Data re-engineering using formal transformations

Richard Eric Mortimer

Data Re-engineering using Formal Transformations

Richard Eric Mortimer

Ph.D. Thesis

Centre for Software Maintenance

Department of Computer Science

University of Durham

1998

Abstract

This thesis presents and analyses a solution to the problem of formally re-engineering program data structures, allowing new representations of a program to be developed. The work is based around Ward's theory of program transformations which uses a Wide Spectrum Language, WSL, whose semantics were specially developed for use in proof of program transformations. The re-engineered code exhibits equivalent functionality to the original but differs in the degree of data abstraction and representation.

Previous transformational re-engineering work has concentrated upon control flow restructuring, which has highlighted a lack of support for data restructuring in the maintainer's tool-set. Problems have been encountered during program transformation due to the lack of support for data re-engineering. A lack of strict data semantics and manipulation capabilities has left the maintainer unable to produce optimally re-engineered solutions. It has also hindered the migration of programs into other languages because it has not been possible to convert data structures into an appropriate form in the target language.

The main contribution of the thesis is the Data Re-Engineering and Abstraction Mechanism (DREAM) which allows theories about type equivalence to be represented and used in a re-engineering environment. DREAM is based around the technique of "ghosting", a way of introducing different representations of data, which provides the theoretical underpinning of the changes applied to the program. A second major contribution is the introduction of data typing into the WSL language. This allows DREAM to be integrated into the existing transformation theories within WSL.

These theoretical extensions of the original work have been shown to be practically viable by implementation within a prototype transformation tool, the Maintainer's Assistant. The extended tool has been used to re-engineer heavily modified, commercial legacy code. The results of this have shown that useful re-engineering work can be performed and that DREAM integrates well with existing control flow transformations.

Acknowledgements

I would not have been able to produce this thesis without the help and support of a number of people. I thank Professor Keith Bennett for his help, guidance and supervision throughout the course of the research for this thesis.

The academic content of this thesis was influenced by a number of different people. Tim Bull, Brendan Hodgson, Zhaohui Luo, Eddy Younger and Martin Ward provided numerous hours of stimulating discussion into the formal and practical aspects of program transformation in WSL.

Ron Cain and Pete Collins at the IBM United Kingdom Laboratories provided assistance during the case studies and the staff at Software Migrations Ltd. provided help with and information about the FermaT program transformation tool.

During the course of the research the staff and students at the Centre for Software Maintenance provided numerous opportunities for taking a much wider view of computing research. Special thanks go to Malcolm Munro, Liz Burd and Peter Biggs.

I gratefully acknowledge the financial support from IBM (UK) Ltd. and the Engineering and Physical Sciences Research Council.

The final preparation of the thesis would not have been possible without the help of Jill Munro and my colleagues at Sun Microsystems.

I could not have completed this thesis without an opportunity to relax and unwind in the company of my many friends at Durham Amateur Rowing Club and The Graduate Society. Special mention must go to Iwan Jones, Ros Martin, Andrew Adams, Jed and Allie Gargan, Clyde Burgess, Jim Dulling, Chris Cooper, Richard Dodson, Anna Hindhaugh and Dr. Mike Richardson.

Finally, may I extend my warmest thanks to Deborah Robson who has provided endless encouragement, help and support over the past few years. I will never be able to fully repay the debt I owe her.

This thesis has been produced using LaTeX, xfig and XEmacs.

To Eric, Mary and Sarah.

Contents

List of Tables

List of Figures

List of Examples

Chapter 1

Introduction

Computer software is an inherently complex product which must conform to very high standards of precision to operate correctly. Many examples of the failures of high profile systems are widely reported in computing literature. The reasons for these failures are varied but usually stem from seemingly trivial errors in the coding or system design.

Many solutions to these problems have been proposed which tackle a wide variety of issues such as: management of software projects; better software languages and tools; and methods for mapping high level descriptions of systems into executable code. None of these issues is sufficient to ensure success and a combination of techniques is usually necessary. Individual excellence is still important for each technique, however, if the chances of errors are to be minimised. This thesis examines one such area, that of formal methods, specialising in the area of data re-engineering. This provides a tightly focussed area of research which makes it feasible to perform a comprehensive examination of relevant issues and problems.

A **formal method** is defined by Lano [61] as

> "the use of mathematical notation to describe both the requirements and the design of software systems in a precise manner".

In other words mathematical symbols and formulae are used to describe the operation of the software. This allows arithmetic laws to be used to show consistency between different parts of the code throughout the various stages of development. This use of mathematical notation in well-defined frameworks brings about the name formal methods.

Mention of the name formal methods often evokes reactions that they are very abstract and therefore do not apply well to the practical tasks of software development and maintenance. Hall [49] challenges many of these arguments stating that "some of the beliefs about formal methods have been exaggerated and have acquired almost the status of myths". The key to the successful application of formal methods is the realisation of their limits and the resulting use of them along with other, traditional methods of software maintenance.

Reverse engineering is an area which can benefit significantly from the use of formal methods. The scale of many software maintenance projects makes human understanding and redevelopment difficult. Tool support is therefore essential to aid the process by tracking information, checking that changes made to the software do not affect other parts of the system and by automation of low-level clerical tasks. In particular it is desirable to have a formal means which allows verification that changes have been made successfully. A full discussion of these issues is provided in chapter 2.

Data presents a number of problems for the maintainer when considered in the realm of an executable program. At this level the abstract relationships between high level data elements is blurred because these relationships are embedded within the program's instructions. Abstract data structures may also have countless numbers of ways in which they can be implemented. These are further differentiated by subtle differences in the semantics of individual variables and their data types. This makes the identification of data structures and extraction of them from the program difficult.

1.1 Statement of the Problem

This thesis uses formal methods to address the problems of data re-engineering. **Data re-engineering** is defined as "the change in representation[1] of program data to aid the maintenance process while still providing equivalent functionality during program execution". These changes in representation include the following three

[1] Data representation is the form of coding used to represent a real-world value within the computer. This includes primitive representations such as integers and strings as well as more complex representations such as structures and algebraic specifications.

types of data transformation:

- **Abstraction** — representation of the data in terms which are more abstract than the original. This means that the data representation does not map directly onto a directly-executable form. It may involve representation in terms of more abstract data structures such as lists and stacks or in terms of the actual relationships between different data objects, e.g. set theoretic descriptions.
- **Refinement** — this is the opposite of abstraction and involves the introduction of more concrete representations of the data. That is the data format is closer to an executable machine version.
- **Restructuring** — a change in the representation without a change in the level of data abstraction. This is the simplest form of manipulation and involves the use of different data concepts, e.g. integers and floating point numbers, to represent a piece of data.

The main thrust of the problem is the integration of theories of data type equivalence and refinement relationships into existing theories for the specification of program control flow behaviour. The important part of this is the harnessing of the relationships to allow machine checked conversion of program representation into functionally equivalent versions which are suitable for the maintainer's needs. This thesis focuses upon reverse engineering[2] and code migration[3] activities, continuing the extensive software maintenance research which has already been performed at Durham.

1.2 Motivation

This interest in data re-engineering stems from previous work done on the ReForm project investigating the use of formal program transformations during software

[2] Reverse engineering is the recovery of design information and high level program code from low level code[32], typically assembly code.

[3] Code migration is the conversion of code from one specific language/machine into a form suitable for use in another environment.

maintenance. This work is well reported in the literature by Ward [88], Bull [23] and others. The work concentrates upon the manipulation of program control flow to allow restructuring of legacy code thus recovering implicit uses of common programming paradigms such as conditionals, loops and subroutines. A full discussion of the theory and tools which demonstrate this work can be found in chapter 3.

The conclusions of ReForm included the realisation that the ability to restructure data was lacking and that this was significantly hindering other restructuring work. The properties of the data representations were not sufficiently well specified within the WSL language[4] and associated theories, both of which were developed for the explicit purpose of program transformation. The tools which implemented this theory did not, therefore, permit formal manipulation of data. This meant that the properties of particular data types could not be used to aid restructuring and that the ad-hoc machine implementations of data reasoning were not sufficiently well developed for reverse engineering purposes.

For example, without data type information it is difficult to differentiate between numbers which are stored as discrete or real values because of differences in the precision and accuracy of each representation. Other examples, at higher levels of abstraction, include situations where the semantics of complex data structures can only be determined when their exact form is known.

1.3 Statement of Contribution

This thesis shows how correctness preserving program transformations can be used for data re-engineering. The work builds upon previous successful work in the program reverse engineering area and extends a number of tools and theories to cope successfully with data reasoning. The main areas of contribution are:

- Typed WSL has been defined. It extends the WSL language to include explicit data typing. The extended language, typed WSL, allows data types to be defined and bound statically to variables in a program. This involves changes

[4] WSL is a custom designed language used in the ReForm transformation work. Code which is being transformed is first translated into WSL using a simple one-to-one mapping. The transformation is performed on the resulting WSL and the final version is translated into a suitable target language.

to the transformation meta language, $\mathcal{META}$WSL allowing type information to be included within the existing transformation system.

- The DREAM (Data Re-Engineering and Abstraction Mechanism) has been developed to allow theories about data equivalence to be harnessed and integrated into a program transformation environment. DREAM is based upon the technique of "ghosting" which was originally used by Owicke to aid program verification [45]. It provides a way of converting assignments-to and uses-of a variable into equivalent assignments-to and uses-of another. The use of ghosting for data transformation was first proposed by Ward [92] in his derivation of the Schorr-Waite graph marking algorithm.

  Ghosting uses **ghost variables** which "mirror" the values held within a variable. The ghost variable has values assigned to it which are equivalent to the original variable although they may have differing data types/representations. These equivalent values can then be used in place of the original effectively changing the representation of the data.

- A prototype transformation tool has been developed to demonstrate the use of DREAM in practical re-engineering situations. This is based on the Maintainer's Assistant [23] transformation tool and has been developed in a way which complements and enhances the existing coding practices which are used within the WSL and $\mathcal{META}$WSL[5] languages.

1.4 Criteria for Success

The contributions presented above must be judged to determine how well they perform as a solution to the maintenance problems set out earlier. The following questions present a set of criteria which cover the main aspects of the work. These criteria provide a basis for examining the success of the thesis. The questions have been divided into three groups which correspond with the different aspects of contribution shown above.

[5] $\mathcal{META}$WSL is an extension of WSL developed by Bull [11] to aid the implementation of a program transformation system

- **Data types in WSL**
  - Can typed WSL represent a full range of data types which may be encountered in common programming languages?
  - Is the set of data types that may be represented in typed WSL extensible?
  - Does typed WSL have a detrimental effect upon Ward's transformations?
- **Data transformations using DREAM**
  - Is data transformation possible for a full range of data types?
  - How easy is it to add new data transformation theories?
  - Do data transformations rely upon control flow transformations?
- **Data Re-engineering using Formal Transformations**
  - Can DREAM perform all of the classes of transformation which are necessary for re-engineering?
  - Does DREAM scale to practical re-engineering tasks?
  - Does DREAM complement existing software maintenance activities?

The answers to these questions will be discussed in chapter 8.

1.5 Thesis Outline

The remainder of this thesis describes the research in detail.

Chapter 2 examines the work in the wider context of software engineering and maintenance. It highlights the large proportion of a software product's lifetime which is spent in maintenance and shows some of the key management, technical and process-oriented problems which are encountered. Legacy code is highlighted as a key area which is vital for re-engineering. Techniques which are used to maintain legacy code are examined along with the different aspects of legacy code which can be maintained.

Chapter 3 looks at formal program transformations and shows how they can be used during software maintenance. It goes on to highlight a lack of research in the field of data transformation and looks at some of the methods which are currently used to re-engineer program data.

Chapter 4 focuses upon the Maintainer's Assistant transformation system. It examines Ward's [88] WSL transformation language and the semantic theory upon which it is based. The transformation engine is also examined in detail and a number of accomplishments/deficiencies are highlighted (full details of these can be found in appendix A). A key deficiency which is identified is the lack of support for data transformation.

The second half of the chapter presents extensions to the WSL language which result in the typed WSL language. This is used as the basis for the data transformations which are presented within this thesis.

Chapter 5 examines the different data transformations which can be performed and then uses the typed WSL language to develop DREAM. The central part of DREAM is a data transformation which separates the change of data representation from the transformation of a program. This allows a generic data transformation to be used to transform data of any type providing that a suitable data type equivalence theory is available.

Chapter 6 shows how the Maintainer's Assistant transformation engine has been extended to allow representation of typed data and to provide an implementation of the DREAM data transformations.

Individual data types and type equivalences are examined in chapter 7. The chapter concentrates upon the integer and record data types and we demonstrate how the types can be described in a form which is suitable for use during data transformation. These types are then used to show how type equivalence theories are constructed and a number of examples of their use for data transformation are given. The chapter also outlines how the semantics of a number of other data types may be represented and transformed in typed WSL.

Chapter 8 presents the results of two case studies which were performed using DREAM to reverse engineer substantial amounts of commercial source code. The first case study examines the overall reverse engineering of a portion of code. This was performed by hand and shows how transformations can be used to recover the design of that code. The second case study uses the extended transformation engine to examine the practical aspects of DREAM. Specific examples of data transformations are identified and the tool is used to perform these. The final part of the chapter re-examines the criteria for success which has been set out in chapter 1. It uses the results of the case studies and the work presented in the other chapters to answer the questions which have been posed.

Chapter 9 summarises the contents of the thesis and uses the criteria for success to show that this thesis has presented a novel contribution to the field of data re-engineering. The thesis closes with a number of ideas for further work which will continue the research into data re-engineering using formal transformations.

Chapter 2

A Perspective on Software Maintenance

Software maintenance is a major activity for companies involved in computer technology. This chapter presents a perspective on the maintenance activity which shows how re-engineering can be used as an effective vehicle for reducing the costs, risks and time-scales involved. In particular the effective use of tools, which are based upon formal theories of program structure, can help the engineer to understand and restructure code without affecting the operation of the program.

The chapter begins with an examination of the software lifecycle which shows that maintenance is a key component in this process. This leads into the identification of three key areas: management, process and technical. The work presented in this thesis can be used to help solve some of the software maintenance problems found in these areas. The biggest gains are realised in the technical area where formal restructuring tools can be used to improve the accuracy of work and ensure that seemingly trivial details are not ignored.

Software re-engineering is discussed in the second half of the chapter. This technique is a vital component of the work presented here allowing legacy software, old code which performs vital portions of a companies computing work, to be examined and converted into different forms for future re-implementation and re-documentation. Re-engineering is performed upon a number of different types of systems which are written in many different programming languages. The application of this varies from project to project: some use the results as an aid to understanding the original system; whereas others use the results to directly re-implement the system in a new language or on a new computing platform.

The chapter concludes that re-engineering is a key component of software maintenance providing significant benefits in a number of areas. Formal tool support is especially attractive because it reduces the reliance upon human effort to achieve complete, correct results. These tools can perform much of the general housekeeping and analysis work which forms a large portion of re-engineering. This allows the maintainer to focus upon the overall result rather than upon minor details.

2.1 The Software Lifecycle

The lifetime of a software system often covers many years and in many cases expands to cover a number of decades. This includes a number of distinct phases [15, 67] which can be summarised as:

- **Development** — where the software is specified, designed, written and tested.
- **Commissioning** — when the system is brought into service. This often involves the replacement of an existing automated or manual system.
- **Maintenance** — the software is modified to meet new requirements and to correct errors found within it.
- **De-commissioning** — at the end of its useful life the system must be phased out. There is often a changeover period where the commissioning stage of the replacement system is performed alongside the de-commissioning of the old system.

The majority of the lifetime of a system is spent in the maintenance phase. Only a small proportion of this lifetime is spent in the other phases: development typically takes between one and five years; commissioning and de-commissioning may be done overnight or over a number of months depending upon the complexity of the changeover and the risks involved in performing it. It is not surprising therefore that studies have shown that the maintenance costs of the system can be up to 80% of the total cost of the system [64, 65]. This is clearly a major proportion of the overall expenditure on the system and is therefore a major target for cost savings.

Unfortunately research into software maintenance is not as popular as the need for it may suggest. Schneidewind [77] notes that in 1987 the IEEE Transactions on Software Engineering did not receive sufficient numbers of good submissions to fill a special issue on software maintenance.

Fortunately recent years have seen an increase in interest in software maintenance research. The annual International Conference on Software Maintenance is flourishing with 42 papers, out of over 85 submitted, being included in the 1997 conference [54]. There is a journal devoted to software maintenance [12] and a yearly European workshop [79, 80] which focuses upon industrial aspects of software maintenance. Each year this workshop attracts a number of industrial and academic participants who share practical and theoretical experience with each other.

2.2 Aspects of Software Maintenance

There are many different aspects of software maintenance work which, when combined together, have a significant impact upon the success of a project. Three key aspects which occur in most projects are:

- **Management issues** — how resources are arranged and coordinated to provide the correct tools and capabilities at the right time, ensuring that projects keep to cost and schedule.

- **The process of maintenance** — arranging individual tasks into an order which minimises wasted effort and ensures that the results are consistent with the goals and with appropriate standards.

- **Technical aspects** — key software engineering techniques and associated knowledge which underlies the work being performed.

Each of these areas has a large number of component parts which are potential areas for in-depth research. The sheer scale of each area makes general research very difficult and specialisation into a small area is necessary if sensible results are to be produced. This thesis specialises in the technical area looking at the recovery of program structure from legacy code.

The technical aspects of software development and maintenance are perhaps the most researched area. Common goals within this field are to develop new languages, tools and techniques which help to provide more concise and less error-prone access to computer technology. This research is often done in conjunction with work from the other research areas mentioned above bringing some process and management issues directly into the engineer's working environment as part of the standard way of operation.

One example of this is the use of self-documenting code [58, 94] where comments within the program can be turned into documentation for modules, routines and procedures. The comments are therefore part of the standard program and can be changed at the same time as changes are made to the code.

Another important technical aspect of software maintenance is that of compatibility with previous versions of a program. It is rare that a program is replaced wholly throughout an organisation and previous versions must still be supported by systems which interact with the original. There are a number of approaches to solving this problem. A direct solution is to add code into the systems to determine which version of the interface they are using and then take appropriate action based upon this.

2.3 Legacy Code

The previous section has shown that software maintenance encompasses many different aspects which are all inter-related and form substantial research areas within their own right. The remainder of this chapter primarily focuses upon the technical aspects of maintenance giving an overview of individual areas which bring challenging research opportunities. Readers interested in pursuing the management and process areas further are directed towards an excellent introduction by Capretz [28].

Legacy code is a category of code which receives much attention during maintenance. There is no single precise definition of a legacy system and the name has derived from the fact that the software tends to be quite old and has been handed down as a legacy from previous generations of software engineers. Legacy code tends to be costly to maintain because it is written in old low level languages and has very poor (if any) documentation. In many cases it may be that the code was designed to run on machines which are no longer available or are expensive to run.

It is not unusual to find that legacy code is central to a companies operation and thus cannot easily be replaced. Bennett [24] presents the following as being typical reasons for continued use of legacy systems:

- The software represents, however inconveniently, years of accumulated experience and knowledge which is unavailable elsewhere;
- The manual system which was replaced by the software no longer exists, so systems analysis must be undertaken on the software itself;
- The software may actually work well and its behaviour may be accurately understood but since a replacement system may not initially work as effectively, some features of the legacy system may be worth recovering;
- Users of legacy code could be exploiting undocumented "features" and side effects, so it may be important to retain these features; and
- Users may prefer an evolutionary, rather than a revolutionary, approach.

When it does finally become necessary to make substantial changes to legacy systems a company must take steps to ensure that these changes do not unduly affect their business.

2.4 Re-engineering

When legacy systems approach the end of their life they are often replaced by a new system which performs the functions of the old system. This new system may be a direct replacement of the old, providing the same functionality, or it may involve substantial changes to the way in which the system operates, such as using graphical user interfaces. Whichever of these upgrade paths is taken it is vital that the new system replicates the behaviour of the old in a well defined manner. Slight differences in processing can cause major deviations from the expected results.

In order for the re-development to be successfully performed the old system must be sufficiently well understood. A thorough understanding enables each possible processing scenario to be taken into account allowing corresponding new behaviours to be specified. This process of re-development is known as **re-engineering**. Chikofsky and Cross [32] define it as

> "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form".

The important factor in this definition is that the original system is used as a basis for design and development of the new one. The key to the success of re-engineering projects is therefore the extraction of appropriate information from the original program. Re-engineering activities upon the original code take a number of different forms, including:

- **Code migration** — the most direct approach for re-engineering is to re-implement the original system on another platform or in another language. This is known as **code migration** or "porting". The reason for performing this type of operation is generally to allow the program to be run in a different, often more up to date, environment while retaining the exact semantics of the original system.

  Typical examples of this include Sneed's [78] work on migration from COBOL to OO-COBOL and the work of Software Migrations Ltd. [91] who convert from assembly code to a number of high-level languages.

- **Reverse engineering** — a less direct approach is to use the old system to extract information about the design, structure and operation of a program. This is then used as an aid to re-development using traditional methods. Chikofsky and Cross [32] note that typical operations include:

  - identification of the system's components and their inter-relationships; and
  - creations of representations of the system in another form or at a higher level of abstraction.

Examples of this type of work come from the field of de-compilation where object/assembly code is converted back into the original source language, e.g. REDO [18] and Lake & Blanchard [60] who use a number of techniques to recognise and extract common patterns of code which are produced by compilers.

Other work in this area, e.g. ReForm [43], allows reverse engineering of any assembly program. These programs include hand written assembly code which may have many programmer specific characteristics. The code is restructured into a high level representation of the original using sophisticated techniques which manipulate the form of the code rather than just by using pattern matching.

- **Program understanding** — a less direct use of the old system is to use the code to gain a better understanding of its operation without necessarily changing the code itself. The code contains precise details about how data is used and information about the business rules that the system implements. Understanding the code can help to clarify these enabling design documents to be retrospectively produced. In many cases the program understanding work will only concentrate upon the overall structure of the code although a few parts of the program may be examined in greater detail.

  The tools and methods for aiding program understanding use a number of different techniques for presenting and extracting information from source code. Some present the code verbatim allowing easy traversal of subroutine calls and data dependencies within the program. The code can often be annotated by the maintainer to record interesting details about the program. Recent work in this area includes Vifor 2 [76], ReThree C++ [13] and MuMMi [66] all of which use world wide web browsers as a base for program display.

  Other tools present the code in a variety of different formats which highlight different aspects of the program. The information which is presented often includes details about the structure-of or relationships-between different parts of the program. The presentation of the data often includes visual metaphors such as call graphs and dependence trees [16, 26]. More recent work is investigating the aspect of three dimensional graphics; a good summary of this work can be found in Young's survey [97].

The above examples exhibit a common theme which involves using information from the original source code to aid future work. Tool support for these operations is desirable to reduce the costs and timescales involved during re-engineering. This thesis concentrates upon theoretical aspects of tool support for these activities, in particular for reverse engineering. It examines how existing formal techniques can be extended to deal with program data structuring information. A full examination of these existing techniques is presented in chapter 3.

Legacy code is a prime target for re-engineering because very little of the original design information is generally available and correct performance of the replacement system may be crucial for future success. The code which is being re-engineered can be examined from two main viewpoints:

- **Information oriented** — concentrating upon information which is processed by the program and stored within its data files. This typically involves deriving the relationships between data objects and converting these into different storage forms, e.g. relational databases, flat files, etc.

- **Source code oriented** — involving a primary emphasis upon the source code. Identifying the order in which operations are performed to process the information which is presented as input to the system.

The former is more oriented towards business process re-engineering focusing upon the information that a company holds. The latter is concerned with the actual program itself and is more relevant to the work presented within this thesis. It involves an examination of both the control and data structures present within the code to determine the actual operation of the software.

Note the distinction between information and program data: information is the data when viewed from the point of view of the business; and program data is viewed in relation to the execution of the program. The lifespan of the information will generally be greater than that of the program data which is limited to the execution time of the program. The following discussion is restricted to the area of source code oriented re-engineering. Further details about information oriented analysis is available in the literature, e.g. Yang & Bennett [95].

2.4.1 Source Languages

Source code re-engineering is performed upon a number of different programming languages each of which have their own characteristics and demand different capabilities from re-engineering tools. Languages which are closer to the machine architecture, such as binary and assembly formats, typically require reasoning about the low level interaction with the machine. Higher level languages require less machine specific information but may rely upon characteristics of particular data types. Typical re-engineering operations upon these classes of languages are described below.

- **Binary code** — the object code for a particular machine. This often has very little information about the structure of the program and does not conform to structured programming conventions. Much of the code being re-engineered in this way is often legacy code where the original source code has been lost. This type of problem has been approached by Bowen et al. [18] as part of the REDO project examining the de-compilation of code to reproduce the original source.

