)))
      ((Else)   (Proc_Call P () ()) (Assign (X 0))))
```

into

```
(Cond ((= A B) (Skip))
      ((Else)   (Proc_Call P () ())))
(Assign (X 0))
```

by extracting the Assign statement.

The header information for this transformation is as follows:

Name	Take_Out->>
Generic Type	Any
Specific Type	Any

The *MetA*WSL code for the applicability test would be as follows:

```
((Cond ((Or ([_Trans?_] Forward_Factor_Cond)
            ([_Trans?_] Forward_Factor_D_If)
            ([_Trans?_] Forward_Factor_While)
            ([_Trans?_] Take_Outside_Loop)
            ([_Trans?_] Forward_Factor_Loop_1)
            ([_Trans?_] Forward_Factor_Loop_2)
            ([_Trans?_] Forward_Factor_For)
            ([_Trans?_] Forward_Factor_Actions)
            ([_Trans?_] Forward_Factor_Where)
            ([_Trans?_] Forward_Factor_Var)
            (And ([_S_Type?_] Cond)
                 ([_Trans?_] Take_Call_Out_Of_Cond)))
         (@Pass))
       ((Else)
         (@Fail))))
```

In the last condition of the Cond statement, there is a test for the selected statement to be a Cond. This is because there is also a transformation called "Take_Call_Out_Of_Cond" that which works on a Call statement and which has a slightly different effect, and thus should not be included as a possibility in this transformation.

The *MetA*WSL code for performing the transformation would be as follows:

```
((Cond ((([_Trans?_] Forward_Factor_Cond)
         (@Trans Forward_Factor_Cond))
       ((([_Trans?_] Forward_Factor_D_If)
         (@Trans Forward_Factor_D_If))
       ((([_Trans?_] Forward_Factor_While)
         (@Trans Forward_Factor_While))
       ((([_Trans?_] Take_Outside_Loop)
         (@Trans Take_Outside_Loop))
       ((([_Trans?_] Forward_Factor_Loop_1)
         (@Trans Forward_Factor_Loop_1))
       ((([_Trans?_] Forward_Factor_Loop_2)
    (@Trans Forward_Factor_Loop_2))
((([_Trans?_] Forward_Factor_For)
    (@Trans Forward_Factor_For))
((([_Trans?_] Forward_Factor_Actions)
    (@Trans Forward_Factor_Actions))
((([_Trans?_] Forward_Factor_Where)
    (@Trans Forward_Factor_Where))
((([_Trans?_] Forward_Factor_Proc)
    (@Trans Forward_Factor_Proc))
((([_Trans?_] Forward_Factor_Var)
    (@Trans Forward_Factor_Var))
((And ([_S_Type?_] Cond)
        ([_Trans?_] Take_Call_Out_Of_Cond))
    (@Trans Take_Call_Out_Of_Cond))))
```

7.5.3 Summary

Generic transformations can be expressed in MetaWSL as lists of transformation applicability testing functions and statements for performing the corresponding transformations. Thus, provided it is known which transformations need to be included in the generic transformations, they are very easy to construct using MetaWSL.

7.6 High-Level Transformations

As was seen in Section 4.4.5, for the Maintainer's Assistant to be suitable for all forms of reverse engineering, its transformation catalogue needs to incorporate "transformations" for crossing levels of abstraction. In fact not all of these are transformations in the true sense of the word since they are not in general reversible; they are in effect "reverse refinements". The steps that are required in order to "transform" code to a more abstract specification were given in Section 4.4.5. To summarise, they are:

- Procedurisation and parameterisation;

- Recursion introduction;
- Determination of invariants;
- Introduction of specification statements;
- Introduction of specifications from assertions; and
- Data abstraction.

7.6.1 Method

Each of these types of transformation would be coded using the same techniques as for earlier transformations.

7.6.2 An Example

Due to the similarity in method to examples earlier in this chapter, just one example is given.

Demonstrating an Invariant

This transformation makes appropriate copies of an assertion, which is true before a While loop, which remains true after each iteration, and is therefore also true outside the end of the loop. (The While loop and the Assert statement are put inside a Cond statement, to take into account the case in which the loop is not executed.) For example,

```
(Assert (> X 10))
(While (<> A B)
        ...
        (Assign (X (+ X 1))))
```

can have assertions and a Cond statement inserted so as to rewrite is as

```
(Assert (> X 10))
(Cond ((<> A B)
         (While (<> A B)
                  (Assert (> X 10))
                  ...
                  (Assign (X (+ X 1)))
                  (Assert (> X 10)))
         (Assert (> X 10))))
```

The header information for this transformation is as follows:

Name	Insert_Invariant
Generic Type	Statement
Specific Type	While

The transformation's applicability condition is as follows:

```
((Cond ((Not ([_Left?_]))
             (@Fail))
        ((Else)
             (@Left)
             (Var ((Table ([_Match_] Statement
                                     (Assert (~>?~ B))
                                     Empty)))
                  (Cond ((Empty? Table)
                              (@Fail))
                          ((Else)
                              (@Right)
                              (@Mark)
                              (@Down)
                              (@Ins_After ([_Fill_In_] Statement
                                                       (Assert (~<?~ B))
                                                       Table))
                              (@Right)
                              (@Trans Duplicate_Assertion)
                              (Loop (Cond (([_Trans?_] Move_Assertion_Forward)
                                             (@Trans Move_Assertion_Forward))
                                           ((Else)
                                             (Exit 1))))
                              (@Up)
                              (@Trans Simplify_All_Expressions)
```

```
                    (Cond (([_Trans?_] Add_Invariant_After)
                               (@Pass))
                           ((Else)
                               (@Fail)))
                    (@Undo))))))
```

The applicability condition does some initial testing to ensure that there is a statement before the While loop and that it is an Assert statement. Having done that, the test marks the current version of the program, since the test changes it, and adds two copies of the assertion inside the body of the loop. The second of these it moves through the loop using the transformation "Move_Assertion_Forward". This transformation swaps an assertion with the statement following it, changing the assertion as necessary. For example it would replace the statements:

```
(Assert (<= X 1))
(Assign (X (+ X 1)))
```

by

```
(Assign (X (+ X 1)))
(Assert (<= X 2)).
```

Having moved the assertion as far through the loop as possible (which may be to the end) the transformation "Move_Assertion_Forward" is no longer valid so the remainder of the transformation test is executed. The applicability condition will have succeeded if, and only if, the transformation "Add_Invariant_After" is then valid. This transformation adds an Assert statement after a While loop, provided that the assertion is true at the beginning and end of the loop body as explicitly indicated by assertions in the While loop. The transformation also puts both the While and the Assert inside a Cond statement, to take into account the case in which the loop is not executed. For example, the statement:

```
(While (< A B)
       (Assert (< X 10))
       ...
       (Assert (= X 4)))
```

can be replaced by the statement:

```
(Cond ((< A B)
        (While (< A B)
                (Assert (< X 10))
                ...
                (Assert (= X 4)))
        (Assert (< X 10))))
```

Finally, the test undoes all the changes to the program using an @Undo statement.

The code for performing the transformation works on the same lines, except that it does not revert to the original version of the program at the end. Also, it needs to do less checking, since this can be deduced from the validity of the transformation's applicability test. The $\mathcal{META}$WSL code is as follows:

```
((@Left)
 (Var ((Table ([_Match_] Statement
                         (Assert (~>?~ B))
                         Empty)))
    (@Right)
    (@Down)
    (@Ins_After ([_Fill_In_] Statement
                             (Assert (~<?~ B))
                             Table))
    (@Right)
    (@Trans Duplicate_Assertion)
    (Loop (Cond (([_Trans?_] Move_Assertion_Forward)
                  (@Trans Move_Assertion_Forward))
                ((Else)
                  (Exit 1))))
    (@Up)
    (@Trans Simplify_All_Expressions)
    (@Trans Add_Invariant_After)))
```

7.6.3 Summary

The purpose of high-level transformations is to enable a user to extract a specification from a program by means of abstraction. During this process it is necessary to *add* information about the application domain and also to *remove* information which relates to the implementation alone. Although this work is at an early stage, it would appear from this example that $\mathcal{M}\varepsilon\tau\mathcal{A}$WSL provides all the facilities that would be required — although not all the actual transformations have been implemented — since the construction of the $\mathcal{M}\varepsilon\tau\mathcal{A}$WSL code for these transformations is very similar to that for compound transformations.

One enhancement to the system that would certainly be needed is an extension to the routines for performing symbolic mathematics, so that they are capable of inductively proving more complex invariants. An example of this would be proving the invariance of an assertion over a loop which contains a recursive call to the procedure containing the loop. To prove this invariance it would be necessary to *assume* that the assertion was preserved over the call and use this knowledge to complete the proof, inductively.

7.7 Analysis

A catalogue of 601 transformations has been implemented using $\mathcal{M}\varepsilon\tau\mathcal{A}$WSL. From the implementation, the following tables have been produced.

7.7.1 Number of Statements

The table in Figure 7.1 gives the number of transformation applicability tests (given in the "test" column) and the number of transformation "perform" routines that are represented using the indicated number of (Meta-)statements.

It can be seen that over half the transformations can be expressed in fewer than ten statements. This suggests that the transformations can be expressed *concisely* using $\mathcal{M}\varepsilon\tau\mathcal{A}$WSL.

Number of Statements	Number of Transformations	
	Test	Perform
1	66	24
2–5	185	201
6–10	136	118
11–20	120	126
21–30	53	51
>30	41	81

Figure 7.1: The Number of Statements

7.7.2 Number of MetaWSL Operations

The tables in Figures 7.2 and 7.3 give the number of transformation applicability tests and the number of transformation "perform" routines whose MetaWSL representations use the indicated number of pattern matching or template filling functions. (Applicability tests which perform template fill operations must necessarily undo the fill, using @Undo.)

Number of Pattern Matches	Number of Transformations	
	Test	Perform
0	387	314
1	129	120
2	47	67
3	12	43
4	14	13
5	4	10
>5	8	34

Figure 7.2: The Number of Pattern Matches

The tables in Figures 7.4 and 7.5 give the number of transformation applicability tests and the number of transformation "perform" routines whose MetaWSL representations use the indicated number database queries, first using functions concerned with variable usage and then using functions concerned with Loops and

Number of Template Fills	Number of Transformations	
	Test	Perform
0	531	259
1	40	134
2	18	81
3	4	31
4	3	38
5	3	13
>5	2	45

Figure 7.3: The Number of Template Fills

action systems.

Number of Variable Queries	Number of Transformations	
	Test	Perform
0	483	407
1	38	97
2	25	33
3	17	20
4	10	9
5	9	14
>5	19	21

Figure 7.4: The Number of Variable Queries

It can be seen that few transformations require more than two or three pattern matches, template fills or query operations in their representations. This, again, suggests that transformations can be expressed *concisely* using MetaWSL.

7.7.3 Spread of Usage of MetaWSL Constructs

The table in Figure 7.6 gives the number of transformation applicability tests and the number of transformation "perform" routines which use particular combina-

Number of Loop	Number of Transformations	
or Action Queries	Test	Perform
0	521	419
1	50	104
2	21	35
3	6	17
4	2	6
5	1	11
>5	0	9

Figure 7.5: The Number of Loop or Action Queries

tions of *MetA*WSL constructs:

- Pattern matching and template filling operations;
- Database query operations;
- Type testing operations; and
- Operations for calling other transformations.

A "■" indicates that the *MetA*WSL code uses the construct.

It has been observed that transformations are generally easier to understand if they do not mix different styles of *MetA*WSL construct. From the table it can be seen, for example, that over 90% of transformations do not mix pattern matching and data base operations. This provides further evidence that transformations can be expressed *clearly* using *MetA*WSL.

7.8 Conclusion

During the implementation of the catalogue of 601 transformations, no transformations have been found that cannot be expressed using *MetA*WSL. Indeed, all but

Type of Operation				Number of Transformations	
Pattern	Query	Type Test	Trans.	Test	Perform
■	■	■	■	14	29
■	■	■	□	33	15
■	■	□	■	3	6
■	□	■	■	11	28
□	■	■	■	10	9
■	■	□	□	10	6
■	□	■	□	38	48
■	□	□	■	9	31
□	■	■	□	25	4
□	■	□	■	5	4
□	□	■	■	64	41
■	□	□	□	96	124
□	■	□	□	13	0
□	□	■	□	88	6
□	□	□	■	68	135
□	□	□	□	114	115

Figure 7.6: The Spread of MetaWSL Constructs

those transformations which incorporate complex algoithms, have been represented in both a clear and concise manner. This is because MetaWSL provides not only the program editing statements and program analysis functions that are needed to construct simple transformations, but it also incorporates additional control statements and all the structures of WSL (which enable sophisticated control of the transformation process). These features, together with the mathematics and logic routines, mean that elementary, compound, generic and high-level transformations can be written without any difficulty.

The next chapter will outline the methods by which many of the important components of the system are implemented.

Chapter 8

Implementation of the Maintainer's Assistant

8.1 Introduction

This chapter considers some of the implementation issues involved with the construction of the Maintainer's Assistant. It addresses the method of constructing the tool, the system architecture and the work's contributions in terms of data structures and algorithms.

8.2 Approach to Building the Tool

The traditional "structured" methods of building a software system are the "**top down**" approach and the "**bottom up**" approach. The top down method starts with a high-level description of the system to be developed which is then refined into a structure expressed in terms of "big" operations. These operations are then re-expressed in terms of operations with simpler functionality, and the process is repeated until an implementation is obtained. Although this approach has been

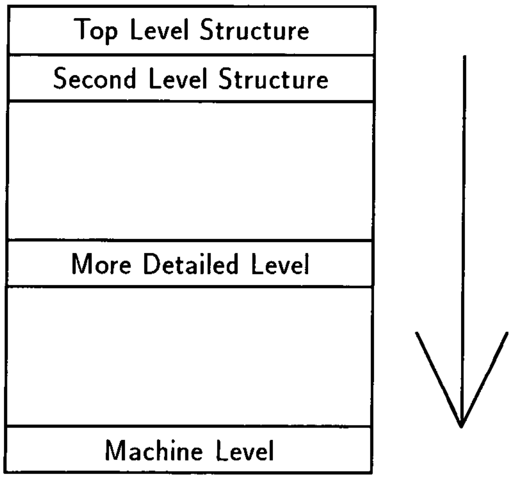


Figure 8.1: Top Down Development

used successfully, there are a number of problems [183]; in particular a problem which arises when building a prototype, such as the Maintainer's Assistant, is that there is only a vague top-level description. In addition, choosing the wrong structures in the early stages can have serious repercussions which will only be uncovered much later in the development.

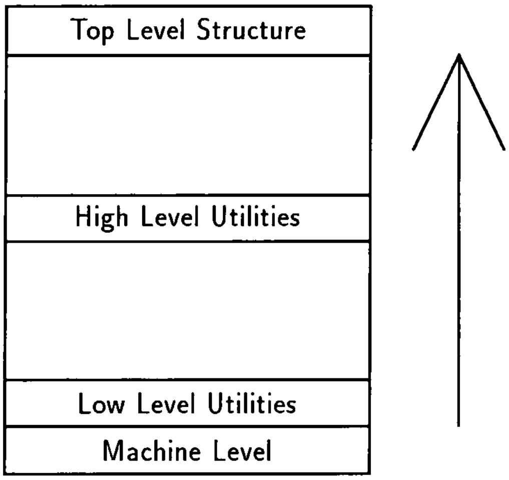


Figure 8.2: Bottom Up Development

A bottom up development starts by implementing the lowest-level, most general "utility" functions. From these, higher-level functions, routines and abstract data types are constructed. The process is repeated, creating increasingly domain-specific routines, until the top-level structure of the program can be implemented. The advantages of this approach include the ability to perform unit testing and the possibility that routines may be reusable. Among the problems [183] is the difficulty, especially in the middle stages of development, of determining what to build next in order to make progress. The high-level routines may do "too much" or "not enough", and they may not be required at all. Again, these problems are exacerbated when the application domain is new, as in the case of the Maintainer's Assistant.

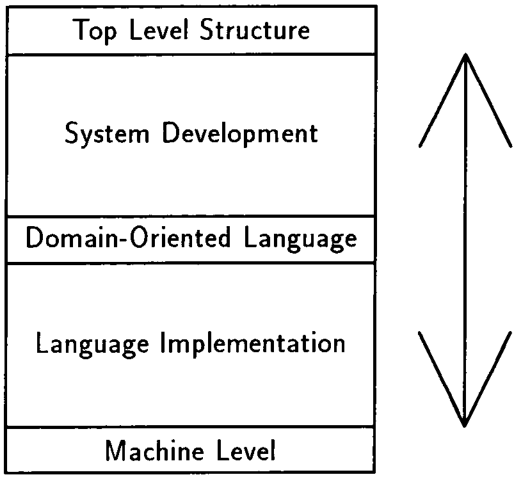


Figure 8.3: Middle Out Development

An alternative method of building a software system was adopted in the construction of the Maintainer's Assistant: "**middle out**" development [183], in which the middle layer forms the starting point. This middle layer takes the form of an abstract machine, specially designed to facilitate the implementation of the kind of software required. In the case of the Maintainer's Assistant, this middle layer takes the form of $\mathcal{META}$WSL.

In general, using a high-level language rather than writing everything in assembler has a number of advantages: the program requires an order of magnitude fewer lines, it is easier to understand and it is easier to change. Using a very-high-level, domain-oriented language such as $\mathcal{META}$WSL has given similar improvements over traditional languages. Moreover, the implementation of $\mathcal{META}$WSL and the implementation of transformations *using* $\mathcal{META}$WSL can be carried out independently.

During development, the Maintainer's Assistant was continuously tested on many small, example programs to determine whether the ideas that it embodied were practical and efficient. These examples were constructed (a) so as to incorporate artificially as much complexity in a small program as was possible, and (b) so that all the different WSL program constructs and transformations in the system were tested.[1] This resulted in information which was fed back into the design of $\mathcal{META}$WSL and thence into the design of the entire system — a process that was simplified by using a middle-out design.

[1] As these examples were created, and the transformations needed to be applied to them determined, this information was put into an executable file so as to provide a regression test for the system.

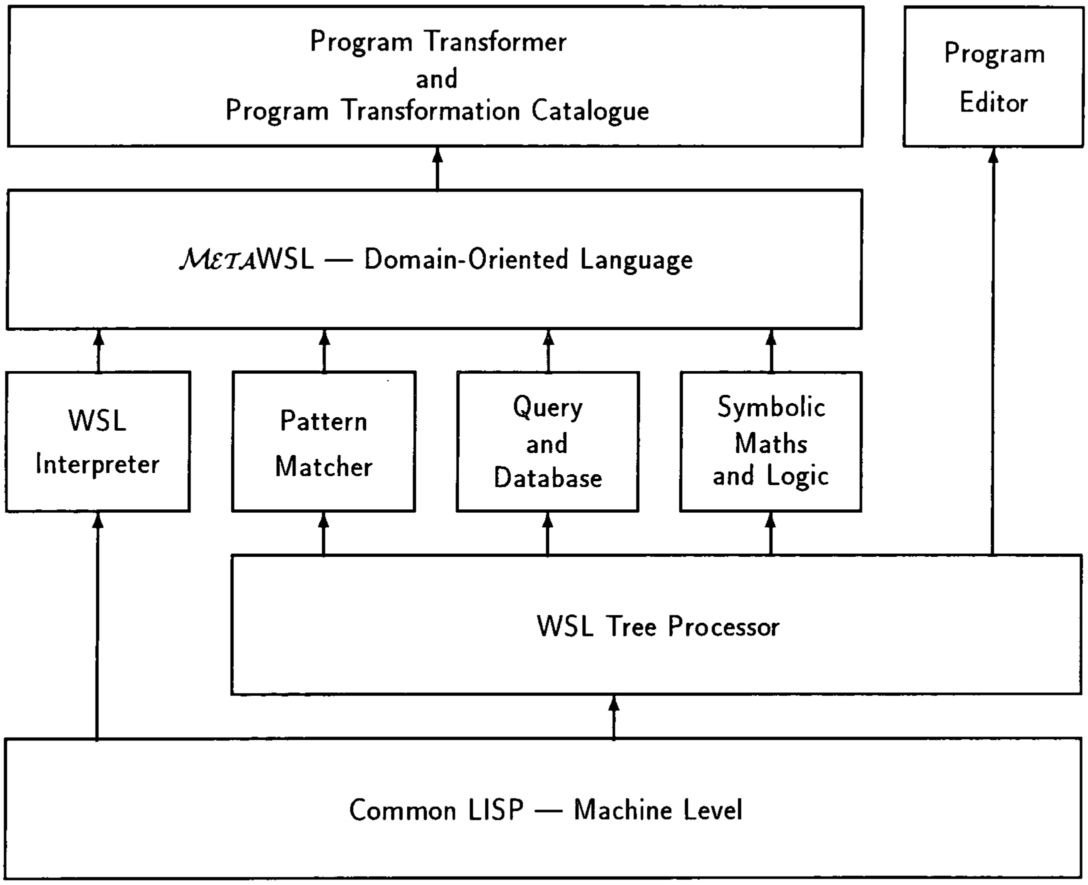


Figure 8.4: The Architecture of the Maintainer's Assistant

8.3 The System Architecture

The architecture of the Maintainer's Assistant, which is shown in Figure 8.4, reflects a middle out design and construction which starts with the definition of MetaWSL. Working upwards, this is used to build the transformations, and working downwards MetaWSL is composed of a number of simpler components. These are the WSL interpreter (since MetaWSL incorporates the whole of WSL), the pattern matcher and template filler, the query and database functions, and the symbolic mathematics and logic routines.

These components were also constructed in a middle out manner by starting with a middle layer which consisted of some elementary WSL tree-processing functions (since WSL programs are represented as trees). These functions proved *reusable* in the construction of a syntax-directed program editor.[2]

At the lowest level of the system, the tree processing functions were implemented in Common LISP. Although any language could have been used as at the lowest level, LISP has four important advantages which make it eminently suitable for constructing this kind of system:

1. LISP is a good language for manipulating tree structures. Programs written in WSL and transformed by the system are much easier to work with if they are represented as a *syntax tree*.
2. LISP works using implicit pointers to allow different data structures to share the same memory, provided that they have common data. This is important when it is necessary to update one data structure and have the corresponding ones updated in the same way since it allows the programmer to make the change only once.
3. LISP allows data structures to store executable code. Since the transformations consist of executable code (written in $\mathcal{META}$WSL) and are stored in a data structure, this facility is essential.
4. LISP is portable across different platforms.

LISP's chief disadvantage is its inefficiency; much time is spent garbage collecting.

The following sections will consider specific details of the implementation, looking in turn at each component.

8.3.1 The WSL Tree Processor

This component implements an abstract data type for WSL program trees (the exact form of which is given in Section 8.4.1). So as to hide the implementation

[2] It may sometimes be necessary to edit the program which is undergoing transformation in order to correct faults or to change the program's functionality.

Function	Parameters	Description
`Tables`	`Item`	Returns the database tables of the item.
`Comments`	`Item`	Returns the comments information of the item.
`Types`	`Item`	Returns the types information of the item.
`Leaf_Item?`	`Item`	Tests whether the item is a leaf node.
`Branch_Item?`	`Item`	Tests whether the item is a branch node.
`Size`	`Item`	Returns the size (number of components) of the item.
`Comps`	`Item`	Returns the components of the item.
`Gen_Type`	`Item`	Returns the generic type of the item.
`Specific_Type`	`Item`	Returns the specific type of the item.
`Leaf_Name`	`Item`	Returns the name (or value) of the leaf item.
`Max_Size`	`Type`	Returns the maximum number of components allowed in an item of the given type.
`Min_Size`	`Type`	Returns the minimum number of components allowed in an item of the given type.
`Type_N`	`Type, N`	Returns the type of the nth component of an item of the given type.
`Get_N`	`Item, N`	Returns the nth component of the item.
`Eq_Items`	`Item_1, Item_2`	Tests the two items for equality, ignoring tables and comments.

Figure 8.5: Basic Tree Examination Functions

of the abstract data type for WSL program trees, a number of operations are provided to access and update these trees. Thus, if the manner in which the extra information — types, database and comments information — is stored at each node is changed, then only the basic tree manipulation functions need to be updated. The functions for examining and comparing trees are given in Figure 8.5, the functions for building and altering trees are given in Figure 8.6 and the functions for working with tables and comments information are given in Figure 8.7.

Function	Parameters	Description
LISP->Int	Item, Type	Converts the item of the given type from LISP-like form to internal form.
Int->LISP	Item	Converts the item from internal form to LISP-like form.
Change_N	Item, N, New_Elt	Changes the nth component of the item to the given new element.
Delete_N	Item, N	Deletes the nth component of the item.
Insert_N	Item, N, New_Elt	Inserts before the nth component of the item, the given new element.
Splice_N	Item, N, New_Elts	Inserts before the nth component of the item, the given list of new elements.

Figure 8.6: Basic Tree Building Functions

Function	Parameters	Description
Set_Table!	Item, Table	Resets the database table of the item to the given value.
Add_To_Table!	Item, Key, Data	Stores a table entry, indexed by the key, for the item.
Get_From_Table	Item, Key	Returns the value, indexed by the key, in the table of a item.
Set_Comment!	Item, Comments	Resets the comments information of the item to the given value.
Edit_Comment!	Item, Key, Data	Stores a comment, indexed by the key, for the item.
Get_Comment	Item, Key	Returns the comment, indexed by the key, of a item.

Figure 8.7: Basic Database and Comment Functions

Higher Tree Manipulation Functions

The basic tree manipulation functions all take as an argument an item which represents a WSL program tree. From these functions, higher-level routines are constructed to operate on the *specific* program trees which represent the current program and currently selected item, and which are held in the global variables `%Program%` and `%Posn%`. These higher-level routines are, in fact, MetaWSL statements and functions in that they examine and edit the program being transformed. However, to facilitate their implementation and to improve the efficiency and effectiveness of the tool, a few intermediate functions have been constructed; these relate to movement within the program tree (see Section 8.4.2) and to the syntax checking of WSL programs.

8.3.2 The WSL Interpreter

Since MetaWSL incorporates the whole of WSL, it is necessary to include a method of executing WSL programs. Rather than building a translator to produce from WSL code in an existing language, or a WSL compiler, a WSL interpreter was used. Although less efficient in terms of program execution times, this has the advantage of being simple to implement and change; an important consideration when building a prototype.

The interpreter consists of a number of LISP functions and macros, each of which defines a WSL construct. An example — the definitions which implement unbounded Loops with Exit statements — is shown in Figure 8.8.

8.3.3 The Pattern Matcher and Template Filler

This section describes the mechanisms for pattern matching and template filling. A pattern, for either matching or for using as a template, could contain tokens for components that have been matched or which need to be filled. These components