  More recent interest in this type of re-engineering has centred upon retargeting binary object files onto other platforms. Sun Microsystems [84] have developed compatibility libraries which allows virtually any code written for Solaris 1.x to run on Solaris 2.x based systems. Work by Cifuentes at the University of Queensland [33] takes a more general view of this examining retargeting onto other machines with different processor architectures. This is done by converting each machine instruction in the object file into an equivalent instruction for another machine. The result is a new object file which runs natively on the other machine.

- **Assembly language** — a large amount of assembly code is still in use today. Many organisations are taking steps to re-engineer this code into higher level languages. Some are attempting to do this using a very low level approach utilising tools to convert the code. Others are extracting the design and then re-implementing using traditional software development techniques. The ReForm project [43] concentrates upon the former type of method and has developed methods for the recovery of high level descriptions of legacy assembly code.

Problems occur when performing these operations because programming shortcuts such as macros have to be taken into account. These can severely change the structure of a program from a simple, easily decipherable, version into a complex structure which has many side effects and is difficult to understand.

- **Legacy languages (COBOL & Fortran)** — many organisations see the code written in these languages as a major problem. This arises from a combination of the relatively unstructured aspects of the language (i.e. lack of subroutines in COBOL) and decreasing use of these languages for new software engineering tasks. The latter means that educational establishments are not teaching older languages as part of their standard curriculum and therefore the numbers of software engineers and application writers who understand these languages is declining.

  Re-engineering of software written in these languages concentrates upon conversion into newer languages such as C and Ada. This allows structured programming concepts to be introduced into software as part of the re-engineering work. Sward [85] reports work in progress developing a formal solution to this problem when applied to legacy Fortran systems. The Fortran code is converted into OO-Fortran with suitable classes and methods being extracted for key data items. Similar work was also performed on the REDO project [63] converting COBOL into Z++, an object oriented version of Z.

- **C, Ada, Modula-2, Pascal** — these newer languages have not yet achieved legacy status. There is very little work aimed at re-engineering from these languages although much of the work mentioned above is targeted at re-implementation into this category of language.

  Research into the maintenance of these languages tends to concentrate upon methods for understanding code to aid the tasks of defect correction and functionality extension. This research is not directly related to this thesis although the theoretical methods presented within this thesis could be used to provide a solid formal foundation. Many methods of program understanding are based around heuristical processes which recognise common scenarios presenting them as possible interpretations of the program.

- **Object-oriented languages** — these languages have rapidly become a popular choice for systems implementation. As such the main focus of re-engineering research in this area has been focussed upon converting procedural programs into data-oriented versions.

All of the above language categories provide a number of challenging problems for data re-engineering. This thesis focuses upon assembly languages to help in the development of the research. The use of assembly languages is especially beneficial because it allows previous work upon control flow re-engineering to be extended to allow the data to be manipulated using similar techniques.

Assembly languages have a number of features which are useful for ease of working. The code is easy to read and convert into a format which can be manipulated using tools. Each instruction usually performs one purpose with well defined side effects and a behaviour which can be seen to be equivalent to the representation within the transformation system. This is in contrast to languages like C and Ada which have a number of simple constructs (i.e. loops and conditionals) but also have a number of attributes which can be applied to these to make the behaviour much more complex.

Conceptually binary code is produced using a direct mapping from assembly code. Thus it would be simple to use the binary code for re-engineering purposes. Unfortunately binary object code often has linking and other machine specific information included which distorts the view of the code. This can be processed reasonably easily but it is difficult to visually inspect the results of re-engineering to ensure that they are true representations of the original code.

2.4.2 Re-engineering Techniques

The re-engineering tools and projects described in the previous sections use differing approaches to achieve their goals. Each varies in the amount of complexity, formality and change in the program's semantics. Some perform very direct changes using a simple translation from one language to another. Other systems use very sophisticated restructuring to present the final code in an appropriate, efficient way. These different techniques are presented below along with a brief description of their characteristics.

- **Direct conversion** — this is the simplest method of conversion which involves a statement by statement translation and does not necessarily take the language features of the source and target systems into account.
- **Stepwise conversion** — where a small amount of restructuring is performed on local areas of code. This would typically involve the translation of a small portion of assembly code into loops and conditionals.
- **Adding procedures and modules** — involving the identification and extraction of modules from the source code. Analogous operations can be performed upon data to recover/improve structure.
- **Event driven** — many systems are being converted from procedural into event driven paradigms which requires the separation of systems into areas which are related to different actions and events. This type of work may include the conversion of text-based user interfaces into window-based systems.
- **Full restructuring** — the integrated restructuring of many system aspects has been attempted by a few projects and tools, e.g. ReForm [93, 90] and REDO [62, 63]. These combine many aspects of the first three methods into one.

This thesis continues previous work on the ReForm project extending the restructuring methods for program control flow to include support for data structuring. This allows the properties of program data to be used to aid the re-engineering process.

2.4.3 Formality of Re-engineering

Correctness and the ability to reproduce results are key factors during re-engineering. The final result needs to have a well known relationship to the original system to allow planning for testing and introduction of the new system. The use of formal techniques brings a solid theoretical underpinning into this process.

They provide an auditable trail from basic axioms about program structure and behaviour to the theories which are used to manipulate the program. This allows the consistency of each theory to be checked and gives a deterministic description of the behaviour of the theory.

The use of formally defined re-engineering methods does not guarantee the correctness of program manipulations. Each theory depends upon the correctness of the axioms that it is derived from. If these axioms are not entirely correct then the theory may fail under certain circumstances. This is not necessarily a bad thing provided that the possibilities are explored in advance.

Ensuring absolute correctness has costs in terms of capital expenditure, effort and elapsed time. These have to be weighed against the needs of a project. For instance, the control systems of an aircraft may require absolute correctness whereas failure of a digital watch under certain extreme circumstances may not justify increased expenditure.

Not every re-engineering method uses formal techniques as a basis for correctness. The following three point scale shows how projects can be classified according to the degree of formality that they use. Each method will lie somewhere on a scale from methods which are based upon heuristics and therefore have very little formality, to methods which use a full theory to perform the changes.

- **Informal** — has little, if any, formal theory underlying the method and relies on general rule-of-thumb heuristics. These are often shown to work in practice but many depend upon adherence to certain programming standards or other generally accepted ways of working. The aim of these systems is to aid the re-engineering process using an interactive approach.

- **Semi-formal** — this type of method is based upon some formal theory but is not worked through in detail to a final solution. The links between method and theory have often been shown to be valid by peer scrutiny and possibly by a limited number of worked examples.

- **Formal** — these methods are based almost entirely upon a solid theoretical foundation and the details of the method have been worked through from the key fundamental concepts. Note, however, that as mentioned earlier the theoretical foundation is often not worked through to a full implementation because of time constraints and the lack of theoretical definitions of languages.

One place where formality cannot easily be used is the actual correction of an error or the addition of new functionality. This requires user input to ensure that the corrections are made. The correctness and consistency of the new version can then be checked using the original machine and textual versions of the specification.

2.5 Summary

Software maintenance forms a large proportion of a software system's cost and continues over most of the lifetime of a product. It is essential that a high quality product can be maintained over a number of years to ensure that the product does not get left behind by new technologies and requirements that affect its operation.

Problems which have affected maintenance over the years still remain and show no sign of being eradicated completely, if at all. In many cases systems are becoming more complex because of the distributed nature of new operating environments and the ever increasing number of programming languages that are being used.

In this chapter we have identified the need for re-engineering of computer systems and have examined techniques which are being used to aid the maintainer's tasks. These techniques have ranged from:

- simple one-to-one conversion of program statements which allow programs to be re-engineered quickly but produce inefficient code because the features of the target languages and architectures cannot be fully utilised; to
- complex restructuring systems which ensure that the code is in a format which can fully utilise the target system capabilities. This work often takes much more time and involves more support for the maintenance activities.

The latter is seen as a key technique but benefits from the use of formal techniques to ensure that the re-engineered system performs correctly without any unexpected changes in the program's semantics. In particular there is a need to be able to manipulate both program statements and data structures. In the next chapter the use of formal transformations within the re-engineering process will be examined.

Chapter 3

Transformation Systems

Re-engineering can take up a large portion of software maintenance budgets requiring considerable effort to ensure that the correct results are produced. The subtle complexity of software means that many minor details must be checked to ensure that side effects from changes to the program do not unduly affect the operation of the program.

In the previous chapter formal tool support was shown to be desirable because it takes much of the effort and risk away from maintenance activities. This chapter examines formal program transformations which provide a way of solving many of the problems which have been identified.

Transformations allow a program to be restructured into a form which makes future maintenance easier. They are often used to clean up heavily modified areas of code which have lost much of their clear, consistent, use of data and control flow paths. Transformations can also be used to isolate specific areas/aspects of the program which are to be modified. This makes the replacement or modification of that code much easier and reduces the risk of unwanted side-effects.

The first part of this chapter presents an overview of how transformations work and describes how they are represented in theory. The effects of transformations are divided into three categories: refinement, abstraction and restructuring. These help to classify the effect that the transformation has upon the program semantics.

A detailed analysis of the practical aspects of transformation systems is presented. A number of key features are highlighted which affect the usefulness of the system. These factors will be taken into account during prototyping work later in the thesis.

Data is one major aspect of programs that is typically overlooked in transformation systems. This is especially true for software maintenance where the main effort has been aimed at re-engineering low-level, typically assembly language, code.

The second part of this chapter looks at data re-engineering in detail. It includes an examination of a number of different techniques for performing re-engineering. These techniques are not limited to program transformations and include both practical and theoretical work.

This chapter concludes that data re-engineering is a desirable feature in transformation systems for software maintenance. This must be integrated with control flow transformations to allow full exploitation of the data semantics during re-engineering.

3.1 What are Formal Transformations?

Formal transformations are a means of describing semantic equivalence and refinement relations between two program fragments. A program can be restructured by applying the transformation relation to the initial program. This produces a modified program whose structure is defined by a combination of the transformation relation and the original program. The resulting program will typically differ by one or more aspects of syntactic representation or execution characteristic.

The transformation relation does not necessarily map every possible program fragment onto an equivalent fragment. The transformation mapping may only be valid for a restricted set of programs which have certain characteristics. These characteristics determine whether the code exhibits the specific behaviour that the transformation is manipulating. Without this behaviour the resulting program would not function correctly.

Preconditions are used to guard each transformation application ensuring that the code has the appropriate characteristics to guarantee the correctness of the transformation. These may require that certain statements, e.g. gotos, do not occur in the program fragment, or that a variable has a specific range of possible values.

The transformation is described formally as

$$T : P \leftrightarrow P'$$

providing that

$$(f_1(P) \wedge f_2(P) \wedge \ldots \wedge f_n(P)) = true$$

In this description P is the initial program fragment and P' is the final output from the transformation. The program fragments can be described in any suitable logic. Chapter 4 details the logic used to describe the semantics of programs in the Maintainer's Assistant. The remainder of this chapter will assume that a suitable logic is available without going into specific details.

$f_1() \ldots f_n()$ are functions which take program fragments and return a boolean truth value depending upon the characteristics of the fragment. These functions are the preconditions of the transformation and must evaluate to true for it to be valid.

The proof of the transformation is done by showing that relation T always holds when the preconditions are true, i.e.

$$(f_1(P) \wedge f_2(P) \wedge \ldots \wedge f_n(P)) \Rightarrow Valid(T)$$

Valid is a function, in the logic used to define programs, that determines whether all possible mappings between program fragments in T are equivalent[1].

The structure of the initial program fragment P can be included as an applicability condition to the transformation. This provides an assertion to say that the program actually has an appropriate structure as required by the transformation. The applicability conditions can then be used as part of a pattern matching procedure for automated application of transformations.

Patterns provide a convenient way to describe the effect of a transformation, figure 3.1 shows an example of this. The transformation reorders a conditional statement by inverting the boolean condition which is evaluated at the start of the

[1] Note that in this general description of transformations, equivalence also includes refinement operations.

block. The pattern contains placeholders for the boolean condition, B, and two sequences of statements, $S1$ and $S2$.

```
if (B)                  if (Not(B))
  then S1                 then S2
   else S2      ≡          else S1
fi                      fi

  Pattern 1               Pattern 2
```

provided that applicability conditions $f_1(), \ldots, f_n()$ hold.

Figure 3.1: A Program Transformation

The resulting program fragment, pattern 2, has the original B replaced by $Not(B)$ and the two sequences of statements swapped. The applicability conditions for this particular transformation are trivial, requiring that statements and boolean objects are properly formed constructs. The proof is correspondingly simple using the definition of conditionals which states that the execution of each branch is determined by the truth value of the boolean expression.

Other transformations require more complex applicability conditions which reflect uses of particular variables or the use of constructs which cause the flow of execution to change, i.e. loop exit and goto statements. The presence of these within specific fragments of code may cause different control flow paths to be taken or may corrupt the values held within particular memory locations.

3.2 Re-engineering using Transformations

Transformations are useful for re-engineering work because they provide the capability to restructure code with confidence that the semantics of the program will not change. Much of the effort required in re-engineering is directed towards preparation for the modification. This involves restructuring the code to either understand it or to isolate the specific functionality which is being changed.

Formally proven transformations provide a means of using formal methods without needing to understand the proof techniques involved. There is a clear separation between the proof-of and the use-of transformations. The theory is encompassed in a clear description of the change that is to be made to the program. This description can be coded to produce a transformation engine which provides support for the maintainer.

The effect of changes to the program are generally broken down into three categories: refinement, abstraction and restructuring. This classification is especially useful for re-engineering transformations because it provides a way to describe the effect that the transformation has upon the program's semantics. Each of these three categories is examined in the following section and some important issues are discussed.

3.2.1 Refinement

Refinement is a key concept in formal methods application. It is the means by which a specification is transformed into a more precise form. This involves the introduction of more specific statements about how operations are performed rather than just what is performed. Refinement is often described in terms of levels of abstraction. A program is said to be refined when an abstract program is replaced by a less abstract program which performs the same task as the original. The new version specifies the behaviour of the program in more precise terms than the original. This results in a different description of either the algorithms or data items present within the program. The new description is presented in terms of concepts which are closer to the machine representation of the program and therefore are nearer to a directly executable implementation.

Non-determinism can be used to describe how refinement affects a program's semantics. Non-determinism occurs because the specification allows a number of possible outcomes for an operation or allows the operations to be performed in a number of different ways. These all satisfy the specification and an implementation is free to use any of the possible methods. Refinement of these non-deterministic programs constrains the behaviour of the program down into a subset of the original behaviour. The final result must still satisfy the specification but does not have to allow all possible behaviours that were present in the original.

It may be thought that refinement is not relevant to software maintenance activities because refinement is used to perform forward engineering activities. Maintenance and re-engineering activities are not limited to reverse engineering, however, and contain a substantial amount of development activity to implement new features in code.

3.2.2 Abstraction

Abstraction is another term which is usually used in the concept of forward engineering, e.g. Meyer [69]. It is used to describe the design of a system in terms of components which have correspondence with either real-world objects or common implementations of high-level data structures. In this context abstractions are developed to make the implementation of a system easier by separating the design of the system into smaller components which reflect natural structure within the concepts that the system is implementing.

The work presented here uses the same definition of abstraction but re-interprets it in terms of reverse engineering where abstractions are produced from existing code which implicitly contains possible abstractions. Analysis of large amounts of legacy code has shown that there are often a number of potential abstractions which can be recovered from the code. Work such as RE^2 [34, 27], Ident [26] and Griswold et al. [46] have used a number of techniques for analysing code to identify these abstractions.

Abstraction is the reverse of refinement involving the crossing of abstraction boundaries from less to more abstract. There is one key difference between the two, abstraction cannot weaken the specifications which govern the operation of the program. If this were the case then the operation of the abstracted code could exhibit different characteristics to the original. This would be a clear breach of correctness for the program and could result in the program being reverse engineered into a specification which allowed the program to perform a totally different function.

The key to useful abstraction work in a formally-based system is to define the properties of the program which must remain unchanged after transformations have been applied. Other properties can then be changed to restructure the program in the desired manner. One example of this includes specifying that the printed output from the program must be unchanged but that the time taken to produce the result, the program's time efficiency, may change. Alternatively constraints may be placed upon memory usage, instruction set usage or data type usage.

3.2.3 Restructuring

The two previous categories of transformation have considered the change in the level of abstraction of program data. Restructuring covers the other areas where the data is converted to a different representation at the same level of abstraction. This means that the code retains the same form with the differences being in the use of different operations upon the data.

Restructuring of control flow may include transformation of loops to isolate invariants or to unravel initial or final iterations. These operations may make further simplifications of the program possible or may allow complex cases to be separated from simpler operations.

Data restructuring involves the change of the representation of a piece of data. For instance, a number which is represented as an unsigned value may be restructured into a signed representation. Alternatively the representation of an integer may be changed to allow different ranges of values to be stored, e.g. from 16-bit to 32-bit integers.

Transformations which perform restructuring operations are not limited to executable languages which have a direct representation in the machine. Restructuring can also be usefully applied to specification languages to change the characteristics of the description of the program.

Refinement, abstraction and restructuring when combined into a series of transformations allow complex maintenance tasks to be performed. The use of transformation techniques allows the maintainer to concentrate upon the strategic aspects of the changes to the code rather than upon general "housekeeping" tasks.

3.3 Transformation Systems

The successful use of transformations for re-engineering requires that a usable transformation environment is provided which complements and extends the maintainer's normal work practices. Maximum productivity is gained when the system presents information at appropriate times and performs as much work as possible without user intervention.

A number of transformation systems have been developed. These are not only for software maintenance, but for other areas including software development. Each system has characteristic features which affects its suitability for use in the environment which it is intended for. The following section examines these areas with regards to their applicability to transformations for re-engineering programs.

Languages which are transformed — re-engineering involves a number of different programming languages. Some projects may deal entirely with one language; others may involve migration from one language into another; yet others may deal with a number of different source and target languages. It is important that a transformation system is capable of handling a sufficient range of languages to allow these types of projects to be performed.

The exact languages used varies from system to system ranging from specification languages down to low level languages such as assembly code. Systems which have formally defined semantics tend to work with a very limited number of languages which are often custom designed for ease of transformation proof. Non-formal transformation systems tend to work with wider ranges of languages although individual transformations are not usually transferable between languages.

Automatic transformation — many transformation engines, e.g. the Maintainer's Assistant [23] and ZAP [41, 42] are able to make suggestions about which transformation should be applied to a particular piece of code. Some, particularly those which form parts of compilers, perform this totally automatically and do not require any user interaction. This approach is made possible because transformation applicability constraints can be checked to determine which transformations are valid. Some form of weighting algorithm is usually applied to choose between any transformations which are applicable at the same time.

Other systems are interactive and allow the user to guide transformation. These vary in their degree of automation. In the simplest methods the user selects a piece of code and the transformation engine suggests valid transformations. At higher levels of sophistication heuristics are used to perform common sequences of transformations automatically. Even higher levels may involve the analysis of programs to search for best case solutions, e.g. Griswold et al. [46]. The more sophisticated approaches typically involve greater amounts of computation but utilise program analysis techniques to reduce the search space.

Size of the transformation catalogue — most transformation systems have a catalogue of transformations which allows the storage and retrieval of individual transformations. The catalogue serves as a basis for searching for particular operations and ensuring that appropriate transformations are applied.

The number of transformations which are available affects the usability of a system. If the catalogue is too restricted then useful work is limited because either too many small steps will have to be applied or some essential steps may not be available. The converse is also true, too many transformations makes the selection of appropriate operations difficult and places too great a strain upon the transformation systems implementation.

Re-engineering transformation systems are typical of this requiring a well balanced set of transformations to ensure that maximum advantage is gained. It is therefore necessary to take care when developing the set of transformations and it is best to use an expert who can use his knowledge of the subject domain to include strategically important and the most frequently used transformations. Ward [89] supports this, saying that a domain expert is essential to develop the most appropriate catalogue of transformations.

It is also necessary to ensure that the set of transformations is complete, providing a comprehensive set of operations which cover any transformation requirement. Some less common transformations may have to be built up from the repeated application of simple transformations.

User extensibility — a transformation system is not necessarily limited to the catalogue of transformations which are supplied with it. User extension may be desirable to allow new transformations to be added. These may take one of two forms:

1. Those which are found to be necessary for continued work or desirable to incorporate new types of transformation. In a formally based system these would have to be proved by the user.

2. "Super transformations" which combine the effects of a number of simpler transformations into a commonly used transformation. This may involve the use of heuristics to guide transformation and would require some form of "glue" language to allow the combination of individual transformations and the unwinding of failed sequences.

The capability to extend the transformation catalogue is especially useful in maintenance work when new patterns of coding are discovered or when changing to different source and target languages.

Efficiency of operation — efficiency can be a prime factor in the operation of any computer system and transformation engines must be designed with this in mind. Much of the implementation of the transformation engine involves traversing programs checking that transformation applicability conditions hold. Automatic systems have extra efficiency requirements because they have to perform searches to find suitable candidate program fragments to transform.

The transformation approach can therefore have a high order of complexity. Transformation engines must utilise innovative implementations to reduce this complexity and, therefore, give reasonable performance when working with medium to large scale transformation projects.

An example of this is Semantic Design's use of combined control and data flow graphs for internal representation of the program [9]. This allows an easier search of the program for dependencies and even allows some reordering transformations to be performed with almost no computation overhead. This storage method does not directly represent the textual structure of the program. If two consecutive statements do not depend upon each other they will not have any control or data inter-dependencies and can therefore be reordered in the textual structure of the program.

Applicability conditions provide an opportunity for increasing the efficiency of the transformation search engine. Some conditions may be easier to compute than others, relying upon static properties of the program. These properties can be cached within the internal structure of the program and used whenever necessary. A careful ordering of applicability checks can ensure that these static checks are made before more complex dynamic ones. This ensures that search time is minimised thus increasing efficiency.

Formal or informal — transformation systems do not have to be based upon formal theories to provide a useful method of re-engineering programs. Informal systems can be developed based upon similar transformation operations. These do not have full proofs which show that they do not change the semantics of the program.

An informal system may be designed to get the transformation correct 99% of the time. In the remaining 1% of cases the program may be too complex or may have subtle features which invalidate the result. The reasoning behind this is that it is better to perform most of the work automatically leaving a small amount of work to be done by hand. In many cases the transformation engine may be able to determine that the transformation may not produce a correct result. Lake and Blanchard's work [59] takes this approach and their tool flags assembler code which it determines is probably going to be converted incorrectly.

There is a tradeoff to be made between informal and formal transformation systems. In general it is much quicker to develop an informal system. Confidence in the results will be gained after testing and after experience using the system on real projects. Formal transformations may take longer to develop but provide a much greater degree of confidence in the results and give a solid formal underpinning which can be used to examine any suspected errors.

Individual transformation systems take different approaches for each of the aspects discussed above. The choices usually depend upon the type of work that the system is designed to perform, e.g. forward engineering or reverse engineering. Table 3.1 summarises a number of transformation systems, it is based around a similar table presented by Bull [23, figure 3.2] but has been extended to include recent work (identified by appropriate citations).

Name of System	**Type of Transform**	**Catalogue Size**	**Automatic or User-Driven**	**Languages**	**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 + Intermediate	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
Maintainer's Assistant	Spec. ↔ Code	Large	User Driven	Wide Spectrum	Yes
Refine [22]	Various	User-Constructed	Semi-Automatic	Various	No
Sward's Work [85]	Code → OO-Code	Being Developed	Semi-Automatic	Fortran	Yes
Grundy's Work [48]	Algebra → Algebra	User-Extensible	User-Driven	Algebra	Yes

Table 3.1: A Summary of Transformation Systems

A lot of the early work in transformation systems concentrated upon forward engineering. More recent work has started to apply transformation techniques to re-engineering. These tend to concentrate upon control flow transformation and only manipulate data when necessary. The next section examines existing data re-engineering work.

3.4 Data Re-engineering

Most of the transformation systems examined in the first half of this chapter concentrate upon program control flow rather than upon aspects of data structuring and representation. This is mainly because control flow has historically been the major focus in programming. More recently data has come into the forefront of computer technology. Object-oriented programming and abstract data types have been used to break the structure of a program down into manageable chunks.

In this section a number of different techniques for data manipulation are presented. These are not limited to transformational or software maintenance techniques but represent the range of technology which can be used to manipulate data. For the purposes of this discussion the work is grouped into three categories:

- **Theoretical work** — which uses various techniques to manipulate the data. These are generally reported as "text book" techniques although other work uses these techniques practically.
- **Transformational work** — which uses transformations to manipulate the data representations.
- **Proof-oriented work** — using goal-based techniques to show that data manipulations are valid.

Each category shows the range of work which is performed. Specific examples are given where appropriate.

3.4.1 Theoretical Work

There are a number of major pieces of theoretical work which have been influential in the development of data re-engineering. Ideas from these have been used in other work which has taken a more practical approach.