```
(Defmacro Loop (&Rest Body)
  '(Let ((%Exit%
          (Catch 'Exit (Tagbody %Here% ,@Body (Go %Here%)))))
     (Or (= 1 %Exit%)
          (Exit (1- %Exit%)))))

(Defun Exit (N)
   (Throw 'Exit N))
```

Figure 8.8: The Implementation of Unbounded Loops

are stored in an association table, with one entry for each token. The entries consist of pairs, each pair containing the name of a token and the value with which it corresponds. An association table can be held as a single object in a WSL variable. There are two main advantages of storing the results of matches in this way:

- The result of a pattern match can be worked on as a single object, for example when it is passed to a template filling routine;
- Different matches can stores their results independently, even though they may have tokens with the same name.

Converting Patterns to Tree Form

In $\mathcal{METAWSL}$ programs, the WSL code and the patterns are expressed as executable LISP-form. However, there are a number of advantages of performing the pattern matching and template filling on the internal tree form.

Foremost among these is that the database tables and comments that are attached to the program components being matched are stored in the association tables along with the actual WSL.[3] This means that when a template fill is performed, the database tables and comments form part of the new piece of WSL. In the case

[3] See Section 8.4.1 for details of the internal form of WSL program trees.

of tables this is a benefit since the information does not need to be recalculated for this newly filled in code. It is also a benefit from the point of view of comments, since the user does want them to disappear as soon as he starts moving code around by transformation.

As a result of performing pattern matching using the tree form, the patterns need also to be converted into tree form, complete with their own tables and so on. This need not be done every time the pattern is used, however. Instead it is done when the transformation is loaded in the system.

8.3.4 The Query and Database Functions

The functions which access and update the database tables that are stored in WSL program trees form part of the WSL tree processor. The functions provided by this component are those that determine the information to be stored in the tables and are, in fact, functions provided as part of *MetA*WSL. They cover four particular areas:

1. Functions for examining the program undergoing transformed: `[_Statements_]`, `[_Calls_]` and `[_Total_Size_]` (see Appendix C.11);
2. Functions relating to variable usage: are: `[_Variables_]`, `[_Used_]`, `[_Assigned_]`, `[_Used_Only_]`, `[_Assd_Only_]` and `[_Assd_To_Self_]` (see Appendix C.12).
3. Functions relating to Loops: `[_Depth_]`, `[_Terminal_Value_]`, `[_Terminal?_]`, `[_Reducible?_]`, `[_Proper?_]`, `[_Improper?_]` and `[_Dummy?_]` (see Appendix C.14); and
4. Functions for testing action systems: `[_Regular?_]`, `[_Regular_System?_]` and `[_Calls_Terminal?_]` (see Appendix C.15).

Each function is implemented by recursively walking down the tree to determine the result for each node. If at any stage the result is already stored in a table, then the rest of that subtree need not be walked through. As the intermediate results for each node are calculated, they are stored in the corresponding database tables for later use. Currently, the system stores this information at *every* node, but for leaf nodes or small subtrees, it might be more efficient just to recalculate the results. The tradeoffs between storage and calculation could be determined empirically and a later system might only store database tables at, say, statement nodes.

8.3.5 The Symbolic Mathematics and Logic Routines

The primary mathematical routines are [_Simplify_], [_Isolate_], [_->T?_] and [_->F?_]. Foremost among these is [_Simplify_] which takes an expression or condition as returns an equivalent expression or condition that has been simplified as much as possible.

As part of the transformation process one may wish to rewrite the statements

```
(Cond ((= X 0) (Abort))) (Assign (X (+ X 1)))
```

as

```
(Assign (X (+ X 1))) (Cond ((= (- X 1) 0) (Abort))).
```

To do this, it is necessary to be able to rewrite an expression of the form (Expression1 = Expression2) so that a particular variable appears on its own on one side of the = sign. The function [_Isolate_] accomplishes this. For example, with the left expression as Y, the right expression as (+ X 1), then isolating X would return (- Y 1).

Two functions are needed for testing logical implication. Both of these take as their arguments two conditional expressions, an assertion and a test. The functions [_->T?_] is used to determine whether the assertion implies the logical truth of the test, and it returns true or false accordingly. For example, (= a 0) logically implies that (< a 5) is true, whereas (<> a 0) does not logically imply that (<>

a 1) is true. The function [_->F?_] is used to determine whether the assertion implies the logical falsehood of the test, and it returns true or false accordingly. For example, (> a 1) logically implies that (< a 0) is false, whereas (<> a 0) does not logically imply that (<> a 1) is false.

8.3.6 *MetA*WSL

*MetA*WSL contains the union of the functions provide by the WSL interpreter, the pattern matcher and template filler, the query and database functions, and the symbolic mathematics and logic routines. There are, in addition, some extra routines which provide the "glue" to hold everything together. These implement WSL statements such as @When which provide additional programming structures; @Pass and @Fail which provide control of the transformation process; [_Replace_] and [_Diff_] which provide extra functions on WSL program trees; and @Trans and [_Trans?_] which permit the combining of transformations.

All these routines are implemented in the same way as the WSL interpreter using LISP macro and function definitions, but they also make use of the lower-level functions so as not to need to refer explicitly to the implementation of the abstract data type for WSL program trees.

8.3.7 The Program Transformer

The program transformation catalogue consists of pieces of *MetA*WSL code stored in appropriate data structures, as was described in Chapter 5. The program transformer provides functions which extract the transformations from these data structures and execute the relevant pieces of code. The execution is performed by means of the LISP "eval" function; *MetA*WSL code maps into LISP code since the definitions of WSL and *MetA*WSL are in terms of LISP functions and macros.

8.3.8 The Program Editor

The program editor uses combinations of the functions provided by the WSL tree processor in order to construct functions for editing WSL program trees. Unlike the corresponding $\mathcal{META}$WSL statements, these functions *do* provide error checking so as to prevent the user from constructing syntactically invalid programs.

8.4 Contributions

This section looks at five areas in which this work has contributed to the study of transformation systems by it use of new-developed data structures or algorithms.

8.4.1 The Representation of WSL Program Trees

Transforming, editing or changing a program in any way involves manipulating the variable in which the program is stored, while moving through a program involves changing the variable which records the current position. These are the variables `%Program%` and `%Posn%`, respectively and their use is described in Appendix C.2. In addition, the currently selected item and other information relating to it is often required. Although this could be calculated from knowledge about the program and the current position, it is stored in a number of variables (see Appendix C.2) to save having to calculate it "on the fly" each time it is needed.

The Maintainer's Assistant represents both the WSL programs that are being transformed, and the selected item, as syntax trees. An example of a portion of the tree for a parallel Assign statement was given in Figure 5.2. There are, however, two forms of these trees. The first, the "internal" form, stores at each node additional information, such as its database table, which is used when transforming the program. The second, the "LISP-like" form, omits the extra information so that programs in this form can be executed via a number of macro and function definitions. All the programs being transformed are stored using an abstract data type which implements the first form. The information held at each node is grouped into various categories as follows.

Types

In order for the system to function, it is necessary to record the specific syntactic type of each node, such as Assign or Assignment, at the node. The generic type of the node is also stored for efficiency. For an Assign node, this would be Statement, whereas for an Assignment node, it would also be Assignment. For leaf nodes this information may be supplemented by a value. Examples are shown in Figure 8.9.

```
(Assign Statement)
(Assignment Assignment)
(Number Expression 0)
```

Figure 8.9: Type Pairs

Database Information

Working with database information requires that each node hold an arbitrary number of data pairs, each pair consisting of a query and its result. These pairs are held in a list with an identifier ("--") as its head to indicate that it is a database.

The database update functions modify the program structures destructively without creating new copies of the structures. Thus the "history" and "future" versions of the program are updated with the same information since, where the node is shared between them, the database results are still valid, and so it is unnecessary to recalculate them. Also if the same node occurs several times in the *current* program version — through the

(--
Id_Number
(Query1 Result1)
(Query2 Result2)
⋮
(Query$_n$ Result$_n$))

Figure 8.10: Queries and Id Numbers

use of transformations which have copied it — then the result will also be valid at those nodes too. This saves time-consuming recalculation of the results.

Item Identification Number

In order to enhance communication with the user interface — in particular to prevent the same piece of code from being passed to the interface several times — each node is given an identification number. This is initially zero, but once the node has been passed to the interface, it is set to a unique, non-zero value. The identification number gets changed in a similar way to the database tables when an item's sub-components are changed. Thus the two pieces of information share the same data structure. The representation is of the form shown in Figure 8.10.

Comments

(| |

(Category$_1$ "Text$_1$")

(Category$_2$ "Text$_2$")

⋮

(Category$_n$ "Text$_n$"))

Figure 8.11: Comments

A helpful feature of the system is the ability to link a comment to each node. Thus if the node moved to a different place in the program (due to it being transformed) the comment would move with it.

A representation that allows for different categories of comment is more flexible and is the one that was adopted. It is implemented in a similar way to the database structure, but stored separately and with a different identifier ("| |"), as shown in Figure 8.11.

This extra information is stored at each node in the order: database tables, comments, types, and is followed by all the components of the node. Thus a node might be of the form shown in Figure 8.12 where the dots indicate that there is a sequence of components, each of the type Assignment.

```
( (__ 0 (Variables X Y A B))
  (|| (Label "An Example"))
  (Assign Statement)
          ⋮
  )
```

Figure 8.12: An Example WSL Program Node

8.4.2 Movement within the Program Tree

Each component in the program tree has a position relative to the root of the tree which is represented as a list of positive integers as described in Chapter 6.

When moving through the program tree, not only is it necessary to update this position — stored in the variable %Posn% — it is also necessary to update the other predefined global variables (see Appendix C.2). In order to do this efficiently another global variable, %P_Data%, is used.

It is always possible to set the values of the global variables described above by starting at the root of the tree and walking downwards through it to the current position. However, this is time consuming, especially when a great deal of movement is to positions adjacent to, within, or containing, the current position. Thus, rather than recalculating the values by walking through the whole tree, all the values of the variables at each stage in the walk through the tree are recorded in %P_Data% and reused.

%P_Data% is a list items, one for each position in the tree that has been walked through, up to and including the actual current position. Each item in %P_Data% is itself a list consisting of the values of certain variables for that position, including, for example, the context variables.

Two functions are needed for moving through trees. The first, All_New_Position_Data, is used after the program has been changed, or the selection has been moved to a completely new point in the program. In this case, it is *not* possible to make use of values of these global variables higher up in the program tree (which are recorded in %P_Data%). The second function, Set_Position_Data, is used after the current position has been moved to one near to the previous position, for example with one of the *MetaWSL* statements @Up, @Down, @Down_Last, @Left, @Right or @To.

The second of these functions takes a parameter which indicates the "direction" of the movement:

0 — Movement downwards through the program tree;

1 — Movement left or right an arbitrary number of positions within the current parent component; and

2 — Movement up one level in the program tree.

The "movement" argument allows the function to strip out[4] any information from the beginning of %P_Data% that cannot be used to regenerate the values of the global variables, thus ensuring that the first item of %P_Data% is the lowest node in the tree which is common to both the old program position and the new program position.

Starting with the position relating to the first item of %P_Data% there is a subsidiary function which steps through the tree until the selected position is reached. For each step, the values of the global variables that the system needs are recalculated and put, together with their position information, as a new element, into %P_Data%.

8.4.3 Global or Local Scope Transformations

Testing the applicability of transformations can be a time consuming process. Thus, if the same transformation needs to be tested for applicability at the same

[4] This can be achieved by setting %P_Data% to be equal to (Nthcdr Direction %P_Data%)

point in the program a number of times, this could become very slow. The chosen solution to this problem is to store in the database table associated with the item, the name of the transformation and the result of its applicability test, then when the transformation's applicability is tested on subsequent occasions, provided the components within the item have not changed (in which case the database table will have been emptied anyway), the result of the test can be obtained simply by looking in the database table.

However, not all transformations rely solely on the form of the current item for their applicability test. Many of them also rely on adjacent components, for example. Thus the results of transformation applicability tests are only stored in a database table if their respective tests relies *solely* on the current item. In order to distinguish between transformation whose applicability can be stored and those for which it cannot, there is a flag associated with each transformation. This is either "Local" if the result can be stored or "Global" if it cannot.

8.4.4 The Symbolic Simplifier

All the symbolic mathematics and logic routines use the generic symbolic simplification function, `[_Simplify_]` which takes an expression or condition and simplifies it as much as possible. If the expression is a number or simple variable then no simplification can be performed, so the expression is returned. If the expression is a compound expression (for example a "+") the arguments of the expression are simplified by recursive calls to `[_Simplify_]` and are then passed to a function for specifically simplifying additions. (There are other functions for simplifying other types of expression and condition.)

The specific simplification functions look at the form of their arguments for common patterns that can be simplified. For example, the function for simplifying additions would contain the rules listed in Figure 8.13, among others: (Here "X", "Y" and "Z" represent any expressions, while "A" and "B" represent numbers. In the results column, where expressions such as "(A+B)" occur — i.e. where both arguments are numeric — these are taken to have been evaluated.)

First Argument	Second Argument	Result
A	B	(A+B)
X	0	X
X	X	(2*X)
X	-X	0
(X+A)	B	(X+(A+B))
(X+Y)	X	((2*X)+Y)
(X+Y)	Y	((2*Y)+X)
(X-A)	B	(X+(B-A))
(A-X)	B	((A+B)-X)
(X-Y)	X	((2*X)-Y)
(X-Y)	Y	X
(X+A)	(Y+B)	((X+Y)+(A+B))
(X+Y)	(X+Z)	((2*X)+(Y+Z))
(X+Y)	(Z+X)	((2*X)+(Y+Z))
(X+Y)	(Y+Z)	((2*Y)+(X+Z))
(X+Y)	(Z+Y)	((2*Y)+(X+Z))

Figure 8.13: Some Simplifications of Additions

8.4.5 Communication with the Interface

This thesis only describes the LISP "engine" which lies behind the Maintainer's Assistant but there is also a user interface which displays the WSL code in a suitable pretty-printed form. These two processes need to be able to communicate, and this section describes the mechanism by which this is accomplished.

The interface is the only way that a user is able to communicate with the LISP-based transformation system. The interface converts the user's actions into LISP expressions which are piped, just as if they had been typed, into a child process that is running LISP.

How LISP Returns Results to the Interface

The LISP system is not able to display its output directly. Instead it produces results of expression evaluation in the conventional way. However, each result is prefixed by one of several key sequences of characters. The interface process examines all the output from the LISP process and, on identifying such a sequence of characters, uses the next piece of output from the LISP process as input for the display routines. These display routines include operations for displaying the WSL program and for building menus of transformations.

The most important form of communication between the LISP system and the interface is the passing of WSL programs. The interface does not need to know the details of the internal database tables, although it does need the comments information in order to display them. Also, each time the interface displays the program, most of it will remain unchanged. Thus only those parts of the program which have changed are passed to the interface.

In order to achieve this enhancement in the efficiency of communication, each node in the program tree is assigned an identity number. Initially this number is zero. The first time a node is passed to the interface this zero is changed to a new, unique, number. All subnodes are similarly passed in this way.

However, if a node is reached which has a non-zero identity number, then that node must already have been passed to the interface — which would have stored it in its own table. Thus, just the number is passed and not the contents and sub-components of the node.

Finally, when a node in the tree is modified, either by transformation or by editing, that node has its identity number reset to zero. Not only that, but all the nodes above it in the tree will have changed — they have a new sub-component, sub-sub-component, and so on — so these nodes must also have their identity numbers reset to zero.

In addition to the identity number system, each type is given a number (which can be determined from the syntax table in Appendix B) so that this can be passed in preference to passing the name of the type. Actual values of leaf items, such as numbers, strings and variables, still need to be passed explicitly.

The overall form of a program when passed to the interface is as a LISP tree in which each node is represented as a list containing (a) it's unique identification number, (b) a list of any attached comments with their types, (c) a number which represents the type of the item, and (d) either the node's value or its components. For example, if the program

```
((Assign (X (+ A B)) (Y 0)))
```

(which contains no comment information) had not already been passed to the interface, then it should be represented as shown in figure 8.14. The "Ipp..."

```
(Ipp...  (1 () 17
               (2 () 33
                     (3 () 8 (4 () 27 X)
                             (5 () 51 (6 () 26 A)
                                      (7 () 26 B)))
                     (8 () 8 (9 () 27 Y)
                             (10 () 24 0))
               )
         ))
```

Figure 8.14: A Program as Passed to the Interface — Version 1

is a key sequence of characters which tells the interface that the following LISP output is to be regarded as a program tree.

If the interface were to request that this information be sent again, then the LISP system would only pass the information shown in Figure 8.15. However, if the zero in the second assignment had been edited, changing it to a one, while all the rest of the program had remained the same, then the LISP system would pass in information shown in Figure 8.16.

```
(Ipp...  (1))
```

Figure 8.15: A Program as Passed to the Interface — Version 2

In this case that items "3" and "9" do not get passed again — only their identity

```
(Ipp...  (11 () 17
                (12 () 33
                       (3)
                       (13 () 8 (9)
                                (14 () 25 1))
                )
         ))
```

Figure 8.16: A Program as Passed to the Interface — Version 3

numbers get passed — but all the other items do get passed with new identity numbers, since changing the leaf node has to be reflected at all the higher levels in the tree.

8.5 Summary and Conclusions

The Maintainer's Assistant consists of a user interface and a transformation engine. The latter, which is the subject of this thesis, incorporates a number of advances over the implementation of other transformation systems.

First, the system was developed in a middle out manner and, as a result, is structured as a series of abstract machines with well-defined interfaces. This enhanced the development of the tool by rapid prototyping, which in turn meant that a variety of data structures and algorithms could be tried so as to find efficient ones. The components that make up the system are the low-level WSL tree processor, the $\mathcal{META}$WSL language, the program transformer and the program editor. $\mathcal{META}$WSL is, in turn, composed of the WSL interpreter, the pattern matcher and template filler, the query and database functions, and the symbolic mathematics and logic routines. Thus, the implementation is modular and has all the advantages, including maintainability, which were given in Section 1.3.2.

Second, the implementation of the Maintainer's Assistant incorporates certain data structures and algorithms which have been developed particularly for this work. These cover the areas of WSL program representations, efficient movement within program trees, the dichotomy of global and local transformations, symbolic simplification and efficient communication with the interface.

The whole system is implemented using Common LISP, and usage suggests that the implementation is fairly efficient.[5]

[5] Precise figures are given in Section 9.6.3.

Chapter 9

Results

9.1 Introduction

The previous chapters of this thesis explained the reasons for creating a transformation system for software maintenance, highlighted some of the important design decisions that had to be made and gave details of the implementation of the Maintainer's Assistant — the tool which embodies these ideas. This chapter presents results from the Maintainer's Assistant applied to various example programs ranging from simple examples to large programs taken from the real world. In doing this, it considers the more pragmatic questions, which can be summarised as:

- Does this approach result in a usable tool? i.e. how much training is required, is the interface easy to understand and does the tool respond promptly to user requests?
- Is the implementation of the transformation catalogue efficient (in that the algorithms employed are at worst polynomial in time), reliable (in that errors are found increasingly infrequently), correct (in that the implementation of the system has been proved) and complete (in that all possible transforma-

tions could be built from those that have been included)?

- Does the method scale up to larger programs? i.e. are the transformations as applicable to large programs as to small? Does the system remain "fast enough" with large programs? Is it possible for a user to view and comprehend a large program?

- What weaknesses does the system have?

9.2 Applying the Tool to Real Programs

In addition to the simple test examples, the Maintainer's Assistant has been used on a number of small published programs. For example, Ward [177] shows the transformation of a WSL program that was transcribed from DataFlex and then transformed so as to reveal a potential fault that was not readily apparent in the original version. This example has been successfully performed using the Maintainer's Assistant.

9.3 Applying the Tool to Larger Programs

In order to tackle programs of more than a few hundred lines it is necessary to have a strategy, or method, of using the Maintainer's Assistant. Such a strategy, which was described in section 4.4, will now be assessed.

Since the Maintainer's Assistant has been produced as part of the ReForm project which has been partially funded by IBM UK Laboratories Limited part of the project has involved taking source code written in IBM 370 Assembler and using the system on it. This has provided more than twenty large real-world examples which have helped to assess the system's power and ease of use. Assembler programs will, therefore, be used to illustrate the following sections, but the four-stage process that will be described would apply to programs written originally in any language.

In addition, IBM Hursley have conducted two case studies on the use of the tool to assess its potential in a commercial environment.

9.3.1 Translation into WSL

Simple translators have been written for BASIC and and PASCAL. However the only complete translator is for 370 Assembler.[1] Translation is largely automatic and very simple-minded. Each instruction of assembly language maps into one or more statements of WSL. The WSL has to *model* every functional[2] aspect of the instruction, such as the setting of flags, even if these extra aspects are not needed in a particular situation. In this way, it is possible to check informally the correctness of the translation of each instruction. (It is *not* possible to check the translation *formally* since there is no formal description of IBM 370 Assembler.) For example the assembler instructions:

```
TM  VALUE1,XYZ  TEST FOR CHECK
BO  DONE1       LEAVE CLEAR
```

are translated into the WSL code:

```
(Comment "TEST FOR CHECK")
(Cond ((= (And_Bit XYZ (Aref A VALUE1)) 0)
        (Assign (CC 0)))
      ((= (And_Bit XYZ (Aref A VALUE1)) XYZ)
        (Assign (CC 3)))
      ((Else)
        (Assign (CC 1))))
(Comment "LEAVE CLEAR ")
(Cond ((And (= CC 0) (= (And_Bit 1 8) 8))
        (Call DONE1 0))
     ((And (= CC 1) (= (And_Bit 1 4) 4))
        (Call DONE1 0))
     ((And (= CC 2) (= (And_Bit 1 2) 2))
        (Call DONE1 0))
     ((And (And (<> CC 0) (<> CC 1)) (And (<> CC 2) (= (And_Bit 1 1) 1)))
```

[1] This translator has been developed as part of the ReForm project by Scriven.

[2] Timing is not modeled.

```
        (Call DONE1 0)))
```

Here the condition inside the local variable structure is used set the condition code — represented by the variable CC — according to the result of a test. The second condition tests the condition code and jumps depending on its value. It is clear from this example, that for the translation to preserve *all* the semantics of the assembler program, a great deal of extra WSL code may be required. Methods have been developed by Scriven for modeling labels and branches in terms of action systems and, in particular, branches to addresses stored in registers which represent subroutines.

There are currently two classes of unsolved problems in the modelling of other programming languages in WSL. The first class of problems are those which are fundamentally solvable by means of extending WSL's *syntax* or by mapping the source language in more complex fashions, since the WSL kernel language can be used to represent the semantics of these language features. This class includes the modelling of pointers (as in C) and overlapping data areas (as in COBOL). The second class consists of those problems that the WSL kernel is unable to express and which would, therefore, require fundamental extensions to the system. These problems include self-modifying code, arbitrary indirect branches and parallelism.

9.3.2 Automatic Removal of Idiosyncrasies

Once the program has been translated into WSL, the Maintainer's Assistant can be used to simplify it by removing all the extra, unnecessary, code that was introduced through translation. In the case of assembler, this would involve removing references to the condition code and replacing the array of registers by individual register variables (if possible).

In order to perform this type of simplification, powerful compound transformations have been constructed. Such a transformation[3] is "Fix_Assembler" which would,

[3] The techniques employed by this transformation are similar to the peephole optimisation techniques used by compilers [3]. However, the transformations in the Maintainer's Assistant

for example, simplify the WSL code above to:

```
(Comment "TEST FOR CHECK")
(Comment "LEAVE CLEAR ")
(Cond ((And (= XYZ (And_Bit XYZ (Aref A VALUE1))) (<> XYZ 0))
        (Call Done1 0)))
```

When "Fix_Assembler" is performed on a WSL program[4] it reduces its size considerably. (See the figures in Section 9.6.1.) Once the program is in a simpler form, the maintainer can then work on it, simplifying and restructuring it.

9.3.3 Manual Transformation

The traditional automatic tools for control flow restructuring do not provide a good basis for subsequent abstraction transformations. The shortcoming of such tools have been pointed out by Calliss [52]:

1. They may replace complex control flow with complex data flow, by introducing flag variables;
2. They may result in programs that are considerably larger; and
3. They do not help human understanding of the system.

The previous, automatic, stage did not attempt to *restructure* the program, only to *simplify* it within the same structure. For restructuring of a program in the Maintainer's Assistant, interaction is used.

are formally proven, whereas the optimisations used by compilers are usually informal.

[4] The program must have been been translated from assembler for "Fix_Assembler" to perform *useful* modifications to it.

Avoiding the Introduction of Flag Variables

When restructuring control flow, the system introduces no flag variables unless explicitly instructed to do so by the user. Extended use of the Maintainer's Assistant shows that for many programs the use of flag variables can be avoided by the judicious combination of conditions and loops, and by the creation of suitably-parameterised procedures. However, this is not always the case with programs which have been translated from assembler and which have multiple exit points. When restructuring such programs into a hierarchy of procedures, if an exit were to occur within a nested procedure, then a flag would be needed to pass this information to the outer procedures. This is the problem of **exception handling**. It is clear from these case studies that the problem lies not with the transformation system *per se*, but with the fact that WSL is not an ideal language for representing exception handling; the introduction of exceptions obscures the normal control flow of the program. For this drawback to be overcome, WSL would need extending with the appropriate constructs.

Avoiding an Increase in Size

Most of the transformations applied with the Maintainer's Assistant reduce the size of the program to which they are applied. There are, however, transformation which may *increase* the size of the program.[5] Use of the Maintainer's Assistant indicate that very often this is a temporary measure which facilitates some further restructuring later. For example, the user must expand all occurrences of a procedure, making the program larger, before he is able to construct a new procedural decomposition, reducing its size again. If at any stage it is not possible to reduce the size of the program, the user can undo the transformations which caused the increase in size, thereby giving much more control over the final size of the program than he would have with a purely automatic system.

[5] Sometimes a large program with a simple structure is easier to understand than a small one with complex structure.

Increasing User Understanding

A user who is unfamiliar with a program can nevertheless use the Maintainer's Assistant to perform transformations on it. This enables the user to obtain different versions, or views, of the program and in so doing this helps him in his understanding of it. Furthermore, the fact of knowing which transformations are applicable and which are not provides the user with information about the code. For example, knowing that the transformation to delete a section of code is applicable indicates that it is unreachable, whereas knowing that the transformation is not applicable indicates that the code is reachable.

Transformation to Aid Abstraction

Thus, the program can be transformed to an appropriate starting point for subsequent abstraction transformations. For example, the Maintainer's Assistant helps with the identification of suitable code sections to fold into procedures; typically, a large monolithic unstructured program can readily be transformed into a short main block from which calls to a set of sub-procedures are made. No known automatic restructurers can achieve this but it *is* an important intermediate step in identifying abstract data types. (There is an example in Section 9.6.1 in which an initially unstructured program is transformed into 39 hierarchically organised procedures.)

During the manual restructuring stage a number of "rules of thumb" have been identified as being helpful in guiding the transformation process.

1. General (i.e. side-effecting) expressions and conditions should be replaced by sequences of statements followed by the relevant expression or condition.
2. Action systems, especially those which embody complex control flow, should be collapsed, i.e. replaced by a series of nested loops and conditions.
3. If any conditions can be simplified to True or False, then this should be done.
4. If any section of code is redundant, then it should be deleted.

5. If there are procedures which are only called once, then they should probably be expanded and removed.

6. If there are local variables that can be removed without making the program larger, then they should be removed.

7. If there are two (or more) identical sections of code then they should be replaced by a single copy either by restructuring the program so that they can be merged or, if this is not possible or would greatly increase the size of the program, by making them into a procedure.

8. If there are Loops with multiple exits then the Exit statements should be merged where this is possible and does not increase the size of the program.

As would be expected, there are exceptions to all these rules and other rules have also been found to be useful in some cases, for example, replacing Loops by While loops. At some future stage, these heuristics could be incorporated into a knowledge base which would provide advice on the selection of transformations.

Thus, interactively applied transformations do avoid the problems described at the beginning of Section 9.3.3.

9.3.4 Abstraction to a Specification

Producing a specification involves two complementary approaches: information removal and information introduction. The information that is removed relates to the implementation of the program: variables names, concrete data structures and algorithms. The information that is introduced imposes *meaning* which relates to the application domain. For example, a program in which a set of variables may always sum to zero may represent the forces on a body in equilibrium or that debits and credits cancel one another in an accounting system. Knowledge of the domain is required to determine which is the case. Human expertise of the application domain and software engineering is required to provide a strategy in both these stages to decide what information can be safely discarded and also what new information (which was originally lost in moving from a design to an implementation) needs to be introduced.

The methods for abstracting a structured program to a specification have yet to be examined in any more detail in this thesis. However, as was described in Chapter 4, some important techniques have been identified:

- Procedurisation and parameterisation;
- Recursion introduction;
- Invariant determination;
- The introduction of specification statements; and
- The production of specifications from assertions.

These techniques have been used in simple examples; however they have yet to be applied to real-world programs.

9.4 Case Study

The following program is a short example to demonstrate some of the techniques described in Sections 4.4.5. While work on functional abstraction is still at an early stage, this example shows that, at least for some programs, it is possible within the Maintainer's Assistant.

```
((Var ((N N0))
    (Assign (K 0))
    (While (= (Mod N 2) 0)
          (Assign (K (+ K 1)))
          (Assign (N (/ N 2))))))
```

By looking at the loop, the user guesses at a suitable loop invariant and asks the system to insert that invariant before the beginning of the loop. The system proves the assertion's validity and inserts it. (%N represents the set of natural numbers.)

```
((Var ((N N0))
    (Assign (K 0))
    (Assert (And (= N0 (* N (** 2 K))) (Member? K %N)))
    (While (= (Mod N 2) 0)
          (Assign (K (+ K 1)))
          (Assign (N (/ N 2))))))
```

The user requests the system to demonstrate that the newly inserted assertion is a loop invariant, which it does in about two seconds. It then inserts new assertions at the beginning and at the end of the body of the loop, and also inserts the invariant after the end of the loop but it also forms the conjunction of it with the loop's terminating condition.

```
((Var ((N N0))
    (Assign (K 0))
    (Assert (And (= N0 (* N (** 2 K))) (Member? K %N)))
    (While (Even? N)
          (Assert (And (= N0 (* N (** 2 K))) (Member? K %N)))
          (Assign (K (+ 1 K)))
          (Assign (N (/ N 2)))
          (Assert (And (= N0 (* N (** 2 K))) (Member? (- K 1) %N))))
    (Assert (And (Odd? N)
                 (= N0 (* N (** 2 K)))
                 (Member? K %N)))))
```

So far no abstraction has been performed. However, at this stage the user selects to replace the initial assignment to K, the loop and the assertion after the loop by a specification statement. The system checks that all the variables which are assigned in the section of code are also referred to in the assertion, requests that the user enter some text explaining the abstraction, and performs the change to the code[6].

```
((Var ((N N0))
    (Assn_Spec (K N)
```

[6] The statement Assign X Such That C is written in LISP form as: (Assn_Spec (X) (C))

```
(And (Odd? N)
     (= N0 (* N (** 2 K)))
     (Member? K %N)))))
```

Finally, the user replaces the local variable structure which contains the specification statement by another specification statement.

```
((Assn_Spec (K) (Exists (N) (And (Odd? N)
                                 (= N0 (* N (** 2 K)))
                                 (Member? K %N)))))
```

This is the specification of the whole program. It calculates the number of trailing zeros in the binary representation of an integer.

9.5 IBM Hursley's Evaluation of the Maintainer's Assistant

IBM Hursley's evaluation[7] of the tool involved having several people in the organisation translate real assembler modules into WSL, and then work on them with the tool to improve the code and their understanding of it. In addition, demonstrations of the tool were organised so as to provide further feedback. To bring some structure to the evaluation, IBM's CUPRIMD (Capability, Usability, Performance, Reliability, Installability, Maintainability and Documentation) assessment categories were used.

9.5.1 Capability

The system was tried on a number of modules, of a few thousand lines each, which have a reputation of being poorly structured and difficult to maintain. After being

[7] This is not necessarily IBM's full position on reverse engineering tools since other approaches may be under development within the company.

transformed with the tool, these modules were hand-translated back to assembler. The resulting code was about 10% shorter and the rate of coding into assembler was "much" faster than it had been previously. However, the resulting assembler, while it passed some of the simpler regression tests, failed the more stringent ones. Although not all the reasons for this were established, some of the failures were due to errors in the hand-translation process. None were shown to be due to problems with the Maintainer's Assistant, although this could not be ruled out.

From the experience with the system, it was clear that when used on large tracts of "spaghetti code" it offers "great benefits" in terms of resulting code quality and programmer productivity.

9.5.2 Usability

The tool was found to be easy to use by maintainers with previous knowledge of the code being maintained, but without previous knowledge of program transformations. This reflects the inherent interactive nature of the tool.

9.5.3 Performance

Running on an RS/6000 (Model 320) under AIX 3.1 the performance was found to be "good" for a single user with many actions being "immediate". The response time was "a little longer" for some of the more complex transformations, but increased significantly (to more than an hour) when certain complex transformations were used on programs of a few thousand lines. However, these operations, such as "Collapse_Action_System" to remove "spaghetti code", tend only to be performed only once on each program. Thus, if viewed as a batch process, before the use of manually selected transformations, the times become more tolerable.

As more users attempted to use the machine simultaneously, response times also dropped.

9.5.4 Reliability

An early version of the tool was found to contain a number of faults, but the latest version was much better and the transformation engine in particular was found to be "very stable".

9.5.5 Installability

The Maintainer's Assistant was found to be easy to install, requiring no user intervention apart from starting the (two hour[8]) process.

9.5.6 Maintainability

IBM performed no maintenance on the tool since this was undertaken by the Durham members of the ReForm project.

9.5.7 Documentation

The online documentation was thought to be good, and the hard-copy documentation was thought to be very good.

9.5.8 Conclusion

As a result of their evaluation, IBM concluded that "the concept behind ReForm works", i.e. that transformation-based software maintenance "offers the first viable opportunity to renovate old code in a cost effective way". The Maintainer's

[8] This includes the time required for compilation.

Assistant was found to be easy to use and for some developers and maintainers was "just what they are waiting for".

In particular, the following areas were identified as those for which the Maintainer's Assistant offers the greatest potential benefits:

- Code re-structuring and optimisation;
- Testing and "debugging";
- Code validation;
- Quality assurance; and
- Software reuse.

9.6 An Assessment of Success

The Maintainer's Assistant has been used on a wide selection of programs (from a few lines to a few thousand lines, and with both simple and complex control structures) and by a number of users; not only the author, but other members of the ReForm project at the University of Durham and at IBM Hursley. Based on these experiences a critical assessment of the work can be obtained.

9.6.1 Is Maintenance by Transformation Plausible?

Experience, particularly at IBM, shows that in a typical situation in which a maintainer, initially unfamiliar with a particular module, uses the Maintainer's Assistant then significant improvements can usually be made to that module. These improvements are in the areas of program structure — which can be measured by the metrics facility built into the system — and comprehensibility. Comprehensibility, being a human factor, can only be measured indirectly; however, response from users is favourable. The final transformed (but not abstracted) versions of program have been shown to the maintainers working on the systems transformed, and were found to provide a clear overview of the structure and function of the software.

During the stages described, the program being transformed changes in both size and complexity. The figures in Figure 9.1 are typical of how the program characteristics change during these processes. The "McCabe" figure measures McCabes cyclometric complexity, the "structural" figure is a measure of the structural complexity devised by Yang [194] and "size" is the number of nodes in the program tree. In this example, which was originally 3,107 lines of assembler (including comment lines), the final version of the program consisted of 39 hierarchically organised procedures.

Stage	Lines	McCabe	Structural	Size
After Translation	2,330	1,030	48,175	24,736
After Initial Simplification	1,381	245	17,021	8,404
After Manual Transformation	1,227	156	11,990	7,120

Figure 9.1: The Effect of Transformation on Program Metrics

Figure 9.2 shows how one of the metrics changes as a small, but complex, program is transformed. From the figure it can be seen that generally the size decreases as transformations are applied, but that there are occasions when the size needs to increase to allow further transformation to take place. In this example, the increase in size was due to the insertion of Assert statements. The other size and complexity metrics change according to a similar pattern, the general form of which is shown in Figure 9.3. The longest program module on which the system has currently been used on is over 20,000 lines of WSL.

All the indications are that software maintenance by transformation is a plausible approach.

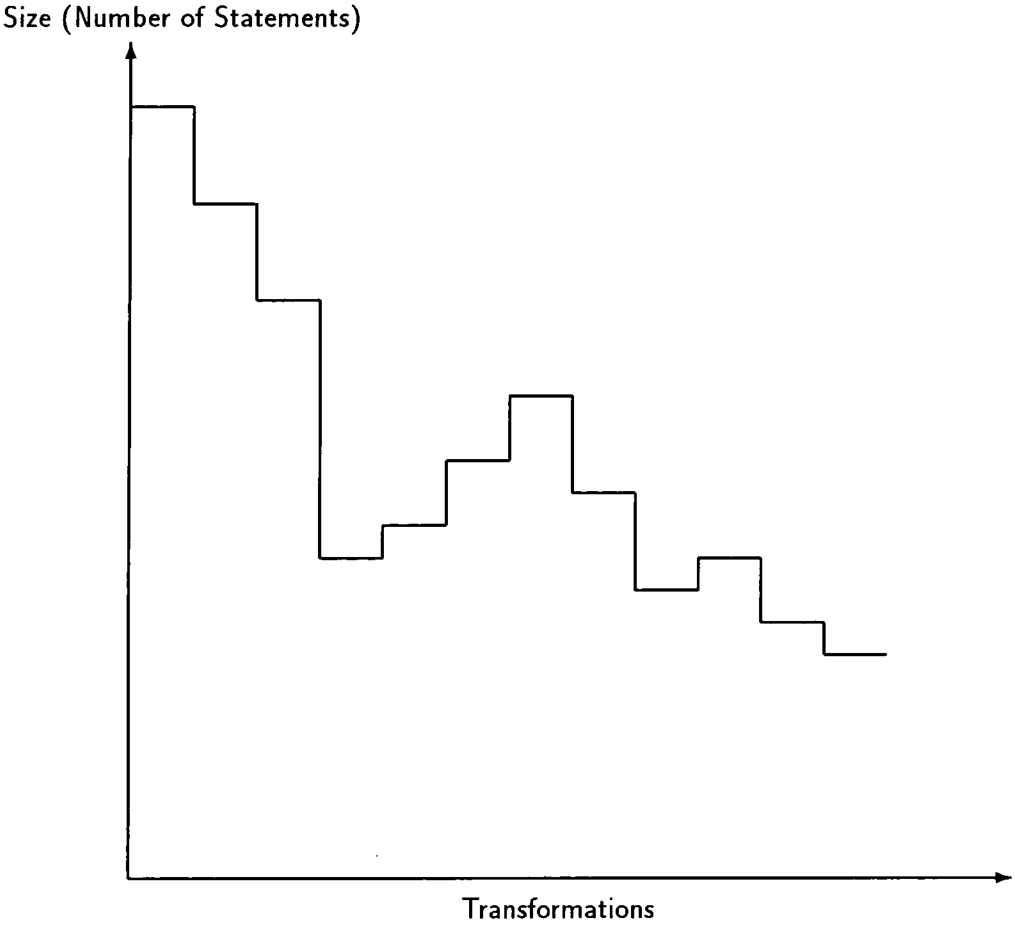


Figure 9.2: The Change in Size with Transformation

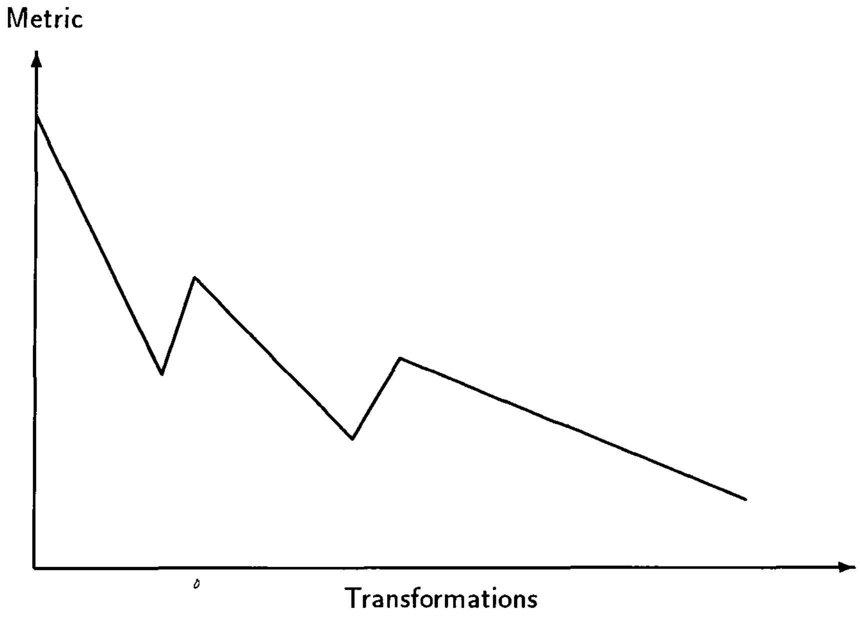


Figure 9.3: The General Change with Transformation

9.6.2 Is WSL a Good Language for Program Transformations?

As Ward showed [177] WSL is a language in which many powerful and flexible transformations can be proved. This leads to a similarly powerful and flexible transformation tool. WSL is also capable of modeling other languages; in particular it has been used to model assembler. There is, however, a drawback to WSL; it does not easily allow exception handling to be expressed, leading to the unavoidable introduction of flag variables in certain situations.

Overall, since it was designed to simplify proofs of program equivalence (for example, by using infinitary logic), WSL forms a better basis for the construction of a program transformation system than existing programming languages. It also has the advantage over other languages designed specifically for program transformation, in that its semantics are based on an *imperative* kernel language which is extended using definitional transformations, and is thus applicable to real world programs which are, generally, imperative.

9.6.3 Is the Tool Usable?

To assess the usability of the tool, two areas must be considered: the amount of training that is required to use it, and whether it responds promptly to user requests.

Training Issues

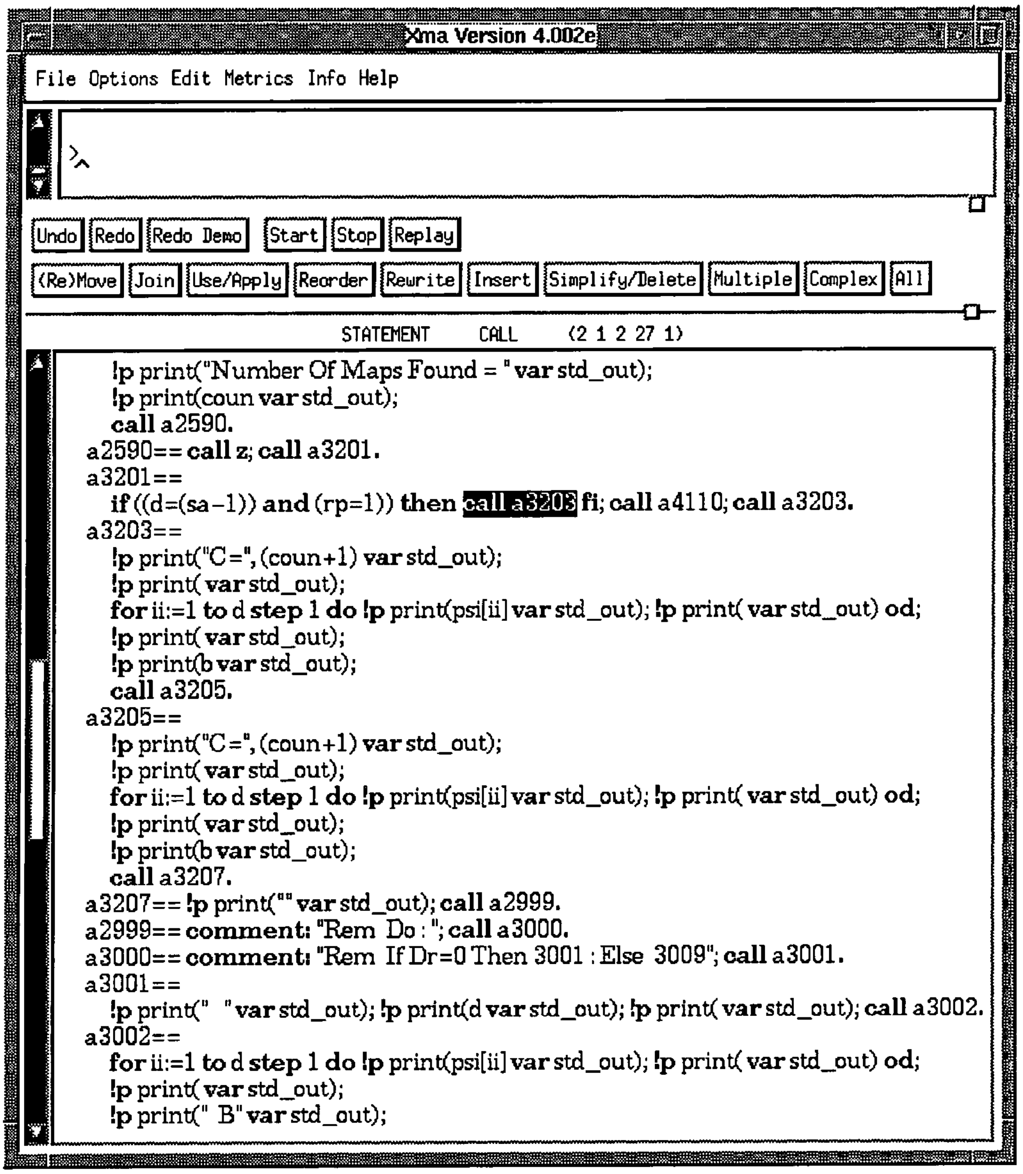
For a user to become proficient, there is a need to form a mental model of a program as an object which can be manipulated, and not just as a static object to be executed. This process has taken less than two weeks with most users, and less than a week with some. It has also been found that most transformations (particularly the generic transformations) can be understood in a similar length of time. However, a few transformations (predominantly those relating to Loops) require a deeper knowledge of the underlying mathematical theory, so a longer course of formal training would be required in order to use the tool most effectively.

Interface Issues

Because of the tool's interactive nature and graphical user interface it is easy to use even by inexperienced programmers. The layout of the interface (shown in Figure 9.4) is clear and there have been few requests to change it.

Speed Issues

The table in Figure 9.5 gives some figures for the speed of the Maintainer's Assistant in performing certain operations. From the table it is clear that for small and medium-sized programs (up to a few thousand lines), response times are acceptable. However, as program size increases the speed of response drops. This is particularly the case with the transformation for collapsing an action system, but as was seen in Section 9.5.3, if viewed as a batch process, the times become more tolerable.