Data reification — reification is a term used to represent

> "the transition from abstract to concrete data types and the justification of the transition (Jones [57])"

It has been applied widely in research on the Vienna Development Method, VDM. Data Reification is used to refine specifications into implementable versions. Andrews [3] gives the following example of the typical operations performed using reification.

$$\begin{array}{c}\underline{Abstract}\\ \underline{Data}\\ \bullet Sets\\ \bullet Sequences\\ \bullet Maps\end{array}\left\{\begin{array}{c}\underline{Data}\\ \underline{Representation}\\ \bullet Scalars\\ \bullet Arrays\\ \bullet Structures\\ \bullet Pointers\end{array}\right.$$

Figure 3.2: Data Reification Operations

The links between abstract and concrete types are described using **retrieve functions**. These are based upon the premise that the relation between abstract and concrete representations is one-to-many (a particular abstract type has many implementations). Reversal of the retrieve function is not generally possible because machine implementation limits constrain the behaviour of general abstract types.

In some cases reversal is possible, especially for composite types which have definite representations. Finding the specific instances of a type within a program does however prove to be difficult.

The refinement calculus — a major piece of work in the area of program refinement is Morgan's refinement calculus [71]. This work is based around a wide spectrum language which allows the semantics of both executable and non-executable constructs to be specified consistently.

Morgan notes that executable programs are characterised by the use of assignment commands which denote that a specified value is given to a particular variable.

This command is deterministic and is used as the basis for one of Morgan's primitive refinements, figure 3.3. This refinement states that as a result of the refinement then all occurrences of w will have been replaced by E. That is, the specification $W : [pre, post]$ can be replaced by code $w := E$. Also note that the inclusion of x in the specification allows variables which are not assigned to in the code.

assignment

If $pre \Rrightarrow post[w\backslash E]$ then

$$w, x[pre, post] \;\sqsubseteq\; w := E.$$

□

Figure 3.3: Morgan's Law 1.3 [71]

Data refinement is done by the introduction of concrete variables which implement the more abstract variables. These are introduced into the program using primitive variable introductions and are linked to the abstract variables using invariants. The properties of the program can then be rewritten in terms of the concrete variable using the invariant as a link between the two.

Data normalisation — data normalisation is a technique used extensively in database environments. It allows duplicated copies of specific pieces of information to be removed from a database and reduces the complexity of the relationships between different database tables.

There are a number of different levels of data normalisation. These are described as **normal forms** which specify the allowable relationships between different data items. The initial work in normalisation was performed by Codd [37] and consisted of three normal forms. Subsequent work has extended the number of normal forms. Different works in the area use differing definitions of each form. Further information on these can be found in Date's "An Introduction to Database Systems" [39].

The relationships between different normal forms are well defined and a number of techniques can be used to convert the tables within the database from one form to another.

3.4.2 Transformational Work

There is little work in the program transformation field which has primarily addressed the maintenance of data structures. A small number of transformations are used which lie on the boundary between data structure and control flow manipulation. These tend to be transformations such as "rename variable", "create local variable" and "replace variable with expression". Each performs simple changes to the program by moving assignments and uses of data around the program. They do not generally affect the values stored within data locations. Doing this would require knowledge of the semantics of the data values and their uses. For this reason existing transformation systems tend to treat the data as "black box" units which cannot be opened but may be transfered to different areas of the program. This approach has stemmed from early maintenance transformation work which needed to limit semantic and computation complexity. It also fitted in with the main aims of restructuring existing programs which tended to be written in assembler and other low level languages.

More substantial work has been performed using program transformations for the development of software. In this work the specification of the program is usually transformed into a more concrete form and the representation of data is changed into a description in terms of directly implementable data types. The transformations are not necessarily used to create the final program implementation but are used to produce a design which can be used by a programmer.

The Munich CIP project [7, 8] is one such example. It is based around a wide spectrum language, CIP-L. This work develops the software in a number of stages [6]. These start from a formal problem specification, which then passes through intermediate recursive solutions and is finally transformed into iterative procedural, machine-oriented programs.

An example given in [6] shows the development of an iterative implementation of a routine to determine whether graphs contains cycles. The implementation of this routine uses primitive high level data types, e.g. sets, which were defined within the original specification. The main development of the program was done using transformations which introduced control flow constructs such as recursion and iteration.

Recently practical re-engineering work using data transformation has been performed by Xinotech [55]. This company is using transformations as the solution to a number of problems including year 2000 and European currency conversions. There is very little information available about the technical aspects of this work but their advertising documentation claims to handle a wide number of legacy languages, including COBOL. The work appears to be based upon pattern matching of common representations of particular features of programs and data.

3.4.3 Proof-Oriented Work

Proof-oriented techniques are the traditional way of applying formal methods. They differ from transformational techniques because the final result is initially selected using informal techniques, e.g. heuristics. A proof is then constructed to show that the final goal is a correct representation of the original program. The proof may take a number of steps and may involve a number of sub-goals which reduce the complexity of the proof. These types of proof are similar to proofs in mathematics.

Maintenance research includes a considerable amount of data re-engineering work done using proof-oriented techniques. The REDO project is well reported and provides some of the most comprehensive data re-engineering in the field. The REDO work [63, 21, 20] examines the extraction of data structure from legacy programs and shows how COBOL code can be reverse engineered to produce specifications in Z++. The data restructuring aspects of this revolve around the identification of groups of data items which can be used to form abstract data types. The work includes the fundamental theoretical steps which are needed for extracting data groupings from the original program.

Program development using proof-oriented techniques has been well researched. This work has generally been used for development of initial specifications into forms which are almost directly implementable. The final stage of implementing this has often been performed by hand to ensure that efficient implementations are produced. Newer work, e.g. RAISE [47] and B [1, 61], is starting to use automated code generation techniques to provide a more complete development environment.

Refinement in Z — Z [74, 82], along with VDM [56, 14], was at the forefront of the surge in interest in formal methods. Z is built around set-theoretic concepts which are used to represent changing relationships between data items. Specifications represent the changes in these relationships as the program is executed.

Refinement in Z consists of the re-expression of data items in terms of other more implementation specific data items and more explicit specification of the operations upon this data. The correctness of the refinement steps is shown in two ways:

- Data representations are changed by specifying an invariant which describes the relationship between new and old versions. This is then used to show that the new form can be deduced from the original.
- Refinement of the relationships between data items is done in a similar way. In this case explicit invariants are not needed because the original definition acts as an invariant. Algebraic laws are used to show that the new relationship is a correct refinement of the original.

Z has been used successfully in a number of projects. One of the most prominent was in the specification of a significant portion of the IBM CICS product [73] during a major redevelopment operation. The specification describes the behaviour of individual subsystems and helps to specify the interfaces between subsystems. These specifications were refined to form detailed designs of the final code.

RAISE and B — the RAISE method [47] and more recently the B Method [1, 61] have examined a more implementation-oriented approach to refinement. Both methods include the ability to represent executable constructs within system specifications. This allows greater scope for refinement from abstract specifications down into representations which are directly related to the implementation language concerned. RAISE and B are conceptually similar to each other, only B will be described in detail in this document.

B is centred around the Abstract Machine Notion, AMN, which encapsulates data, its relationships (invariants), initialisation constraints and operations upon that data. Abstract machines can be nested, a machine may use, or extend, another allowing hierarchies to be formed. These combine to produce detailed specifications of the full system.

Work	Operations			Can be used for			Application	
	Refinement	Abstraction	Restructuring	Specification	Design	Implementation	Practical	Theoretic
Data Reification [3]	✓			✓	✓			✓
Refinement Calculus [71]	✓			✓		✓		✓
Data Normalisation [37, 39]			✓		✓	✓	✓	
CIP [7, 8, 6]	✓		✓	✓	✓		✓	
Xinotech [55]	✓	✓	✓			✓	✓	
REDO [63, 21]		✓	✓	—	✓	✓	✓	
Z [74, 82]	✓			✓	✓		✓	✓
VDM [56, 14]	✓		✓	✓	✓		✓	—
RAISE [47]	✓		✓	✓	✓	—	✓	
B [1, 61]	✓		✓	✓	✓	✓	✓	—

Key: ✓ = Applicable, — = Sometimes Applicable.

Table 3.2: Data Re-engineering Methods

Refinement is performed within an *implementation* of a specific machine. In this context implementation involves importing extra, more implementation specific, modules and introducing extra internal data items. Relations between the new objects are specified and proof techniques are used to show that the refinements are correct.

The use of the AMN allows stepwise refinement to be brought into a context which is suitable for use in industrial system development from specification down to implementation. IBM [52] has performed preliminary trials using the B-Method to specify and develop a substantial portion of their CICS transaction processing system. They report that the trials produced usable low level designs of the code but found that the implementation produced required a substantial amount of manual re-coding to perform efficiently.

This section has presented a number of different techniques for performing data re-engineering. Table 3.2 summarises these highlighting three aspects which are relevant to this thesis.

The first aspect shows the types of data re-engineering which can be performed by each technique. It is quite clear from the table that only a small proportion of these techniques are oriented towards performing abstraction. This represents a general emphasis upon forward engineering and highlights the need for more work on support for reverse engineering.

There is a more even emphasis upon the different phases of program development. Most work allows the crossing of abstraction boundaries but a significant number do not allow machine implementations to be produced/manipulated. They tend to work towards producing designs which have clear equivalences with an executable form and the actual conversion to/from an executable form is done by hand. This ensures that efficient implementations can be produced which take advantage of aspects which are obvious to the programmer but not to the machine.

The final aspect is the type of each piece of research. This is split between practical techniques which are used to aid the engineer on a day-to-day basis and theoretical techniques which are primarily aimed at providing the underlying theory for use in practice. The split between these is fairly even. Overall there is a fair balance of emphasis in the techniques but there is a slight lack of detailed work in the area of re-engineering of practical code.

3.5 Summary

Formal transformations are presented as a key technique for use during software maintenance. They allow code to be restructured to “clean up” after a number of years of modifications or to prepare for future maintenance. The transformations have well-defined effects upon the semantics of the program ensuring that the program’s behaviour is changed as expected.

A number of key factors which affect the usefulness of transformation systems were examined. These highlight the need for flexibility in the catalogue of transformations which are available and for efficiency of operation to minimise intrusion into the maintainer’s work.

The chapter shows that there is a lack of research into data transformation. Many methods of data re-engineering exist but they are not very compatible with the transformational approach to maintenance. The next chapter examines the Maintainer’s Assistant, a transformation system developed at Durham, in detail. It highlights the need for data transformation in the tool and the chapter looks at the changes which are required to enable data to be represented fully within the system and its associated languages.

Chapter 4

The Maintainer's Assistant

This chapter examines one particular transformation system, the Maintainer's Assistant, in detail. The Maintainer's Assistant was developed with the aim of aiding re-engineering, providing many of the program manipulation features discussed in the previous chapter. It does however lack data manipulation facilities which makes it an ideal base for the research presented in this thesis.

The chapter begins with an introduction to the theory behind WSL, the language around which the Maintainer's Assistant is based. This is accompanied by the examination of some of the main aspects of the transformations, and their implementation, which are relevant to the thesis.

The achievements and deficiencies of the Maintainer's Assistant work are summarised and show that WSL lacks data typing constructs. Without these it is difficult to represent and reason about the properties of program data making it infeasible to develop transformations for data re-engineering.

The second part of the chapter shows how data typing can be added to the WSL language. Different options are considered and a solution is presented which allows an extensible type system to be implemented. The system is based around a taxonomy of data types which groups individual types according to general properties of their semantics. These groups are: elementary types, composite types, structural types and dynamic types.

Data type semantics are incorporated within the WSL language using a shallow embedding. This allows data type theories to be imported axiomatically into the language without reproof in terms of the WSL semantics. The properties of these data types are then treated as primitive assumptions in WSL. These theories are then used to reason about any operations which involve data items of that type. Use of a shallow embedding does not invalidate the correctness of existing WSL transformations and also allows new types to be added when necessary without reproof of the transformations. These are major advantages over the use of a deep embedding.

Syntactically data types are assigned statically to variables in a similar manner to that found in many programming languages, e.g. Ada. Each variable holds values which belong to its own type. Types are defined statically within the program and are derived from the base data type from whose semantics they take their meaning. Each type definition can constrain the range of values which may be held within a variable. This provides extra information for use when transforming the data.

4.1 Theory and Implementation

The Maintainer's Assistant was developed, as part of the ReForm project [10], to explore the feasibility of program transformation for software maintenance tasks. The main aim of the work was to provide support for reverse engineering of code, particularly for the migration of assembler code to higher level languages. The work is based around Ward's theory of program refinement and transformation [88] which uses a custom designed domain specific language, WSL. This is designed to allow easy, efficient proof of program transformations as well as to represent the source code that is to be re-engineered. The theory and practice of the Maintainer's Assistant is divided into three sections:

- **WSL and Transformations** — the core theory behind the transformation system. All code is first translated into WSL from the source language. The WSL is then transformed. Once transformation is complete the WSL is translated back into a target language.

- $\mathcal{METAWSL}$— $\mathcal{META}$WSL is the domain specific language used to describe the implementation of transformations. It was developed by Bull [23] specifically for the Maintainer's Assistant.

- **The User Interface** — the user interface is designed to aid the maintainer in the selection of appropriate transformations and to hide many of the implementation details of the transformation engine.

Each of these is discussed in the following sections.

4.1.1 WSL and Transformations

Ward's [88] theory of transformations is based around the language WSL. This language was designed specifically for program transformation and has semantics which are designed to aid transformation proof.

WSL is a **wide spectrum language** which is capable of representing both directly executable, imperative statements and specifications of execution behaviour. Both of these may be mixed within one program allowing stepwise transformation between program representations at different levels of abstraction. This is especially useful for re-engineering because it allows a specification to be produced from legacy code without a change in programming language.

Executable programs are represented in WSL using a Pascal like syntax which includes a wide variety of common statements such as:

- Assignments,
- Conditionals,
- Bounded and unbounded loops,
- Subroutines,
- Action systems (for representing goto statements) and
- Local variables.

Specifications of program behaviour are represented using atomic descriptions. These allow the effect of a program to be described using a predicate which specifies the relationship between the input and output set of variables.

Semantics

Each of the WSL constructs are defined in terms of a small kernel language[1]. Definition of the WSL constructs in this way means that they inherit the combined semantics of the kernel constructs that they are composed of. Transformations can then be produced which describe changes to the WSL constructs but are proved using the underlying kernel's semantics.

The semantics are expressed as weakest preconditions using formulae of infinitary logic. A **weakest precondition** [40] for a given program S with final state R is denoted $WP(S, R)$. This states the weakest condition, on the initial state, that guarantees that the program fragment will terminate in output state R.

The use of **infinitary logic**, which is an extension of first order logic, is necessary to represent a general (non-terminating) loop in the language. Infinitary logic allows infinitely long formulae to be used within proofs. These formulae simulate the effect of infinite iteration and their uses makes proof of transformations involving loops easier because the maintainer does not need to supply suitable invariants which describe the effect of the loop.

The semantics of a programming language describe the changes which are made to the state of a program, i.e. variables, after execution of a series of instructions. In WSL this is done using Back's notion of an "atomic description" [4]. An **atomic description** is written as

$$<x_1, \dots, x_n> / <y_1, \dots, y_n>: [Q]$$

and consists of three parts:

- $< x_1, \dots, x_n >$ — a list of variables whose values are changed as a result of execution of the description. If any of these do not already exist then they are introduced into the state of the program.
- $< y_1, \dots, y_n >$ — a list of variables which are removed from the state of the program after execution of the atomic description.

[1] In actual fact the kernel language is part of WSL but for this paragraph it is best to think of the kernel as a separate language.

- $[Q]$ — a condition which must be true when execution completes.

The important feature of an atomic description is the fact that there is no concept of how the state of the program is changed. The statement specifies what the state will be after execution but any method can be used to produce the correct result. If there are multiple states which satisfy the condition then any of these is equally valid, i.e. the choice of result is non-deterministic. If no states satisfy the condition then the atomic description does not terminate and the program is said to **abort**.

Atomic descriptions allow temporary variables to be introduced into the program state. This is necessary to allow intermediate values, which are not given in the specification of a program, to be used in implementable versions of the program. An example of the use of temporary variables is in a block where local variables are defined for use within it but then cease to exist at the end of the block.

Each WSL construct has its semantics defined in terms of atomic descriptions. These are connected together using sequence, non-deterministic choice and recursion constructs. These constructs use infinitary logic to describe how the semantics of individual atomic descriptions are related. Describing the semantics of every WSL statement is beyond the scope of this thesis; instead only assignment and assertion statements will be examined in detail. These are central to the proofs of data equivalence and refinement which are presented in the next chapter.

Assignment — assignments are the main type of statement which change the state of a program[2]. An assignment replaces the value of the variable on the left hand side of the construct with a value which satisfies the expression on the right hand side of the program.

The semantics of assignment are defined as the sequential composition of two atomic descriptions as follows:

$$x := x + y \;\equiv\; \begin{array}{l} <z>/<>\!: [z = x + y]; \\ <x>/<z>\!: [x = z] \end{array}$$

[2] Specifications also change program state but are not needed for the purposes of this discussion.

The first atomic description introduces a temporary variable z whose value satisfies the condition part of the atomic description. The second one removes z from the state space and replaces the original value of x with the value of z produced in the previous stage (using the condition $x = z$).

Two stages are required to describe the semantics of an assignment because execution of an atomic description results in a state that satisfies the condition (after variables have been added-to the state space). If the assignment, e.g. $x := x + y$, involves uses of the original value of the variable (which is also to be assigned-to) then a single condition would be recursive and would assert that the new value is used rather than the original. For example, the assignment above would have a condition of $[x = x+y]$ which would only be satisfied when $y = 0$. At other times the statement would be equivalent to an **abort** and the program would not terminate.

Assertion — assertions are closely related to assignments but they do not cause any changes to the program state. They assert that a condition is true when the statement completes execution. If the condition cannot be satisfied, e.g. false, then the assertion is equivalent to **abort** and it does not terminate. In cases where the condition is satisfied the assertion acts as a **skip** statement.

An assertion is written as

$$\{(x = y)\} \quad \equiv \quad <>/<>: [x = y]$$

Here there is no change in the program state (no variables changed or removed) and the condition $(x = y)$ must be true if the program terminates. The condition in the assertion can be any boolean expression of infinitary logic.

Assertions are useful in a transformation environment because they allow properties about the program state to be stated explicitly. The information contained within the assertion can be moved around the program to places where it can be used to aid the application of a transformation.

Values

Each variable within a program holds one value at any particular point in time. A collection of the values of all variables at that particular time is known as the **state** of the program. Atomic descriptions describe how this state changes when constructs/programs are executed. Transformations do not generally reason about specific individual states. Instead they are defined for a range of possible states which have some property in common. This makes a transformation more general and applicable in many situations.

To make this description of transformations possible values are described as formulae of infinitary logic which represent their abstract or physical properties. From this it follows that variables in WSL programs are capable of representing any value which can be described by a formula in infinitary logic. Sets of possible values can also be described in this logic allowing generalisation of the state of a program. The formulae which describe these possible values are used in the condition part of atomic descriptions to specify the final state of the program after execution of that atomic description. These formulae are often written in terms of the initial state of the program thus providing an ordering of statements during execution.

If the initial values are such that the atomic description's condition cannot be satisfied then the program is in error and will not terminate. At first sight it seems that this is not desirable for a programming language because non-termination is usually considered to be an incorrect behaviour. The possibility of non-termination is not necessarily bad for a language designed for software maintenance however because the aim is to preserve the original behaviour of the program. If the program originally did not terminate then it is acceptable to transform the program into an **abort** statement.

Proof of transformations in WSL is done in the manner described in section 3.1. Each construct within the program fragment is mapped onto its kernel language semantics. These are then used to show that the transformation does not corrupt the semantics of the initial program fragment.

To make the proof easier a number of lemmas are used which describe useful properties of the program and allow considerable proof reuse. These include lemmas which state that if two program fragments do not use or assign to common variables then they are independent and do not depend upon each other. Full details about the proof of transformations and the semantics of WSL can be found in Ward's thesis [88].

4.1.2 $\mathcal{META}$WSL

Implementation of the transformations is done in a variant of WSL called $\mathcal{META}$WSL. This language is an extension of WSL containing all of the statements in the core language plus extra ones for representing, manipulating, traversing and querying WSL programs. The use of $\mathcal{META}$WSL brings two major advantages to the implementation of the transformation engine:

1. The transformations are implemented in a language which is independent of any particular programming language or machine and

2. It is possible to transform the implementation of the transformation during future maintenance.

Implementation consists of conversion from the theoretical representation into a machine representation. Efficiency is a key requirement in this process, a transformation must take the minimum amount of processing time because repeated application of transformations can cause exponential growth in processing time. $\mathcal{META}$WSL helps constrain the amount of work required to produce efficient and concise transformations by providing a number of constructs which perform the operations that are most commonly used during transformations. Development time can then be concentrated upon making these efficient.

To aid the maintainer in the selection of suitable transformations the Maintainer's Assistant is able to check applicability conditions for a number of transformations. These are targeted upon a user selected area of code and the system is able to present a list of applicable transformations which will make similar changes to the program.

The applicability tests are used to guard the main transformation code. They do not guarantee that application of the transformation will succeed but reduce the likelihood of failure substantially. The main code is responsible for checking the remaining transformation preconditions. These are typically those which take substantial amounts of processing time, and can signal that the transformation has failed. If the transformation fails then any changes that have already been made by the transformation are unwound leaving the program in its initial state.

Further details about *MetA*WSL and its innovative use in the implementation of a transformation engine are given in Bull's thesis [23].

4.1.3 The User Interface

The user interface plays a major role in the usefulness of the transformation system. It helps the maintainer by providing easy access to the program which is being transformed and helps in the selection of appropriate transformations.

Transformations have been grouped according to the effect that they have upon the program. Each transformation is centred around a key component of the program fragment being transformed. This takes the emphasis away from incidental details of the transformation and focuses upon the primary effect of the transformation. The groupings are shown in table 4.1. They make it easier for a novice maintainer to use the Maintainer's Assistant by cutting down the search space when choosing a suitable transformation.

Type	**Description**
Move	Moving constructs around the program.
Join	Join adjacent constructs forming a composite construct.
Use/Apply	Use or apply the construct to neighbouring constructs.
Reorder	Rearrange the order of statements or expressions.
Rewrite	Rewrite the current construct in a different form.
Insert	Insert a new construct into the program.
Simplify/Delete	Simplify or remove constructs.
Multiple	Perform an action many times.
Complex	Complex restructuring operation.

Table 4.1: Transformation Groupings

4.1.4 Analysis of the Maintainer's Assistant

Over a number of years a large amount of practical experience of using the Maintainer's Assistant, and its commercial counterpart FermaT [81], has been collected.

This highlights a number of areas where the tool excels and also shows a few places where it lacks functionality. A summary of these findings is shown in table 4.2; more detail on each aspect can be found in appendix A.

Accomplishments		
	Code Restructuring	The rearrangement of a number of program constructs including gotos, loops and procedures.
	Program Editing	An editor allows errors to be corrected or changes made to the program during transformation. These are recorded in an audit trail.
	Multiple Source Languages	Code originally written in a number of different languages has been transformed.
	Formally Defined Semantics	WSL has formally defined semantics which are used to show correctness of program transformations.
	Practical Experience	The transformation tools have been used in a number of practical situations and the results have shown that the theory is correct.
Deficiencies		
	Data Typing	This is desirable to represent exact semantics of different data objects and to separate the behaviour of logically different activities.
	Data Abstraction	Data abstraction is a key factor in software engineering activities.
	Language Translators	These are not formally defined and insert a weak link into transformation work.
	Selecting Transformations	The order of transformation application is a key factor in the success of the technique.
	Backtracking	In some cases a simple Undo/Redo facility is not enough.
	New Language Constructs	WSL cannot easily represent language constructs such as exceptions.
	Data Reasoning	Stricter reasoning about laws of arithmetic etc. is required.
	Modularisation	The system cannot handle modules and libraries very well.

Table 4.2: Analysis of the Maintainer's Assistant

From this summary we conclude that the Maintainer's Assistant has achieved the primary goal of enabling useful restructuring of legacy code in a wide variety of programming languages. The system does this using formally defined transformations which have been demonstrated to be correct in a number of practical situations.

One area which needs further work is the need for increased rigour and formalism in the area of data manipulations. This recurs under a number of different headings in the table and involves the following problems.

- **The current theory and transformation system does not provide sufficient control over data structures and relies upon translators to ensure that data structuring is preserved** — the translators insert markers into the program to represent the data types. This information is used after transformation when the program is being translated into the target programming language.

- **In many cases existing transformations may violate data structure properties because the inbuilt knowledge about the properties of data operators assumes that they are used upon certain data types** — if a program uses a data type which has semantics of operators which differ from those of common data types, e.g. integers, then the transformation system will not be able to recognise this and take appropriate action.

This lack of data transformation support is the subject of this thesis and is covered in detail in the remainder of the thesis.

4.2 Adding Data Typing to WSL

The first stage in addressing the re-engineering of data using formal transformations is to ensure that the transformation language can represent program data in sufficient detail. This allows the transformation engine to explicitly annotate the program with the information that is needed to reason about data. It also makes the language more like current programming languages, e.g. Ada, Modula-2, C, C++, which have strongly typed representations.