```
   !p print("Number Of Maps Found = " var std_out);
   !p print(coun var std_out);
   call a2590.
a2590== call z; call a3201.
a3201==
   if ((d=(sa-1)) and (rp=1)) then call a3203 fi; call a4110; call a3203.
a3203==
   !p print("C=", (coun+1) var std_out);
   !p print( var std_out);
   for ii:=1 to d step 1 do !p print(psi[ii] var std_out); !p print( var std_out) od;
   !p print( var std_out);
   !p print(b var std_out);
   call a3205.
a3205==
   !p print("C=", (coun+1) var std_out);
   !p print( var std_out);
   for ii:=1 to d step 1 do !p print(psi[ii] var std_out); !p print( var std_out) od;
   !p print( var std_out);
   !p print(b var std_out);
   call a3207.
a3207== !p print("" var std_out); call a2999.
a2999== comment: "Rem  Do : "; call a3000.
a3000== comment: "Rem  If Dr=0 Then 3001 : Else  3009"; call a3001.
a3001==
   !p print("   " var std_out); !p print(d var std_out); !p print( var std_out); call a3002.
a3002==
   for ii:=1 to d step 1 do !p print(psi[ii] var std_out); !p print( var std_out) od;
   !p print( var std_out);
   !p print(" B" var std_out);
```

Figure 9.4: The Maintainer's Assistant's Interface

Size of Program in Statements	Time Required for Operations			
	Opening a "Move" Menu	Opening a "Rewrite" Menu	Reversing a "Cond" Statement	Collapsing an Action System
25	< 1 sec	< 1 sec	< 1 sec	4 secs
189	< 1 sec	< 1 sec	2 secs	23 secs
2,624	3 secs	2 secs	4 secs	35 mins
11,123	5 secs	4 secs	13 secs	7 hours

Figure 9.5: The Speed of the System

Also, since the Maintainer's Assistant is a prototype, designed to test the underlying ideas, not all the algorithms adopted are necessarily as efficient as possible.[9]

9.6.4 Efficiency and Reliability

The Maintainer's Assistant has been developed using the method of *rapid prototyping* rather than by formally specifying the system and then (transformationally) implementing it. This is because of its nature as a research prototype — it was not clear at the outset of the project what the specifications for the tool would be. However, the tool is built as a series of abstract machines each with clearly defined inputs and outputs. Thus, the tool is well structured and easily maintainable. This is reflected in the fact that increasingly few faults have been found in the system so that it is now a stable and reliable tool.

The table in Figure 9.6 shows, for each of the last six months of the tool's development, the number of errors found in the system, the number of enhancements made to the system and the percentage of changes that each accounted for. From this it can be seen that the number of errors found each month, and the percentage of changes caused as a result of errors, decreased.

[9] In particular, the tool is wasteful of memory when changing deeply nested program structures, which causes a larger amount of garbage collection than is necessary.

	Errors		Enhancements	
Month	Number	Percentage	Number	Percentage
May	46	28%	119	72%
June	13	29%	32	71%
July	12	27%	33	73%
August	12	29%	30	71%
September	9	22%	31	78%
October	8	20%	33	80%

Figure 9.6: The Number of Errors

Speed is an important consideration for an interactive tool since it should respond to the user with minimal delay. Running on an IBM RS/6000 with 32Mb of RAM, most of the transformations take at most two or three seconds to perform — figures are given in the table in Figure 9.6. More important is whether the algorithms employed are fundamentally efficient. Measurements made on algorithm for collapsing an action system (the slowest part of the system) show that it is approximately of order $n^{1.7}$ where n is a measure of the size of the WSL program. In fact, an inspection of the program revealed no algorithms which were exponential with respect to time, although there are some which are polynomial; mostly of degree two.

Of prime concern is the time taken to create the transformation menus since, generally, many transformation applicability tests must be performed. However, by suitably subdividing the transformation catalogue, as described in Chapter 5, the valid transformations can be determined, and put into menus, in not more than two or three seconds, as is shown in the table in Figure 9.6.

9.6.5 Correctness and Completeness

Validation of the Maintainer's Assistant — whether it is of use — was undertaken by IBM in their CUPRIMD assessment (see Section 9.5) with favourable results. Two other important issues are whether the transformations are correctly implemented and whether the transformation catalogue is complete.

In the ReForm project the transformations have already been proved to be correct [177], but it still remains is to demonstrate the correctness of the implementation; something which could be achieved by giving $\mathcal{M}\mathcal{ETA}$WSL a formal semantics and using program transformations.

The transformation system is complete in the sense that all the fundamental transformations proved by Ward have been implemented. However, a more important gauge is whether the transformations provided in the catalogue constitute a "useful" set. Certainly during the early stages of the project many compound transformations were added. However, as the project progressed, the rate at which transformations were added decreased, as can be seen from the table in Figure 9.7. This would indicate that there is some limit towards which the system was moving. (The actual limit could be a function of the examples on which the system was used, but no evidence was obtained either for or against this.) Moreover, in the latter stages of the project so few transformations were added that it would seem that this limit was (virtually) reached. Barstow [23] obtained similar results with the PECOS system.

Month	Number of Transformations	Number Added
April	512	—
May	533	21
June	560	27
July	574	14
August	591	17
September	600	9
October	601	1

Figure 9.7: The Number of Transformations against Time

9.6.6 Does the Method Scale up to Larger Programs?

Little work has been carried out in this area; however, it can be seen that the problems which would need to be addressed fall into three categories: theoretical, implementational and comprehension problems.

From the theoretical point of view, for the method to scale up, the transformations must be applicable equally to large programs as to small ones. This is certainly so. For example, a conditional statement can be reordered regardless of whether its branches each contain ten statements or ten thousand.

Implementationally, the tool must be efficient, with no inherently complex (for example, exponential) algorithms, taking not much longer to operate on large programs than on small ones. This seems to be the case with the majority of the transformations (collapsing an action system being an exception).

Finally, from a practical perspective, there must be a way for the user to view, and thus comprehend, the whole program. This could be done by means of folding or slicing the code, hiding the parts that are not relevant. For example, the program could be "sliced" on a variable so as only to display the parts of the program that affect the value of that variable. Thus, the method does seem to scale up, although more work would be needed to confirm this.

9.6.7 What Weaknesses does the System have?

The Maintainer's Assistant has four important weaknesses. The first, which has already been discussed, is that WSL is not a good language for modeling exceptions handling.

The second weakness is the system's reliance on the symbolic mathematical functions. Since simplifying a mathematical or logical expression, or demonstrating that one condition implies another are commonly performed tasks, it is essential that this is theoretically sound. So far, no work has been done on proving the implementation of this component (although the mathematical knowledge that it embodies has been proved). Not only that, but some transformations rely on inductive proofs that certain assertions (of invariants) hold; and this is outside the capabilities of the current system. For example, the system would not be able to equate the variable X with the length of L in the following program:

```
(Assign (X 0) (L ()))
(While (< Y 10)
    (Assign (X (+ X 1))
            (Y (Cons E Y))))
```

Third, it can happen that while there is a sequence of transformations that would greatly simplify the program being worked on, the sequence is not intuitively obvious. However, the system is unable to provide any guidance in this area, so only an experienced user would be able to make use of this sequence (other than by chance).

Finally, the system needs greatly extending in the area of crossing levels of abstraction, in particular in the areas of:

- The identification of generic procedures;
- The determination and removal of information which relates solely to the implementation and not to the design; and
- The introduction of abstract data types, including information hiding and inheritance.

9.7 Conclusions

The Maintainer's Assistant has been used successfully to transform both small and large programs into a highly structured form. This form provides the maintainer with valuable information regarding the structure and function of the program. In addition, a number of small programs have been transformed as far as the specification level, but this has yet to be attempted with larger programs. The tool is also reliable and efficient.

Chapter 10

Conclusions

10.1 Introduction

This chapter summarises the thesis, considers the Maintainer's Assistant in use and assesses its success at meeting the original goals of the constructing a transformation system based on Ward's transformations, which can be used primarily for software maintenance.

10.2 Summary of the Thesis

Chapter 1 of this thesis introduced the concept of *software engineering* as an approach to tackling the *software crisis*. In particular, Chapter 1 demonstrated how engineering as applied to software lacked maturity due to two particular problems: not having a satisfactory method of constructing *correct* software (a product attribute) and not having a satisfactory method of maintaining existing software (a process attribute).

In Chapter 2 solutions to these problems were examined. Solutions to the first problem were categorised as *look and see*, *test exhaustively* and *formal methods* including automatic program verification. Solutions to the maintenance problem were split into *management solutions* and *technical solutions*. Reverse engineering was one promising route which could be more efficacious were it not for its lack of formality. Thus the maintenance problem were shown to be united with the correctness problem in that a single solution would be to use a method of moving between specifications and program *in both directions* in formally correct — i.e. semantic-preserving — ways.

Transformation systems, which were reviewed in Chapter 3, claim to provide this kind of functionality but for a variety of reasons have not done so. Ward's method of proving program equivalence, however, seemed to provide a framework for the required solution. Ward's Wide Spectrum Language (WSL), which is based on transformational extensions to a small, imperative, kernel language with formal denotational semantics, and his transformations, which are proved using either semantic or proof-theoretic refinement, were considered in more detail in Chapter 4. Since no tool based on Ward's approach existed, this chapter proposed the subject of the thesis: the creation of such a tool — the *Maintainer's Assistant*. This chapter also looked at a possible method for using the tool.

Chapter 5 addressed the high-level design decisions that have had to be made in order to construct the Maintainer's Assistant. These include: how WSL programs should be represented, how the transformations to apply should be selected and tested for applicability, how transformations should be represented and stored, and why and how certain components, such as a mathematical simplifier, should be incorporated into the system. The chapter concluded that the transformations should be constructed as *programs*, as opposed to, say, pairs of patterns, (a) so that they could easily represent *all* transformations and (b) so that they could include arbitrarily complex algorithms.

The language selected for expressing transformations was designed as an extension of WSL, namely *$\mathcal{META}$WSL*. The language extensions were given in the Chapter 6 and include statements for selecting items within WSL program trees and for changing program items, and also expressions for examining the WSL programs and for performing symbolic computations.

A transformation system such as the Maintainer's Assistant needs a *catalogue of transformations* in order to operate. Chapter 7 described the contents of the catalogue — elementary, compound, generic and high-level transformations — and gave examples of how these might coded using $\mathcal{M}\varepsilon\mathcal{TA}$WSL. From this, it was possible to conclude that $\mathcal{M}\varepsilon\mathcal{TA}$WSL enabled transformations of all types to be expressed clearly and concisely.

The implementation of the Maintainer's Assistant was the topic of Chapter 8 and Chapter 9 gave details of the results that have been had with the Maintainer's Assistant; notably with *IBM 370 assembler* code.

10.2.1 Answering the Engineering Questions

In setting out the goals of this work, two sets of questions were asked. The first set were engineering questions while the second set were more theoretical in nature. These questions have been implicitly summarised in this thesis, but are addressed more explicitly in the following sections.

How should programs undergoing transformations be represented?

WSL programs are represented as abstract syntax trees in which each node (and its corresponding subtree) represents a single syntactic object and the branches of that node are the components of the object. The advantages of this approach are that the transformations work mostly on syntactic objects, and these can easily be selected, identified and manipulated as branches or leaves within the tree structure, obviating the need for much parsing.

In addition, the trees are used to store information about the items' types, database query tables (to enhance the efficiency), identity numbers (to aid in communication with the interface) and attached comments (to provide a documentation facility). The implementation of the tree structures is by way of LISP's nested lists. Having created a transformation system based on tree structures, empirical observations show that this is an efficient approach.

How should the transformations, and their point of application, be selected?

Human expertise of both software engineering and the application domain should influence the direction of the transformation process. Thus, it is necessary for the system to be interactive and to provide some method for selecting *which* transformations should be applied and at *which point* in the program tree. The point of application of transformations can be selected in two ways.

1. The program is presented on a graphics screen and the user points to an item in the program using a mouse and clicks a mouse button. The smallest portion of the tree (syntactic object) containing the item at which the user pointed would then become the selected item. This method has provided the user with a simple and intuitive method of identifying sections of the program.
2. In the second method the system provides a series of commands for *relative* movement within the program tree: LEFT, RIGHT, UP and DOWN. This method, while available to the user of the system, has been harder to use, but has proved to be more efficient and forms an essential part of the mechanism for performing transformations.

Having selected the point of application, it is necessary to select the transformation that is actually required, as in most cases more than one will be applicable.

The transformations are divided by function into a number of different menus. When the user selects a transformation menu, the system checks which of the transformations that could potentially appear in the chosen menu are actually applicable, and creates a menu containing only valid transformations, listed alphabetically, from which the user makes his choice. This arrangement reduces the need for the user to know the names of all the transformations in the system, and the division of transformations into small groups has decreased the time required to construct the menus. A minor drawback is that it is not always clear in which menu a particular transformation should appear.[1]

How should the applicability of the transformations be tested?

In the simplest cases, the applicability condition of a transformation would consist of a pattern that a section of the program has to match against. But in more complex transformations, additional tests on the code would be needed. These could be put into a logical formula, but this approach lacks the flexibility to test, for example, "distant" sections of code. Thus, the Maintainer's Assistant uses a *programming language*, which has been based on WSL, *MetaWSL*, for expressing applicability tests.

How should the transformations be represented?

Simple transformations can be expressed as two patterns, possibly together with actions to be performed on the elements as they are matched. However, these actions are often extremely complex and some, such as those that need to examine sections of code outside the syntactic scope of the section being changed, cannot be represented in this way. Thus, as with applicability tests, the *Meta*WSL *programming language* is used.

How should the transformations be stored in the system?

Transformations are stored in a tree, based on the WSL types to which they apply, so as to reduce the number of transformations that need to be tested for applicability at any stage. Each entry in the tree holds information about the transformation, such as its name, and an index into a number of vectors which store additional information. This combination of a tree and vector structures

[1] There is an "All" menu which lists *all* the applicable transformations.

enables a specific group of transformations to be accessed efficiently and, once found, the information about the specific transformations can also be accessed efficiently.

How can transformations be combined?

A set of elementary transformations forms a basis for constructing further transformations via combination. This is made possible by use of the $\mathcal{M}\varepsilon\mathcal{TA}$WSL statement @Trans and condition [_Trans?_].

Transformation combinations that are not provided must be explicitly performed by the user. Experiments have been carried out whereby the user can record such a sequence of transformations and replay it later, possibly on another part of the program. This has proved useful only in a very few circumstances as it is not usual to want to apply *exactly* the same transformations in two places. A future extension of the system would be to provide the user with a *macro language* so that he can build compound transformations himself using WSL in combination with a subset of the features provided by $\mathcal{M}\varepsilon\mathcal{TA}$WSL; for example, @Trans, [_Trans?_] and movement statements, but *not* editing statements (otherwise the user would be able to construct potentially invalid transformations).

What other facilities should be included in a usable transformation system?

In order to make the Maintainer's Assistant into a usable transformation system, there are a number of other facilities and features that have been incorporated.

First, so that it is simple to operate, the tool works in a fast, efficient and intuitive windows environment with mouse control, buttons, pull-down menus and so on.

Next, since the user may not always choose the correct sequence of operations at a first attempt, the system provides the ability to undo changes made to the program. An "audit trail" facility is also included so that as a program is changed, the operations performed on the program, and the selection of program items, are recorded and could be "replayed" on a modified initial specification.

Since faults may be found in the program being maintained with the system, the Maintainer's Assistant incorporates a structural, syntax-based editor.

The system includes the ability to calculate metrics about the program undergoing transformation thereby providing and a measure of the benefit produced by the transformation process.

Finally, both a user guide and online help are provided.

10.3 Criteria for Success Revisited

In Chapter 4, the criteria for the success of this work were presented. This section will consider whether these criteria have been met.

10.3.1 Preliminary Questions — Maintenance by Transformation

- Is software maintenance made simpler by using transformation-based reverse engineering?
- Is WSL a good language for this purpose; i.e. can existing programs be expressed in WSL and is there a suitable range of WSL transformations?
- Crossing levels of abstraction involves removing details of the program's implementation while retaining details of its function. How can one do this in a transformation-based system?

The first two of these questions were answered in Chapter 9. From the experience gained using the Maintainer's Assistant, in particular within IBM Hursley, it would appear that maintenance by transformation offers great benefits in terms of resulting code quality and programmer productivity.

WSL was designed from the beginning to facilitate program transformation and as a result is better suited to this purpose than traditional languages. However, its elegant semantics lead to difficulties when it is required to express certain inelegant programming concepts, notably exception handling. Overall, it does form a good basis for transformational maintenance.

The aspect of crossing levels of abstraction was addressed in Section 4.4.5. The method adopted involves first restructuring the program into a suitable form (for example, introducing procedures and using recursion in place of iteration). Having done this, the user then needs to identify suitable abstractions which can be introduced via assertions (possibly resulting from the identification of invariants). As yet, not enough work has been carried out to determine how successful this approach might be but, on very simple examples, it looks promising.

10.3.2 Central Questions — The Assessment of MetaWSL

- What constructs should MetaWSL include so as to be flexible enough to express program transformations without becoming overburdened with little-used constructs? i.e. What constructs should MetaWSL include so as to be simple yet complete?
- Can MetaWSL clearly and concisely represent Ward's transformations?
- What other transformations are required?
- Can MetaWSL be used to express clearly and concisely these transformations?

As was seen in Chapter 5, and expanded on in Chapter 6, MetaWSL has been constructed so as to code both the transformations' applicability *and* the code for performing the transformations. MetaWSL includes (for reasons given in Chapter 5) the following type of construct over and above those provided by

WSL: program editing statements, pattern matching and template filling functions, movement statements, movement applicability testing functions, predefined global variables, query functions, symbolic mathematics and logic functions, and repetition statements.

Through the implementation of the transformation's of Ward's thesis [177], Chapter 7 demonstrated that $\mathcal{METAWSL}$ *is* suitable for coding these transformations. Moreover, the representation of these transformations using $\mathcal{META}$WSL was seen to be both clear and concise.

There are four types of transformation in the system. First, all Ward's transformations [177] are incorporated.

Transformations of the the next type combine these in forms that have been identified through studies as being useful. These are compound transformations. Their effects include fully factorising a Cond statement, removing Exit statements from a loop or restructuring an action system, and they are large in number.

Generic transformations combine into single transformations elementary or compound transformations which have a similar function such as merging program items. Thus, the user does not need the experience to know which specific transformation has to be selected.

Finally, high-level transformations enable a user to extract a specification from a program.

As was seen in Chapter 7, each transformation can be written concisely, without difficulty and no less clearly than a purely mathematical statement of the transformation. This is because of the availability in $\mathcal{META}$WSL of all the structures of WSL, together with additional control statements and the mathematics and logic routines.

10.3.3 Questions on the Effectiveness of the Tool

- Does the approach result in a usable tool? In particular, what training is required?
- Is the implementation of the transformation catalogue efficient, reliable, correct and complete?
- Does the method scale up to larger programs?
- How well does the system work on real programs in an industrial environment?
- What weaknesses does the system have?
- How does the use of the tool fit into the software process?
- Does this system add to the study of transformation systems in general?
- Can the system be used to maintain itself?

Chapter 9 specifically address the first five of these questions. As was seen, the approach does result in a usable tool which is efficient and reliable. Correctness and completeness need to be proved, but the evidence available is favourable. The evidence also shows that the method would, unless any unforeseen problems arose, scale up to larger programs. The system has four weaknesses in the areas of: WSL's representation of certain types of behaviour, the reliance on the symbolic mathematics system and lack of a method of performing inductive proofs, the difficulty of finding optimal transformation sequences, and in crossing levels of abstraction.

The other three questions in this section will be considered at greater length.

How does the Maintainer's Assistant fit into the software process?

The Maintainer's Assistant is a tool that takes a program as its input an transformations it to produce another program as its output. As was seen earlier, this could be extended to take specifications as input and/or produce specifications as output. Thus, the tool could, potentially, be used at any stage in the software life cycle at which specifications or programs form both the input and output. This would exclude the requirements analysis and definition and both specification stages, but the tool could be used in the production of a design from the specification and in the production of code from the design. As a side effect of using the Maintainer's Assistant in these phases of the life cycle, the need for verification in the testing stage would be reduced. Finally, one of the original aims was to create a tool to help with maintenance, and it is indeed also of use in this stage of the life cycle.

At all these stages, it is not *necessary* to use the Maintainer's Assistant; rather, it can be seen as a extra weapon in the software engineer's armoury to be used when deemed suitable. The reason for this is that it can be applied to *any* code or specification. Thus, the software engineer may choose to develop (or maintain) certain parts of his system using traditional methods, and just use the Maintainer's Assistant on those parts where provable correctness is essential. However, by using the tool only on *parts* of the system does reduce the benefits that it can give.

Does this system add to the study of transformation systems in general?

There are four main ways in which this work has added to the study of transformation systems in general. First, it has been designed with maintenance as its *primary* field of use. This has resulted in a system which is designed to take as its input unstructured code and produce a design; the opposite of most systems which take as their input a design and produce as their output efficient code. As a result of this approach a number of issues have had to be addressed, notably in the area of formal restructuring and in the crossing of levels of abstraction.

The second contribution of this work is in the method in which transformation are expressed. While $\mathcal{METAWSL}$ was not the first *language* to be used for writing program transformations (see HOPE [51] and Hildum and Cohen's work [95]), it is the first language which contains more than pattern matching, template filling and simple iteration facilities. In particular, *METΑ*WSL includes statements for movement within program tree. These allow the determination of context information about a part of the program other than the part that the transformation changes as would be the case with a transformation which replaces a procedure call by the body of the corresponding procedure.

Other *METΑ*WSL constructs reflect the use of Ward's approach to proving program equivalence. These expressions for testing whether a program item is "terminal", "reducible", "proper", "improper", "dummy", "regular" and so on enable more sophisticated transformations to be included which take into account more than just the superficial syntax of the program. *METΑ*WSL could be adopted and reused in future transformation-based work.[2]

Another way in which this work has added to the study of transformation systems has been in the area of usability. While many other transformation systems require the user to test the validity of the application of the transformations, the Maintainer's Assistant does this itself, presenting the user with a menu of only correct transformations. This is a benefit of coding each transformation as two pieces of *METΑ*WSL code: the first part which tests its applicability and the second which performs the changes to the text of the program.

Finally, the Maintainer's Assistant has been used with effect in a commercial organisation (IBM Hursley) and on an actual "live" system. This gives the system a degree of credibility that other transformation systems lack.

Can the system be used to maintain itself?

Since the transformations in the Maintainer's Assistant are written in an extension of WSL, the transformations can be applied to themselves. The exception is that there are as yet no transformations relating explicitly to *METΑ*WSL constructs.[3] However, once a formal definition has been produced for *METΑ*WSL, transforma-

[2] This is borne out by the fact that *METΑ*WSL has been adopted, albeit with some stylistic changes, by Durham Software Engineering's Fermat project [182].

[3] Such a transformation might remove a "@Left; @Right" pair.

tions on $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL could be produced and the tool would become more usefully applicable to its own code.

10.4 The Final Analysis

The majority of the software industry, and software maintenance in particular, still lacks maturity for the reasons described in Chapter 1. However, there are a few organisations and projects — for example the Space Shuttle software project — that have managed to achieve a higher level of maturity through a disciplined use of *informal* methods. Formal methods, as has been shown in this thesis, appear to offer more benefits than these informal methods, so the challenge is to realise these benefits. Transformation systems have the potential to harness the advantages of formal methods, making valuable tools, but until now have, for various reasons, failed. The Maintainer's Assistant, being based on a purpose-designed programming language (WSL), an improved method of proving program equivalence [177], a more flexible method of expressing program transformations (using $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL), and a simple, interactive interface would, at least on paper, seem to overcome these problems. So far, experiments on commercial systems bear this out.

Thus, software maintenance by program transformation in a wide spectrum language not only seems plausible, but offers important benefits over other approaches.

10.5 Future Directions

The most important extension to this work is the production of a *formal* definition of $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL. This would enable proofs of the correctness for the implementation of the transformations to be produced. In addition, it would allow transformations on $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL to be constructed, thus enabling the tool to be used more effectively in its own maintenance.

There is a need to extend this work in the area of crossing levels of abstraction. In particular simpler methods must be introduced for identifying *suitable* abstractions. This method must also cover data abstraction which is an area that has not been considered in this thesis.

Other enhancements revolve around usability issues, for example in helping a user who is untrained in program transformations to benefit from the system. These include:

- An interactive assistant to help in the selection of transformation. This could be implemented by storing with each transformation a list of possible "next candidate" transformations. An alternative method would be to make use of the metrics facility to guide a "hill climbing" algorithm.
- A "jittering" mechanism similar to that used by the TI (Transformational Implementation) System [18]. This automatically modifies a program to match a transformation that previously failed to match because of some technical detail. Mostow [139] proposes a similar system for automating transformations in which the user selects the transformation he wants to apply. The system calculates the changes that need to be made first, and looks for suitable rules to apply.
- An additional class of transformations which embody knowledge of programming goals: "divide and conquer", formal differentiation, embedding, recursion removal, backtracking, function tabulation, function inversion, dynamic programming (store versus recompute) and so on.
- A facility to allow the user to build up his own catalogue of compound transformations consisting of combinations of existing transformations. Thus, he could customise the system for the particular problems that he has to deal with. This would follow the trend which is away from huge catalogues, and toward individual, problem-oriented sub-systems based on small sets of powerful rules allied with advanced metalanguages [152]. Such an approach was also proposed by Bauer [24].

The ultimate goal of a transformation system is to achieve a symbiosis between the talents of a skilled human, who is better able to make strategic decisions, and the mechanical abilities of the system to carry out flawlessly numerous trivial low-level manipulations [73]. Adding the features described above to the Maintainer's Assistant would represent considerable progress towards this goal.

Appendix A

A Survey of Transformation Systems

In the following review, only those systems which use transformations in interesting or original ways will be considered. Simple optimising compilers and partial evaluation will not be considered. The systems reviewed will be assessed in terms of their formality, how much guidance and informal reasoning the user has to provide, the scope of the examples the systems have been applied to, and whether the system is being used other than by its developers. Each transformation system will be described under five headings:

Background — The first section give the details of where the transformation system was developed.

Purpose — Since transformation systems have been constructed to achieve many different goals, the second section gives these goals.

Details — The third section describes how the transformation system works, its transformation catalogue and any special characteristics the system has.

Results — The results section analyses the results that have been obtained with the system.

Conclusion — The final section assesses whether the transformation system has been successful, and what, if anything, it adds to the field of knowledge about such systems.

A.1 The SETL Project

Background

This long-running project at the Courant Institute of New York University [62] has served as the context for a wide variety of transformation research. SETL is one outcome of this work.

Purpose

SETL is a very high level programming language which has syntax and semantics based on standard set-theoretic mathematics. It is possible to execute SETL programs, but naïve execution of programs that make liberal use of the high-level language features may be very inefficient. There is a SETL compiler which produces efficient interpretable code or machine code. A key part of the compiler is an optimiser which uses ideas from transformation systems.

Details

SETL uses transformations for code optimisation, for selecting data structures and for ensuring that data types are efficiently used. Although the work is of relevance to transformation theory, it is not a general purpose transformation system.

Results

The examples on which SETL has been tried are described by Partsch and Steinbrügen [152], and include finding the shortest path in a graph, some garbage collection algorithms and the Cocke-Younger parsing algorithm.

Conclusion

As has been described earlier, there are many benefits to be gained from writing programs at a high level and performing transformations to produce executable code. SETL is one system that supports such an approach, but it does not have the power required to be of general use, since the examples cited above were only transformed from high-level specification to executable code with a great deal of informal reasoning. Thus, while SETL has worthwhile objectives, the transformation system employed is too limited in both its scope and power to be able to draw many lessons from it.

A.2 RAPTS

Background

RAPTS (Rutgers Abstract Program Transformation System) was developed by Paige [148].

Purpose

It is a supercompiler that takes as its input an abstract program specification and outputs efficient object code and a description of its performance. It is claimed that because specifications are more likely to be correct than programs, the output from this system is more likely to be bug-free than that from an ordinary compiler.

Details

RAPTS is a running transformational programming system that embodies many of the features of supercompilers. It constructs programs in a sequence of stages, each of which implements an essential program characteristic. The first stage introduces computability, while the following stages add strategy, data structures, optimal control flow and so on. The theory behind RAPTS is that of expressing specifications in purely set-theoretic terms and then using fixed point theorems[1] on these sets as the basis for transformation rules.

RAPTS has a small catalogue of transformations which does not support lower level code manipulations.

Slight changes in the form of the input specification can result in significant differences in the performance of the object code. Thus, it seems essential for the success of transformational systems which use large scale automation that they both compile the code and provide a performance analysis of this code [148]. This is possible in RAPTS since the implementation of a program characteristic at each stage is accompanied by a sufficiently precise increase in efficiency that it is possible to combine the efficiency information to create a time complexity formula.

In contrast to the SETL optimiser, reported to be 24,000 lines of SETL code, RAPTS is only a few hundred lines of SETL [148].

Results

No results are available.

[1] In lattice theory, a fixed point is a point which satisfies the equation $x = f(x)$ for some monotonic function f. By expressing recursion as a function, f, in this form, where x is the "solution" of the recursion, it is possible to use the theorems of fixed points to transform the recursive program.

Conclusion

The major problem is the applicability of the approach. In particular fixed point transformations only apply to specifications written in set-theoretic terms, which is not always a convenient way of expressing a specification. A second weakness is the lack of published results.

A.3 The TAMPR System

Background

TAMPR (Transformation-Assisted Multiple Program Realisation) was developed at the Argonne National Laboratory.

Purpose

It is not a general transformation system, but is a special purpose system whose primary goal is to adapt numerical algorithms to work with particular hardware and software environments. In particular, the TAMPR transformation system is used to transform declarative specifications written in pure applicative LISP (which has no side-effects) into efficient executable programs written in FORTRAN (which may be either sequential or parallel). The advantage of pure LISP specifications is that they are based on the mathematics of the lambda calculus and recursive function theory. Thus, from the point of view of program transformation, it is possible to write and prove interesting transformations, and it also prevents problems from being overspecified.

Details

The TAMPR System uses 20 language levels between pure applicative LISP and FORTRAN [42]. These levels include pure applicative LISP, expressed in extended-FORTRAN syntax and recursive FORTRAN with one function call per statement.

Transformations in TAMPR are rewrite rules consisting of a *pattern* and a *replacement* [41]. In addition, transformations can have *applicability conditions* on the semantics of the code but, according to Boyle, these are seldom needed. TAMPR contains 90 major correctness preserving transformation rules divided into 20 independent groups corresponding to the different language levels. Thus, at each level there is only a small number of transformations available.

Both transformations and the programs on which they work are represented as tree structures internally in the system [41].

Results

In [42] an example is given in which a program is produced whose resulting code is 3,150 lines. This takes 15,639 transformations for a sequential version of the program, and 23,583 for a parallel version. These are too many for the user to have to apply by hand, so a strategy is used to automate completely the system. This strategy is described in [42].

TAMPR has been used successfully to convert single-precision to double-precision arithmetic and to change the dimensions of an array, for example to use a one-dimensional array in place of a two-dimensional one. TAMPR has also been used to transform FORTRAN programs to uncover structure inherent in them and to make other changes to FORTRAN programs [41]. However, the most significant accomplishment of the system is that TAMPR itself is written in LISP and has been transformed into FORTRAN. When new functionality was added to TAMPR, it was easy to retransform it into FORTRAN.

Conclusion

The use of many different language levels is in contrast to systems which use a wide spectrum language and would appear to be a serious drawback if the system were to be extended to become general-purpose since different transformations are needed for each language level (and also for moving between different levels). Nevertheless, in its restricted domain TAMPR has been used with some success, producing programs which are much more efficient than the original LISP code. It has also been used to perform a limited amount of legacy code restructuring.

A.4 The Restructurizer

Background

A special type of transformation system is the structuring engine, which is a software tool with two properties [124]:

- It transforms an executable program written in a given language, but of undetermined structure, into another program written in the same language with a well-defined structure; and
- The resulting program produces the same transformation on any set of input data as does the original program.

Such tools which have already been developed include Superstructure, Structuring Facility and, described here, the Restructurizer developed by Sneed [7] [168].

Purpose

The Restructurizer takes, as its input, code written in COBOL-74 and produces restructured code written in COBOL-85. This is used as the third of five stages in a more general strategy for software recycling (Sneed's own variation on restructuring).

Details

The task of the restructurizer is to restructure the flow of control within COBOL modules by means of the successive application of seven syntactic rules which are mainly designed to remove Goto statements [168].

A key factor in the recycling process is the presence of an intermediate design language to which the COBOL-74 is first converted by hand before being regenerated as COBOL-85 code. This allows all the work to be carried out at the same "level" with the same set of rules (i.e. transformations). This is the opposite of the TAMPR system described in Section A.3.

The Restructurizer has no firm theoretical foundation for the transformation rules that are used in the system. Instead these are "rules of thumb" such as:

> All forward branching conditional Gotos should be deleted, the If condition negated and all the statements up to the label referenced should be nested by one.

Results

The Restructurizer has been used successfully to restructure various real-world COBOL programs.

Conclusion

The Restructurizer does all that it sets out to do; that is, it restructures unstructured COBOL programs. However, it is only able to consider the *syntactic form* of the code and not its *meaning*. Thus, unlike a true transformations system, it is unable to remove redundant tests and so on; this is partly due to its not being a rigorously-based system.

A.5 Burstall and Darlington's Work

Background

A great deal of the pioneering work in transformation systems was undertaken by Burstall and Darlington, and their work is continuing. Their ideas have heavily influenced today's transformation systems.

Purpose

The first version of their work was a schema-driven method for transformation in which transformations were applied to code which matched certain patterns. The system transformed from applicative recursive programs to imperative ones, the goal being improved efficiency. On the whole their system worked automatically with little user interaction. However, it had a very incomplete (and difficult to extend) set of transformation rules.

Details

Their second system is, like their first, based on the generative set approach and used only six rules:

Definition — which introduces a new recursion equation;

Instantiation — which introduces a substitution instance of an existing equation by replacing a parameter by a value;

Unfolding — in which a (recursive) call to one of the recursion equations is replaced by the body of that equation;

Folding — in which the body of an equation is replaced by a (recursive) procedure call;

Abstraction — which introduces a "Where" clause by deriving a new equation from a previous equation by replacing specific values by parameters; and

Laws — which are any set of data-structure-specific rules such as associativity, commutativity etc.

The system works largely automatically on the language NPL [49]. It uses "forced folding" in which the system suggests a fold which the user accepts or rejects, asking for another. The system is not fully automatic, and the user must supply definitions, clever ideas in the form of "eurekas", laws for data structures, explicit reduction rules, and switches which permit or forbid various searching criteria and expression generalisations.

Results

The recursive functions which have been transformed by the system are all fairly simple and mostly mathematical.

Conclusion

This system is in itself very primitive by today's standards; in particular it is limited to transforming from recursion equations (imposing a restriction on the kinds of program that can be transformed) to improved recursion equations (necessitating, in most cases, further processing before the program can be expressed in a conventional language). Nevertheless, the system was important and influential in providing the inspiration for later transformation systems.

A.6 The ZAP System

Background

The ZAP system [70] [71] [74] is based on the Burstall and Darlington system.

Purpose

There is special emphasis in the ZAP System on software development by supporting large-scale program transformations which perform dramatic, global changes to the program as opposed to small, local changes.

Details

The principle is still that of "fold/unfold" and the language is still NPL. In ZAP, however, meta-programs can be written using HOPE (a purely applicative programming language, developed from NPL) which apply a directed series of transformations to NPL programs in a high-level hierarchical fashion. The system is based on pattern-directed transformations, i.e. transformations in which the user gives only the approximate form of the expected answer, as a pattern. These are expanded into a variant on the six basic rules of the previous Burstall and Darlington system which the system then applies to perform the transformation.

There are other new facilities, too, and these include an extended control language, defaults (notably default patterns), a bookkeeping facility to record the sequence of operations, and a "discovery" capability so that the system can suggest alternative transformations.

Results

The examples on which ZAP has been tried are more ambitious than the examples tackled with most other transformation systems: the "telegram problem" (which involves decoding an incoming stream of characters) [70], a very small compiler and a text formatter [71]. However, these examples leave the resulting program as a series of recursion equations which must be further modified to produce programs in conventional programming languages.

Experiments with transformations applied to maintenance have also been performed with ZAP. However, these have been limited to performing new transformational developments from slightly different starting conditions. No attempt has been made to use the system to perform reverse engineering.

Research is also being conducted into "paradigm algorithms", such as the general divide and conquer paradigm and other general strategies.

Conclusion

Perhaps the most important feature is the use of a meta-language for expressing transformation tactics since it gives the system a great deal of flexibility. However, the user needs to write scripts in the meta-language for each transformational development, making the system cumbersome to use, especially as large and complex patterns needs to be created in non-trivial cases.

The use of ZAP for maintenance is another important idea but, as with meta-language scripts, the theory is better than the practice, since no work has been done on transforming existing code and the system is, therefore, of little use in the real world.

A.7 The SAFE and TI Projects

Background

SAFE (Specification Acquisition From Experts) is Balzer's project from the Information Science Institute (ISI) in Los Angeles and deals with the synthesis of formal specifications from informal ones [190]. Although not a transformation system itself, it forms part of a larger system with TI (Transformational Implementation) in which transformations play an important part.

Purpose

TI works on the derivation of efficient programs from formal specifications by means of transformations [18]. Thus, the SAFE/TI combination allows for informal specifications to be transformed through to executable programs.

Details

TI does not produce programs directly, but instead produces output in a subset of the specification language GIST, which is translatable into an existing programming language. Abandoning automatic compilation allows more freedom in the language in which programs are presented to the computer. GIST is the result of adopting this approach [81] [19], and is a wide spectrum language. It has been developed to provide the flexibility and ease of expression necessary for describing the full range of acceptable system behaviour.

When using the TI system, it is the programmer's task is to select transformations from a pre-existing catalogue [18]. If a required transformation does not exist, the programmer may extend the catalogue or edit the program manually. In both cases it is up to him to ensure the resulting program's correctness.

Among the features of the TI system are:

- An interactive transformation engine;
- An automatic documentation facility that allows one to replay a development using a modified specification;
- A catalogue of transformations which reflect how to implement certain specification constructs and optimisation techniques; and

- A mechanism for translating a fully developed program into some target language.

TI includes one other important feature: "jittering". Jittering is the process by which the system automatically modifies a program to match a transformation that previously failed to match because of some technical detail. This uses standard artificial intelligence back-tracking techniques.

Results

Some examples which have been tackled with the TI system include a text editor [18], some special versions of a line justifier [190], the "eight queens" problem [15] and a package router [122].

Conclusion

TI's key strengths are the GIST wide spectrum language and the "jittering" technique. GIST, being formal and wide spectrum, provides an enormous degree of expressiveness and power (even if it lacks clarity and structure and is thus hard to understand, as one of its developers admits [72]).

However, although working with GIST is clear in principle and has been demonstrated with a number of substantial case studies, there are still several non-trivial technical omissions; i.e. (a) a sound theoretical foundation for the notion of a valid transformation (the user may add unproven transformations to the catalogue and to make unchecked edits to the code), (b) suitable collections of rules, (c) appropriate strategies for the development of an implementation [151].

A.8 GLITTER

Background

GLITTER, like the SAFE and TI systems, originated in the Transformation Based Maintenance project at Information Science Institute (ISI).

Purpose

This project addresses the problem of understanding, reusing and maintaining previous specifications and optimisations. *Paddle* is the language used to record the formal development of GIST specifications. These developments can then be reused. The major weakness of this system is not in the Paddle itself but in the interpretation mechanism for Paddle [189]. Hence Fickas created an automated development system that selects and applies transformations to achieve developer-stated goals. The language for stating these goals is GLITTER (GoaL-directed jITTERer).

Details

GLITTER [75] was designed within the TI environment. The user begins by stating (in GIST) some design goal. This is then processed by four subsystems.

- The problem solver either asks the user for more details or checks the method catalogue;
- The method catalogue contains methods for achieving goals and contains transformations and planning knowledge;
- The rule selection catalogue chooses between rules from the method catalogue; and
- The applier applies the chosen method.

GLITTER is an interactive, rather than fully automatic system. For example, it may be necessary in the process of using it for the user to add to one of the catalogues.

Results

GLITTER has been tried on a large number of toy examples and a few larger ones. The latter included a controller for a mechanical postal package router in which the system was able to generate automatically a significant number of steps [75].

Conclusion

GLITTER extends the capabilities of TI, but is still let down by a lack of formality epitomised by the fact that users can add unproven transformations and methods to the catalogues.

A.9 The PSI and CHI Systems

Background

The PSI system [87], [85], [86] [88] and CHI [89] system have been produced mainly at Stanford.

Purpose

The purpose of both systems is to synthesise efficient programs, by taking as input a specification obtained from a dialogue with the user. This may include natural language or partial traces of computations (given by sample input-output pairs).

Details

In PSI, the dialogue with the user is processed by various "experts" which are software modules which together form a large LISP system. Each expert performs a particular function such as parsing the input or applying domain knowledge.

The result is a number of program fragments which act as input for the Program Model Builder (PMB) [129]. The result of this is a complete program model which is coded by a coding expert and an efficiency expert. PMB builds a complete and consistent program model which is an abstract, implementation-independent, annotated program in a high-level language. The program, thus, corresponds to the desires of the user. PMB's expertise is coded as a set of about 200 procedural rules which are scheduled by a rule interpreter.

The coding expert (PECOS) [20] [21] takes an abstract program description produced by PMB and successively refines it using transformation rules which reflect coding knowledge. This has two parts:

- A catalogue of about 400 transformations relating to symbolic programming [21] together with an extensible knowledge base [20]; and
- A task-oriented control structure based on program development by successive refinements.

The efficiency expert (LIBRA) [104] [105] gives advice to the coding expert, thus helping it to make decisions. Its expertise is coded as an extensible knowledge base of about 100 rules. New rules can be derived semi-automatically from new transformations or can be gained by asking the user appropriate questions.

The synthesis phase of PSI (PSI/SYN) transforms specifications given in PMB's formal high-level language [106]. This reverses the röles of PECOS and LIBRA, so that PECOS advises LIBRA.

Results

Partsch and Steinbrügen [152] give a list of examples on which the PSI system has been tried, all of which are only moderately sized and very heavily biased towards number-theoretic algorithms. These include a prime number generator, some set manipulation algorithms, a simple retrieval program, an algorithm to determine the reachability of nodes in a graph and a variety of sorting algorithms.

Conclusion

PSI was a useful testbed for program transformations, but was superseded by the CHI system. This differs from PSI in that it uses a wide spectrum language called "V", which is easier to read. Also, instead of using autonomous experts, CHI uses a homogeneous collection of tools sharing a common database. Although CHI has had very limited success in use (having only been applied to number-theoretic examples), it makes two important contributions to transformation theory: rules can be applied by analogy (by indicating a related rule having a similar effect) and it is a self-describing system, in that it can be modified using its own rules.

A.10 The CIP Project

Background

The work has been carried out since 1975 in Munich by Bauer [27] and others.

Purpose

The CIP acronym (Computer-aided, Intuition-guided Programming) indicates the project's primary aim; to produce a system which allows the user to construct programs by transformation, obviating the need for much clerical work ("computer-aided"), while using his own experience ("intuition-guided") to direct the process. Secondary aims, which have been accompanied by thorough investigations of the theoretical issues, are:

- To use a sound method (based on a formal calculus) for guiding the process of formal reasoning in program development;
- To design and define formally a wide spectrum language, CIP-L, in order to provide a uniform framework for the formulation and transformation of both specifications and programs;
- To develop an interactive system for supporting the process by performing the transformations mechanically, doing administration, and producing the documentation.

Details.

CIP-L, has different programming "styles". It has a core imperative language, based on the algebraic semantics [154], which is extended with applicative constructs by definitional transformations; each language construct is defined in terms of how it can be transformed into a combination of lower-level constructs [153]. Other transformations include [26] fundamental transformations on the kernel language, and derived transformations which are sequences of fundamental or definitional transformations.

Transformations are represented as input and output templates together with additional "semantic" predicates on the code being transformed. Transformations can be applied to program schemes, thus producing a new transformation rule consisting of the original program scheme as the input template and the generated program scheme as the output template. The applicability conditions for the new rule are induced by those of the applied rule [26].

Internally, program schemes are stored as abstract terms which are tree-like structures.

Results

Pepper [155] gives three examples of the use of CIP. These are the elimination of quantifiers, recursion removal using data types, and a data flow program. But perhaps the most significant is that it has been used to prove parts of its own development.

Conclusion

The CIP project has made substantial contributions to the field of program transformation systems such as the symbiosis of the human and computer contributions and the use of a wide spectrum language. Nevertheless there are certain areas of weakness of the system.

First, in performing transformations it may be necessary to obtain information about a part of the program other than the part that the transformation changes. This would be the case with a transformation which replaces a procedure call by the body of the corresponding procedure. In the CIP project, this non-local information is obtained by way of "contexts" and "theory propagation" [155]. However, these have been added later and are rather clumsy, since, for example, the definition of a procedure must be stored at every point in the program.

Second, in order to understand abstract types it is helpful to form a conceptual model [45], but CIP's semantics are not model-based, and thus require an extra level of description.

Third, the method can only deal with the equivalence of full programs and not arbitrary program parts [153].

Finally, in order to deal with imperative programs (such as are used in virtually all real-world situations) the applicative kernel needs to be extended with many "implicit axioms" such as: $(S_1;\ S_2);\ S_3 \ \equiv\ S_1;\ (S_2;\ S_3)$ [177]. Extra axioms necessitate extra conditions on the applicability of the transformations reducing the scope and usefulness of the approach.

A.11 DEDALUS

Background

Various experimental transformation systems have been developed by Dershowitz and Manna [60] the most important of which is DEDALUS (DEDuctive ALgorithm Ur-Synthesizer).

Purpose

The goal of DEDALUS [127] is to derive LISP programs automatically and deductively from high-level input-output specifications in a LISP-like representation of mathematical-logical notation. Manna [125] claims that the methods employed by DEDALUS can also be used for program transformation, data abstraction, program modification and structured programming. However, the system was not intended to be used other than as a testbed for these ideas.

Details

The task is represented as a goal which can be modified by using transformations. This produces sub-goals which are handled in the same way, introducing recursion where necessary. A program, correctness proof and proof of termination (using well-founded sets) are all produced simultaneously for non-mutually-recursive programs.

Of the transformations, of which there are more than 100, some represent knowledge about the program's subject domain (for example, numbers, lists or sets); others represent the meaning of the constructs of the specification language and the target programming language; and some represent basic programming knowledge [125]. Transformations are represented as simple input and output templates, or patterns. The system has been implemented in QLISP [60].

Results

DEDALUS has only been tried on toy examples like the greatest-common-divisor and the intersection of two lists [126].

Conclusion

Since DEDALUS has only been tried on small examples, it is difficult to know whether it would work on large programs. Another drawback is the degree to which the user is expected to check the application of transformations and to hand-modify code, making any formality dependent on the user's ability. The main advantage, however, is that the system is applicable to some areas of software maintenance, since from a modified (maintained) specification, a new program can be relatively easily derived using this system. This advantage, however, has to be seen in the context of the trivial examples with which that DEDALUS has actually be used. Also, the system is not suitable for reverse engineering existing code.

A new deduction-oriented system will regard program-synthesis as a theorem-proving task that uses unification, mathematical induction and transformation rules.

A.12 Hildum and Cohen's Work

Background

This is work that was carried out by Hildum and Cohen [95].

Purpose

Although not a transformation system as such, Hildum and Cohen propose a language for writing transformations so that the transformations are applicable to a variety of programming languages.

Details

In this system, it is the user's responsibility to ensure that the transformation rules defined preserve correctness. However, Hildum and Cohen have proposed that the system could be extended so that it could present the user with a menu of appropriate transformations.

Like most other systems, transformations are expressed using two patterns [95]:

- A series of elements to be found and actions to be performed while finding these elements; and
- A new ordering of the elements that describes the result of applying the transformation.

In this system, a program to apply transformations is seen essentially as a text editor that executes a series of commands (in this case specified by the program transformation) and produces an altered version of the original text which represents the new code sequence.

The language for writing transformations [95] includes a number of important features:

- A pattern matcher for matching literals, single variable items and sequences of variable items;
- A "repeat" construct for performing a replacement several times. It provides a feature for specifying the minimum number of times that a repeated pattern must be found. (Nested repeats have not yet proved necessary.)

- A feature to prescribe "actions" to be performed on items matched by patterns (such as adding two constants) before doing the replacement;
- Multiple-pass transformations for moving a piece of code from one point to another, for example. (The first pass would find the code and the second would perform the replacement.)
- Transformations to be repeated up to a maximum number of times, for example, for unrolling the body of a loop.

Results

Since no transformation system has been written using this language, no results are available.

Conclusion

A *language* for writing transformations offers more flexibility over a catalogue of initial and final patterns and, thus, this work looks promising. However, since programs are stored as text sequences, rather than in a more structured form such as a tree, a system based on this approach would be inefficient due to the parsing overhead required. This work would be more credible if there were a working transformation system based on it.

A.13 Kozaczynski's Work

Background

Kozaczynski *et al*'s program transformation system [113] forms part of the software re-engineering program at Anderson Consulting's Center for Strategic Technology and Research (CSTaR).

Purpose

The purpose is to provide a general transformation system that can be used in software maintenance; in particular at higher, conceptual levels. The core of the work is based on concept recognition using an "ISA" hierarchy [173].

Details

The system is constructed with four levels — the text level, the syntactic level, the semantic level and the concept level — each of which has corresponding program transformations.

Text-level transformations work on the source text directly and can be performed by means of string matching (for example, using string replace in the EMACS editor). For example, the user may wish to replace all occurrences of the number 100 by 200. This can cause problems, however. First, these transformations may replace components that should have remained the same; for example, in the case of a string which also forms part of an identifier name. Second, these transformations rarely preserve the program's semantics.

Syntactic-level transformations overcome many of these problems by putting the program into the form of an abstract syntax tree and using pattern variables are used to identify matched components.

Semantic-level transformations require that semantic properties of the program be considered. For example, a loop may be executed only once, but to determine this, the semantic properties of the loop need to be analysed. Correctness-preserving transformations, such as code optimisation and restructuring, are supported at the semantic level.

Concept-level transformations are needed for software maintenance activities such as fault correction, functional enhancements and platform migration. These transformations require knowledge about abstract concepts concerning programming, problem solving and application domains. For example, the user may only wish to perform a certain transformation to sections of the program related to a particular concept or function.

Transformations are expressed as a left-side pattern, a right-side pattern and possibly some transformation conditions. They may also include calls to functions "Delete", "Replace", "Insert-Before", "Insert-After" and "Insert-Into".

Results

The system has been used in the porting of an 8,000 module COBOL system; predominantly to make changes to the interfaces. The amount of transformed code was 7–50 lines per 1,000 lines [67].

Conclusion

From the example given of the use of this tool, it is clear that it really performs very little transformation of the code — at most 5% is changed. Also, formal features such as correctness and completeness have been omitted. Thus, while it has worthwhile goals, notably in the area of concept recognition, it is not a viable general purpose transformation system.

A.14 Ward's Work

Background

This work was conducted by Ward in his D.Phil. thesis [177].

Purpose

Ward, in his thesis, develops a theory of program refinement and equivalence, based on a wide spectrum language, which can be used as develop practical tools for program development and modification. This can be achieved by implementing the refinements and equivalences as transformations within some suitable system. However, this has not yet been done.

Details

The theory is based on the use of a wide spectrum language which is defined in terms of an imperative kernel language of atomic specification statements. This is extended using definitional transformations to define new concepts in terms of those already present. The resulting language, known as "WSL", covers the whole range of operations from general specifications to assignments, jumps and labels, and expressions with side effects.

Program equivalence is proved in one of two different ways: either by using the denotational semantics of WSL directly, or by using the method of weakest preconditions, expressed as formulae within a framework of first order infinitary logic [177]. The means by which this is accomplished is explained in more detail in Chapter 4.

The theorems which Ward has proved in his thesis [177], and which form a foundation for building a transformation system, cover a wide range of areas. These include the following:

- Theorems on proving the termination of recursive and iterative programs;
- The recursive implementation of specifications, enabling the transformation of general specifications into programs;
- A rigorous framework for reasoning about programs with nested loops, terminated by Exit statements;

- Selective and entire unrolling of loops;
- Selective folding and unfolding of procedures, which form the basis for a rigorous treatment of "action systems" [9] [8] — parameterless, recursive procedures which can be used, among other things, as the equivalent of Goto statements;
- A wide range of theorems and techniques for recursion removal; and
- Techniques which use the theorems above for deriving algorithms from specifications, and for obtaining specifications from existing programs.

More details are given in Chapter 4.

Results

These theorems and techniques have successfully been applied by hand to a number of examples of varying complexity [176] [175] [179] [196]. These have included the "greatest true square" problem, topological sorting and some real-world programs translated into WSL from assembler.

Conclusion

Ward's work provides a rigorous and formal basis on which a transformation system could be based. Although Ward did not develop such a system, this is by no means an indication of the impracticality of such a system.

One of the motivating aims of Ward's work [177] was to develop a theory which can be applied to any program written using any methods. This is so that the theory can be applied to the development of practical systems for software maintenance as well as for the development of programs from specifications. Thus, if a system could be constructed based on these transformations, it would have both the formal foundations and the scope of application of the kind of system that has been identified as being useful.

A.15 Other Work on Program Transformations

The above survey of transformation systems is certainly not exhaustive. Other systems include ALICE [59] which aimed to develop a complete programming environment, using HOPE [51], for a highly parallel graph reduction machine; ΦNIX [22] which is an automatic programming system for writing programs which interact with external devices through temporally-ordered streams of values; REFINE [112] which uses transformations for program analysis and testing; KIDS [6] which uses transformations to express machine-independent optimisations in a compiler for a purely functional parallel language; and Fradet and Le Métayer's [77] work which uses transformations in the compilation of functional languages. Both Lu [123], and Yang and Choo [195] use transformation methods in the compilation of parallel languages. Work which is of more relevance to software maintenance includes that carried out by Arango *et al* [5], Keller [107] and Overstreet *et al* [147].

Appendix B

A Syntax Table for WSL and *METAWSL*

The following table summarises the syntax of the LISP form of *META*WSL. In the table are the following entries:

Number — This is the type number that is passed to the pretty-printer as a more efficient alternative to passing the actual type of the object. (Its use is described in Chapter 8.)

Name — This is the name of the item.

Generic Type — This is the class of program item to which the named item belongs. For example, Skip is a type of Statement and a Number is a type of Expression.

Leading Token — This is if and only if the type of the item is the first part of the printed form, otherwise it is "No". For example, as @When statement begins with the word "@When", but an assignment does not begin with the word Assignment (or any other word).

Minimum Size — This is the smallest number of components that the type can have. Examples are an assignment which must have at least two (in fact only two) components and a For loop which must have at least five components, whereas a list of variables can contain any number of variables.

Component Types — This gives the types of components of the given type (if there are any). For example, the components of an assignment are a variable and and expression. If there is an unlimited number of components for a given item, indicated by this entry finishing with "...", then any additional components have the same type as the last component. For example, a While statement must have a condition as its first argument followed by any number of statements.

The table also includes information about the generic types Statement, Expression and so on, as well as the most generic type, Thing. These are not part of WSL itself but are present in the table in order to simplify the system's implementation.

Num	Name	Generic Type	Leading Token	Min Size	Component Types
1	Thing	—	No	0	—
2	A_List	Thing	No	0	Thing ...
3	Symbol	Thing	No	0	—
4	Name	Thing	No	0	—
5	Statement	Thing	Yes	0	—
6	Expression	Thing	Yes	0	Expression ...
7	Condition	Thing	Yes	0	Condition ...
8	Assignment	Thing	No	2	Assd_Var Expression
9	Guarded	Thing	No	2	Condition Statement ...
10	Action	Thing	No	2	Name Statement ...
11	Definition	Thing	Yes	0	Name Variables Variables Statement ...
12	$Statement$	Statement	Yes	0	—
13	$Expn$	Expression	Yes	0	—
14	Var	Expression	Yes	0	—
15	$Condition$	Condition	Yes	0	—
16	$Name$	Name	No	0	—
17	Statements	A_List	No	1	Statement ...
18	Expressions	A_List	No	0	Expression ...
19	Variables	A_List	No	0	Variable ...
20	Assd_Vars	A_List	No	0	Assd_Var ...
21	Assignments	A_List	No	1	Assignment ...
22	Guardeds	A_List	No	1	Guarded ...
23	Names	A_List	No	1	Name ...
24	!L	Expression	Yes	1	A_List
25	Number	Expression	No	0	—
26	String	Expression	No	0	—
27	Variable	Expression	No	0	—
28	Assd_Var	Variable	No	0	—
29	Aref	Variable	Yes	2	Variable Expression
30	Abort	Statement	Yes	0	—

Num	Name	Generic Type	Leading Token	Min Size	Component Types
31	Actions	Statement	Yes	2	Names Action ...
32	Array	Statement	Yes	2	Assd_Var Expression
33	Assert	Statement	Yes	1	Condition
34	Assign	Statement	Yes	1	Assignment ...
35	Call	Statement	Yes	2	Name Number
36	Comment	Statement	Yes	1	String
37	Cond	Statement	Yes	1	Guarded ...
38	D_If	Statement	Yes	1	Guarded ...
39	D_Do	Statement	Yes	1	Guarded ...
40	Exit	Statement	Yes	1	Number
41	Loop	Statement	Yes	1	Statement ...
42	For	Statement	Yes	5	Assd_Var Expression Expression Expression Statement ...
43	!Xp	Statement	Yes	2	Name Expressions
44	!P	Statement	Yes	3	Name Expressions Assd_Vars
45	Proc_Call	Statement	Yes	3	Name Expressions Variables
46	Skip	Statement	Yes	0	—
47	Var	Statement	Yes	2	Assignments Statement ...
48	Where	Statement	Yes	2	Statements Definition ...
49	While	Statement	Yes	2	Condition Statement ...
50	Proc	Definition	Yes	4	Name Variables Variables Statement ...
51	Funct	Definition	Yes	3	Name Variables Expression
52	B_Funct	Definition	Yes	3	Name Variables Condition
53	+	Expression	Yes	2	Expression ...
54	-	Expression	Yes	2	Expression
55	*	Expression	Yes	2	Expression ...
56	/	Expression	Yes	2	Expression
57	**	Expression	Yes	2	Expression
58	Min	Expression	Yes	2	Expression ...
59	Max	Expression	Yes	2	Expression ...
60	Div	Expression	Yes	2	Expression
61	Mod	Expression	Yes	2	Expression
62	If	Expression	Yes	3	Condition Expression
63	Funct_Call	Expression	Yes	2	Name Expressions
64	!F	Expression	Yes	2	Name Expressions
65	Gen_Expr	Expression	Yes	3	Assignments Statements Expression
66	Int	Expression	Yes	1	Expression
67	Frac	Expression	Yes	1	Expression
68	Abs	Expression	Yes	1	Expression
69	Sgn	Expression	Yes	1	Expression
70	True	Condition	No	0	—
71	False	Condition	No	0	—
72	Else	Condition	Yes	0	—
73	=	Condition	Yes	2	Expression
74	<>	Condition	Yes	2	Expression
75	<	Condition	Yes	2	Expression
76	>	Condition	Yes	2	Expression
77	<=	Condition	Yes	2	Expression
78	>=	Condition	Yes	2	Expression
79	==	Condition	Yes	2	Expression
80	Even?	Condition	Yes	1	Expression
81	Odd?	Condition	Yes	1	Expression
82	True?	Condition	Yes	1	Expression
83	False?	Condition	Yes	1	Expression
84	And	Condition	Yes	1	Condition ...
85	Or	Condition	Yes	1	Condition ...
86	Not	Condition	Yes	1	Condition
87	B_Funct_Call	Condition	Yes	2	Name Expressions
88	!C	Condition	Yes	2	Name Expressions
89	Gen_Cond	Condition	Yes	3	Assignments Statements Condition
90	Empty	Expression	Yes	0	—

Num	Name	Generic Type	Leading Token	Min Size	Component Types
91	Cons	Expression	Yes	2	Expression
92	Append	Expression	Yes	2	Expression
93	Intersection	Expression	Yes	2	Expression ...
94	Union	Expression	Yes	2	Expression ...
95	Set_Diff	Expression	Yes	2	Expression
96	List	Expression	Yes	1	Expression ...
97	Hd	Expression	Yes	1	Expression
98	Tl	Expression	Yes	1	Expression
99	Length	Expression	Yes	1	Expression
100	Reverse	Expression	Yes	1	Expression
101	Empty?	Condition	Yes	1	Expression
102	Non_Empty?	Condition	Yes	1	Expression
103	Member?	Condition	Yes	2	Expression
104	Some_Member?	Condition	Yes	2	Expression
105	Any_Member?	Condition	Yes	2	Expression
106	Subset?	Condition	Yes	2	Expression
107	Same?	Condition	Yes	2	Expression
108	Push	Statement	Yes	2	Expression Assd_Var
109	Pop	Expression	Yes	1	Assd_Var
110	[_S+_]	Expression	Yes	2	Expression ...
111	Assn_Spec	Statement	Yes	2	Assd_Vars Condition
112	Old	Variable	Yes	1	Variable
113	%N	Expression	Yes	0	—
114	%Z	Expression	Yes	0	—
115	%Q	Expression	Yes	0	—
116	%R	Expression	Yes	0	—
117	Map	Expression	Yes	4	Name Name Variable Expression
118	Reduce	Expression	Yes	4	Name Name Variable Expression
119	Set	Expression	Yes	2	Expression Condition
120	For_All	Condition	Yes	2	Variable Condition
121	Exists	Condition	Yes	2	Variables Condition
122	@Up	Statement	Yes	0	—
123	@Down	Statement	Yes	0	—
124	@Left	Statement	Yes	0	—
125	@Right	Statement	Yes	0	—
126	@To_Last	Statement	Yes	0	—
127	@Down_Last	Statement	Yes	0	—
128	@To	Statement	Yes	1	Expression
129	@Goto	Statement	Yes	1	Expression
130	@Follow	Statement	Yes	0	—
131	@Return	Statement	Yes	0	—
132	@++Span	Statement	Yes	0	—
133	@–Span	Statement	Yes	0	—
134	@Set_Span	Statement	Yes	1	Expression
135	@All_Span	Statement	Yes	0	—
136	@Span_Flagged	Statement	Yes	0	—
137	@Del	Statement	Yes	0	—
138	@Del_Back	Statement	Yes	0	—
139	@Del_Rest	Statement	Yes	0	—
140	@Undel_After	Statement	Yes	0	—
141	@Undel_Before	Statement	Yes	0	—
142	@Ins_After	Statement	Yes	1	Expression
143	@Ins_Before	Statement	Yes	1	Expression
144	@Change_To	Statement	Yes	1	Expression
145	@When	Statement	Yes	2	Number Guarded ...
146	@When_Terminal	Statement	Yes	0	Statement ...
147	@When_Terminal_0	Statement	Yes	0	Statement ...
148	@Exit_When	Statement	Yes	0	—
149	@No_Deeper	Statement	Yes	0	—
150	@Trans	Statement	Yes	1	Name Expression ...

Num	Name	Generic Type	Leading Token	Min Size	Component Types
151	@Pass	Statement	Yes	0	—
152	@Fail	Statement	Yes	0	—
153	@Mark	Statement	Yes	0	—
154	@Undo	Statement	Yes	0	—
155	@Reposition	Statement	Yes	0	—
156	@Drop	Statement	Yes	0	—
157	@Wrong	Statement	Yes	0	String ...
158	[_Up?_]	Condition	Yes	0	—
159	[_Down?_]	Condition	Yes	0	—
160	[_Left?_]	Condition	Yes	0	—
161	[_Right?_]	Condition	Yes	0	—
162	[_With_Else?_]	Condition	Yes	0	—
163	[_Size_]	Expression	Yes	1	Expression
164	[_Body_]	Expression	Yes	1	Expression
165	[_Comps_]	Expression	Yes	1	Expression
166	[_Contents_]	Expression	Yes	1	Expression
167	[_All_Contents_]	Expression	Yes	1	Expression
168	[_Variables_]	Expression	Yes	1	Expression
169	[_Used_]	Expression	Yes	1	Expression
170	[_Assigned_]	Expression	Yes	1	Expression
171	[_Used_Only_]	Expression	Yes	1	Expression
172	[_Assd_Only_]	Expression	Yes	1	Expression
173	[_Assd_To_Self_]	Expression	Yes	1	Expression
174	[_Statements_]	Expression	Yes	1	Expression
175	[_Calls_]	Expression	Yes	1	Expression
176	[_Total_Size_]	Expression	Yes	1	Expression
177	[_Depth_]	Expression	Yes	2	Expression
178	[_Terminal_Value_]	Expression	Yes	2	Expression
179	[_Arguments_]	Expression	Yes	2	Expression ...
180	[_Occ_]	Expression	Yes	2	Expression ...
181	[_Diff_]	Expression	Yes	2	Expression ...
182	[_Replace_]	Expression	Yes	3	Expression
183	[_Rplc_All_]	Expression	Yes	2	Expression
184	[_Isolate_]	Expression	Yes	3	Expression
185	[_Number?_]	Condition	Yes	1	Expression
186	[_Variable?_]	Condition	Yes	1	Expression
187	[_Syntax?_]	Condition	Yes	2	Name Expression
188	[_S_Type?_]	Condition	Yes	1	Symbol ...
189	[_G_Type?_]	Condition	Yes	1	Symbol ...
190	[_P_Type?_]	Condition	Yes	1	Symbol ...
191	[_Primitive?_]	Condition	Yes	1	Expression
192	[_Reducible?_]	Condition	Yes	1	Expression
193	[_Proper?_]	Condition	Yes	1	Expression
194	[_Improper?_]	Condition	Yes	1	Expression
195	[_Regular?_]	Condition	Yes	1	Expression
196	[_Regular_System?_]	Condition	Yes	1	Expression
197	[_Dummy?_]	Condition	Yes	1	Expression
198	[_Calls_Terminal?_]	Condition	Yes	1	Expression
199	[_Terminal?_]	Condition	Yes	2	Expression
200	[_Trans?_]	Condition	Yes	1	Name
201	Pattern	Expression	Yes	0	Expression ...
202	~?~	Pattern	Yes	0	—
203	~*~	Pattern	Yes	0	—
204	~?*~	Pattern	Yes	0	—
205	~>?~	Pattern	Yes	1	Symbol
206	~>*~	Pattern	Yes	1	Symbol
207	~>?*~	Pattern	Yes	1	Symbol
208	~<?~	Pattern	Yes	1	Symbol
209	~<*~	Pattern	Yes	1	Symbol
210	~<Se~	Pattern	Yes	1	Pattern

Num	Name	Generic Type	Leading Token	Min Size	Component Types
211	~<Sc~	Pattern	Yes	1	Pattern
212	~Or~	Pattern	Yes	2	Pattern ...
213	[_Put_]	Expression	Yes	3	Symbol Expression Expression
214	[_Get_]	Expression	Yes	2	Symbol Expression
215	[_Val_]	Expression	Yes	2	Symbol Expression
216	[_Check?_]	Condition	Yes	2	Symbol Pattern
217	[_Match_]	Expression	Yes	3	Symbol Pattern Expression
218	[_Fill_In_]	Expression	Yes	3	Symbol Pattern Expression
219	[_Fill_Args_]	Expression	Yes	3	Symbol Pattern Expression
220	[_And_]	Expression	Yes	2	Expression ...
221	[_Or_]	Expression	Yes	2	Expression ...
222	[_Not_]	Expression	Yes	1	Expression
223	[_->T?_]	Condition	Yes	2	Expression
224	[_->F?_]	Condition	Yes	2	Expression
225	[_Simplify_]	Expression	Yes	1	Expression
226	[_Simplex?_]	Condition	Yes	1	Expression
227	[_Increment_]	Expression	Yes	2	Expression
228	[_Decrement_]	Expression	Yes	1	Expression

Appendix C

$\mathcal{META}$WSL in Detail

C.1 Introduction

This appendix gives the specific details of $\mathcal{META}$WSL.

C.2 Predefined Variables

C.2.1 %Program%

This variable holds the whole program that is currently being transformed. The program is stored in a particular, efficient, way (which is described in Chapter 8). The variable should, therefore, not be used except as a parameter to any of the statements and functions in this chapter that require a piece of WSL as an argument. For example, one could write ([_Total_Size_] %Program%) to determine the number of nodes in the program tree, but *not* (Hd %Program%) to obtain the first node.

C.2.2 %Item%

This variable holds the currently selected syntactic program item. Like the variable %Program% it is should only be used as a parameter to any of the METAWSL statements and functions that require a piece of WSL as an argument.

C.2.3 %Posn%

This variable holds the position, relative to the root node, of the currently selected program item as described in Chapter 6.

C.2.4 %Length%

This variable holds the number of components of the current item's parent node.

C.2.5 %Data%

This variable holds any input to the transformation that was provided by the user.

C.3 Statements for Movement in the Program Tree

C.3.1 @Up

This METAWSL statement moves up through the program structure.

C.3.2 @Down

This METAWSL statement moves down through the program structure to the first component of the current item.

C.3.3 @Down_Last

This METAWSL statement moves down through the program structure to the last component of the current item.

C.3.4 @Left

This METAWSL statement moves left through the program structure.

C.3.5 @Right

This METAWSL statement moves right through the program structure.

C.3.6 @To_Last

This METAWSL statement moves to the last item at the current level in the program structure.

C.3.7 @To

This METAWSL statement moves to the nth component at the current level in the program, where n is the statement's argument, i.e. the nth component of the parent node of the current item.

C.3.8 @Goto

This *MetaWSL* statement moves to an arbitrary position in the program tree. This function is rarely be used in coding transformations, its main use being when the transformation has moved to some distant point in the tree and needs to return to the original position quickly. In the example below, the original position could be stored in a variable and jumped to with @Goto.