Note that unlike other programming languages the use of data types to allow type checking during compilation is not a primary consideration in WSL. The transformation system assumes that programs which are being transformed are already type correct and any discrepancies are taken to be part of the program's semantics.

The addition of types into WSL can be broken down into three components:

- **Available types** — the types which are representable within the language. These must provide a complete set of types which are used in both programming and specification languages.
- **Semantics** — the semantics of the data types need to be integrated with the semantics of existing WSL constructs. This makes proof of transformations possible.
- **Syntax** — the syntax of the standard WSL language needs to be extended to allow data types to be explicitly associated with values and variables.

The following sections examine each of these points in detail. The set of types required is identified using a taxonomy of data types. This classifies types into groups which represent the semantics of the values stored and their typical uses within a program.

Integration of syntax and semantics is explored by examining different strategies and evaluating them to ensure that the existing language and transformations are not unduly affected by the changes. It is also important that the resulting programming language is capable of representing code in a form that maintainers can work with.

4.2.1 Available Types

Many different data types are provided by programming languages. These **data types** provide a way of describing the set of possible values that an instance of that data type could assume. Each data type has a number of operations associated with it which allow manipulation of data instances and reasoning about them. Each language is designed with the needs of different domains in mind and as such each language provides data types which differ subtly from those provided by similar languages.

Implementation constraints also affect the semantics of data types due to word sizes, arithmetic operations and memory allocation policies. If transformation of data types is to be feasible then it is necessary to be able to cope with the varying demands of different environments while still providing a stable platform around which to base work.

Many languages allow **subtyping** of data types to create logically distinct instances of a type which inherit the properties of the original type but are treated as different types when describing the semantics of the program. This enables separation of the program into a number of segments which represent the logical design of the program.

A Taxonomy of Data Types

There are a number of primitive data types which form an elementary group of types within a language. These are combined together by other types which allow the representation of groups of values. These groupings have two semantic properties which are of interest for our data transformations:

1. **number of components** — the grouping may have a static number of components (fixed at compile time) or the number of components may vary dynamically.

2. **naming of components** — a particular component may be accessed via a static name, i.e. a record component, or it may be accessed via a dynamic name which is computed at runtime, i.e. array elements.

These properties of groupings affect data transformation because they influence the ease with which a specific transformation may be shown to be valid. In general data types with static properties are easier to transform because the properties can be determined without extensive analysis of the program.

Category	**Primitive**	**Number of Components**		**Naming of Components**	
		Static	**Dynamic**	**Static**	**Dynamic**
Elementary	✓				
Composite		✓		✓	
Structural		✓			✓
Dynamic		✓	✓		✓

Table 4.3: Data Type Categories

For the purposes of this thesis four categories of data type are defined. Table 4.3 shows how these categories correspond to the semantic properties. Each category is also discussed below:

- **Elementary types** — these are the most basic that occur within programs and are the types which are commonly found as primitives within programming languages. They are used to describe individual aspects of a logical entity. Examples include: discrete types (including integers), real numbers and sets in Pascal.

- **Composite types** — these allow grouping of individual variables together to form a new object which can be used as a single entity. The structure of the data is therefore represented in a more manageable format which provides basic functionality for data abstraction. The most common example within this category is the record (structure) where the components are usually instances of elementary types or of other composite types. Programs often provide subroutines to allow controlled access to composite data.

  The static nature of composite type semantics makes it possible to apply control flow transformations to each component of the data type individually. This requires special support from the WSL semantics. Sections 4.2.2 and 4.2.5 show how this is allowed as part of the definition of typed WSL semantics.

- **Structural types** — structural types differ from composite types because they have component names which are computed dynamically. The most common example of this data type is the static array. Note that a static number of components is important because it provides a fixed range of items which can be accessed.

- **Dynamic types** — a dynamic type is more semantically complex than the other types because it is not generally possible to determine if a particular component exists at any point in time. If a component which does not exist is accessed then the semantics of the data may be undefined. Types with either static or dynamic naming are included within this type category because the dynamic number of components has more effect upon the data type's semantics than component naming semantics. Common examples of these data types are dynamic arrays and lists.

Table 4.4 lists common data types and shows which categories each type lies within. This grouping has been extracted from an analysis [72] of a number of common programming and specification languages, including: Ada, Modula-2, C, C++, Java, Pascal, Lisp, VDM, Z and B.

Data Format	**Category**			
	Elementary	**Composite**	**Structural**	**Dynamic**
Bit	✓	✗	✗	✗
Integer	✓	✓	✗	✗
Boolean	✓	✗	✗	✗
Enumeration	✓	✗	✗	✗
Character	✓	✗	✗	✗
Real — Fixed	✓	—	✗	✗
Real — Float	✓	—	✗	✗
Set (in Pascal)	✓	✗	✗	✗
Array	—	✗	✓	✓
String	—	✗	✓	✓
Record	—	✓	✗	✗
Tuple	—	✓	✗	✗
List	✗	✗	✗	✓
Tree	✗	✗	✗	✓
Graph	✗	✗	✗	✓
Relation	✗	✗	✗	✓
C-Union	✗	✗	✗	✗
C-Pointer	✗	✗	✗	✗
First-class function	✗	✗	✗	✗
Object-oriented class	✗	✗	✗	✗

Key:

✓	Denotes that a type is available in that category.
—	Denotes that the type could reasonably be expected to appear in that category, but in practice it is not usually found there.
✗	Denotes that the type is not found in that category.

Table 4.4: Common Data Types

Some data types may appear in more than one category. This is because the data may have more than one possible interpretation which governs the operations which may be performed upon that data value. A typical example is an integer which is treated in many languages as a number, an array of bits or a truth value. These may often be mixed together, especially in C or assembly code, to reflect the current needs of the programmer. This is seen by Hatton [51] as laziness on the programmer's part, leading to error-prone code. However, there are a number of occasions where the representation of these data values must be mixed, especially when interfacing directly with hardware.

The last four entries in table 4.4 represent data types whose semantics are complex and are therefore not fully represented within our classification. The c-union is complex because it allows its components to share the same storage. This means that an assignment to one component may affect the value stored within another. The c-pointer is complex because it does not represent a particular value but merely allows a value to be identified by dereferencing the pointer.

A first-class function is one which may be stored as a primitive data value. This is not supported by the WSL kernel language. An object-oriented class is complex because data type inheritance introduces problems for static determination of the exact type of an object. This means that it is not possible to statically determine which instance of a method (subroutine) should be called at a particular point within the program. Support for both of these would require significant extensions to WSL.

Each of these data types presents difficulties which are not easily overcome. This thesis will concentrate upon the core data type categories and will not examine the other data types further.

4.2.2 Data Type Semantics in WSL

The semantics of WSL needs altering to accommodate the semantics of the data types described above. This is a crucial prerequisite for successful data transformation and it must be done in a manner which minimises the effect upon existing transformation work. The main factors which affect the choice of a method for extending the WSL semantics are listed below:

1. **The introduction of data typing should have a minimal effect upon the original theory** — our goal is to extend the previous work rather than to redevelop it.

2. **The set of data types should be extensible** — to make the system as flexible as possible it is desirable that new data types can be added easily to

support new source and target languages.

3. **Reuse of data type theories is desirable** — it is sensible to use other peoples' theories about type properties. There is little point in the reproof of complex data type theories.

4. **Future work may involve the integration of a theorem prover with the Maintainer's Assistant** — any transformations and theories about data types could be checked during transformation to ensure correctness.

These factors are taken into account in the remainder of this chapter when a suitable method of adding data typing to WSL is discussed. The extended language is known as **typed WSL** and we begin by examining how data typing affects the values which are stored within a variable.

Values in Typed WSL

Typed WSL has a different model of data values to that of untyped WSL. In typed WSL the entire set of possible values is segmented into a number of disjoint sets. These subsets group together values which share common properties and provide the basic description of a **data type** in typed WSL.

The set of values which represent data types are deliberately disjoint because the physical characteristics of data types make logically equivalent values distinct when used on a physical computer. For instance, an integer which is represented as a floating point number does not have the same binary representation as an integer represented as a two's complement value. The use of disjoint sets of values is also beneficial because it allows the data types to be integrated into WSL using a shallow semantic embedding. This gives extra flexibility for the addition of new data types and is discussed fully on page 68.

As a consequence of the disjoint nature of sets of data values it is not possible to compare the semantics of one data type directly with those of another. Instead equivalence of data type semantics is shown by demonstrating that two expressions produce equivalent output. Both expressions must produce output of the same type and need only produce equivalent results for the possible ranges of input values associated with a specific instance of that expression.

This model of type equivalence presents the data type as a "black box" whose semantics can only be judged against other data types by comparing the output for a given input. This corresponds to Ward's [88] model of program transformations where the fragment of code which is transformed is taken to be a "black box". The inside of the box can be transformed at will providing that there is no visible change in the operation of the program outside of the box.

The typed model of WSL data is represented in Ward's original untyped model by encoding individual data values as a value-type pair (full details of this can be found on page 65). The type serves to differentiate between different sets of values. This allows a countably infinite number of different types whose set of component values may also be countably infinite. The typed model can be shown to be representable in Ward's untyped value model using Gödel numbers [87][3].

Data Types

Data types serve to constrain the values which may be stored within particular variables. The data type specifies the properties of values and as such acts as an invariant[4] over all of the values of that type. This means that each atomic description will contain appropriate type invariants for those variables which are assigned-to within the operation.

An assignment statement is therefore described as

$$<z>/<>: [(z = x + y) \wedge (z \in \mathbb{Z})];$$
$$<x>/<z>: [(x = z) \wedge (x \in \mathbb{Z})]$$

In this example the assignment is to an integer variable and the assigned values are constrained to be members of the set of integers ($z \in \mathbb{Z}$). The type predicate can

[3] Gödel numbers allow any countably infinite set of values to be represented distinctly from any number of other countably infinite sets of values. This requires an infinite number of possible combinations (values) but this is still countably infinite and therefore fits into Ward's model of data values.

[4] The invariant serves to identify the set of values to which an individual type belongs. This is necessary to retrieve the value from the unique data value which has been encoded using Gödel numbers.

take any appropriate form and may be any valid expression in infinitary first-order logic.

In typed WSL programs the type predicate will be composed of a number of individual parts. These are generated from the type theory and the syntactic components of the language. The components of the predicate include:

- **The resulting type** — showing the type to which the result belongs.
- **The general description of the result** — a description of the result of the expression in general. This serves to identify the legal return values[5] of the particular type.
- **Subtype constraints** — more specific constraints for a particular subtype which is derived from a base type.

This approach to type definition allows reuse of type theories while allowing introduction of more specific information for particular program fragments.

Embedding Data Type Theories

The principal change required to the theory of WSL is to incorporate the data type theories into the existing semantics. Melham [68] and others [17, 19] describe two approaches to this: deep embedding and shallow embedding. These differ in the degree to which the semantics of the two are intertwined.

- **Deep embedding** involves the definition of both the syntax and the semantics of the embedded theory within the host language. This makes the embedded theory become part of the host's semantics and allows proof of new theorems.
- By contrast **shallow embedding** only involves the immersion of the syntax of the embedded theory within the host language. Any theorems about properties of the embedded theory are imported into the host language and used as though they are primitive definitions.

Boulton et al. [17] and Bowen & Gordon [19] discuss advantages and disadvantages of each method. These have been summarised in table 4.5. When viewed in

[5] The return values will all belong to one data type.

our context within WSL a deep embedding would involve the definition of individual data types in terms of the kernel language semantics. Equivalence of these structures could then be proven using the semantic definitions.

Alternatively the use of a shallow embedding would involve the proof of the data type properties outside of WSL and these would have to be imported along with their related equivalence theorems. The transformations would then rely upon the correctness of these.

	Advantage	**Disadvantage**
Deep	Allows reasoning about classes of programs.	Setting up semantic functions can involve a lot of work.
Shallow	The WSL-to-data type interface handles the mapping between programs and their semantics.	Mechanised proof of the combined system is not possible — the system is less secure.
		It is not possible to state generalised theorems about the embedded types.

Table 4.5: Advantages and Disadvantages of Different Embeddings

The criteria presented above (on page 60) supports the use of a shallow embedding to bring the data types into the periphery of WSL because it minimises the impact of data types upon the language. The main disadvantage of this is the lack of direct proof of theories of data types within WSL. Upon further examination, however, shallow embeddings have already been used within the existing implementation of the transformation engine and the design of the language tends towards this type of approach.

The current (untyped) transformation engine uses a symbolic mathematics and logic module to provide reasoning about data values. This embodies primitive knowledge about common operators such as associativity and commutativity and also allows evaluation of expressions using constants wherever possible. The module does not consider the limits of data types and is therefore limited in usefulness. For example, it assumes that integers are unbounded and therefore overflow and wrap-around semantics cannot be taken into account.

This model of data properties is in practice an implicit use of a shallow embedding within the language. It provides a proxy for the theorems which govern the behaviour of primitive data operations. Bull [23, section 5.14] acknowledges the need for proof of these theorems stating that a more formal version is necessary for high degrees of confidence in the implementation. Currently the transformations trust that the symbolic maths routines produce correct responses to requests for verification of assertions about the equivalence of data items.

The design of WSL itself tends towards the use of a shallow embedding because it does not deal directly with the contents of data locations treating them as though they are black boxes whose equivalence is known.

Extending the WSL Data Model

The addition of data type semantics into WSL requires three main changes to the underlying semantic definition.

- The first is a change to the model which is used to represent the data space within the language. This involves the association of a data type with every value within the language which makes it feasible to check the correctness of operations and transformations which originally made explicit assumptions about data type properties.
- The second change is to use these type associations to represent the shallow embedding of data types within the language.
- The third is to extend the name to variable mapping semantics to allow composite types to be represented in the embedded semantics. This is not strictly necessary but it allows the individual components of a composite data type to be re-engineered using the original WSL control flow transformations.

The state space of a WSL program consists of a number of variables which have a value, or one of a set of values[6], assigned to each. Conceptually this consists of two relations: *env* and *store* (see figure 4.1).

[6] In a non-deterministic program the precise value which is assigned to a variable may not be defined.

The *env* function represents the mapping between a variable name and a storable location, the **l-value**. The result of applying this function returns a value which can be used on the left-hand side of an assignment statement. This l-value can then be used to retrieve the value stored at that location, the **r-value**. The r-value is retrieved using the *store* relation. This returns the actual data value which is stored within the variable. It may then be stored in another location or used as an argument to a subroutine.

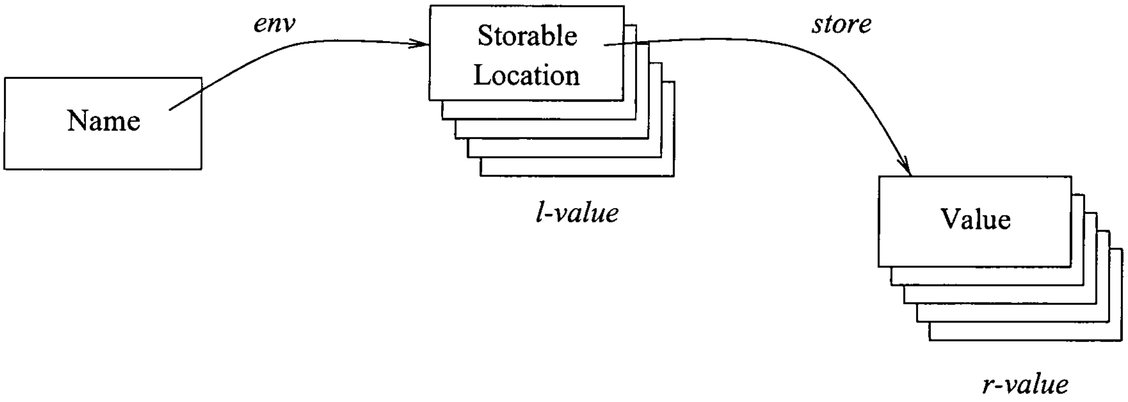


Figure 4.1: The WSL Data Model

Semantically, types are introduced into the environment as part of the r-value. This is done by redefining the r-value as the Cartesian product of a data value and its type;

$$\text{r--value} ::= \textit{value} \times \textit{type}.$$

Subroutines and operators are now passed a number of these value-type pairs as their parameters. The type is used during execution to select the correct version of the operation[7] and the value is used as a parameter to the operation. During transformation the type is used to select appropriate semantics for the operation. These semantics are then used to determine whether the transformation is valid.

This use of value-type pairs makes the data type of parameters to subroutines explicit. This is different to the original scheme where the data type was implicit.

[7] Note that in practice selection of the correct operation is made at compile time, i.e. it is a static binding.

Note that WSL is not intended for software development and therefore does not benefit from the ability to check for consistent use of data types at compilation time. It does however benefit when transformation of data is performed because it allows direct checking of type semantics rather than the original implicit assumptions that the semantics were appropriate to the operation.

Composite Data Types

A composite type is composed of a number of statically named components. Each one of these components is independent of the others (except by virtue of their grouping) and in principle it could be transformed using WSL's control flow transformations. Unfortunately the use of a shallow semantic embedding makes this difficult because a WSL data object is treated as an indivisible entity. This means that an assignment to one component of the composite type would have to be written as

$$x := \mathit{comp\_assign}(x, \mathit{comp\_name}, \mathit{value});$$

instead of

$$x.\mathit{comp\_name} := \mathit{value};$$

In the former case the *comp_assign* function takes the original value of x and changes the value of component *comp_name* to a new value. This is a self-referential assignment to x and WSL does not understand that this is actually just changing one component of x.

To alleviate this problem it is necessary to extend the variable naming mechanism by adding an extra level of indirection into the semantics. This is done using definitional transformations which ensure that the underlying transformation theory is not invalidated. Figure 4.2 shows the new data model. In this model the names which are used in the transformation theory are not directly available at the syntactic level. Instead a new class of name (*name*$*$) is defined. This can be used in two ways:

1. to map onto a primitive name (via the *id*() function) or

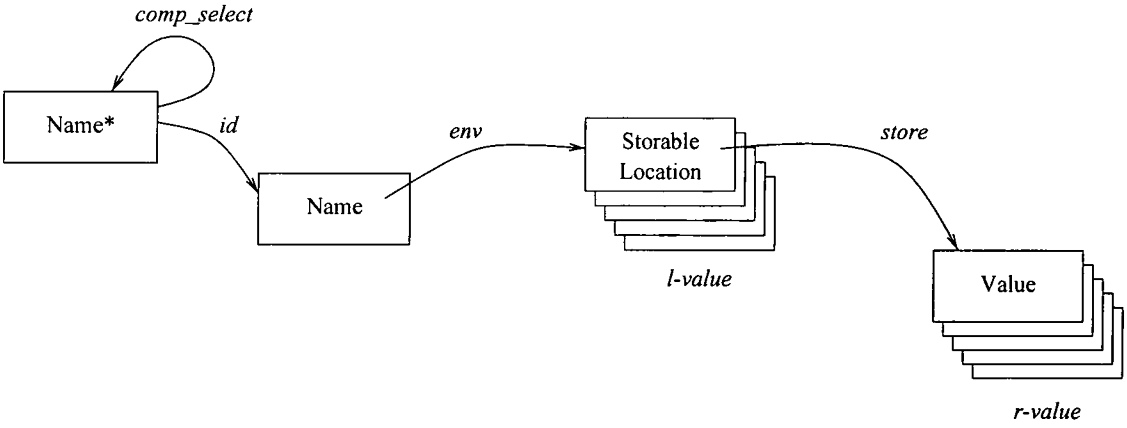


Figure 4.2: Composite Type Semantic Extensions

2. to map onto another *name**, via the *comp_select*() function, when used in conjunction with a component name.

This allows a tree of names to be constructed. Each node of this tree identifies either a unique variable or a group (subtree) of other variables. Only the leaf nodes in the tree are allowed to map onto a semantic name and there are no circularities in the tree. This latter fact ensures that composite types cannot be recursive and prevents the representation of dynamic data types using this method. Section 7.3.6 shows how dynamic data types can be represented in typed WSL.

The l- and r-values of individual components of a composite type can be retrieved using the **l-select** and **r-select** constructs. The former returns the variable name associated with the component and the latter returns the value that a specific instance of the component contains.

4.2.3 Using a Shallow Embedding

The shallow embedding of type semantics now comes into play with all theories about data values and their equivalences being defined as axioms which are referenced by the type portion of the r-value pair. These axioms are imported from the external proofs of the data type theories and used as primitive facts by data transformations. The net result is that the semantics of the data types are separated from the transformation theory although the axiomatic definitions of the data type properties must be expressed in terms of the infinitary logic that the WSL theory is based upon.

The r-value represents data values which correspond to a particular data type theory. These values may be arbitrarily complex as required by the theory. In particular dynamic types, those which represent dynamic data structures, have a model of their environment built into the type theory. This contains memory models which denote the storage model present within the theory of the data type. This neatly separates the potentially complex model of dynamic types from the WSL theory making proof of both easier.

A side benefit of using a shallow embedding is that type theories do not have to be fully proven before transformation work can be performed. This gives some degree of leeway in the choice of balance between formality and practicality.

It is not the intention of this thesis to show proofs of each data type but merely to describe them in sufficient detail for the underlying principles to be clear (see chapter 7).

4.2.4 WSL Data Type Syntax

The syntax of data typing constructs plays a major role in the effectiveness of data transformations. Syntax is closely related to the language's semantics and provides the bridge between theoretical concepts and practical programming concerns. It constrains the effect of changes to the program and helps to provide a program structure that the maintainer can understand.

Each syntactic construct maps onto a semantic description of the program which describes the effect of the execution of that construct. This mapping can be arbitrarily complex but must ensure that there is only one possible meaning for each program. Note that the reverse is not necessarily true because the same semantics can be represented in more than one way.

WSL's syntax has been extended using a static data typing mechanism which binds individual variables to a data type. Each variable can only hold values of a specific type and may only be used in expressions which accept input of that type. Data types are derived from primitive type categories. This allows subtyping to create logically distinct data types allowing the use of primitive types for different purposes within the program.

Many constructs within the language have changed to allow the data type syntax to be represented. These changes include:

- the addition of a type definition operator;
- binding of variables to types whenever a new variable or formal parameter is declared;
- explicit typing of all operator parameters;
- explicit type conversion where types do not match;
- selection of components for composite types and
- a construct which defines the types of external variables.

Each change has been made after careful consideration of a number of factors which affect the mapping of syntax onto the underlying syntactic program representations. These factors have been expressed as two questions which guide the choice of type binding operators:

- **How should types be declared?** — should a type be declared to allow reuse of that type or should it be declared each time it is used?
- **When and how are types bound to values?** — this defines the lifetime of the binding of a type with a value. Should the type be: bound dynamically as execution proceeds; bound once when a variable is declared; or bound statically when the program is compiled? What syntactic mechanism is used to show the binding?

Each question has been answered after consideration of the needs of transformation implementations and user requirements. When transformations are implemented the transformation engine must be able to efficiently extract data typing information from the program that is being transformed. This requires extensions to $\mathcal{META}$WSL and the code which implements transformations needs to be able to look in well defined places for the data typing information.

The user of the transformation system is affected by the syntax of the language. It should resemble common programming languages to make it easier to understand the concepts present within the language. If constructs do not resemble similar ones in other languages the user may find it difficult to work effectively with WSL.

How should types be declared?

Declaration of types is a primary requirement for introducing data typing into WSL. It allows an explicit definition of the properties of a particular data value. These properties are represented by the general category of the data type, for example integers, real numbers or records, and specific constraints upon the range of values within the base category.

Each value needs to be associated with one of these types and there is typically a large amount of repetitive use of each type. To reduce this repetition it is convenient to give each specific type a name. This name is then used wherever the full type definition is needed, making the program less complex and providing a basic method of determining the equivalence of data types.

Type declarations have been added to WSL to provide the mapping between names and specific data types. These declarations occur in **where** statements and are declared in a similar manner to procedures and functions. The data type is given a **type_name** which can be used within the **where** block.

The syntax of the actual data type in the declaration depends upon the base type category. In general it consists of the type category name and an appropriate representation of the constraints upon the types within the category. This loose definition of type expression syntax makes extension of the set of available types easy.

The siting of type declarations within a **where** block has a number of advantages for implementation of transformations it provides a well-defined location where the details of any type can be found and allows reuse of much of the $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL code which searches for procedures and functions.

Another benefit of the use of type declarations is that it provides a convenient means by which **subtypes** may be declared. The subtypes are used to provide extra information during transformation. They constrain the range of values which may be held within a variable further than the base type theory. This may make it easier to determine if a specific transformation is applicable.

When and how should types be bound to values?

Transformations must be able to determine the types of individual values within the program. This allows reasoning about the properties of these values and the subsequent use of this information to determine the result of a transformation. Values are generally stored within variables providing a means to represent the program state. The type of a variable is therefore a primitive concern of a data transformation. Extraction of this type from the program is an important operation which must be efficient and easy to perform.