```
(Var ((P %Posn%))
     (Loop  ⋮
           (Cond (([_Left?_]) (@Left))
                 (([_@Up_]) (@Up))
                 ((Else) (Exit 1)))
             ⋮
              )
     (@Goto P)).
```

C.3.9 @Follow and @Return

It is frequently necessary, when a procedure call or function call is the currently selected item, to move to the corresponding definition. The @Follow *MetaWSL* statement moves to the definition of the currently selected Proc_Call, Funct_Call or B_Funct_Call. For example, with the program

```
((Where (...(Proc_Call P () ())...)
     (Proc P () () ...)))
```

and the Proc_Call selected, executing the statement @Follow would move to move to the definition of P.

When a call is followed, the global variable %Followed% is updated. This variable holds a list of the names of the definitions that are currently being followed with @Follow statements. Since *MetaWSL* programs can access the value of this variable, it allows transformations to check whether a call has already been followed. Thus transformations can be written which prevent the system getting trapped in an infinite loop while checking recursive calls.

It is possible to return to the place from which the @Follow was executed by means of the @Return statement. When a @Return is executed, the name which had been added to the variable %Followed% is removed.

C.4 Functions for Testing for Valid Movements

C.4.1 [_Up?_]

This *MetA*WSL returns true if and only if it is valid to move up through the program structure.

C.4.2 [_Down?_]

This *MetA*WSL returns true if and only if it is valid to move down through the program structure to the first component of the current item.

C.4.3 [_Left?_]

This *MetA*WSL returns true if and only if it is valid to move left through the program structure.

C.4.4 [_Right?_]

This *MetA*WSL returns true if and only if it is valid to move right through the program structure.

C.5 Statements and Variables Relating to Spans

C.5.1 @Inc_Span

This *MetA*WSL statement increases the number of items in the current span by one.

C.5.2 @Dec_Span

This *MetA*WSL statement decreases the number of items in the current span by one.

C.5.3 @Set_Span

This *MetA*WSL statement sets the span of items in the current sequence to include the number of items given by the statement's argument.

C.5.4 @All_Span

This *MetA*WSL statement increases the span of items in the current sequence so as to extend as far to the right as possible. Thus it will it include all the rest of the items in current structure and at the current level.

C.5.5 %Span%

When a sequence of items has been selected as a span, this variable holds the number of items in that span.

C.5.6 %Items%

When a sequence of items has been selected as a span, this variable holds a list of those items in the span.

C.6 Editing Statements

C.6.1 @Del

This *MetA*WSL statement deletes the item at the current position. The last deleted object of is stored so that it can be undeleted later at a different point in the program using one of the undelete *MetA*WSL statements.

C.6.2 @Del_Back

This *MetA*WSL statement deletes the item at the current position and moves back to the previous item if there was one. Thus the current position always moves to the left (unless that is not possible, in which case it either stays at the same point or moves up) unlike the case with the @Delete statement for which the current position always stays the same (unless that is not possible in which case it moves either to the left or up).

The last deleted object is stored so that it can be undeleted later at a different point in the program with one of the undelete *MetA*WSL statements.

C.6.3 @Del_Rest

This *MetA*WSL statement deletes all the items in the current branch after (but not including) the item at the current position.

C.6.4 @UnDel_Before and @UnDel_After

These *MetaWSL* statements insert into the program tree a copy of the last thing which was deleted with a *MetaWSL* deletion statement.

C.6.5 @Ins_Before and @Ins_After

These *MetaWSL* statements insert some code into the program tree. The argument can either be a single item or a list of several items.

C.6.6 @Change_To

This *MetaWSL* statement changes the current item in the program by replacing it with a new item. The argument to this statement must be a single program item, which would probably be created using the [_Fill_In_] function.

C.7 Statements for Repeating an Operation at Different Nodes

C.7.1 @When

This *MetaWSL* statement performs some actions at each item, within a program item, which meets any of a set of given criteria.

The statement takes a numeric argument, followed by a number of guarded clauses. The current item is searched to find each place at which a test at the start of one of the guards is true, and the corresponding set of statements is executed at that point. If the numeric argument is non-zero then, after the statements have been executed, the new current item is in turn searched for more places where one of the conditions is true. If the numeric argument is zero, then the other items on the same level (and their components) are considered, but not items inside the item which has just been acted upon.

The following example changes all the Comment statements within a particular section of code into Skip statements. It is assumed that the variable S holds a WSL Skip statement. Note also that the [_S_Type?_] function (which is described later) tests the type of the selected item. Finally, Comment statements do not include other Comment statements, so a numeric option of zero is used.

```
(@When  0  (([_S_Type?_] Comment) (@Change_To S))).
```

The second example reverses the order of the first two components of all additions within the current item. It uses the @Del statement to delete the first component of the +, and this leaves the second component selected so that the @UnDel_After statement reinserts this expression. A numeric argument of one is used since an addition may contain other additions as components.

```
(@When  1  ((And ([_S_Type?_] +) (>= ([_Size_] %Item%) 2))
            (@Down)
            (@Del)
            (@UnDel_After)
            (@Up)))
```

Within a @When statement it is possible to access two extra variables. These variables are: %Top% which hold the item from which execution of the @When statement started; and %Offset% which holds the offset of the current item relative to the position of the item from which execution of the @When statement started.

It is also desirable to be able to leave prematurely a @When statement or to abandon looking inside a particular node. These are done by the statements @Exit_When and @No_Deeper which are described in more detail later.

C.7.2 @When_Terminal

This MetaWSL statement performs a set of actions on each terminal statement[1] of the current item within that item.

Since calculating whether an item is terminal within some containing item involves walking down through the program tree, for @When queries which have this as their only test, writing an explicit @When using the context variables %Top% and %Offset% would be very inefficient. Thus, there is a separate statement, @When_Terminal, specially designed for situations in which the only test is for terminal statements.

Rather than taking guards as its arguments, the statement takes a sequence of statements to perform when a terminal statement is found. Also a terminal statement will not include another terminal statement so this is effectively a @When with a numeric option of zero.

C.7.3 @When_Terminal_0

This MetaWSL statement performs a set of actions on each terminal statement of the current item whose terminal value is zero within that item. It is very similar to the MetaWSL statement above.

C.7.4 @Exit_When

This MetaWSL statement causes the immediate termination of a @When, @When_Terminal or @When_Terminal_0 statement and a return to the position in the program from which the repetition statement began. An example of the use of this statement is given in the next section.

[1] A statement is a terminal statement of some structure if it could be the last statement to be executed as part of that structure.

C.7.5 @No_Deeper

This MetaWSL statement prevents a @When, @When_Terminal or @When_Terminal_0 construct from looking any further down the current branch of the program tree.

The following example fragment of MetaWSL code sets the variable OK to 1 if there are more than two Call statements in the current program item, and to 0 otherwise. It uses @No_Deeper to avoid searching in branches which do not contain Call statements and @Exit_When to quit when more than two Call statements have been found.

```
(Assign (OK 0))
(Var ((Counter 0))
   (@When 0 ((Not (Member (!L Call) ([_Statements_] %Item%)))
              (@No_Deeper))
             (([_S_Type?_] Call)
               (Assign (Counter (+ Counter 1)))
               (Cond ((> Counter 2)
                       (Assign (OK 1))
                       (@Exit_When)))))).
```

C.8 Other MetaWSL Statements

C.8.1 @Pass and @Fail

These MetaWSL statements set the applicability condition to true of false, respectively. The following example is a simple applicability condition which illustrates the use of these statements.

```
(Cond ((And ([_S_Type?_] +) (>= ([_Size_] %Item%) 2))
        (@Pass))
      ((Else)
        (@Fail)))
```

C.8.2 @Wrong

This *MetaWSL* statement displays an error message caused because a transformation has aborted during the program modification stage (without changing anything if it has been implemented correctly). The error could have been due, for example, to incorrect data that has been provided by the user and which could not, therefore, have been tested earlier. For example, a transformation might include the following statement:

```
(Cond (([_Syntax?_] %Data% Statement)
                :
             Perform the transformation
                :
                  )
          ((Else)
            (@Wrong "The input did not have the syntax of a statement."))).
```

C.8.3 @Mark

A .transformation may attempt to make some changes, only to find that they do not work, or are not suitable, and goes back to an earlier version of the program so as to try another course of action. The @Mark statement records the version of the program to which the transformation may have to revert with one of the following statements. An example of the use of this statement is given later.

C.8.4 @Reposition

This *MetaWSL* statement moves back to the position in the program, but not the version of the program, that was current when the last @Mark was executed. There is no checking that a @Mark has actually been executed — this is left to the implementor of the transformations.

C.8.5 @Undo

This is the MetaWSL statement to undo any edits that have been made to a program since the last @Mark was executed. There is no checking that a @Mark has actually been executed — this is left to the implementor of the transformations. An example of the use of this statement is given in the next section.

C.8.6 @Drop

This MetaWSL statement removes a previous version of a program from the list of versions created with @Mark. Thus, if two @Mark statements have been executed and the transformation needs to revert to the state of the program at the first one, then before it executes an @Undo it must execute a @Drop. As before, there is no error checking that a @Mark has actually been executed — this is left to the implementor of the transformations.

In the following example, the transformation makes some changes to a program repeatedly, until the changes no longer cause the program to reduce in size. At that point, the program reverts to the last version after which it had reduced in size. The @Drop statement ensures that unneeded program versions are discarded.

```
(Loop (Var ((S ([_Total_Size_] %Program%)))
           (@Mark)
             ⋮
           Perform some changes to the program
             ⋮
           (Cond ((< ([_Total_Size_] %Program%) S)
                    (@Drop))
                 ((Else)
                    (@Undo)
                    (Exit 1)))))
```

C.9 Pattern Matching and Template Filling

C.9.1 [_Check?_]

This MetaWSL condition takes two arguments, a generic type (such as Statement) and a pattern, and returns true if and only if the current item is of the correct type and also matches the given pattern. If any tokens are used, then their values cannot be determined later since this is a Boolean function and does not return an association table.

For example, in MetaWSL a transformation might be written as:

```
(Cond (([_Check?_] Statement  (Assign ((~?~) (~?~))) )
         "Do some action")
      ((Else)
         "Do some other action")).
```

C.9.2 [_Match_]

This MetaWSL function takes three arguments, a type (such as Statement) a pattern and an association table (to which to add the result of the match), and returns the result of pattern matching the current item with the given pattern. If the current item is of the incorrect type, then an empty table is returned. The matches are added to the association table which is given as the function's third argument. An example is given in Section C.9.3.

C.9.3 [_Fill_In_]

This MetaWSL function takes three arguments, a type (such as Statement) a template and an association table (from which to get the values of any tokens), and returns the result of filling a template by replacing tokens with values taken from the association table. For example, a transformation for exchanging the arms of a Cond statement could be written in MetaWSL as:

```
(Assign (Table ([_Match_] Statement
                          (Cond ((~>?~ B) (~>*~ S1))
                                ((Else) (~>*~ S2)))
                           Empty)))
(@Change_To ([_Fill_In_] Statement
                         (Cond ((Not (~<?~ B)) (~<*~ S2))
                               ((Else) (~<*~ S1)))
                         Table)).
```

C.9.4 [_Fill_Args_]

This MetaWSL function is similar to the previous one, except that it returns just the arguments of the template in which tokens have been replaced by values taken from another table. This function could, for example, produce a list of statements from an item of type Statements.

C.10 Functions for Association Tables

C.10.1 [_Put_]

This MetaWSL function takes as its arguments a token name, a value and an association table, and returns the table with the extra value added under the given token name. For example, a transformation might include the following code:

```
(Assign (T1 ([_Put_] S %Item% Empty)))
```

which would put the current item into an empty table under the token label S. The resulting table would be stored in a variable called T1.

C.10.2 [_Get_]

This $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL function takes as its arguments a token name and an association table, and returns the value of the given token, as found in the table.

C.10.3 [_Val_]

This $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL function takes as its arguments a token name and an association table, and returns the "contents" (i.e. the name) of the item stored under the given token, as found in the table.

C.11 Functions for Examining the Program being Transformed

C.11.1 [_With_Else?_]

This $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL function returns true if and only if the given item is a Cond statement with an Else clause. Although this information could be determined by a pattern match, it is information that is often required and this function provides it more efficiently.

C.11.2 [_Size_]

This $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL function returns the size, i.e. the number of components, of the given item.

C.11.3 [_Comps_]

This MetaWSL function returns a list of the components of the given item. For example, given the statement

```
(Assign (X (+ A B)) (Y 0))
```

the function would return the list of two assignments:

```
((X (+ A B)) (Y 0)).
```

C.11.4 [_Contents_]

This MetaWSL function returns the name of a leaf item in a program, for example, the name of a variable.

C.11.5 [_All_Contents_]

This is the MetaWSL function which takes as its argument a WSL item that represents a list of leaf items and returns a list of atoms representing the names of the "leaf items". For example, given the first component of an action system, the list of possible starting actions, it would return a list of atoms corresponding to the names of the possible starting actions.

C.11.6 [_Statements_]

This MetaWSL function returns a list of all the types of statement which are used within the given piece of code. For example, given the item:

```
(Cond ((= A B) (Assign (X 1)))
      ((Else) (Assign (Y 2)) (Skip)))
```

the function would return the list (Cond Assign Skip). The order of the list is unspecified.

C.11.7 [_Calls_]

This MetaWSL function returns a list of all the actions which are called from within a given piece of code, together with the number of times each action is called. For example, given the item:

```
(Cond ((= A B) (Call P 0))
      ((= C D) (Call Q 0))
      ((Else) (Call P 0)))
```

the function would return the list ((P 2) (Q 1)). The order of the list is unspecified.

C.11.8 [_Total_Size_]

This MetaWSL function returns the total number of nodes and keywords within a given piece of code.

C.11.9 [_Body_]

This MetaWSL function returns the statements which form part of the given item. The result is a WSL item of type Statements. If the item contains no item of type of type Statement then the function returns the empty list. The function does not look inside components to find statements, so the [_Body_] of a Cond statement would be the empty list.

For example, given the statement,

```
(For J 0 10 2 (Assign (X (+ X J))) (Skip) (Assign (Y (* Y X))))
```

the function would return the WSL Statements item,

```
((Assign (X (+ X J))) (Skip) (Assign (Y (* Y X)))).
```

C.12 Functions Relating to Variable Usage

C.12.1 [_Variables_]

This $\mathcal{META}$WSL function returns a list of all the variables in the current item.

C.12.2 [_Used_]

This $\mathcal{META}$WSL function returns a list of all the variables "used" (i.e. referred to, but not necessarily assigned to) in the current item.

C.12.3 [_Assigned_]

This $\mathcal{META}$WSL function returns a list of all the variables assigned to in the current item.

C.12.4 [_Used_Only_]

This $\mathcal{META}$WSL function returns a list of all the variables "used", but definitely not assigned to, in the current item.

C.12.5 [_Assd_Only_]

This MetaWSL function returns a list of all the variables assigned to, but not used otherwise, in the current item.

C.12.6 [_Assd_To_Self_]

This MetaWSL function returns a list of all the variables in the current item that are only "used" in assignments to themselves.

Given the follow WSL statement,

```
(Cond ((= A B) (Assign (X 1) (A 1)))
      ((Else) (Assign (Y (+ Y 1)))))
```

the functions above would return the following results:

[_Variables_]	would return	(A B X Y),
[_Used_]	would return	(A B Y),
[_Assigned_]	would return	(A X Y),
[_Used_Only_]	would return	(B),
[_Assd_Only_]	would return	(X),
[_Assd_To_Self_]	would return	(X Y).

C.13 Functions for Testing Types and Syntax

C.13.1 [_Number?_]

This MetaWSL function returns true if and only if the given WSL item represents a number.

C.13.2 [_Variable?_]

This MetaWSL function returns true if and only if the given WSL item represents a variable.

C.13.3 [_Syntax?_]

This MetaWSL function takes two arguments and returns true if and only if its first argument has the syntax indicated by its second argument. The primary use of this function would be to test the syntax of input provide by the user (and held in the variable %Data%). The MetaWSL code:

```
([_Syntax?_] %Data% Statement)
```

would return true if and only if the user had supplied a syntactically correct statement.

C.13.4 [_S_Type?_]

This MetaWSL function takes a list of types and returns true if and only if the specific type of the current item matches any of the indicated types.

C.13.5 [_G_Type?_]

This MetaWSL function takes a list of types and returns true if and returns true if and only if the generic type of the current item matches any of the indicated types.

C.13.6 [_P_Type?_]

This MetaWSL function takes a list of types and returns true if and returns true if and only if the specific type of the parent of the current item matches any of the indicated types.

C.13.7 ==

This MetaWSL function returns true if and only if its two arguments are identical or represent the same piece of WSL code.

C.14 Functions Relating to Loops

C.14.1 [_Primitive?_]

This MetaWSL function returns true if and only if the given item is a "primitive" statement.

Here a primitive statement is defined to be one which cannot be terminated from within by an Exit statement. Thus a primitive statement is any statement except one of the following: Cond, D_If or Loop.

C.14.2 [_Depth_]

This MetaWSL function returns the "depth" of an item within a structure. It takes as its arguments the top-level item and the relative position of the lower item. The relative position is of the same form as the variable %Posn% (see Chapter 6), but differs in that here the empty list represents the given item (not the whole program) and positions are calculated relative to that.

The function returns the number of Loops within the given item, that enclose the item at the given relative position is inside. (If the item is itself inside a number of Loops then these are not counted.)

C.14.3 [_Terminal_Value_]

This MetaWSL function returns the "terminal value" of a statement within a structure. It takes as its arguments the top-level item and the relative position of the lower item.

The terminal value is defined as the number of Loops, outside the enclosing item, that the statement at the relative position would leave. In other words, it is the "exit value" of an Exit or Call statement (or zero for other statements) less the depth of the item.

C.14.4 [_Terminal?_]

This MetaWSL function returns the true if and only if the indicated statement is a terminal statement within a structure. It takes as its arguments the top-level item and the relative position of the lower item.

A statement is terminal if it is in a terminal position or causes the termination of an Loop which is in a terminal position.

C.14.5 [_Reducible?_]

This MetaWSL function returns true if and only if the given item is a "reducible".

Here a statement S is defined to reducible if replacing any terminal statement of the form (Exit K) or (Call N K), which has terminal value one, by (Exit K-1) or (Call N K-1), respectively, gives a terminal statement of S.

C.14.6 [_Proper?_]

This MetaWSL function returns true if and only if the given item is a "proper".

Here a statement S is defined to be proper if every terminal statement of S has terminal value zero. In other words, control flow will *never* leave the statement by means of an exit out of an Loop.

C.14.7 [_Improper?_]

This MetaWSL function returns true if and only if the given item is a "improper".

Here a statement S is defined to be improper if every terminal statement of S has terminal value greater than zero. In other words, control flow will *always* leave the statement by means of an exit out of an Loop. (A piece of code which is not "proper" is not necessarily "improper".)

C.14.8 [_Dummy?_]

This MetaWSL function returns true if and only if the given item is a dummy loop.

Here a statement S is defined to be a dummy loop if every terminal statement of S has terminal value greater than zero, and S is also reducible.

C.15 Functions for Testing Action Systems

C.15.1 [_Regular?_]

This MetaWSL function returns true if and only if the given item is a "regular".

Here a program item is defined as regular if every execution of that item leads to an action Call. The item need not be a statement, it could be a guard or an action, for example. A regular action system is not regular in this sense since its execution finishes normally; on termination of the action system, the next statement to be executed is the one which follows it.

C.15.2 [_Regular_System?_]

This $\mathcal{M}\varepsilon\mathcal{TA}$WSL function returns true if and only if the given item is a "regular" action system, i.e. one in which every action is regular.

C.15.3 [_Calls_Terminal?_]

This $\mathcal{M}\varepsilon\mathcal{TA}$WSL function returns true if and only if all the Call statements in the the given item are in terminal positions.

C.16 Functions for Symbolic Mathematics and Logic

C.16.1 [_And_]

This is the $\mathcal{M}\varepsilon\mathcal{TA}$WSL function for symbolic logical conjunction. It takes as its arguments two or more pieces of WSL code which represent conditions, and returns a new piece of WSL which represents their conjunction, but in so doing it makes any simplifications as necessary.

C.16.2 [_Or_]

This is the *MετA*WSL function for symbolic logical disjunction. It works in a similar way to the function [_And_], taking two or more WSL conditions and returning a new condition.

C.16.3 [_Not_]

This is the *MετA*WSL function for symbolic logical negation. It takes a single argument which is a piece of WSL code which represents a condition, and returns a new piece of WSL which represents its negation, but in so doing it makes any simplifications as necessary.

C.16.4 [_->T?_]

This *MετA*WSL function takes as its arguments two pieces of WSL code which represent conditions. It then determines whether its first argument (an assertion) implies the logical truth of its second (a test), and it returns true of false. For example, (= a 0) logically implies that (< a 5) is true, whereas (<> a 0) does not logically imply that (<> a 1) is true.

C.16.5 [_->F?_]

This *MετA*WSL function takes as its arguments two pieces of WSL code which represent conditions. It then determines whether its first argument (an assertion) implies the logical falsehood of its second (a test), and it returns true of false. For example, (> a 1) logically implies that (< a 0) is false, whereas (<> a 0) does not logically imply that (<> a 1) is false.

C.16.6 [_Simplify_]

This *MetA*WSL function returns a simplified version of the given expression or condition (and flags the expression or condition internally as having been simplified).

C.16.7 [_Simplex?_]

This *MetA*WSL function returns true if and only if the expression has been simplified (as determined by the internal flag).

C.16.8 [_Isolate_]

This function takes three arguments, which represent a variable, the left-hand-side of an expression and the right-hand-side of as expression. The function returns an expression which represents the result of isolating the given variable from the expression (Left = Right). For example, with the left expression as Y, the right expression as (+ X 1), then isolating X would return (- Y 1).

C.17 Other Sundry *MetA*WSL Functions

In addition to the *MetA*WSL functions already described, there are some which do not fit into any particular category. These are described below.

C.17.1 [_Replace_]

This is the *MetA*WSL function which replaces all the occurrences of its first argument which appear in its second argument with its third argument. For example, if the variables P, Q and R held the the WSL expressions X, (* X (+ A B)) and Y respectively, then

```
([_Replace_] P Q R)
```

would return the WSL expression (* Y (+ A Y)).

C.17.2 [_Rplc_All_]

This MetaWSL function takes two arguments: a piece of WSL code and some replacements in the form of a list of pairs each consisting of an old value and a new value. It returns a piece of WSL in which the indicated replacements have been made. For example, if the variable P held the WSL expression (* X (+ A X)) and the variable R held the list of pairs ((X Y) (A B)), then the function

```
([_Replace_] P R)
```

would return the WSL expression (* Y (+ B Y)).

C.17.3 [_Arguments_]

This is the MetaWSL function which takes as its arguments the name of an operation and a WSL program item. It returns a list containing one element for each occurrence of the operation within the item. Each element in the list is a list of the arguments of that occurrence of the operation. For example, with the operation + and the program item

```
(Assign (X (+ A B)) (Y (* C (+ D E))))
```

the function would return the list ((A B) (D E)).

C.17.4 [_Occ_]

This MetaWSL function returns the number of occurrences of its first argument within its second. For example, given the two arguments: A and the program item

```
(Assign (X (+ A B)) (Y (* A (+ C D))))
```

the function would return 2.

C.17.5 [_Diff_]

This MetaWSL function uses a unification algorithm to return a list of the differences between its two arguments, provided that the differences are just atomic. i.e. it returns the list of replacements that would need to be made (using the function [_Rplc_All_]) in the first argument to give its second argument. These replacements would have to be consistent, and if there is no suitable replacement, the function returns the string "Fail". For example, if the variables E1 and E2 held the WSL expressions (* (+ A B) (- A B)) and (* (+ X Y) (- X Y)) respectively, then the function

```
([_Diff_] E1 E2)
```

would return the list ((A X) (B Y)) meaning that in the second argument A has been replaced by X, and B has been replaced by Y. However,

```
([_Diff_] '(* (+ A B) (- A B)) '(* (+ X Y) (- Y X)))
```

would return the string "Fail" since there is no possible replacement that can be made.

C.17.6 [_Increment_]

This is the *MetA*WSL function which when given an item of the type Statements and a number, returns the sequence of statements incremented the given number of times. The definition of incrementing a sequence of statements is given in the papers by Ward [177] [174].

C.17.7 [_Decrement_]

This is the *MetA*WSL function which when given an item of the type Statements and a number, returns the sequence of statements decremented the given number of times. The definition of decrementing a sequence of statements is given in the papers by Ward [177] [174].

C.18 Calling other Transformations

C.18.1 [_Trans?_]

This *MetA*WSL function returns true if and only if the named transformation is applicable at the current point in the program.

C.18.2 @Trans

This *MetA*WSL statement performs a named transformation without testing the transformation's applicability. However, this statement can be combined with the previous function to ensure that only applicable transformations are performed. For example, a transformation might include the following:

```
(Cond (([_Trans?_] Delete_Item) (@Trans Delete_Item))).
```

Appendix D

The Transformation for Collapsing an Action System

The following *MetaWSL* is the code for performing a "Collapse_Action_System" transformation.

```
((Comment "Most of this transformation assumes that the action system in
            question is regular, so if it is not, regularise it.")

 (Cond ((Not ([_Regular_System?_] %Item%))
          (@Trans Simplify_Non_Regular_Action_System)))

 (Cond
   (([_Trans?_] Remove_Action_System)
        (Comment "If we can remove the action system by virtue of it only having
                   one action, then we do that.")
        (@Trans Remove_Action_System))

    ((Else)
       (@Down)
       (Var
        ((Start_Call %Comp_1%)
         (Name Empty)
         (Names Empty)
         (Temp_Num 0)
         (Best_Name Empty)
         (Best_Num 0)
```

```
           (Best_Posn 0)
           (Fill Empty)
           (Fixed_Fill Empty)
           (Value 0)
           (T_Calls Empty)
           (Calls Empty))

          (@Up)
          (Loop (Comment "If we have only one action then we can stop...")
                (Cond ((= ([_Size_] %Item%) 2) (Exit 1)))
                (Comment "otherwise we try to do some simplification.")
                (@Trans Merge_Action_Calls)
                (Comment "If we have only one action then we can stop...")
                (Cond ((= ([_Size_] %Item%) 2) (Exit 1)))
                (Comment "otherwise we try to do some simplification.")
                (@Trans Simplify_Action_System)
                (Comment "If we have only one action then we can stop...")
                (Cond ((= ([_Size_] %Item%) 2) (Exit 1)))
                (Comment "otherwise we do some simplification and then
                            remove an action which will minimise the increase
                            in program size.")

                (@Down)
                (Comment "First we build a list of the names of all the
                              actions which it may be possible to remove - e.g.
those which are not part of the list of possible
                            entry actions.")

                (While ([_Right?_])
                        (@Right)
                        (Assign (Names (Cons %Comp_1% Names))))
                (Assign (T_Calls ([_Calls_] %Item%)))
                (While (Non_Empty? T_Calls)
                        (Cond ((Not (== (Hd (Hd T_Calls)) (!L Z)))
                                (Assign (Calls (Cons (Hd (Hd T_Calls))
                                                Calls)))))
                        (Assign (T_Calls (Tl T_Calls))))
                (Cond ((Not (Subset? Calls Names))
                        (@Up)
                        (Exit 1)))
                (Assign (Names (Set_Diff Names (List Start_Call))))
                (Cond ((Empty? Names)
                        (@Up)
                        (Exit 1)))
                (Assign (Best_Name Empty)
                        (Best_Num %Big_Num%))
                (@To 1)
```

```
(Comment "For each action we either remove it, because it is
          small and its body can be made into a procedure, or
          we determine whether it is the smallest action that
          we have so far come across.")

(While ([_Right?_])
     (@Right)
     (Cond ((And (< ([_Total_Size_] %Item%) 40)
                 (= (Length ([_Calls_] %Item%)) 1)
                 ([_Check?_] Action (( * ) (Call ( ? ) ( ? ))))
                 ([_Trans?_] Substitute_And_Delete))
            (Var ((P (- %C_Posn% 1)))
                (Cond (([_Trans?_] Automatically_Procedurise)
                       (!XP Show ("p"))
                       (@Trans Automatically_Procedurise)))
                (@Trans Substitute_And_Delete)
                (!XP Show ("*"))
                (@Down)
                (@To P))))
     (Assign (Temp_Num (* ([_Total_Size_] %Item%)
                          (Length ([_Calls_] %Item%)))))
     (Cond ((And (< Temp_Num Best_Num)
                 (Member? %Comp_1% Names))
            (Assign (Best_Name %Comp_1%)
                    (Best_Posn %C_Posn%)
                    (Best_Num Temp_Num)))))

(Comment "If we have only one action then we can stop...")
(Cond ((= %Length% 2) (@Up) (Exit 1)))

(Comment "We now move to the smallest action that we found
          and did not remove, Make its body into one or more
          procedures (where applicable) and then remove the
          action.  This will reduce the number of actions in
          the program, but minimise the increase in size of
          the program.")

(@To Best_Posn)
(Cond ((Some_Member? Best_Name ([_Calls_] %Item%))
       (@Trans Recursion->Loop)))
(While ([_Trans?_] Automatically_Procedurise)
       (!XP Show ("p"))
       (@Trans Automatically_Procedurise))
(Assign (Name %Comp_1%)
        (Fill (Body %Item%)))
(@Del)
```

```
            (!XP Fresh-Line ())
            (!XP Show ((- %Length% 1)))
            (@Up)

            (Comment "Next we do some general simplification of the action.")
            (@When 0 ((Not (Some_Member? Name ([_Calls_] %Item%)))
                      (@No_Deeper))
                     ((And ([_S_Type?_] Call)
                           (== %Comp_1% Name)
                           ([_Trans?_] Take_Call_Out_Of_Cond))
                      (@Trans Take_Call_Out_Of_Cond)))
            (@When 0 ((Not (Some_Member? Name ([_Calls_] %Item%)))
                      (@No_Deeper))
                     ((And ([_S_Type?_] Call) (== %Comp_1% Name))
                      (Assign (Value %Comp_2%)
                              (Fixed_Fill Fill))
                      (Assign (Fixed_Fill ([_Increment_] Fixed_Fill Value)))
                      (@Ins_Before (Args Fixed_Fill))
                      (@Del_Back)))

            (Comment "Finally, we go back to try to remove another of
                      the remaining actions."))

   (Comment "Having removed all the actions that we possibly can, we
             should have only one (possibly recursive) action left,
             which we can deal with in the usual way - i.e. by removing
             the recursion and then removing the (now unnecessary)
             action system.  Finally, we simplify any loops that we
             may have introduced")