Types may be bound to values in a number of ways. Two possibilities for this are: association of types directly with values and allowing variables to hold values of only one type. The former is known as dynamic typing and is used extensively in languages such as Common Lisp [83]. Conversely variables could be given a static type and may then only hold values of that type. This is used in languages such as Ada [2] and Modula-2 [38] which have strong data typing models.

There are a number of ways of representing both dynamic and static type binding within the syntax of the language. These are presented below along with an analysis of their suitability for use in typed WSL.

- **Associating a type with each value** — this method associates a type with a value whenever the value is mentioned within the program. The value may then be assigned to variables or used as a parameter to functions and procedures. Each value must be associated with a type allowing explicit determination of its properties. Use of this method introduces a large amount of extra complexity into the code and would be appropriate for a dynamic typing scheme. It does, however, have a very clear mapping onto the underlying semantics of the language.

- **Associating types with some values** — not every value needs a type associated to it. Those operations which retain the type of a value, or only read the value, do not change the type and are unnecessary. This removes a lot of the extra complexity present in the previous method but makes processing of the code to determine the type of a value more difficult. The type associations are not placed at fixed places within the code which means that the execution flow of the code must be traced to find the correct type.

- **Associating types with values at declaration points** — another variant of the explicit association of types with values is when the type must be stated in the first assignment to a variable (i.e. at the definition). This makes the association occur at a deterministic place and ensures that each variable has a specific type. Problems occur in WSL when functions are considered because the return value of a function is defined by the type of the expression that it contains. The return type of the function would therefore be hidden within the function definition.

- **Associating types with variables** — an alternative approach is to introduce an explicit type binding construct which binds variables to specific types. This gives a deterministic place where the type of a specific variable can be retrieved. The type binding is explicitly stated for each variable declaration and formal parameter to a subroutine. This means that subroutines become specific to a particular type. Some programs may become semantically incorrect using this scheme if they overload subroutine parameters with different data types. This is a minor problem and most, well-behaved, programs do not utilise this capability. Those programs which do can be converted by defining a "union" type which allows variables to have any appropriate type.

- **Associating types using statements** — some languages, e.g. Common Lisp, allow types to be associated with variables using statements. These statements are usually placed at the beginning of blocks and declare the type of a variable from that point onwards. In Common Lisp this is an annotation for the compiler and allows optimisation of any subsequent commands. Using this method it is not easy to find the type of a particular variable but it allows the language to be amended easily because the only addition to the language is a new category of statement.

- **Using assertions[8] to associate types** — This is similar to the previous entry and involves virtually no change to the language. The only change is

[8] An assertion is a type of statement whose successful execution ensures that the program state satisfies a specified boolean condition.

the addition of a new assertion condition which explicitly states the type of a variable. This suffers from the problems discussed above but fits well into the transformation system because assertions are designed to carry information around the program.

Binding Method	**Binding Class**	**Link with underlying semantics**	**Syntax**	**Legible**	**Auto-mation**
Type ➔ Value (all)	dynamic	✓✓	✓	✗✗	✓✓
Type ➔ Value (some)	dynamic	✓	✓	✓	—
Type ➔ Value (decl)	static	✓	✗	✓	✓
Type ➔ Variable	static	✓	✓	✓	✓
Statements	both	—	✓	✗	✗
Assertions	both	—	✓✓	✗	✗

Key: ✓✓ = Very Good, ✓ = Good, — = Average, ✗ = Poor, ✗✗ = Bad

Table 4.6: The Pros and Cons of Type Binding

Each of the methods presented above have a number of strengths and weaknesses. Table 4.6 summarises these showing how well each fits in with various aspects of the transformation system. Many provide suitable solutions for use in a transformation environment but association of types with variables has been chosen for implementation of data typing in WSL. It provides a static data typing environment which is similar to that used in many programming languages. Use of this method provides a direct relationship between variables and their associated types which can be retrieved easily from the program when necessary. These changes to the language do affect a number of transformation implementations but their correction is simple.

Type equivalence in WSL is defined as simple textual equivalence of the types' definitions. No attempt is made to infer any internal equivalences as this would mean that data types could no longer be treated as black-box entities.

Table 4.7 shows the syntax of the changed language constructs and two new constructs which represent external variables and data type conversions. External variables are those variables which form the program's interface with the outside world. Data type conversions are used to ensure that a program remains type-correct when transforming data.

Component	**Appearance**
Type Declaration	**begin** *$statement$* **where** **type** *$type_name$* ≡ *$type_defn$.* **end**
Variable Declaration	**var** < *var* : *$type_name$* := *$expn$* >: *$statement$* **end**
General Expression	[**e-** <*var* : *$type_name$* := *$expn$* >; *$statement$* : *$expn$* **-e**]
General Condition	[**c-** <*var* : *$type_name$* := *$expn$* >; *$statement$* : *$condition$* **-c**]
Procedure Definition	**proc** *$name$*(*var* : *$type_name$* **var** *var* : *$type_name$*) ≡ *$statement$.*
Function Definition	**funct** *$name$*(*var* : *$type_name$*) ≡ *$expn$* : *$type_name$.*
External Variables	**external** < *var* : *$type_name$* >: *$statement$* **end**
Component Selection	*var.$comp_name$*
Type Conversion	**&** *$type_name$*(*$expn$*)

Table 4.7: Data Typing Components of WSL

The typed constructs include place holders for type names (*\$type_name\$*) and type definitions (*\$type_defn\$*). Type names are simple text strings which uniquely identify a type at any point within the program and type definitions represent a specific type category and the constraints for its associated data type. Examples of these will be given in chapter 7.

4.2.5 Mapping the Syntax onto the Underlying Semantics

At this point both the syntax and underlying semantics have been defined. The final section of this chapter shows how these are linked together to represent the semantics of the syntactic elements of the language. The links are made using definitional transformations which map each typed syntactic unit onto the original WSL syntax. In this original format all data values are associated with a data type, as described in section 4.2.2. The definitional mapping is done in three stages:

1. Type declarations are expanded to rewrite all occurrences of a type name with the corresponding type definition.

2. Variable declarations and typed constructs are replaced with their untyped counterparts and each occurrence of a variable is replaced by a typed version.

3. Assignments to variables with a composite type are expanded into parallel assignments to the variable which corresponds to each component of the type. This is an additional step which is specifically for composite types and it will be discussed separately.

The result of the mapping is a WSL program with all of the data values (expressions) replaced with value-type pairs. The original WSL transformations then treat these as primitive data values. The symbolic maths and logic unit is modified to handle these but the remainder of the system is unaffected.

Example 4.1 shows how the first two stages of the mapping is performed. These convert between the new and old syntax. Version A shows an outline program which uses the new typed WSL language. This has a typed local variable block which uses a data type declared within a **where** block.

```
begin
  var < $var$ : $type_name$ := $expn$ >:
    $var$ := $expn$
  end
where
  type $type_name$ ≡ $type_defn$.
  $definition$
end
```

Version A — Typed WSL program

```
begin
  var < $var$ : $type_defn$ := $expn$ >:
    $var$ := $expn$
  end
where
  $definition$
end
```

Version B — After Step 1
(Type Declarations Expanded)

```
begin
  var < $var$ := ($expn$ × $type_defn$) >:
    $var$ := ($expn$ × $type_defn$)
  end
where
  $definition$
end
```

Version C — After Step 2
(Typed Variables Expanded)

Example 4.1: Definitional Semantics of Data Types

In version B the **where** block has been expanded to replace each occurrence of the type name, *\$type_name\$*, with the type definition (in bold type).

The final step (version C) rewrites the typed local variable block replacing all expressions (r-values) with an expression-type pair which represents the typed data values. The program is now represented in the original (untyped) program format which can be transformed using standard program transformations.

Composite Data Types

Example 4.2 shows the third step of the semantic transform where composite variables are mapped onto the individual variables that they contain. Version C is the output from the first two stages of the transform where the type definition and expression are tuples with one entry per element of the composite type. Version D shows the program with the *\$type_defn\$* and *\$expn\$* placeholders replaced by the type's definition. Version E shows the expanded form of the variable declaration and assignment. Each component of the type has been expanded into an equivalent operation on the variable which corresponds to that component.

Composite types may contain components which are of a composite type. This means that the transform has to be applied recursively until all of the composite types have been expanded into their component variables. This does not result in the possibility of unbounded expansion because a composite type is not allowed to have a component which has the same type as itself.

Individual statements can now be rewritten in terms of atomic descriptions as shown in example 4.3. In this note that each of the atomic descriptions' conditions contain the type predicate which represents the properties of the data type. This is not strictly necessary because the second step is a straight assignment and the value of *tmp* will already conform to the type predicate. It is included because it serves to remind that the value has a particular type.

In the example the type definition predicate is written in terms of the value of the expression (*expn* or *tmp*) to show that it is associated with that value.

```
var < $var$ : $type_defn$ := $expn_a$ >:
  $var$ := $expn_b$
end
```

```
where $type_defn$ ≡ ⟨ ($name_1$, $type_defn_1$),
                        . . . ,
                        ($name_n$, $type_defn_n$)⟩ and
      $expn$ ≡ ⟨$expn_1$, . . . , $expn_n$⟩
```

Version C — After Step 2 (in example 4.1)

```
var < $var$ : ⟨ ($name_1$, $type_defn_1$),
                 . . . ,
                 ($name_n$, $type_defn_n$)⟩
              := ⟨$expn_1a$, . . . , $expn_na$⟩ >:
  $var$ := ⟨$expn_1b$, . . . , $expn_nb$⟩
end
```

Version D — Type Definitions and Expressions Expanded

```
var <$var$.$name_1$ : $type_defn_1$ := $expn_1a$,
       . . . ,
       $var$.$name_n$ : $type_defn_n$ := $expn_na$ >:
  <$var$.$name_1$ := $expn_1b$,
   . . . ,
   $var$.$name_2$ := $expn_nb$ >
  end
```

Version E — After Step 3
(Composite Types Expanded)

Example 4.2: Definitional Semantics of Composite Data Types

$$var := (expn \times type\_defn) \equiv \begin{array}{l} <tmp>/<>: [(tmp = expn) \wedge type\_defn(expn)]; \\ <var>/<tmp>: [(var = tmp) \wedge type\_defn(tmp)] \end{array}$$

Example 4.3: The Relationship between Type-Value Pairs and Atomic Descriptions

4.3 Summary

The Maintainer's Assistant has been identified as an ideal candidate for further work on data re-engineering program transformations. It has a strong track record for performing control flow restructuring and has well understood and carefully defined semantics. The transformation system currently lacks the ability to restructure data and the transformation language, WSL, is not capable of representing data typing information.

This chapter has shown how the ability to represent data typing can be added to WSL. Data types have been classified into four categories: elementary types; composite types; structural types and dynamic types. These represent the different semantic uses of data within a program.

WSL's syntax and semantics have been extended to provide a statically typed language which can be transformed in a similar way to the original WSL. These extensions have been made by rewriting all the values (expressions) within the language as a value-type pair. The type is used to determine the exact properties of the data value and to provide information for use during transformation. The next chapter shows how this can be used to implement data re-engineering transformations.

Chapter 5

Data Transformation in DREAM

Data re-engineering has been identified as a key capability for software maintenance because it is important to be able to restructure program data thus complementing control flow manipulation capabilities. Chapter 3 showed that program transformation systems typically tend to lack the ability to re-engineer program data. Chapter 4 looked at one transformation system, the Maintainer's Assistant, and showed how the capability to represent data type information can be added to the transformation language, WSL.

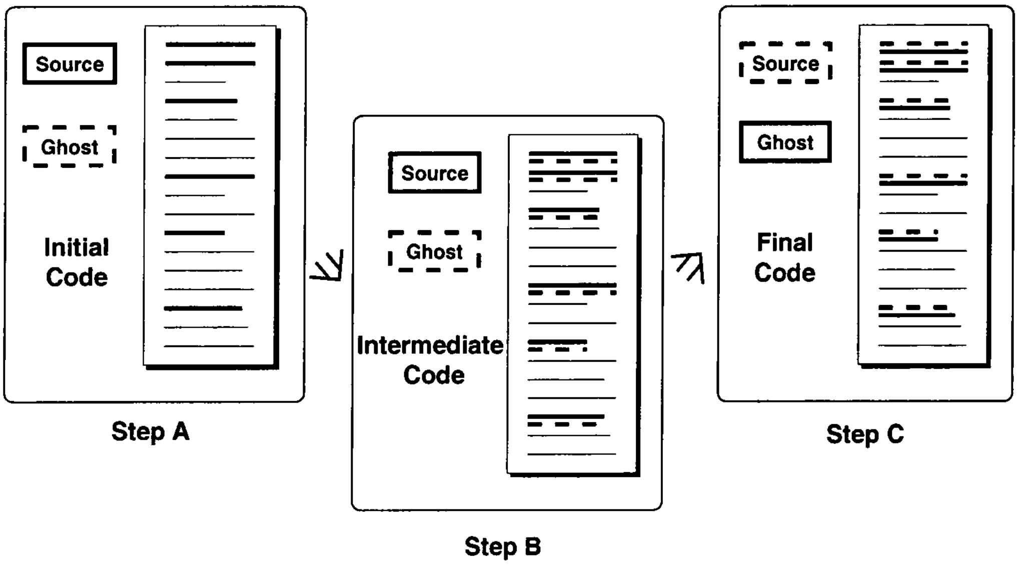
This chapter uses the resulting typed WSL language to develop formal transformations for data re-engineering. The techniques used for control flow transformation are combined with the semantics of typed WSL to produce DREAM (the Data Re-engineering and Abstraction Mechanism). This provides a powerful way of manipulating program data which is suited to the needs of software maintenance.

5.1 Overview

DREAM provides a transformation mechanism which can be used to change the representation of program data. It is based around the technique of ghosting whose suitability for use during data transformation was demonstrated by Ward [92] in his derivation of an efficient iterative implementation of the Schorr-Waite graph marking algorithm.

This thesis extends Ward's [88] work by development of a **data expression refinement relation** which describes the correspondence between an original and final (ghost) representation of the data. This relation describes a refined, or equivalent version of an expression which has the desired change in data format. Note that the data format change may correspond to a data refinement, abstraction or restructuring operation. Multiple data expression refinement relations are collected together into a **type equivalence theory** which describes transformations between specific data types.

Ghosting involves the introduction of new, ghost variables into the program. The ghost variables are introduced to replace the variables which are to be transformed. Assignments are made to these ghost variables using values which are a refinement-of or equivalent-to those values assigned to the original variables. The equivalence between the values contained within the variables is shown formally at all places where the original variable is used. Once this equivalence has been demonstrated transformations are used to replace the expressions involving the ghost variable with expressions involving the original variable throughout the program. This brings about the desired change in the data representation.



Each box within the diagram represents the scope of ghosting within the program. The horizontal lines within the boxes are lines of code where the variables in question are mentioned. The thick lines are assignments to a variable and the thin lines represent uses of the variable. The solid and dashed lines represent either the source or ghost variable which are named at the top left-hand corner.

Figure 5.1: The Ghosting Process

Figure 5.1 shows a pictorial representation of a program which is being ghosted. It shows how a source variable is replaced by a ghost variable. To illustrate this further consider the following example: the program initially uses two "real" numbers (the source variables) to represent a complex number. A transformation is invoked to replace these reals with an abstract type. The first stage of this transformation is to introduce a new variable, the ghost, which has the appropriate "complex" type. A type equivalence theory is now used to define a mapping between the two reals and the complex number. This mapping is used to introduce assignments to the ghost variable at all places where the original reals are assigned to (step B in figure 5.1). Expressions which will replace the original uses of the source variables are now developed and a data expression refinement relation is used to show that these are equivalent to the original. At this point the expressions involving the ghost variable replace those of the source variables completing the change in data representation. After this has been performed the source variables can be removed from the program because they are no longer used.

The transformation mechanism is developed with the aim of allowing four basic types of data manipulation to be performed. These types of transformation are:

- **Changing data representation** — to represent data using a different data type which has equivalent[1] semantics to the original format. This may involve refinement, abstraction or restructuring of the data type representation.

- **Changing the relationship between logical data objects** — to allow the use of a variable for a number of different purposes. For example, during graph traversal a data location may be used to mark that the current item has been visited and to store the location of the previous node that was visited.

- **Changing the scope of data** — making data visible at different points within the program. This allows data hiding and encapsulation to be introduced. Note that scope is a syntactic device but it can be a very important tool during data re-engineering.

- **Introduction of subtypes** — to constrain the values held by variables into

[1] The use of the word "equivalent" here includes refinement operations.

a subset of the allowable values for the parent type. This allows more precise reasoning about a type and allows the introduction of logical distinctions between different uses of a data type.

The data transformation mechanism is extended to allow all of the transformation types listed above to be performed. This involves removing the requirement to completely replace variables with their ghosted equivalent. To do this the extended ghosting concentrates upon ensuring that the ghosted portion of the original variable's scope satisfies the expression refinement criteria. The remainder of this chapter examines the data transformation mechanism, its theory and the operation of ghosting in detail.

5.2 Types of Transformation

The primary aim of this thesis is to investigate the transformation of program data representation to aid re-engineering. This involves not only transformations which allow data representation to be altered but transformations which affect the relationship between program code and the data. To perform this re-engineering the capability to manipulate the following is required:

1. the representation of the data;

2. the relationship between program data objects and

3. the lifetime/scope of each data item.

The work presented in this thesis does not directly concentrate upon the relationship between data representation and control flow. The application of data transformations may allow changes to the control flow structure of the program but they are side effects and are not the primary purpose of this research.

The three capabilities identified above are combined with a fourth, introduction of data subtypes, to make up four basic types of transformation which are needed for successful data re-engineering. Each of these are examined below and they are drawn together into a powerful transformation tool which allows data re-engineering to be performed.

Introduction of data subtypes is a subset of the transformations which change the data representation. They are treated separately because they introduce explicit information about the properties of data. This explicit information is used by the transformation engine when it is determining the validity of individual transformations.

5.2.1 Changing Data Representation

The main category of data transformation involves making changes to the data representation. These transformations allow substantial re-engineering of the program which includes performing refinement and abstraction operations upon the implementation of data structures. The knowledge that is required to make these changes is encapsulated within theories of equivalence between the source and target data types (see section 5.4).

These theories make use of data type information to show that a change in representation is correct. This information constrains the behaviour of the source and target variables and therefore allows equivalence theories to concentrate upon the specific properties of the allowable data values.

Typed WSL allows extra information about data values to be represented within the program. Each data type definition may place constraints upon that particular instance of a parent type. These constrained data types, known as **subtypes**, are used to make checking of individual transformations easier. Section 5.2.4 describes the category of data transformations which allow subtype information to be introduced into a program.

The transformations which make changes to data representations act upon data types from each of the four data type categories described in chapter 4. They are not limited to changing representations between types in one category but also allow changes from one category to another. Changing to another category is appropriate where data implementations have hidden structure or where a variable could have static or dynamic behaviour. Possible transformations include:

- **Converting to a different primitive format** — for example, changing from an integer representation to an enumeration.

- **Splitting into subcomponents** — breaking a value down into constituent parts. For example, when converting from elapsed seconds into elapsed hours, minutes and seconds.
- **Changing data groupings** — restructuring abstract data types. For example, to move a variable out of a record structure.
- **Static to dynamic structures** — for example, changing a fixed size array into a dynamic array.
- **Converting to abstract dynamic structures** — removing explicit dynamic data types, e.g. pointers, and replacing them with abstract dynamic data types such as lists and sets.
- **Turning structures into relations** — extracting the abstract entity that a data structure implements. For example, to replace a hash table with a relation between the key value and the elements contained within the hash table.

The exact set of transformations which are required depends upon the circumstances of a particular re-engineering task. Migration of assembly code to a high level language will typically require transformations which allow data structures to be extracted whereas those involving the reverse engineering of an implementation into a specification would require operations on higher level data types. Section 7.3 examines some of these transformations.

5.2.2 Changing the Relationship between Logical Data Objects

A variable may be used for a number of different purposes within a program. Each purpose represents a specific logical data object which may be manipulated separately or as part of a combined group. Typical scenarios include: cpu register variables which hold temporary values and variables whose value has two different meanings depending upon the context that it is used in. An example of the latter is a pointer which may be accessed to find the object that it references or it may be used to determine if there is an object which is referenced by it, i.e. if the pointer is null. These logical uses of a data item can be manipulated to separate, or join, them making the structure more appropriate to the maintainer's needs.

Separation of the contexts that a variable is used within requires the introduction of a new variable which holds values appropriate to one of the contexts. Once this transformation has been performed the two contexts/variables are distinct and may be manipulated independently, possibly with changes in the scope of the variable or simplification at the places where the variable is accessed. It also allows the separation of side effects from the actual operation that is provided by a function or procedure. For example, in C integers are used as boolean values. A value which is not zero is treated as being "true" which could be used by programmers where it is not appropriate. If a program is changed without taking account of this side effect the program could become erroneous.

There are cases where the joining of two variables is desirable. This is the reverse of the separation operation described above but is potentially harder to perform because of the possibility of interaction between the original two contexts/variables when they are stored in the same location. If an undesirable interaction occurs and the uses of the two variables interfere with each other then the transformation will fail. This type of operation is commonly applicable when optimising a program for memory usage. For example, two variables may have very similar uses/meanings and using the same value to represent both would save memory.

5.2.3 Changing the Scope of Data

Manipulation of scope is a powerful transformation tool which is based around the syntactic representation of the program. Scope allows data values to be used and referenced only at specific places within the program. Changing the scope of a data item can affect the efficiency of execution by allowing memory usage to be limited and by allowing local variables to be placed in cpu registers. Another major use for scope is to provide facilities for data encapsulation and hiding.

The transformations presented in Ward's theory provide functionality for making changes to the scope of data. They allow local variables to be introduced and allow the scope of these local variable blocks to be changed. They also allow the parameters to functions to be manipulated. This provides a sufficient range of transformations which manipulate scope. It is desirable, however, that these scope operations can be integrated with the other data transformations to provide a single comprehensive data transformation facility.

5.2.4 Introducing Data Subtypes

Subtypes allow the explicit description of the properties of the values which may be held within a variable. These properties aid both the maintainer and the transformation engine to understand the program and ensure that the application of a transformation is valid. Formally a subtype s (which may hold any member of the set of values S) of type t (set of values T) is defined as

$$s < t \equiv S \subseteq T$$

Here the symbol "<" is used to denote a subtype. This notation has been borrowed from type theoretic research [29] but the semantics of data types in typed WSL are not based upon this research.

The effects of transformations which introduce data subtypes can be grouped into two categories:

1. Those which provide explicit distinction between values which are used to represent different logical values in a program, e.g. an integer could have subtypes which represent time and weight. These subtypes have the same data properties, e.g. addition/subtraction operations, as each other but their different names distinguish them from each other.

2. More complex use of subtypes allow the properties of variables to differ. This means that information about the use of a variable can be contained within the data type definition and subsequently used during the application of other transformations.

Use of subtype introduction transformations is desirable in all types of program. Legacy programs generally require a large amount of restructuring. Their original languages do not typically allow expression of much data typing information. In these cases introduction of subtypes is useful to make the program easier to understand.

In cases where legacy programs have undergone initial re-engineering or where programs were originally written in languages with more expressive data typing information the use of subtyping for logical data type separation may not be as important. In these cases subtype information is introduced to aid future transformation. This makes explicit information, about the range of data values which are held in a particular variable, directly available to the transformation engine. It removes much of the need for the transformation engine to analyse programs to find this information.

Four types of data transformation have been identified. Together these provide a comprehensive data transformation capability but to allow these to be combined they must be based around a common underlying transformation mechanism. This mechanism must be compatible with the previous transformation research at Durham. It must integrate into the theory of transformations and must be implementable within the transformation engine with a minimal amount of change. The major aspects which must be considered are:

- **Providing a unified data transformation system** — the maintainer should be able to perform different data transformations using a single mechanism which hides the details of individual operations.
- **Maintaining the transformation approach** — data transformation should be performed in a similar manner to existing transformations.
- **The catalogue of data transformations should be extensible** — there are many different data types and the transformation system should be capable of being extended to allow these.
- **Minimise theoretical changes** — the transformations already proven rely upon complex reasoning about their effects upon a program. Minimal changes to the theory will ensure that existing work is not invalidated.
- **Minimise changes to the transformation engine** — changes to the transformation engine are undesirable because of the complexity and size of the code

which must be changed.

- **Efficiency is important** — the transformations should be executed efficiently and have deterministic time performance. In particular it is important that each transformation will terminate and will not enter infinite loops.

These factors are taken into account in the remainder of this chapter and in the next chapter which describes the implementation of the data transformations.

5.3 Data Expression Refinement Relations

The previous section identified transformations which change data representation as being the main component of data re-engineering work. These transformations involve changing the data types which are used to implement individual variables and the formal theory which describes the transformations must be able to demonstrate that the semantics of the program are not altered.