   (@Down_Last)
   (Cond (([_Trans?_] Recursion->Loop)
          (@Trans Recursion->Loop)))
   (@Up)
   (Cond (([_Trans?_] Remove_Action_System)
          (@Trans Remove_Action_System)
          (@When 1 ((Not (Any_Member? (!L (Cond Skip))
                                      ([_Statements_] %Item%)))
                    (@No_Deeper))
                   ((And ([_S_Type?_] Cond)
                         ([_Trans?_] Fully_Factor_Cond))
                    (@Trans Fully_Factor_Cond))
                   ((And ([_S_Type?_] Skip)
                         ([_Trans?_] Delete_Skip_Simple_Version))
                    (@Trans Delete_Skip_Simple_Version))))))))
```

Appendix E

References

1. *IEEE Standard Glossary of Software Engineering Terminology.* In *ANSI/IEEE Standard 729.* 1983.

2. Software Product Evaluation (DP 9126). ISO/IEC JTC1/SC7 Working Document for DIS, December 1989.

3. AHO, A. V. AND ULLMAN, J. D. *Principles of Compiler Design.* Addison Wesley Publishing Company, 1977.

4. ALDERSON, A. *Configuration Management.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

5. ARANGO, G., BAXTER, I., FREEMAN, P. AND PIDGEON, C. TMM: Software Maintenance by Transformation. *IEEE Software* Vol. 3 No. 3 (1986), 27–39.

6. ARIOLA, Z. M. A Syntactic Approach to Program Transformations. *ACM Sigplan Notices* Vol. 26 No. 9 (September 1991).

7. ARNOLD, R. S. *An Introduction to Software Restructuring.* In *Tutorial on Software Restructuring*. IEEE Computer Society, 1986, 1–11.

8. ARSAC, J. *Transformations of Recursive Procedures.* In *Tools and Notations for Program Construction*, D. Neel, Ed. Cambridge University Press, Cambridge, 1982, 211–265.

9. ARSAC, J. Syntactic Source to-Source Transformations and Program Manipulations. *Communications of the ACM* Vol. 22 No. 1 (January 1979), 43–54.

10. ATKINS, M. C. AND BROWN, A. W. *Principles of Object-Oriented Systems.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

11. BP INTERNATIONAL LIMITED. *B-Tool Version 1.1.* 1991.

12. BACK, R. J. R. Correctness Preserving Program Refinements. Presented at *Mathematical Centre Tracts*(1980).

13. BACK, R. J. R. AND WRIGHT, J. VON. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing* Vol. 2 No. 3 (1990), 247–272.

14. BACKHOUSE, R. C. *Program Construction and Verification.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

15. BALZER, R. Transformational Implementation: An Example. *IEEE Transactions on Software Engineering* Vol. 7 No. 1 (January 1981).

16. BALZER, R. A 15 Year Prospective on Automatic Programming. *IEEE Transactions on Software Engineering* Vol. 11 No. 11 (November 1985), 1257–1267.

17. BALZER, R. AND CHEATHAM, T. E. Editorial: Program Transformations. *IEEE Transactions on Software Engineering* Vol. 7 No. 1 (January 1981), 1–2.

18. BALZER, R., GOLDMAN, N. M. AND WILDE, D. S. *On the Transformational Programming Approach to Programming.* In *Proceedings of the Second International Conference on Software Engineering*. IEEE Computer Society, New York, October 1976.

19. BALZER, R. M. *Final Report on GIST.* In *Information Science Institute*. University of Southern California, Marina del Rey, 1981.

20. BARSTOW, D. R. *A Knowledge-Based System for Automatic Program Construction.* In *Proceedings of the 5th International Conference on Artificial Intelligence*. Cambridge, Mass. , 1977, 382–388.

21. BARSTOW, D. R. An Experiment in Knowledge-Based Automatic Programming. *Artificial Intelligence* Vol. 12 (1979), 73–119.

22. BARSTOW, D. R. Automatic Programming for Streams II: Transformational Implementation. Presented at *Proceedings of the 10th International Conference on Software Engineering*, Singapore (1988).

23. BARSTOW, D. R. On Convergence Toward a Database of Program Transformations. *ACM Transactions on Programming Languages and Systems* Vol. 7 No. 1 (January 1985).

24. BAUER, F. L. The Munich Project CIP. *Lecture Notes in Computer Science* Vol. 183 (1985).

25. BAUER, F. L. The Munich Project CIP. *Lecture Notes in Computer Science* Vol. 292 (1987).

26. BAUER, F. L., BROY, M., PARTSCH, H., PEPPER, P. AND WÖSSNER, H. Systematics of Transformation Rules. *Lecture Notes in Computer Science* Vol. 69 (1979), 273–289.

27. Bauer, F. L., Moller, B., Partsch, H. and Pepper, P. Formal Construction by Transformation — Computer Aided Intuition Guided Programming. *IEEE Transactions on Software Engineering* Vol. 15 No. 2 (February 1989).

28. Belady, L. A. and Lehman, M. M. *The Characteristics of Large Systems.* In *Research Directions in Software Technology*. MIT Press, 1979.

29. Bennett, K. H., Cornelius, B. J., Munro, M. and Robson, D. J. *Software Maintenance.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

30. Bennett, K. H., Martil, R. and Zuylen, H. V. A Model of Software Reconstruction. Centre of Software Maintenance, Durham, Technical Report, 1990.

31. Berg, H. K., Boebert, W. E., Franta, W. R. and Moher, T. G. *Formal Methods of Program Verification and Specification.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

32. Bergland, G. D. A Guided Tour of Program Design Methodologies. *Computer* Vol. 14 No. 10 (October 1981).

33. Bishop, J. *Data Abstraction in Programming Languages.* Addison Wesley Publishing Company, Wokingham, England, 1986.

34. Bjørner, D. On the Use of Formal Methods in Software Development. Presented at *Proceedings of the 9th International Conference on Software Engineering*, Monterey, California (1987).

35. Bjørner, D. and Jones, C. B. *Formal Specification and Software Development.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

36. BLANK, J. AND KRIJGER, M. J. *Software Engineering: Methods and Techniques*. Dutch Computer Association with John Wiley and Sons Ltd, 1983.

37. BOEHM, B. W. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

38. BOEHM, B. W. A Spiral Model of Software Development and Enhancement. Presented at *IEEE Computer*(May 1988).

39. BOYER, R. S. AND MOORE, J. S. *A Computational Logic*. Academic Press, New York, 1979.

40. BOYER, R. S. AND MOORE, J. S. *The Correctness Problem in Computer Science*. Academic Press, 1981.

41. BOYLE, J. M. *Lisp to Fortran — Program Transformation Applied*. In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

42. BOYLE, J. M., DRITZ, K. W., MURALIDHARAN, M. N. AND TAYLOR, R. J. *Deriving Sequential and Parallel Programs from Pure LISP Specifications by Program Transformations.* In *Program Specifications and Transformation*, L. G. L. T. Meertens, Ed. Elsevier Science Publications, 1987, 1–19.

43. BREUER, P. T., LANO, K. AND BOWEN, J. Understanding Programs through Formal Methods. Oxford University Programming Research Group, Technical Report, 1991.

44. BROWN, A. J. Specifications and Reverse Engineering. Presented at *Journal of Software Maintenance*(1992).

45. BROY, M. *Algebraic Methods for Program Construction: The CIP Project.* In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

46. BULL, T. M. *An Introduction to the WSL Program Transformer.* In *Proceedings of the IEEE Conference on Software Maintenance.* San Diego, California, November 1990.

47. BULL, T. M., BENNETT, K. H. AND YANG, H. *A Transformation System for Maintenance — Turning Theory into Practice.* In *Proceedings of the IEEE Conference on Software Maintenance.* Orlando, Florida, November 1992.

48. BURSTALL, R. M. *Program Development by Transformations: An Overview.* In *Proceedings of the CREST Course on Programming.* 1978.

49. BURSTALL, R. M. AND DARLINGTON, J. A. A Transformation System for Developing Recursive Programs. *Journal of the ACM* Vol. 24 No. 1 (January 1977).

50. BURSTALL, R. M. AND GOUGEN, J. A. The Semantics of CLEAR, A Specification Language. *Lecture Notes in Computer Science* Vol. 86 (1980).

51. BURSTALL, R. M., MCQUEEN, D. B. AND SANNELLA, D. T. HOPE: An Experimental Applicative Language. Department of Computer Science, University of Edinburgh, Internal Report, 1980.

52. CALLISS, F. W. Problems with Automatic Restructurers. Durham University, Technical Report, Durham, 1989.

53. CALLISS, F. W., KHALIL, M., MUNRO, M. AND WARD, M. *A Knowledge-Based System for Software Maintenance.* In *Proceedings of the IEEE Conference on Software Maintenance.* Phoenix, Arizona, October 1988.

54. CHAPIN, N. *The Job of Software Maintenance.* In *Proceedings of the IEEE Conference on Software Maintenance.* 1987, 4–12.

55. CHIKOFSKY, E. J. AND CROSS, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* Vol. 7 No. 1 (1990), 13–17.

56. Chu, W. C. and Patel, S. *Software Restructuring by Enforcing Localisation and Information Hiding.* In *Proceedings of the IEEE Conference on Software Maintenance.* 1992, 165–172.

57. Church, A. The Calculi of Lambda Conversions. *Annals of Mathematical Studies* Vol. 6 (1951).

58. Daly, E. B. Organizing for Successful Software Development. *Datamation* Vol. 25 (December 1979).

59. Darlington, J. *Program Transformation in the ALICE Project.* In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1983.

60. Dershowitz, N. and Manna, Z. *On Automating Structured Programming.* In *Proceedings of the International Symposium on Proving and Improving Programs.* France, July 1975.

61. Dewar, R. B. K., Grand, A., Liu, S-C. and Schwartz, J. T. Programming by Refinement as Exemplified by the SETL Representation Sublanguage. *ACM Transactions on Programming Languages and Systems* Vol. 1 No. 1 (July 1979).

62. Dewar, R. B. K., Schonberg, E. and Schwartz, J. T. Higher Level Programming: Introduction to Use of the Set-Theoretic Programming Language SETL. Courant Institute of Mathematical Sciences, New York University Technical Report, 1981.

63. Dietrich, S. W. and Calliss, F. W. A Conceptual Design for a Code Analysis Knowledge Base. *Journal of Software Maintenance* Vol. 4 No. 1 (1992), 19–36.

64. Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, 1976.

65. DIJKSTRA, E. W. On the Interplay between Mathematics and Programming. *Lecture Notes in Computer Science* Vol. 69 (1979), 34–46.

66. DIJKSTRA, E. W. Interplay between Invention and Formal Techniques. *Lecture Notes in Computer Science* Vol. 69 (1979), 1.

67. ENGBERTS, A., KOZACZYNSKI, W. AND NING, J. *Concept Recognition-Based Program Transformation.* In *Proceedings of the IEEE Conference on Software Maintenance.* October 15–17, 1991, 73–82.

68. ERIKSEN, K. E. AND PREHN, S. RAISE Overview. Esprit Project Report Doc. Id. RAISE / DOC / KEE / 5 / V1, November 1989.

69. FAGAN, M. E. Design and Program Inspections to Reduce Errors in Program Development. *IBM Systems Journal* Vol. 12 No. 1 (1976).

70. FEATHER, M. S. ZAP Program Transformation System: Primer and Manual. Department of Artificial Intelligence, University of Edinburgh, Internal Report, 1978.

71. FEATHER, M. S. A Program Transformation System. University of Edinburgh, PhD Thesis, 1979.

72. FEATHER, M. S. *Specification and Transformation: Automated Implementation.* In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

73. FEATHER, M. S. *A Survey and Classification of Some Program Transformation Techniques.* In *Program Specification and Transformation.* Amsterdam, 1987, 165–198.

74. FEATHER, M. S. A System for Assisting Program Transformation. *ACM Transactions on Programming Languages and System* Vol. 4 No. 1 (January 1982), 1–20.

75. FICKAS, S. F. Automating the Transformational Development of Software. University of California, PhD Dissertation, Irvine, 1982.

76. FISHER, A. S. *CASE: Using Software Development Tools.* John Wiley and Sons Ltd, New York, 1988.

77. FRADET, P. AND MÉTAYER, D. L. Compilation of Functional Languages by Program Transformation. *ACM Transactions on Programming Languages and Systems* Vol. 13 No. 1 (January 1991).

78. GADD, R. J. *ReForm — From Assembler to Z using Formal Transformations.* In *Proceedings of the Fourth European Software Maintenance Workshop.* Durham, September 1990.

79. GAUDEL, M-C. *Algebraic Specifications.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

80. GOLDBERG, A. T. Knowledge-Based Programming: A Survey of Program Design and Construction Techniques. *IEEE Transactions on Software Engineering* Vol. 12 No. 7 (1986), 752–768.

81. GOLDMAN, N. M. AND WILDE, D. S. *A Relational Database Foundation for Process Specification.* In *Entity Relationship Approach to System Analysis and Design*, P. P. S. Chen, Ed. Elsevier North-Holland, New York, 1980.

82. GOOD, D. I., LONDON, R. L. AND BLEDSOE, W. W. An Interactive Program Verification System. *IEEE Transactions on Software Engineering* Vol. 1 (March 1975).

83. GORDON, M. J. C. *Programming Language Theory and its Implementation.* Prentice-Hall, Englewood Cliffs, 1988.

84. GOUGEN, J. A. Reusing Interconnecting Software Components. *IEEE Software* Vol. 3 (February 1986).

85. Green, C. *A Summary of the PSI Program Synthesis System.* In *Proceedings of the 5th International Joint Conference on AI.* Cambridge, MA, 1977, 380–381.

86. Green, C. *The PSI Program Synthesis System 1978: An Abstract.* In *Proceedings of the 1978 National Computer Conference.* Anahaim, CA, 1978, 673–674.

87. Green, C. *The Design of the PSI Synthesis System.* In *Proceedings of the 2nd International Conference on Software Engineering.* San Francisco, October 1976, 4–18.

88. Green, C., Gabriel, R. P., Kant, E., Kedzierski, B. J., McCune, B. R., Phillips, J. V., Tappel, S. T. and Westfold, S. J. *Results in Knowledge Based program Synthesis.* In *Proceedings of the 6th International Joint Conference on AI.* Tokyo, August 1979, 342–344.

89. Green, C., Phillips, J. V., Westfold, S. J., Pressburger, T., Kedzierski, B. J., Angerbranndt, S., Mont-Reynaud, B. and Tappel, S. T. Research on Knowledge Based Programming and Algorithmic Design. Kestral Institute, Technical Report U 81. 2, Palo Alto, CA, 1982.

90. Gries, D. Current Ideas in Programming Methodology. *Lecture Notes in Computer Science* Vol. 69 (1979), 77–93.

91. Griffiths, M. Program Production by Successive Transformation. *Lecture Notes in Computer Science* Vol. 46 (1976), 125–152.

92. Halpern, J. D., Owre, S., Proctor, N. and Wilson, W. F. Muse — A Computer Assisted Verification System. *IEEE Transactions of Software Engineering* Vol. 13 No. 2 (February 1987), 151–156.

93. Hannan, J. Staging Transformations for Abstract Machines. *ACM Sigplan Notices* Vol. 26 No. 9 (September 1991).

94. HAYES, I. VDM and Z: A Comparative Case Study. *Formal Aspects of Computing* Vol. 4 No. 1 (1992), 76–99.

95. HILDUM, D. AND COHEN, J. A Language for Specifying Program Transformations. *IEEE Transactions on Software Engineering* Vol. 16 No. 6 (June 1990).

96. HOARE, C. A. R. An Axiomatic Basis for Computer Programming. *Communications of the ACM* Vol. 12 No. 10 (1969), 576–580, 583.

97. HOARE, C. A. R. Proof of Correctness of Data Representations. *Acta Informatica* Vol. 1 No. 4 (1972), 271–281.

98. HOARE, C. A. R. The Emperor's Old Clothes: The 1980 ACM Turing Award Lecture. Presented at *Communications of the ACM*(February 1981).

99. HOARE, C. A. R. AND WIRTH, N. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica* Vol. 2 (1973), 335–355.

100. INCE, D. C. *Software Engineering*. Van Nostrand Reinhold Co Ltd, London, 1989.

101. JACKSON, M. A. *Principles of Program Design*. Academic Press, London, 1975.

102. JOHNSON, J. R. *The Software Factory; Second Edition*. Butterworth Heinemann, Oxford, 1991.

103. JONES, C. B. *Systematic Software Development using VDM — Second Edition*. Prentice-Hall, 1990.

104. KANT, E. The Selection of Efficient Implementations for a High-Level Language. *SIGPLAN Notices (ACM)* Vol. 12 No. 8 (August 1977), 140–156.

105. KANT, E. *A Knowledge-Based Approach to Using Efficiency Estimation in Program Synthesis.* In *Proceedings of the 6th Joint Conference on Artificial Intelligence.* Tokyo, Japan, August 1979, 457–462.

106. KANT, E. AND BARSTOW, D. R. The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Automatic Programming. *IEEE Transactions on Software Engineering* Vol. 7 No. 5 (1981), 458–471.

107. KELLER, S. E. *Grammar-Based Program Transformation* . In *Proceedings of the IEEE Conference on Software Maintenance.* Phoenix, Arizona, October 1988, 110–117.

108. KELLER, T. W. The Importance of Process Improvement in Software Maintenance. Presented at *Conference on Software Maintenance*(November 1993).

109. KING, J. C. Program Correctness: On Inductive Assertion Techniques. *IEEE Transactions on Software Engineering* Vol. 6 No. 5 (September 1980).

110. KNUTH, D. E. Structured Programming with the GOTO Statement. *Computing Surveys* Vol. 6 No. 4 (1974), 261–301.

111. KOSMAN, R. J. *Incorporating the Inspection Process into a Software Maintenance Organisation.* In *Proceedings of the IEEE Conference on Software Maintenance.* 1992, 51–56.

112. KOTIK, G. B. AND MARKOSIAN, L. Z. *Automated Software Analysis and Testing Using a Program Transformation System.* In *Proceedings of the Third Symposium on Software Testing, Analysis and Verification.* 1989.

113. KOZACZYNSKI, W., NING, J. AND ENGBERTS, A. Program Concept Recognition and Transformation. *IEEE Transactions of Software Engineering* Vol. 18 No. 12 (December 1992), 1065–1075.

114. KRIEG-BRUCKNER, B. *Language Comparison and Source-To-Source Translation.* In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

115. LANDIN, P. J. The Mechanical Evaluation of Expressions. *Computer Journal* Vol. 6 No. 4 (1964), 308–320.

116. LANDIN, P. J. A Correspondence between ALGOL 60 and Church's Lambda Notation. *Communications of the ACM* Vol. 8 No. 2 and 3 (February and March 1965), 89–101 and 158–165.

117. LANO, K. AND HAUGHTON, H. Applying Formal Methods to Maintenance. Oxford University Programming Research Group, Technical Report, 1990.

118. LEHMAN, M. M. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE* Vol. 19 (1980), 1060–1076.

119. LEHMAN, M. M. AND BELADY, L. A. A Model of Large Program Development. *IBM Systems Journal* Vol. 15 No. 3 (1976).

120. LEINTZ, B. P. AND SWANSON, E. F. *Software Maintenance Management.* Addison-Wesley Publishing Company, 1980.

121. LISKOV, B. H. AND V., B. *An Appraisal of Program Specifications.* In *Research Directions in Software Technology*, P. Wegner, Ed. 1979, 276–301.

122. LONDON, P. AND FEATHER, M. Implementing Specification Freedoms. Information Science Institute, University of Southern California, Research Report 81-100, Marina del Rey, CA, 1982.

123. LU, L. C. A Unified Framework for Systematic Loop Transformations. *ACM Sigplan Notices* Vol. 26 No. 7 (July 1991), 28–38.

124. LYONS, M. J. *AFIPS Conference Proceedings.* vol. Vol. 50 , National Computer Conference, 1981.

125. MANNA, Z. AND WALDINGER, R. Synthesis: Dreams ⇒ Programs. *IEEE Transactions on Software Engineering* Vol. 4 No. 4 (1979).

126. MANNA, Z. AND WALDINGER, R. DEDALUS — The DEDuctive Algorithm Ur-synthesizer. *Proceedings of the National Computer Conference* Vol. 47 AFIPS Press (June 5–8, 1978), 683–690.

127. MANNA, Z. AND WALDINGER, R. *The Synthesis of Structure-Changing Programs.* In *Proceeding of the 3rd International Conference on Software Engineering*. Atlanta, GA, May 1978, 175–187.

128. MASON, I. A. AND TALCOTT, C. L. Program Transformations for Configuring Components. *ACM Sigplan Notices* Vol. 26 No. 9 (September 1991).

129. MCCUNE, B. P. Building Program Models Incrementally from Informal Descriptions. Department of Computer Science, Stanford University, PhD Dissertation, Standford, California, 1979.

130. MCDERMID, J. *Introduction and Overview to Part II.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

131. MCDERMID, J. AND ROOK, P. *Software Development Process Models.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

132. MCDONNALL, P. AND BENNETT, K. H. *Reverse Engineering IS Library Module (Forthcoming).* CCTA, 1993.

133. MCGETTRICK, A. D. *The Definition of Programming Languages.* Cambridge University Press, Cambridge, UK, 1980.

134. MILLER, J. C. *Software Re-Engineering: Getting it Done is Twice the Fun.* In *Techniques of Program and System Maintenance.* QED Information Services, Inc. , 1979.

135. Miller, J. C. and Strauss, B. M. Implications of Automatic Restructuring of COBOL. *SIGPLAN Notices* Vol. 22 No. 6 (June 1987), 76–82.

136. Monahan, B. and Shaw, R. *Model-Based Specifications.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

137. Morgan, C. *Programming from Specifications.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

138. Morgan, C., Robinson, K. and Gardiner, P. On the Refinement Calculus. Oxford University, Technical Monograph, 1988.

139. Mostow, D. J. Mechanical Transformation of Tasks Heuristics into Operational Procedures. Carnegie-Mellon University, Ph. D. dissertation, Rep. CMU-CS-81-113, Pittsburg, Pa. , 1981.

140. Myer, B. *Introduction to the Theory of Programming Languages.* Prentice-Hall, 1990.

141. Myers, G. J. *The Art of Software Testing*. John Wiley and Sons Ltd, New York, 1979.

142. Myers, W. *Software Engineering.* In *McGraw Hill Encyclopedia of Science and Technology*. McGraw Hill, 1987.

143. Neighbours, J. M. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering* Vol. 10 No. 5 (1984), 564–574.

144. Nielson, H. R. and Nielson, F. *Semantics with Applications.* John Wiley and Sons Ltd, 1992.

145. Nielson, H. R. and Nielson, F. Using Transformations in the Implementation of Higher-Order Functions. *Journal of Functional Programming* Vol. 1 No. 4 (October 1991).

146. OULD, M. A. Testing — A Challenge to Method and Tool Developers. *Software Engineering Journal* Vol. 6 No. 2 (1991), 59–64.

147. OVERSTREET, C. M., CHEN, J. AND BYRUM, F. *Program Maintenance by Safe Transformation* . In *Proceedings of the IEEE Conference on Software Maintenance*. Phoenix, Arizona, October 1988, 118–123.

148. PAIGE, R. *Supercompilers — Extended Abstract*. In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

149. PARIKH, G. What is Software Maintenance?. Presented at *ACM SIGSOFT Software Engineering Notes* (1986).

150. PARKER, S. P. *McGraw-Hill Dictionary of Scientific and Technical Terms (4th Edition)*. McGraw-Hill, New York, 1989.

151. PARTSCH, H. *Specification and Transformation of Programs*. Springer-Verlag, New York, 1990.

152. PARTSCH, H. AND STEINBRÜGEN, R. Program Transformation Systems. *ACM Computing Surveys* Vol. 15 No. 3 (September 1983), 199–236.

153. PEPPER, P. A Study on Transformational Semantics. *Lecture Notes in Computer Science* Vol. 69 (1979), 322–405.

154. PEPPER, P. *Algebraic Techniques for Program Specification*. In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

155. PEPPER, P. *Inferential Techniques for Program Development*. In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

156. PERRY, W. E. *A Structured Approach to System Testing: Second Edition.* QED Information Sciences Inc, Wellesley, Massachusetts, 1988.

157. REDDY, U. S. Transformational Derivation of Programs Using the FOCUS System. Presented at *Symposium on Software Development Environments, ACM*(December 1988).

158. ROBSON, D. J., BENNETT, K. H., CORNELIUS, B. J. AND MUNRO, M. Approaches to Program Comprehension. *Journal of Systems Software* Vol. 14 No. 1 (1991).

159. ROSS, D. T., GOODENOUGH, J. B. AND IRVINE, C. A. Software Engineering: Process, Principles and Goals. Presented at *Computer*(May 1975).

160. RUSSELL, G. W. Experience with Inspection in Ultralarge-Scale Developments. *IEEE Software* Vol. 8 No. 1 (January 1991), 25–31.

161. RUSSINOFF, D. M. A Verification System for Concurrent Programs Based on Boyer-Moore. *Formal Aspects of Computing* Vol. 4 No. 6A (1992), 597–611.

162. SCHERLIS, W. L. *Supercompilers — Extended Abstract.* In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1983.

163. SCHWARTZ, J. T. *On Programming: An Interim Report on the SETL Project.* Courant Institute, New York, 1975.

164. SHOOMAN, M. L. *Software Engineering*. McGraw-Hill, 1983.

165. SHRIVASTAVA, S. K. *Fault-Tolerant System Structuring Concepts.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

166. SNEED, H. M. Economics of Software Re-engineering. *Journal of Software Maintenance* Vol. 3 No. 3 (1991), 163–182.

167. SNEED, H. M. AND JANDRASICS, G. *Inverse Transformation of Software from Code to Specification.* In *Proceedings of the IEEE Conference on Software Maintenance.* IEEE Computer Society Press, Phoenix, Arizona, October 1988, 102–109.

168. SNEED, H. M. AND JANDRASICS, G. *Software Recycling.* In *Proceedings of the IEEE Conference on Software Maintenance.* Austin, Texas, September 1987, 82–90.

169. SOMMERVILLE, I. *Software Engineering — Fourth Edition.* Addison Wesley Publishing Company, 1992.

170. SPIVEY, J. M. *The Z Notation: A Reference Manual — Second Edition.* Prentice-Hall, 1992.

171. STOBART, S. C., THOMPSON, J. B. AND SMITH, P. Use, Problems, Benefits and Future Directions of Computer Aided Software Engineering in United Kingdom. *Information and Software Technology* Vol. 33 No. 9 (1991), 629–636.

172. TAMIR, M. ADI: Automatic Derivation of Invariants. *IEEE Transactions on Software Engineering* Vol. 6 No. 1 (January 1980).

173. TANIMOTO, S. L. *The Elements of Artificial Intelligence using Common Lisp.* Computer Science Press, New York, 1990.

174. WARD, M. A Catalogue of Program Transformations. Durham University, Technical Report, Durham, 1988.

175. WARD, M. Transforming a Program into a Specification. Durham University, Technical Report, Durham, 1988.

176. WARD, M. Constructive Specifications and Program Transformations. Durham University, Technical Report, Durham, 1988.

177. WARD, M. Proving Program Refinements and Transformations. Oxford University, DPhil Thesis, Oxford, 1989.

178. WARD, M. The Largest True Square Problem — An Exercise in the Derivation of an Algorithm. Durham University, Technical Report, Durham, 1990.

179. WARD, M. Specifications and Programs in the Wide Spectrum Language. Durham University, Technical Report, Durham, 1991.

180. WARD, M. *A Recursion Removal Theorem.* In *Proceedings of Fifth Refinement Workshop.* BCS FACS, London, 1992.

181. WARD, M. A Definition of Abstraction. Durham University, Technical Report, Durham, 1992.

182. WARD, M. Foundations for a Practical Theory of Program Refinement and Transformation. Durham University, Technical Report, Durham, 1993.

183. WARD, M. Language Oriented Programming. Durham University, Technical Report, Durham, 1994.

184. WARD, M. AND BENNETT, K. H. A Practical Program Transformation System For Reverse Engineering. Presented at *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (May 1993).

185. WARD, M., CALLISS, F. W. AND MUNRO, M. *The Maintainer's Assistant.* In *Proceedings of the IEEE Conference on Software Maintenance.* Miami, Florida, October 1989.

186. WARNIER, J. D. *Logical Construction of Programs.* Van Nostrand Reinhold, New York, 1977.

187. WEBER-WULFF, D. Proof-Movie — A proof with the Boyer-Moore Prover. *Formal Aspects of Computing* Vol. 5 No. 2 (1993), 121–151.

188. Whysall, P. *Refinement.* In *Software Engineer's Reference Book*, J. McDermid, Ed. Butterworth Heinemann, 1991.

189. Wile, D. S. *Transformation-Based Software Development.* In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag, Berlin Heidelberg, 1984.

190. Wile, D. S., Balzer, R. and Goldman, N. M. *Automated Derivation of Program Control Structures from Natural Language Program Descriptions.* In *Proceedings of the 4th ACM Symposium on Artificial Intelligence and Programming Languages.* vol. Vol. 12 , no. No. 8 , SIGPLAN Notices (ACM), Rochester, New York, August 1977.

191. Wirth, N. Program Development by Stepwise Refinement. *Communications of the ACM* Vol. 14 (1971).

192. Yang, H. How does the Maintainer's Assistant Start?. Durham University, Technical Report, Durham, 1989.

193. Yang, H. *The Supporting Environment for a Reverse Engineering System — The Maintainer's Assistant.* In *Proceedings of the IEEE Conference on Software Maintenance.* Sorento, Italy, 1991.

194. Yang, H. PhD. Thesis, Forthcoming.

195. Yang, J. A. and Choo, Y. Parallel Program Transformation using a Meta-Language. *ACM Sigplan Notices* Vol. 26 No. 7 (July 1991), 11–21.

196. Younger, E. J. and Ward, M. Inverse Engineering a simple Real Time Program. Durham University, Technical Report, Durham, 1992.

197. Yourdon, E. *Structured Walk-Throughs.* Yourdon Press, New York, 1977.

198. Zelkowitz, M. V., Shaw, A. C. and Gannon, J. D. *Principles of Software Engineering and Design.* Prentice-Hall, Inc, Englewood Cliffs, NJ, 1979.