This section defines a **data expression refinement relation** which describes the relationship between source and ghost expressions. This will be used during ghosting to show the correctness of a particular transformation (see section 5.4). Note that the use of the term "refinement" describes the effect of the transformation upon the semantics of the program. The changes to the representation of the data will actually involve refinement, abstraction and restructuring as described in section 3.2.

WSL uses a "black box" model of program operation to perform transformations. A program fragment which is transformed must have identical behaviour after transformation but may change the internal ways in which the program is implemented. Chapter 4 defined typed WSL in a similar manner describing the semantics of data types using a "black box" approach. A data type is treated as a black box whose semantics can only be compared with those of another type when the values of both types are used. This use is central to data expression refinement relations.

We will define the relation in terms of two expressions

$$f(x) \text{ and } g(y)$$

where x is the source variable of type M,
y is the ghost variable of type N,
$f(x)$ is the source expression and
$g(y)$ is the ghost expression.

These expressions both return values of the same type (this allows the values to be compared). The expressions are allowed to exhibit a degree of non-determinism in their output. This means that for a particular input value the expression may return any one of a number of values. The exact one which is returned is chosen non-deterministically. This behaviour is captured by defining functions $\bar{f}(x)$ and $\bar{g}(y)$ which represent the sets of values which may be returned for a particular input. These are defined as

$$\bar{f}(x) = \{p | p = f(x)\} \text{ and}$$
$$\bar{g}(y) = \{q | q = g(y)\}$$

The data expression refinement relation is shown in figure 5.2. This states that for all of the values of x and y which are defined to be equivalent (by relation h) then the output of $g(y)$ is a refinement of $f(x)$. That is $g(y)$ produces a subset of the values that $f(x)$ produces for equivalent inputs.

In addition to this if the source expression, $f(x)$, terminates (i.e. it produces a value) then the refined expression must also terminate. This is required by Ward's definition of transformation semantics and is described by

$$((\bar{g}(y) = \emptyset) \equiv (\bar{f}(x) = \emptyset))$$

In general, the relation "h" does not have to describe the relationship between

$$\exists h : M \leftrightarrow N \bullet$$
$$\forall x, y \bullet (x, y) \in h \implies \left(\left(\bar{g}(y) \subseteq \bar{f}(x)\right) \wedge \left(\left(\bar{g}(y) = \emptyset\right) \equiv \left(\bar{f}(x) = \emptyset\right)\right)\right)$$

where h is the desired relationship between source and ghost types.

Figure 5.2: The Data Expression Refinement Relation

all possible values of the source or ghost types. It need only describe the relationship between those source and ghost values which may occur at the place in the program where the relation is used.

5.4 Data Type Equivalence Theories

Data expression refinement relations will be used at specific places within a program to show that ghosting transformations are correct. These relations are specific instances of more general theories which are used to describe data representation changes. These more general theories are named **data type equivalence theories**. Note that the use of the word "equivalence" is not technically correct because the theories could also describe data refinement and abstraction. We choose the word "equivalence", however, because it is in common use to describe relationships between similar items.

Data type equivalence theories define mappings between source and ghost data representations. These mappings do not have to be one-to-one but could be one-to-many or many-to-one depending upon the type of representation change which is being described. For example, the abstract form of a variable may represent an error as a simple "true" or "false" value. The corresponding concrete form of this may use more descriptive error codes to enable debugging and tracing of the error. In this case the type equivalence theory would represent a one-to-many relationship.

It is not immediately obvious how a many-to-one relationship can be used without affecting the semantics of the program because compressing many states into one loses information within the program state and means that incorrect output could be produced. In practice there are a number of situations where this operation is possible. The first of these is where the information content of a variable is not fully used. For example, consider the error code scenario described above; when reverse engineering this program it would be possible to change the descriptive error codes back into a boolean value because the program does not depend upon the extra information.

This last example may not be applicable in many re-engineering situations because the transformation is being performed using the implemented program as the base for reasoning. In the implemented program the error codes may form part of the output which is visible to the outside world. In this case the error codes cannot be re-compressed because the black box behaviour of the program would be altered. The solution to this problem is to redefine the program's output to separate the output which is critical to the operation of the program and other output which is not important for the aims of the re-engineering task. This latter form of output can then be removed from the program and this would make the transformation possible.

These mappings highlight an important difference between program semantics and data semantics. The data used within a program can be restructured, abstracted and refined without necessarily imposing similar changes upon the semantics of the program. In particular, it is possible to abstract data while maintaining program equivalence or refinement.

Benefits of a Shallow Embedding

The use of a shallow semantic embedding of data types into WSL allows many different data types and equivalence theories to be used during the maintenance of one single program. The equivalence theories link individual data types and can be separated into three categories:

1. Reasoning about expressions with the same base type.

2. Reasoning about expressions with different base types but with type theories which were proven in the same embedded logic.

3. Reasoning about expressions with different base types which were not proven using the same embedded logic.

The first two of these are straightforward because the type theories have the same basic axioms. This means that showing equivalence is performed by direct reference to the base theory. Of course this proof of equivalence may involve a large amount of extra proof if the already proven laws do not describe the desired relationship.

The latter case is more difficult because there is no way to show formal equivalence of the data values due to their differing basic axioms. A solution which may be adopted is to use informal equivalences which describe simple relationships between the two types. This makes transformation possible while still providing some degree of confidence that the transformations are correct.

Chapter 7 shows how the semantics of individual data types can be defined in typed WSL and shows how equivalences between these are demonstrated.

5.5 Ghosting

Ghosting provides a transformation mechanism which is used to implement DREAM data transformations. It is based around the concept of **ghost variables** which was first proposed by Clint [35, 36] to aid the verification of coroutine correctness. In his work ghost variables are added to a program using "virtual statements" which are ignored by the compiler but are used during theorem proving to store useful intermediate results.

The transformations presented within this thesis use ghost variables to allow the representation of data to be changed. Ghost variables are introduced into the program to hold values of the target type. At this point the ghost variables are unused and therefore they may have any desired values assigned to them. In practice assignments are introduced at all points where the initial variable is assigned to. Once all of the assignments to the "source" (initial) variable have been mirrored by one to the "ghost" (final) variable then the two variables have equivalent values at all points within the program and the uses of the source variable can be replaced by uses of the ghost variable. This operation is known as **ghosting**. At this point the source variable becomes redundant (it is no longer used) and the assignments to it can be deleted from the program and the variable removed.

This approach is different to Clint's initial work because in our work the ghost variables actually become used whereas in Clint's work the variables only ever hold extra information which is useful for theorem proving. His variables never become part of the program's output state.

Ghosting was chosen as a data transformation mechanism for this research because of similarities with the existing transformation mechanisms which are used within WSL. Ghosting allows the data to be transformed in a number of stages which perform distinct changes to the program. These stages are performed incrementally (much like control flow transformation application) to produce a combined effect. It is possible to change the exact details of each of the steps to produce specific results. This is in contrast to proof-oriented techniques which require proof of specific changes to a program.

The key novel component which allows ghosting to be used to perform data re-engineering is the use of a data expression refinement relation in the application of a general data transformation. This allows the use of data type equivalence theories without needing to prove specific transformations for each equivalence theory. Typed WSL provides the framework which allows ghosting and equivalence theories to be integrated easily.

An example of the use of ghosting within a transformation environment is given by Ward [92] in his derivation of an efficient implementation of the Schorr-Waite graph marking algorithm. In his work ghost variables are used to refine data structures from abstract lists and stacks into a concrete implementation which has an efficient use of data. The results of this work provided the initial impetus for exploration of the use of ghosting for data re-engineering.

The aim of this thesis is to examine the use of ghosting for data transformation. It is not the aim of this thesis to compare ghosting with the other data re-engineering mechanisms which were presented in section 3.4.

5.5.1 Theory

Ghosting provides the central mechanism for performing DREAM data transformations. The success of this relies upon the semantic correctness of changes which are made to the program. It must be possible to prove that a ghosting transformation does not unduly change the program's semantics. This is done by examining each step of the transformation and by confirming its validity. The steps which must be considered are:

1. Introduce a new "ghost" variable to the program.
2. Add assignments to the ghost variable at each point that the original variable (the "source") is assigned to.
3. Gather information about the contents of source and ghost variables at the point where the source variable is used. This information will be used in the next step to show that the variables are equivalent.
4. Perform the DREAM ghosting transformation operation and replace the uses of the source variable with uses of the ghost variable. This uses the information gathered in the previous step to show that the uses of the variables are actually equivalent.
5. Remove assignments to the source variable.
6. Remove the source variable from the program.

Example 5.1 shows each step of this operation when applied to a simple program. The theory of each step is described below and is shown to fit into the theory of typed WSL. Note that the example shows a simple case where the ghost variable has the same type as the source variable. In a more complex case where the types of source and ghost variables differ the assignments-to and uses-of the variable would use appropriate expressions which produce equivalent values although they have differing types. The validity of this is demonstrated by use of the data expression refinement relation.

Step 1 — Introducing a New Variable

The semantics of transformations which introduce new variables into a program are well understood. Ward's theory provides transformations which allow this operation to be performed.

```
var
  < source : int := 5 >:
  source := z
  retval := source
end
```

Initial Program

```
var
  < source : int := 5,
    ghost : int := undef >:
  source := z
  retval := source
end
```

Step 1: Introduce Ghost Variable (Choice of type by Maintainer)

```
var
  < source : int := 5,
    ghost : int := 5 >:
  source := z;
  ghost := z;
  retval := source
end
```

Step 2: Introduce Assignments to Ghost

```
var
  < source : int := 5,
    ghost : int := 5 >:
  source := z;
  ghost := z;
  {(source ≡ z)};
  {(ghost ≡ z)};
  retval := source
end
```

Step 3: Gather Information to show equivalence

```
var
  < source : int := 5,
    ghost : int := 5 >:
  source := z;
  ghost := z;
  {(source ≡ z)};
  {(ghost ≡ z)};
  retval := ghost
end
```

Step 4: Perform Ghosting Change by transformation

```
var
  < source : int := 5,
    ghost : int := 5 >:
  ghost := z;
  retval := ghost
end
```

Step 5: Remove Assignments to Source

```
var
  < ghost : int := 5 >:
  ghost := z;
  retval := ghost
end
```

Step 6: Remove Source Variable

Example 5.1: An Example of Ghosting

It is important to remember that in typed WSL variable declarations have types associated with them. A type must be supplied as part of the variable introduction transformation and should be appropriate to the desired result of the ghosting operation.

Step 2 — Introducing Assignments to the Ghost Variable

Assignments to the ghost variable can be freely introduced into the program. They do not affect the semantics of the program because the ghost variable is not (initially at least) used. Assertion and assignment introduction transformations allow these to be inserted into the program.

The expressions which are used to assign values to the ghost variables require careful development. There is no definitive rule to specify the form that these should take but they must be provably equivalent to the source assignments to allow the next step to be performed. A careful mixture of maintainer and heuristic guided development of these expressions is required.

Step 3 — Gathering Information to allow Equivalence to be Demonstrated

The central part of the ghosting transformation requires that the source and ghost variables are equivalent at all of the places where the source variable is used within the program. To show this information about the values assigned to the variables must be collected and made available at the appropriate places. This information is gathered in two ways: type and subtype information provides basic information about the variables and constrains the values which could be held in the variables; assertions provide the remainder of the information which is gathered from the places where the variables are assigned to.

The data type information is static within the program and requires very little effort to extract it. This is the reason for presenting "introduction of subtypes" as a basic form of data transformation. The presence of this information removes the need to perform expensive analysis of the program every time a data transformation is performed.

Assertions provide the link between the assignments-to and uses-of the variables.

The previous stages were free to introduce any assignments to the program. There were no checks to ensure that these provided equivalent values because the "black box" nature of WSL programs does not depend upon values which are not used. The next stage corrects this and provides the theoretical link between the source and ghost variables.

Step 4 — Performing the Ghosting Transformation

The ghosting transformation involves replacing an expression involving the source variable with an equivalent (or refined) expression involving the ghost variable. The validity of this is shown by reference to:

- the relevant data type theories;
- an equivalence theory which links the two types in general terms and
- the assertions generated in the previous stage.

A suitable replacement expression must be generated which uses the ghost variable in a manner which is the same as the original use of the source variable. There is no definitive form for this new expression and selection of an appropriate one may involve maintainer guidance or the use of heuristics. The relationship between source and ghost expressions will vary. In simple cases where the source and ghost types are similar the expressions may be almost identical but in more complex cases, where the representation of the data is being changed the replacement expressions could differ substantially.

Theoretically the new version of the program must be shown to be a refinement-of or equivalent-to the original version. This is done by appealing to the data expression refinement relations introduced in section 5.3. At this point it is important to note the difference between the refinement/equivalence of expressions demonstrated in the data expression refinement relation and the refinement/equivalence of program fragments which determines the correctness of the transformation. The former allows reasoning about the values produced by expressions but it does not describe the changes in program state which are performed by execution of a program fragment. For this reason a new transformation theory is needed to provide the link between data expression refinement/equivalence and program refinement/equivalence.

The transformation theory allows the ghosting to be performed in any case where it can be shown that the ghost expression is a refinement of the source expression. This provides a distinction between type equivalences and program transformations allowing new data type theories to be developed in isolation from the transformation engine. This is made possible by the use of shallow semantic data type embedding. The transformation theory is fully discussed in section 5.5.2.

Step 5 — Removing Assignments to the Source Variable

After the roles of the two variables have been reversed it becomes possible to remove the assignments to the source variable. The variable is no longer used within the program allowing statement deletion transformations to remove the assignments.

Step 6 — Removing the Source Variable

Removal of the source variable can be performed easily because it is no longer referenced within the program. It is even possible to combine this step with the previous one because the "remove variable" transformation allows removal of the variable if it is only assigned-to and therefore not used within the program.

5.5.2 The DREAM Data Transformation

The DREAM data transformation is used to perform the ghosting transformation (step 4). It replaces source expressions with ghost expressions in any WSL statement provided that the expressions can be shown to be equivalent to each other or that the ghost expression is a refinement of the source expression. The general form of the data transformation is

$$\begin{aligned}&\text{If } \exists h : M \leftrightarrow N \bullet\\ &\qquad\qquad \forall x, y \bullet \left[(x,y) \in h \implies \left((\bar{g}(y) \subseteq \bar{f}(x)) \wedge \left((\bar{g}(y) = \emptyset) \equiv (\bar{f}(x) = \emptyset)\right)\right)\right]\end{aligned}$$

$$\text{then } S_1;\ S(f(x));\ S_2 \quad \sqsubseteq \quad S_1;\ S(g(y));\ S_2$$

This states that whenever the data expression refinement relation can be shown to be true at a point where the source expression, $f(x)$, is used within the program then that program can be replaced with an equivalent (or refined) version which uses the ghost expression, $g(y)$. Note that assertion information is used to show that the data refinement relation holds. A fully worked example of this is shown in section 7.2.3.

Showing Equivalence of WSL Programs

Ward's theory [88] demonstrates equivalence and refinement using weakest preconditions. These are used to show that the semantics of two atomic descriptions[2] have a suitable relationship between each other during a transformation. The weakest precondition of an atomic description is defined by Ward as

$$WP(<x>/<y>: [Q], R) = \exists x.Q \wedge \forall x.[Q \Longrightarrow R]$$

This weakest precondition is separated into two parts:

- $\exists x.Q$ — this specifies that the atomic description will not terminate if there is no assignment which satisfies Q. That is the weakest precondition is *false* (Dijkstra's [40] "law of the excluded miracle") if the atomic description does not terminate.
- $\forall x.[Q \Longrightarrow R]$ — this part specifies that any assignment of values to x which satisfy Q must also satisfy R. In other words for a specific condition on the final state R any program which terminates (satisfies Q) will also satisfy that condition.

Equivalence and refinement are demonstrated by appealing to the weakest preconditions and showing that they are equivalent (or that one implies the other for refinement). That is

[2] Atomic descriptions were introduced in section 4.1.1

$$WP(S_a, R) \equiv WP(S_b, R) \quad \text{— } S_b \text{ is equivalent to } S_a \quad (S_a \equiv S_b)$$

and

$$WP(S_a, R) \Longrightarrow WP(S_b, R) \quad \text{— } S_b \text{ is a refinement of } S_a \quad (S_a \sqsubseteq S_b)$$

These conditions which demonstrate the validity of a transformation are shown to be true using the laws of infinitary logic and the laws of individual embedded data types (expressed in infinitary logic).

Use of embedded data types does not invalidate the semantics of existing WSL transformations. Any transformations which were proven using the untyped WSL language are still valid because all of the original laws are still valid. The addition of data typing actually makes it possible to prove a wider range of program transformations which rely upon the semantics of individual data types.

Proving the DREAM Data Transformation (Ghosting)

First consider the program fragment which is to be ghosted. The result of the program is the set of possible final states, R, which may be produced by execution of the program fragment, S. Ward's theory states that for a transformation to be valid (i.e. equivalence or refinement) the following must be true.

$$WP(S_a, R) \Longrightarrow WP(S_b, R)$$

where S_a is the source program and

S_b is the ghosted program.

The ghosting operation takes place at points where an expression is used within the program. These occur in any statement where the source variable is mentioned in the atomic description's condition, i.e. most frequently in assignments and assertions. The proof of equivalence for each statement is performed in a similar manner and only the proof of equivalence for assignment statements is shown here.

Consider the semantic expansion of the assignment statement described in section 4.1.1. This is described as a two step instruction which consists of an assignment to a temporary variable, *tmp*, followed by assignment of the temporary value to the variable which will hold the result. The first step performs the assignment of the variable and the second step merely serves to ensure that there are no circularities in the evaluation of the expression (see section 4.1.1). This second stage is a degenerate version of the original assignment and proof of the first step serves to show its correctness.

$$\begin{aligned}\text{Let } S_a &\text{ be } \mathit{tmp} := f(x)\\ \text{and } S_b &\text{ be } \mathit{tmp} := g(y)\end{aligned}$$

Here an expression involving the source variable is described as $f(x)$ where x is the source variable which has possible initial values such that $x \in M$. This set of values is shown by assertion derived from the places where values may have been assigned to the variable. A similar expression, $g(y)$ can be provided for the ghost variable y which has values such that $y \in N$. This corresponds to the following transformation (in terms of WSL statements).

$$\begin{array}{lcl} \{(x \in M)\}; & & \{(x \in M)\}; \\ \{(y \in N)\}; & \sqsubseteq & \{(y \in N)\}; \\ \mathit{tmp} := f(x); & & \mathit{tmp} := g(y); \end{array}$$

To show that this transformation is correct we must prove that

$$WP(<\mathit{tmp}>/<>: [\mathit{tmp} = f(x)], R) \implies WP(<\mathit{tmp}>/<>: [\mathit{tmp} = g(y)], R)$$

Note that the assertion statements, data type assertions and type equivalence invariants have been excluded from this proof. They are contained within the data expression refinement relation presented in section 5.3.

The conditions within the weakest preconditions are rewritten using set notation as described earlier to produce

$$WP(<tmp>/<>: [tmp \in \bar{f}(x)], R) \implies WP(<tmp>/<>: [tmp \in \bar{g}(y)], R)$$

This is then expanded using Ward's definition of WSL semantics to produce

$$\exists tmp \bullet \big(tmp \in \bar{f}(x)\big) \wedge \forall tmp \bullet \big[\big(tmp \in \bar{f}(x)\big) \implies R\big] \implies$$
$$\exists tmp \bullet (tmp \in \bar{g}(y)) \wedge \forall tmp \bullet [(tmp \in \bar{g}(y)) \implies R]$$

Now using the relationship between source and ghost expressions presented in section 5.3 case analysis can be performed on the relationship between $\bar{f}(x)$ and $\bar{g}(y)$. Initial analysis shows that either

$$\bar{f}(x) = \emptyset \wedge \bar{g}(y) = \emptyset$$

or

$$\bar{f}(x) \neq \emptyset \wedge \bar{g}(y) \neq \emptyset$$

Using this the existential part of the weakest precondition formula can be simplified. In the first case, where the sets are empty, this gives

$$false \wedge \forall tmp \bullet \big[\big(tmp \in \bar{f}(x)\big) \implies R\big] \implies false \wedge \forall tmp \bullet [(tmp \in \bar{g}(y)) \implies R]$$

which simplifies to

$$false \implies false$$

which is trivially true.

In the second case where the sets are non-empty the condition is rewritten as

$$true \wedge \forall tmp \bullet \left[\left(tmp \in \bar{f}(x)\right) \implies R\right] \implies true \wedge \forall tmp \bullet [(tmp \in \bar{g}(y)) \implies R]$$

which simplifies to

$$\forall tmp \bullet \left[\left(tmp \in \bar{f}(x)\right) \implies R\right] \implies \forall tmp \bullet [(tmp \in \bar{g}(y)) \implies R]$$

To demonstrate that this is valid we use the truth table shown in table 5.1. This shows that the transformation can only fail if $tmp \in \bar{f}(x)$ is false but $tmp \in \bar{g}(y)$ is true. The data expression refinement relation allows us to show that this can never occur because $\bar{g}(y) \subseteq \bar{f}(x)$.

$tmp \in \bar{f}(x)$	$tmp \in \bar{g}(y)$	R	A	B	C
F	F	F	T	T	T
F	F	T	T	T	T
F	T	F	T	F	**F**
F	T	T	T	T	T
T	F	F	F	T	T
T	F	T	T	T	T
T	T	F	F	F	T
T	T	T	T	T	T

$$\begin{aligned}\text{where } A &= \left(tmp \in \bar{f}(x)\right) \implies R \\ B &= (tmp \in \bar{g}(y)) \implies R \text{ and} \\ C &= \forall tmp \bullet \left[\left(tmp \in \bar{f}(x)\right) \implies R\right] \implies \forall tmp \bullet [(tmp \in \bar{g}(y)) \implies R]\end{aligned}$$

Table 5.1: Truth Table for the Proof of Ghosting

This proof demonstrates that the technique of ghosting performs valid transformations which may be applied at any place within a program where the state is changed. Successful application of the transformation requires that:

- sufficient information is available about the values held in both source and ghost variables;

- the relationship between source and ghost variable is known to allow checking of the equivalence of the values held in each and
- that the ghost expression is a refinement of the source expression.

This information is gathered within the DREAM ghosting framework.

5.5.3 Ghosting as an Algorithm

The theoretical view of ghosting shows that data transformations can be applied using a number of Ward's transformations and semantic knowledge from the typed WSL language. This provides a flexible approach but has a number of limitations and problems which must be addressed before it can be used to practically implement the full range of ghosting transformations. They are as follows:

1. **The desired relationship between the source and ghost variables must be known** — this guides the production of suitable expressions and ensures that sufficient information is available.
2. **Application of ghosting to the entire scope of a variable limits ghosting's flexibility** — in particular transformations which allow separation and merging of logically distinct variable uses would not be possible.
3. **Assertions must be moved around the program from assignment statements to the places where variables are used** — the assertions must be moved down all control flow paths and this involves the use of a large number of computationally expensive transformations.
4. **From a transformation efficiency viewpoint it is desirable to reduce the number of passes over the program** — passing over the program a number of times requires temporary information to be stored within the program tree and uses a large number of position movement instructions.

Each of these points is examined below and their effect upon the ghosting transformation is discussed. They are used to make ghosting more suitable for the purposes of data transformation.

Relationship between Source and Ghost Variables

Knowledge about the desired relationship between the source and ghost variables is vital to the success of a ghosting transformation. In many cases it is not possible for the transformation system to determine the relationship which should be used. For instance if an integer value is being split into a record containing a number of integer values whose combined result is equivalent to the source value then the formula which governs this relationship must be specified explicitly.

Similar problems may also arise in less complex situations where integers are being converted into enumeration values. Here the mapping between the two needs to be determined to ensure that each and every integer is assigned an appropriate name.

Information about these relationships can be provided in two ways:

1. Implicitly by the type equivalence theory and

2. Explicitly by the maintainer.

The information provided by the equivalence theory generally cannot provide all of the information required to make meaningful transformations. It encapsulates the basic knowledge about how two variables could be related but it is not practicable to provide all of the knowledge necessary for a full transformation. Inclusion of full knowledge within the equivalence theory would either make it too specific to one particular transformation instance or make the theory too complex to implement and prove to be correct.

The information provided explicitly by the maintainer fills in the gaps in the implicit knowledge from the equivalence theory. It specifies the precise relationship between the source and ghost variables making formal proof of equivalence possible. Each equivalence theory will require its own specialised format of information which depends upon the exact structure of source and ghost types along with the types of relationship that the theory supports.

Theoretically the relationship between the source and ghost variables is described by an invariant. This must be consistent with the theories of each type and is asserted to be true throughout the area of code which is being ghosted. This assertion is used during the proof of the data expression refinement relation.

Automatic choice of suitable relationships/invariants is a complex task. It would be very difficult to perform this on arbitrary programs but in situations which are well understood it is possible to use heuristics to select appropriate transformations. An example situation where it may be possible is in the conversion of assembly language code into a higher level language. In assembly code registers have specific formats and are typically used for a limited number of specific purposes. Characteristics of each can be recognised and the appropriate transformations are applied to perform a suitable operation.

The Scope of the Ghosting Transformation

The theoretical description of ghosting involves the introduction of the ghost variable into the program and requires that ghosting is performed throughout the whole scope of the variable. This limits the usefulness of the transformation. Specifically it does not allow implementation of transformations which merge and split logically different uses of a variable and those which involve a change of scope.

Data re-engineering transformations are not fundamentally concerned with replacement of variables but are concerned with changing the types which are used to represent data values. Ghosting is a suitable mechanism which can be used to perform these operations but there is no requirement to use the entire theory of ghosting. If a subset of it is sufficiently powerful for the requirements of data re-engineering then this can be used instead.

Analysis of the use of the complete theory of ghosting shows that it is unnecessarily restrictive for two reasons:

1. Variable introduction and variable removal transformations are primitive operations in the theory of WSL. These can be carried out constructively without reference to the theory of ghosting. Ghosting can then use these variables instead of having to introduce them into the program itself.

2. The ghosting operation does not have to replace all assignments-to/uses-of a variable. The theory only requires ghosting of all assignments which affect the variables' value where a specific use occurs. The correctness of this point is demonstrated by the proof of individual DREAM transformations in section 5.5.2.

The central aspects of the ghosting operation are those which involve generating assertions which are used to show that the two variables have equivalent values and which involve using these to rewrite the usage of the source variable with an equivalent expression involving the ghost variable. This involves passing information about the values contained within variables between assignments and uses.

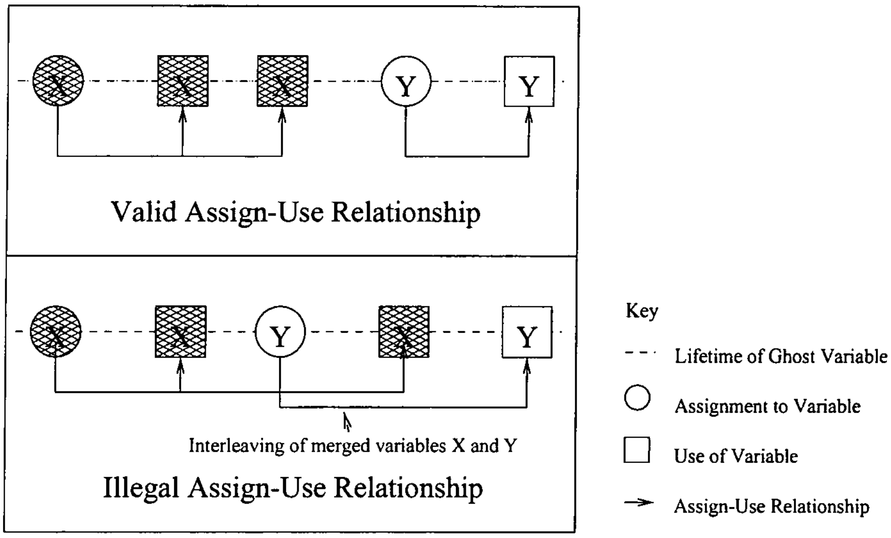

X
X
X
Y
Y
Valid Assign-Use Relationship
X
X
Y
X
Y
Interleaving of merged variables X and Y
Illegal Assign-Use Relationship
Key
Lifetime of Ghost Variable
Assignment to Variable
Use of Variable
Assign-Use Relationship

Figure 5.3: The Interleaving of Ghosted and Original Assign-Use groupings

Figure 5.3 shows an example of this and demonstrates how a transformation which merges two variables together may fail. This figure represents the value stored in one variable over a period of time. In the first case the values assigned to the variable and its uses (marked X and shaded) do not interleave with the new assignment-to and use-of the source variable. In the second case there is an interleaving and the second use of the original value would fail because the variable would contain the value introduced during the ghosting.

Changing the emphasis on ghosting to concentrate on the assign-use relationships allows more flexibility in the type of ghosting operation that can be performed. It makes it possible to implement the whole range of data transformations using one mechanism which in turn makes it much easier to provide a consistent interface to each type of data transformation.

Movement of Assertions around the Program

The description of ghosting on page 95 showed that assertions are the key to demonstrating that a ghosting transformation is valid. The information held within the assertion is used to show that expressions containing the source variable can be replaced by those containing the ghost variable.

Unfortunately moving assertions around the program is expensive in terms of computation effort. Movement of the assertion from one point within the program to another may take a large number of transformations. The assertions must be propagated down every possible path in the program until it is certain that they cannot possibly be used. This is complicated by the need to merge assertions at any point where control flows rejoin, e.g. after each branch of a conditional statement. It is also complicated by the use of external procedure calls[3] past which assertions cannot be moved because of the unspecified behaviour of the call.

The information carried by the assertion is essential to the transformation and therefore cannot be ignored. A solution which increases efficiency is to not add the assertions to the program but to hold the information within the transformation code and ensure that this information is updated at control flow branches and joins. This does not affect the theory but makes implementation more efficient.

The Number of Passes over the Program

In its original form ghosting requires a number of passes over the area of code which is being transformed. These allow increasing amounts of information to be added to the program to enable the ghosting to take place. Reducing the number of passes during the transformation is a desirable solution which saves wasted computation time and also allows the assertion information caching described above to be implemented.

5.5.4 Revised Ghosting Mechanism

The discussion above has highlighted a number of changes to the ghosting transformation mechanism. These result in a new definition which is more suitable to

[3] External procedure calls allow calls to arbitrary pieces of code (which is not in the program being transformed). These calls perform unspecified operations and consequently the state of the program is unknown after they have finished executing.

data transformation. The new form consists of a single pass over the program which performs the following operations:

1. Inserts ghost assignments wherever an assignment to the source variable occurs.

2. Records the assertions which are generated while adding ghost assignments. These must be properly combined with other assertions whenever passing over control flow joins.

3. Verify that the expressions involving the source and ghost variables are equivalent at all points where the source variable is used.

4. Generate appropriate uses of the ghost variable and replace the use of the source variable with it.

As part of these operations the transformation must also check that the assign-use relationships between the source and ghost variables are not violated. This may be slightly complicated because the transformations are no longer tied to the scope of the variables. There may be assignments-to or uses-of the variable outside of the transformation scope which may interfere with the transformation. One likely scenario is that the first use of the source variable within the scope may occur before an assignment has been made to it within the scope. In this case the transformation is invalid and it will fail.

The ghosting transformation must also account for the fact that the source and ghost variables may have a different scope within the program. This is generally not a problem because both variables must exist at the boundaries of the area of the program which is being transformed. However, there are situations where a variable may temporarily go out of scope within the transformation area. For instance, if a local variable or a procedure parameter is declared with the same name as either the source or ghost variable then this causes a change in scope. In these situations the general rule is that any assignments-to/uses-of the source variable will cause the transformation to fail. Extra care must be taken to ensure that variable aliasing is taken care of. If a variable is passed as an actual parameter to a function it may be renamed and used within recursive calls to the same piece of code.

Further details about how the ghosting algorithm is implemented within the Maintainer's Assistant are discussed in chapter 6.

5.5.5 Using Ghosting to Implement Data Transformations

Ghosting has been developed into a very flexible transformation mechanism which is capable of performing many different data transformation operations. This section shows how each of the data transformation categories can be performed using ghosting.

- **Changing data representation** — this category of data transformation is the principle transformation which is used for DREAM data re-engineering. The ghosting transformation mechanism is directly applicable and requires no special action to be taken although the type of the ghost variable must be supplied and the desired relation between source and ghost variables must be identified.

- **Changing the relationship between logical data objects** — these transformations place some of the greatest demands upon the ghosting theory. They require the checking of scope and assign-use relationships between the original ghost variable and the source variable.

- **Changing the scope of data** — a change in the scope of data is easily performed by introducing the ghost variable with an appropriate scope. The ghosting operation will then ensure that all of the data accesses are correct taking the differences in scope into account.

- **Introducing subtypes** — this transformation can be performed straightforwardly using ghosting. The primary operation performed by the transformation is to change the type of the variable to a different subtype. This is handled easily using the semantics of one data type. Ghosting does require the introduction of a new (ghost) variable into the program but in principle this is not needed because only the type is changed and not the variable name. In practice the transformation can hide the introduction of the new variable by replacing the source variable directly with the ghost variable.

Each of these data transformations can also be performed in conjunction with the others to produce complex effects upon the program data representation. For instance a change of scope may be combined with a merging of variables.

5.6 Summary

This chapter has described DREAM (the Data Re-Engineering and Abstraction Mechanism) which provides a transformation mechanism which allows program data to be manipulated. An important part of this is the use of the ghosting technique which provides the underlying formal basis of the transformations.

Four categories of data transformation have been identified. These allow a comprehensive range of manipulations to be applied to program data producing a restructured version of the original. The transformations allow not only the representation of the data to be altered but also allow its relationship with the program scope and other variables to be changed.

Ghosting allows the original implementation of the program data semantics to be replaced by an equivalent version which differs in user selectable ways. Ghost variables are added to the program which have the desired new data format and properties. The DREAM data transformation is then used to replace all of the uses of the original variable with equivalent versions which are represented in the new manner. At the end of the transformation the original version no longer influences the output of the program and can be removed.

The key to the proof of these DREAM transformations is the use of a data expression refinement relation which allows any use of the source expression to be replaced by a target expression without any further knowledge about the structure of the program. The validity of this transformation step is proved in section 5.5.2 for any atomic description. This means that the transformation is valid for all WSL statements because their semantics are all defined in terms of atomic descriptions.

The algorithm used to implement DREAM transformations has been analysed and modified to provide a full support for a number of types of data transformation. This allows the whole range of data transformations to be performed using a single transformation mechanism.

This chapter has addressed the theoretical aspects of data transformation but has not examined the practical details of implementing the transformations within a transformation tool. There are a number of aspects of implementation which must be considered including:

- How should the transformations be integrated into the transformation system?
- How easy is it to extend the transformation tool to allow new data types and equivalence theories to be used?
- What extra $\mathcal{M}$ETAWSL statements are needed to implement data transformations?
- Is it possible to present a similar user interface for both control flow and data transformations?

These practical issues are covered in the next chapter which describes how the Maintainer's Assistant has been extended to allow data transformation to be performed.

Chapter 6

The Prototype Tool

Putting the theoretical aspects of DREAM into practice involves developing an implementation which integrates the theory with algorithms which automatically select appropriate ways of applying the transformations. This allows most data transformations to be performed with little user intervention (except for the initial choice of transformations). Some of the more complex transformations may, however, require more user guidance to ensure that the desired[1] results are produced. One of the benefits of heuristic application of algorithmic solutions is that they allow the efficiency of the transformation application to be bounded rather than relying upon exhaustive application of the theory to produce a final, optimal result.

The main practical extension to the theory is the automatic generation of appropriate ghost assignments and expressions during the transformation. This is combined with an implementation of the ghosting procedure which collects information about source and ghost variables. It is done in a manner which bounds the execution overhead involved by making compromises which limit searches in the program. It is possible to supplement this information by the introduction of assertions at points where extra reasoning is required. The low level details of data transformation application are taken care of by the user interface. The transformation engine automatically determines which transformations are suitable at a particular point and allows the maintainer to apply these.

[1] This does not imply that undesired results are semantically erroneous; it merely states that a different (but correct) result would be produced.

6.1 Introduction

The Maintainer's Assistant is used as a prototype platform to examine the feasibility of automating data transformations using the theory and practical aspects introduced in the previous chapters. The main aim of the extensions to the existing tool is the integration of data and control flow transformations to provide a combined maintenance environment. This is achieved by extending the tool to incorporate typed WSL and by use of the ghosting transformation mechanism to provide the DREAM data transformation capabilities.

These extensions to the Maintainer's Assistant are implemented in a modular manner which complements the original design of the tool. The modules are self contained with well documented interfaces to the other parts of the system. The new modules are:

- **DREAM** — which performs the ghosting transformation ensuring that the appropriate rules are observed. This is responsible for ensuring that the correct type theories are used to generate assignments and expressions.

- **Data types** — which represent the semantics of individual data types allowing simplification of expressions and extraction of type information from the program. The data types module contains a number of sub-modules which provide information for individual data types. The interfaces to these submodules are structured to provide a common way of describing each type.

- **Type equivalence** — which contains the information about the relationship between expressions which are used during ghosting. It is also responsible for producing suitable equivalent expressions given the full details of the required transformation. There are sub-modules for individual instances of type equivalence theories.

In addition to these new modules a number of changes are required in other parts of the transformation engine. The most important change is the introduction of typed WSL constructs, as presented in table 4.7 on page 75, and the addition of extra $\mathcal{META}$WSL statements to provide the capability to extract data type information from the program. DREAM also requires some changes to the user interface to allow selection of ghosting transformations and identification of the variables which will take part in the transformation.

The next section describes the design of these extensions to the transformation engine which provide the data transformation environment. Section 6.3 describes the implementation of the changes to the transformation engine giving details of the steps taken to add data typing constructs to WSL. It also describes how these changes were tested to ensure that the existing control flow transformation were unaffected by the addition of data types.

6.2 Extending the Transformation Engine

The implementation of the transformation engine separates its functionality into a number of subsystems which provide support services allowing programs to be transformed. The structure of the system is shown in figure 6.1 with the unshaded boxes forming the original transformation engine.

The system is based around an abstract representation of the WSL language ("xlang" in the diagram) which specifies the format of constructs. This internal representation is produced automatically from the file "table" which describes the syntax in less implementation specific terms. The "support" module uses this language description to provide an internal program storage facility which holds programs while they are being transformed. The internal representation of a program is manipulated by the "data", "pattern" and "meta" modules. These provide $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL statements which allow efficient reasoning about programs and the searching of programs to find suitable program fragments. In addition to this there is a "maths" module[2] which allows basic simplification of expressions.

The "lang" module provides executable versions of each WSL construct. These are used together with the $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL constructs to implement individual transformations. A program editor, "editor", is provided which allows changes to be made to programs when bugs are found or if extra functionality is being introduced into the program which is being transformed.

DREAM data transformations require that capabilities to represent and reason

[2] The maths module has been replaced in the enhanced version of the Maintainer's Assistant.

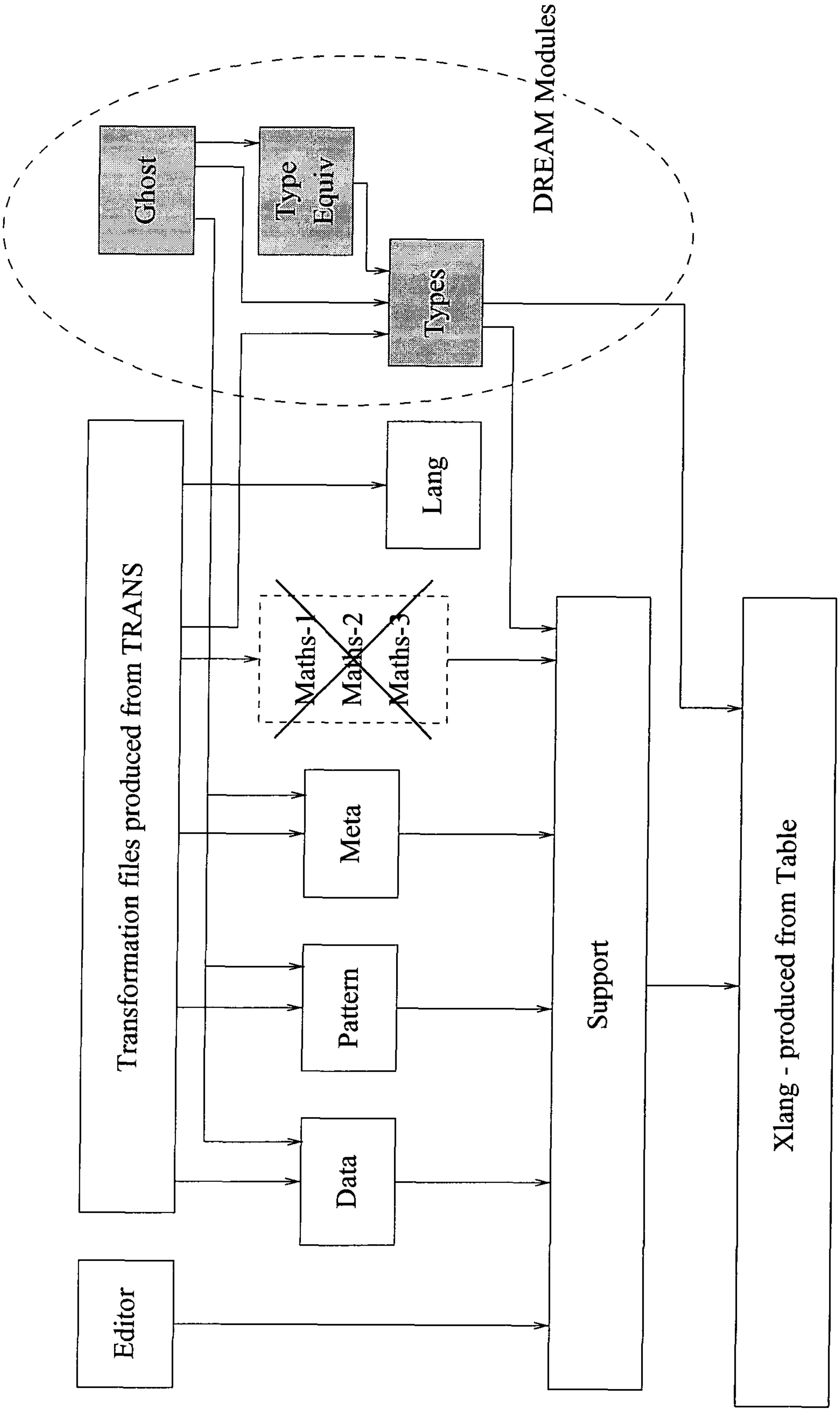


Figure 6.1: The Extended Tool Architecture

about data types and their associated theories are added to the Maintainer's Assistant. These are then used to implement the ghosting transformation and to replace the original mathematics and logic packages. The extensions to the Maintainer's Assistant have been separated into three new modules (shown as shaded boxes in figure 6.1). The "types" and "type equivalence" modules have been designed to allow new types and theories to be added to the transformation engine with minimal effort. To aid this, each data type and equivalence theory must provide a well-defined set of interface procedures. The transformation engine uses these in a manner which is appropriate to the transformation being performed. The DREAM module implements one transformation but due to the amount of code required to implement it and the distinct nature of the transformation it has been placed into a separate module.

6.2.1 DREAM Transformation Module

This module is responsible for coordinating the application of DREAM transformations. It implements the algorithms presented in the previous chapter providing a single transformation which can make use of any type equivalence theory to perform individual ghosting transformations.

A secondary function of the module is to aid the maintainer in the selection of suitable transformations. Information about source and target variables is used to produce a list of transformations which would be applicable, in general, for their respective types. This operation is not part of the main transformation code and involves searching the type equivalence modules to find suitable equivalence theories.

Implementing the DREAM Transformation Algorithm

DREAM transformations have two major constituent operations: adding assignments to the ghost variable and using the ghosting transformation to replace uses of the source variable with uses of the ghost variable. These operations are performed under strict control to ensure that the semantics of the program are unchanged.

Application of the algorithm has been separated into three stages:

1. **Initial phase** — identification of the exact transformation required and checking of the source and ghost variables' state at the entry point to the area of

code which is being transformed.

2. **Main phase** — systematically performing the transformation by identifying all affected program statements and taking appropriate action at those points.

3. **Final phase** — checking validity of final variable states and recovering from any transformation error conditions which may have occurred.

These stages are applied in sequence and the transformation is only guaranteed to be successful if all three stages complete.

The initial phase — before the transformation is performed its parameters are checked to ensure that they are legal given the currently selected program fragment. This involves checking that the source and ghost variables exist and that the equivalence theory and user supplied invariant are valid. These checks do not guarantee that the transformation will succeed but ensure that there are no configuration errors which may cause incorrect action to be taken.

The main phase — the main phase of the transformation involves following possible control flow paths gathering information about the values held in the source variable and instructing the type equivalence modules (see section 6.2.3) to produce appropriate assignments and expressions involving the ghost variable. In the process of doing this, assign-use interleavings between the source variable and the original ghost variable (before the transformation started) are checked. If any of these operations fail then processing in this phase finishes and the final phase is responsible for unwinding any changes which have already been made to the program.

Control flow paths are traversed because they represent the order of execution of the program. This is done for four distinct reasons:

1. **to gather assertions about the contents of the source variable** — these are collected at points where the source variable is assigned-to and from assertions which explicitly state the value of the variable. The structure of these assertions is specific to each data type and the collection of them is performed by the appropriate data type module (on behalf of the ghosting module).

2. **to insert (ghosted) assignments to the ghost variable** — each assignment to the source variable requires an equivalent assignment to the ghost variable. This is generated by the appropriate type equivalence module and is inserted by the ghosting module.

3. **to perform the ghosting transformation on the uses of the source variable** — uses of the source variable are replaced by suitable uses of the ghost variable (they are produced by the type equivalence module).

4. **to check that assign-use relationships are valid** — the DREAM ghosting mechanism allows the ghost variable to be used for different purposes (see section 5.5.3) but checks must be made to ensure that these different uses do not invalidate the transformation.

Processing of the control flow paths is performed by ensuring that paths are examined in the order that they would be executed. This involves a breadth-first traversal of the program fragment with careful checks to ensure that when control flows join all branches which reach that point have been examined.

Loops and other statements which have iterative/recursive calls to themselves must be treated specially. At the entry/re-entry points to these areas the automatically generated assertions about source variable values must contain the union of the values for every *nth* iteration, i.e. the limit value. This can be calculated to varying degrees of accuracy: the least accurate is to assume that the variable may hold any value which is valid for the data type; a more accurate way is to process the loop/construct once and determine whether the assertion at re-entry is more restrictive than that at initial entry. If it is then the initial assertion can be used. This process could be repeated ad infinitum but the prototype implementation attempts one iteration and if that fails then the least accurate case is used.

The final phase — the final phase checks for any invalid assign-use interleavings which occur after the scope of the transformation. If these occur or if the transformation has failed for some other reason then the changes made to the program are unwound and the error is reported to the maintainer.

The application of the transformation relies heavily on data type modules and equivalence modules. These provide information in a standard format to make the DREAM transformation module truly generic. Intermediate information about the source or ghost variable which is specific to one particular data type is stored in assertions which are specific to that type. The ghosting module treats these as black boxes and delivers them back to the data type modules at appropriate points.

6.2.2 Data Type Modules

Data type modules represent the syntax and semantics of individual data types. They are responsible for providing knowledge and reasoning about operations involving specific data values. This knowledge may be used during ghosting to extract appropriate type information from the program or it may be used for simplification of expressions during other transformations[3].

There are a potentially unlimited number of type modules. These correspond to the type categories which are used within the program (see section 4.2.1). Each data type is mostly independent of others although there may be dependencies for composite types which contain values which are of other data types. Even though data types may be dissimilar they are all used in a similar manner within the transformation system. This characteristic makes it possible to simplify the addition of many different data types to the transformation engine by defining a common interface for the operations which each data type must provide.

The Data Type Module Interface

The interface to each data type module is shown pictorially in figure 6.2. Each type category is given a unique name which is used to distinguish it from other type categories within the transformation system. This name is then used to allow the transformation engine to retrieve the details of this type from the full collection of data type theories held within the system.

The structure of the data type interface has been developed to reflect the needs of the transformation engine. It does not provide an exhaustive range of possible

[3] This is a replacement for the symbolic maths and logic modules which were present in the original version of the Maintainer's Assistant.

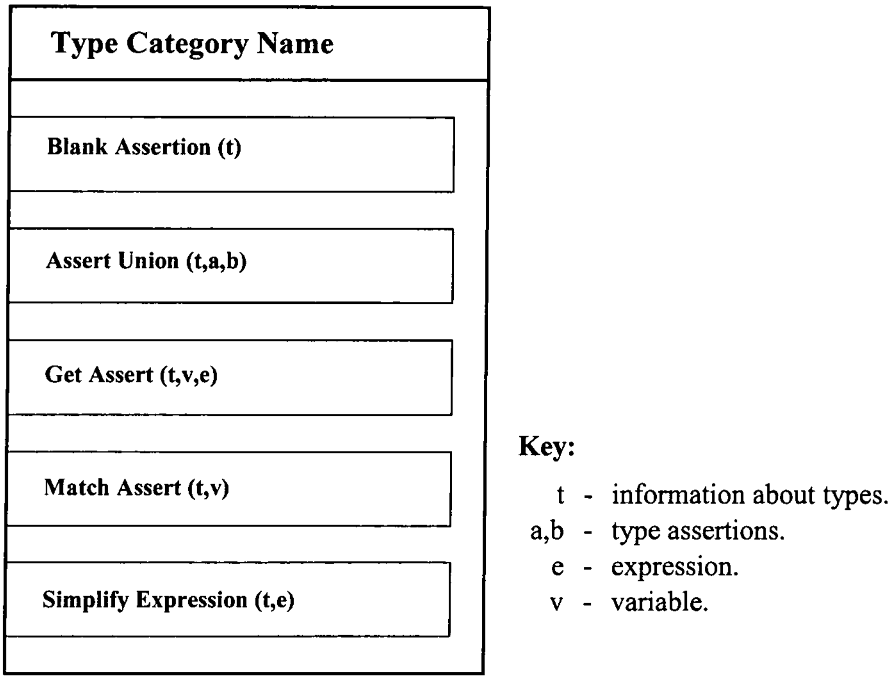


Figure 6.2: The Interfaces to Type Modules

operations upon the data but could be extended if necessary to cope with the requirements of future work. Note that the operation of individual interface functions should be considered in association with the description of the prototype implementation of the DREAM module which is discussed in section 6.2.1. The relationship between these interfaces and the type theories is discussed in the next chapter when a number of example type theories are introduced.

- **Type Category Name** — the type category name provides a way to identify the data type and is used internally to identify the appropriate set of routines for the variable under analysis.

- **Blank Assertion** — blank assertions represent the entire set of data values which may be stored within an individual variable. The assertion takes type and subtype information into account but does not assume anything about what has been assigned in previous statements. This provides a starting point for reasoning about variables before any assignments to it have been processed.

- **Assert Union** — when the control flow of a program rejoins after a branch construct the potential values of the variable after each branch, represented as assertions, must be combined to form an assertion which contains the set of all of these values.

- **Get Assertion** — this routine constructs an assertion which represents the possible values returned by an expression given details of the input to that expression, i.e. its parameters. The expression could be a constant value or it could be a function call. The result returned will typically have varying conciseness depending upon how well the values of the input are known. This routine is typically used to analyse the values assigned to variables.

- **Match Assertion** — each data type has differing formats for assertions. This function examines a given assertion and determines if it is appropriate to the current variable which is being transformed. If so, it extracts any useful information from the assertion for later use. Note that the calling transformation (i.e. the DREAM module) is responsible for storing and using the information.

- **Simplify Expression** — this operation is the replacement for the functionality of the symbolic maths and logic unit which has been replaced in the new version of the transformation engine. The interface to these replacement functions is similar to that in the untyped transformation tool and is used to perform simplification on a supplied expression. Simplification could involve calls to other type modules to allow sub-expressions with differing types to be simplified.

This interface does not directly reflect the structure of the semantic theory of a particular data type. The semantics are hidden within the type module and are used to show the correctness of any manipulations which are performed. In particular the "simplify expression" interface relies on many heuristics to guide the simplification process. The semantics are also used in the type equivalence modules which are discussed later.

Data Type Syntax

Each data type has its own individual syntax to represent the characteristic properties of that type. These syntactic elements are used in a limited number of places within a program as shown in section 4.2.4. This makes it possible to allow modular extension of the syntax of the WSL language as well. The internal, abstract syntax representation of the program is especially easy to extend because its already modular structure allows many different constructs to be used at one particular position within the syntax. Implementation difficulties for this are discussed in section 6.3.1.

Extensions to MetaWSL

Data type information is stored in a number of different places around the program and the existing MetaWSL instructions are not sufficient to provide simple methods of accessing this information. Two extensions have been made to MetaWSL to alleviate these difficulties:

- **An extended @Follow/@Return** — the original **@Follow** and **@Return** constructs allowed a transformation to temporarily move from the current point in the program (which must be a subroutine call) to the definition of that subroutine. This makes the description of a transformation more concise by removing the need to write code to find, and move to, the appropriate definitions. These instructions have been extended to provide a similar capability which moves from uses of type names (i.e. in variable declarations) to the declaration of that type (in a **where** statement) and vice-versa

- **Extended data extraction/caching** — MetaWSL provides constructs which return information about the program such as the variables which are used or assigned-to within the current construct. New constructs have been added which provide information about the data types which are declared at a point within the program and about the binding of types to specific variables. This information is cached and can be retrieved easily with no execution overhead.

These extra constructs provide comparable functionality to that provided in the original version of the Maintainer's Assistant. They make it easier to extract datatype information from the program and to remove the need to write repetitive code which is not directly related to the transformation being implemented. Other aspects of the addition of data typing into the transformation engine are discussed in section 6.3.1.

6.2.3 Data Type Equivalence Modules

Equivalence modules are responsible for determining the exact semantics of the ghosting transformation and for ensuring that appropriate assignments and use-expressions are produced for the ghost variable. This involves checking that the semantics of the data expression refinement relations are used properly and also involves reporting errors to the ghosting control module (DREAM).

The equivalence modules are designed to be self-contained providing assignments and expressions which conform to the data expression refinement relations which are used in the DREAM transformation theory. To do this the interfaces presented below are supplied with information about the context of each individual ghosting operation. This context has two parts: (1) the detailed invariant between source and ghost variables which is supplied by the maintainer (this complements the part of the invariant which is implicit in each equivalence theory); (2) information from the program about the values which may be stored within each variable at the point in the program where ghosting is being performed. This information is collected by the DREAM transformation module which is responsible for ensuring that the appropriate interface routines are called when necessary.

The Type Equivalence Module Interface

Equivalence modules are integrated into the Maintainer's Assistant in a similar way to the type modules. A number of well-defined routines are used to provide access to reasoning about the applicability of particular ghosting operations. Equivalence modules depend upon source and target data type modules to provide reasoning about the types which they represent.

Each equivalence module (figure 6.3) is identified by the source and target type categories that it applies to. This allows easy retrieval of appropriate theories within the transformation system and provides a basic method for filtering out inappropriate type categories. More precise filtering is performed using the exact definition of each type — the properties of an equivalence relation may not be suitable for use

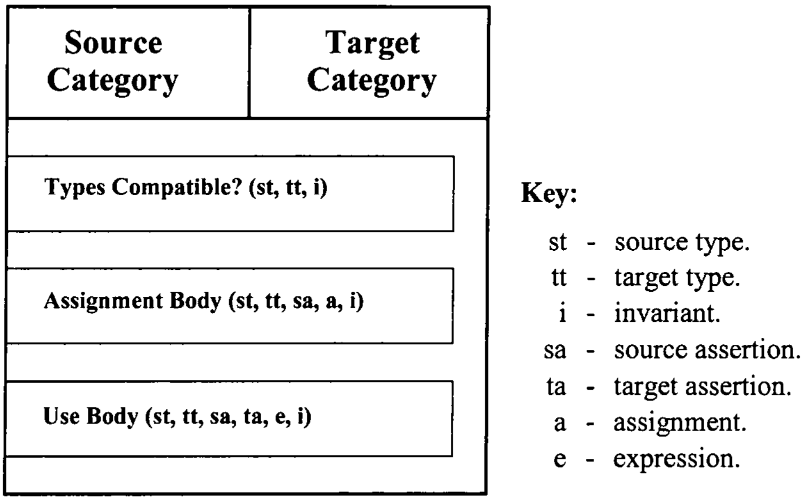


Figure 6.3: The Interfaces to Type Equivalence Modules

and may prohibit further transformation. A final check on applicability is made at each place where assignment or use conversions are made.

- **Source & Target Type Categories** — the source and target categories are used to determine if a particular theory is applicable for consideration in the current conversion. Finer grained checking of applicability is performed using the definition of particular data type instances and assertions about the current values stored in the source and ghost variables (see the following functions).

- **Types Compatible?** — this function determines if the conversion between the specified variables is valid. It uses the equivalence invariant and the precise definition of data types to ensure that the conversion is feasible. This is not a definitive answer to the question of compatibility but is used in a similar manner to the applicability conditions of control flow transformations. The final check on applicability is performed during transformation whenever the next two routines are called.

- **Assignment Body** — this produces an assignment to the ghost variable which is equivalent to the given assignment to the source variable. The ghost assignment need not be valid for any input condition but must be valid for all values which may be presented as input to the assignment's expression at that point within the program. Decisions about the structure of the assignment are guided by the invariant and by heuristics which choose between possible, valid alternatives.

- **Use Body** — this performs a similar operation to the assignment-body function above but produces an equivalent expression which uses the value stored within the ghost variable rather than that stored within the source variable.

One type equivalence theory may be represented by more than one equivalence module. It may be convenient to implement the ghosting operation for a specific subcategory of the equivalence theory rather than implementing a general equivalence module which may require very complex heuristics to allow the simple cases to be handled. An example of how an equivalence theory is used by the transformation engine is given in section 7.2.

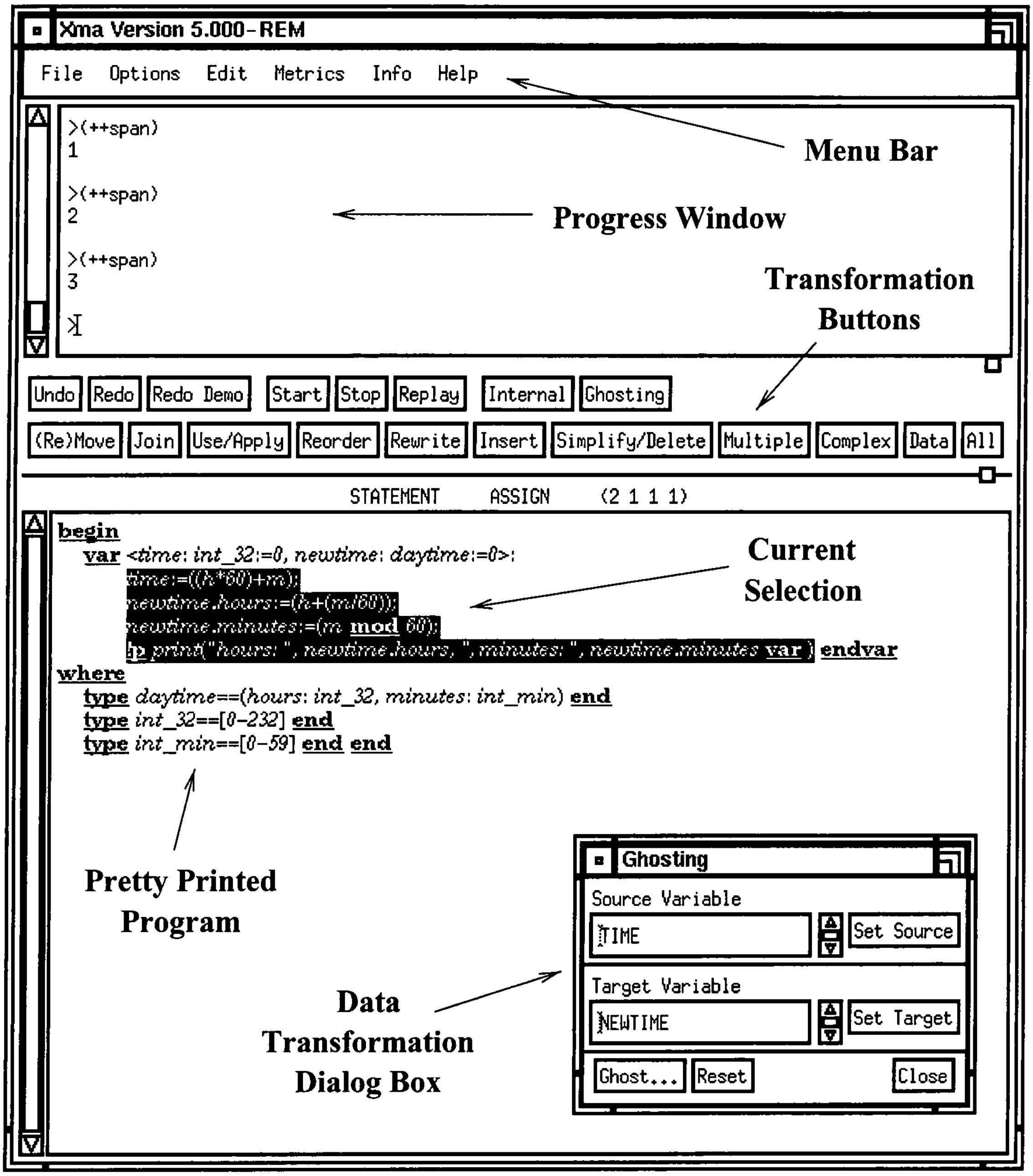
6.2.4 User Interface

The DREAM, data type and type equivalence modules provide the core functionality necessary to perform data transformations. The user interface provides a simple method for applying these data transformations.

Figure 6.4 shows the extended user interface which encapsulates the raw data transformations making it easier to select and apply transformations. The figure shows the main transformation window and the data transformation (ghosting) dialog box. The latter allows the maintainer to select the source and ghost variables from the program window and allows a search for applicable transformations to be initiated.

Source and ghost variables are identified by selecting the declaration of an appropriate variable and then clicking on the "Set Source" or "Set Target" buttons in the dialog box. The code to be transformed is selected by dragging the mouse over the code which is then highlighted in reverse video. When the "Ghost" button is pressed the applicable transformations are presented in a menu (not shown) and when the maintainer selects the desired transformation from the menu it is invoked and progress/error messages appear in the message sub-window (in the top half of the main window).

The user interface removes the need to understand the entire ghosting process.



```
>(++span)
1

>(++span)
2

>(++span)
3

>
```

```
STATEMENT    ASSIGN    (2 1 1 1)
```

```
begin
   var <time: int_32:=0, newtime: daytime:=0>:
        time:=((h*60)+m);
        newtime.hours:=(h+(m/60));
        newtime.minutes:=(m mod 60);
        lp print("hours: ", newtime.hours, ", minutes: ", newtime.minutes var ) endvar
where
   type daytime==(hours: int_32, minutes: int_min) end
   type int_32==[0-232] end
   type int_min==[0-59] end end
```

Source Variable: TIME

Target Variable: NEWTIME

Figure 6.4: The MA User Interface

The labour intensive peripheral tasks involved in performing data transformations are handled by the system allowing the maintainer to concentrate upon selection of suitable transformations.

6.3 Implementing the tool

Implementation of these extensions to the Maintainer's Assistant required considerable planning. The changes to WSL affect many aspects of the transformation engine even though they involve only minor changes to the language itself. In addition to this a staged introduction of type/equivalence modules and the DREAM transformation routine was required. It was impractical to expect that all of the changes could be implemented and tested together. To make the extensions feasible the following implementation plan was used:

- **Add the typing constructs to WSL's abstract syntax** — this involved extending the language definition and making associated changes to the rest of the transformation engine.

- **Testing the extended transformation engine** — to verify that the addition of data typing has not unduly affected the operation of the transformation engine.

- **Adding support for type/equivalence modules** — providing the module interfaces specified in the first part of this chapter.

- **Implementing the DREAM module and providing the user interface** — using the typed language and type/equivalence modules to provide ghosting data transformations.

The issues encountered during the implementation are discussed below.

6.3.1 Abstract Syntax Changes

The main change which affects the original transformation system is the extension of WSL's abstract syntax. These extensions are shown in chapter 4 (table 4.7) and are implemented by adding extra entries into the transformation engine's language description table (file "table.clisp").

Syntax Definition	`(Put_Frame_Lf!`
	`'(Type (AKO Definition)`
	`(Type_X (Type_Name Type_Defn))`
	`(Ins/Del? (No 2))`
	`(Blank (Type $Type_Name$ $Type_Defn$))))`

Example	type *$Type_Name$* ≡ *$Type_Defn$*.

Example 6.1: Type Definition Construct — Abstract Syntax Definition

Example 6.1 show the entry for a type definition. This entry states that the construct is called a "Type" and has a number of properties. These properties define the constructs structure and specify its relationship with other statements. The a-kind-of, AKO, field announces the construct is a specific instance of a definition and therefore may appear within a where construct. The "Type_X" field specifies the type definition's subcomponents which in this case are a name and a type description. The "Ins/Del?" field specifies that no extra subcomponents can be added and that the minimum number of subcomponents is two[4]. Finally the "Blank" field gives a prototype entry which is inserted as a placeholder when the Maintainer's Assistant program editor is used to add a new type definition.

The other data typing constructs are added to the language in a similar way and once this is complete the transformation engine can store typed WSL programs. Unfortunately this does not mean that the transformation engine operates correctly upon typed programs. This requires further changes as described below:

Parsing programs — the Lisp part of the transformation engine[5] uses two program representations: the internal and external formats. The internal format is a direct representation of the program as stored, and manipulated during transformation. This contains a large amount of cached information about the program which is used by the transformation engine. In contrast the external representation has

[4] The Ins/Del? field is mainly used for constructs which contain variable numbers of other constructs, e.g. lists of statements. It is not an important field for type definitions.

[5] Remember that the implementation of the transformation engine is separated into the main Lisp part and the graphical user interface which is implemented in C.

none of this information and is used during the dialogue with the graphical user interface.

Problems arise when parsing the external representation because the transformation engine does not use a look-ahead mechanism to determine the format of the next construct. Instead it relies on the "`Ext_Spec_Type`" (external specific type) routine to determine the format. This routine uses heuristics to make a choice between possible alternatives and is very prone to disruption due to changes in the language.

At numerous times during the extension of the transformation engine these heuristics had to be "adjusted" to ensure correct parsing. This often caused great difficulty because it is not easy to identify the exact cause of the problem or its solution.

Pattern matching — many of the existing transformations rely upon pattern matching and template filling MetaWSL constructs to recognise specific code fragments and to replace these with their transformed versions.

The typed language syntax affects some of the operations which match program fragments containing the typed constructs. **var** and **where** blocks were the worst affected because these contain the main changes to the syntax.

Untyped Version	

MetaWSL	`(Var ((Table`
	`      ([_Match_] Assignment ((~>?~ V) (~>?~ E)) Empty)))`
	`      ...`
	`      )`

Matches	*$Var$ := $Expn$*

Typed Version	

MetaWSL	`(Var ((Table`
	`      ([_Match_] Assignment`
	`       ((Typed_Var (~>?~ V) (~>?~ N)) (~>?~ E)) Empty)))`
	`      ...`
	`      )`

Matches	*$Var$ : $Type_Name$:= $Expn$*

Example 6.2: Changes to Pattern Matching Constructs

Example 6.2 shows an untyped **var** pattern match and its corresponding typed version. The pattern match statement is matching a variable name, and the expression that is assigned to it, for subsequent use during a transformation. The untyped version does not work in typed WSL because the program contains a name-type pair in place of the original name entry. The typed version has been modified to match the name-type information and store it away for future use.

Each of the six hundred and three control flow transformations had to be exampled and modified, if necessary, to take these changes into account. This was done by visual inspection and manual change. Section 6.3.2 describes the tests which were performed to validate that these changes had been made correctly.

Composite names — chapter 4 described the definitional extensions to WSL which allow composite variables to be accessed either as a single entity or as a number of individual entities. This capability is added to the transformation engine by introducing the **l-select** and **r-select** components as described in chapter 4. This change is completed by extending the variable assignment/use analyser (`vua_iter`), and hence the **[_Used_]**/**[_Assigned_]** $\mathcal{META}$WSL constructs. This allows the transformations to determine if specified variables are used or assigned-to within the currently selected program segment.

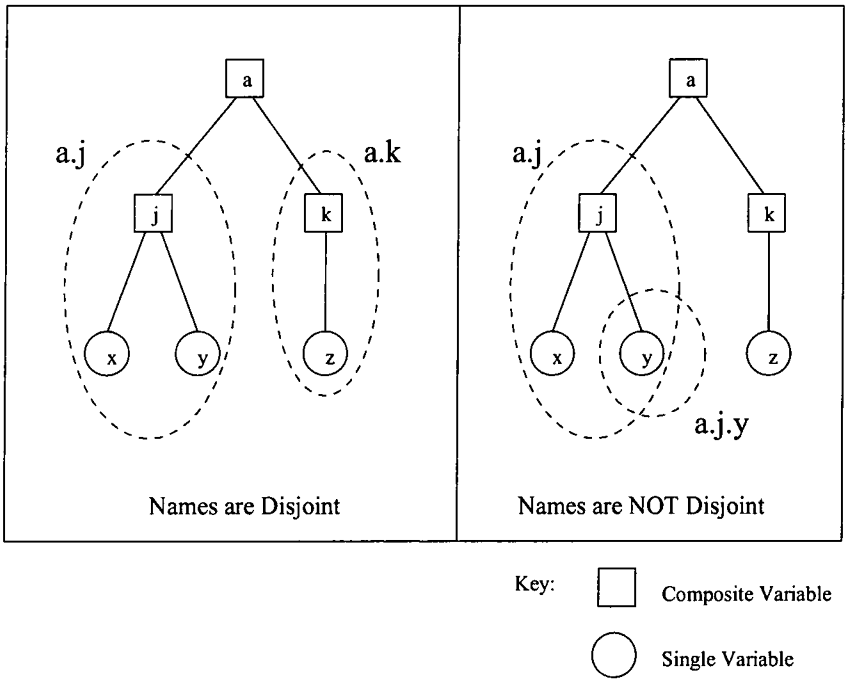


Figure 6.5: Composite Name Usage Analyser

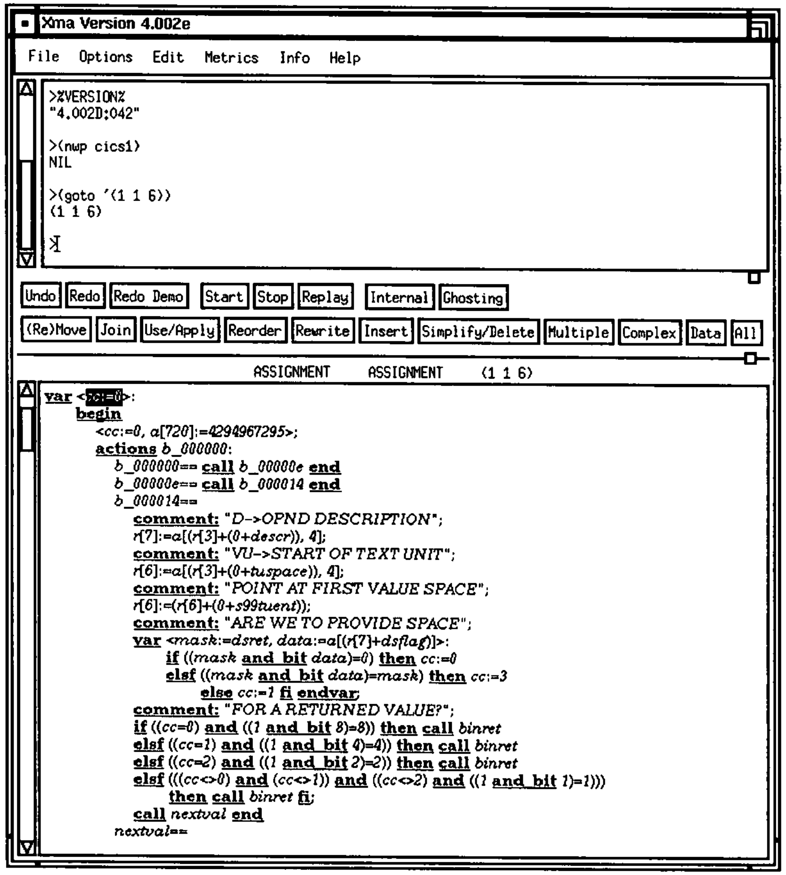
These extensions involve making the analyser store the full variable name (including all component names) and then making the comparison routines check whether two names have any part of the name space tree in common. Figure 6.5 shows two cases which may occur. On the left-hand side $a.j$ and $a.k$ are disjoint because they do not have any variables in common. The right-hand side shows a case where the two are not disjoint because $a.j$ contains the variables in $a.j.y$.

Expression simplifier — expression simplification in typed WSL is separated into individual routines for each data type. One routine is provided by each type module to simplify its own expressions. At the moment these type simplification routines are very primitive and do not handle a full range of possible situations. This is not due to technical, or theoretical difficulties but merely reflects a lack of time for implementation/testing. This is not seen as a major obstacle to the evaluation of DREAM transformations because simplification is not an inherent part of the ghosting operation. Note that the original expression simplifier is provided for the universal type "any" which is described below.

Backward compatibility — the typed WSL transformation engine no longer recognises untyped WSL programs. This makes it difficult to transform any untyped WSL code and makes testing more difficult (see below). To work around this problem we provide a backward compatibility parsing mode. In this mode the external representation parser automatically converts untyped WSL into a typed version. In this version all variables are given the type "any" which represents the semantics of the original data. Any simplification of expressions which contain this type is done by the original expression simplification routines.

Figure 6.6 shows an example of the same program being transformed in both the untyped and typed versions of the Maintainer's Assistant. Note the use of the type "any" in the typed version.

User interface — the user interface has also been extended to allow pretty printing and parsing of the extended constructs. This involved adding extra entries into the output format tables which describe the layout and typesetting of each construct in the table.clisp file. Changes were also required to the lex/yacc parser which reads

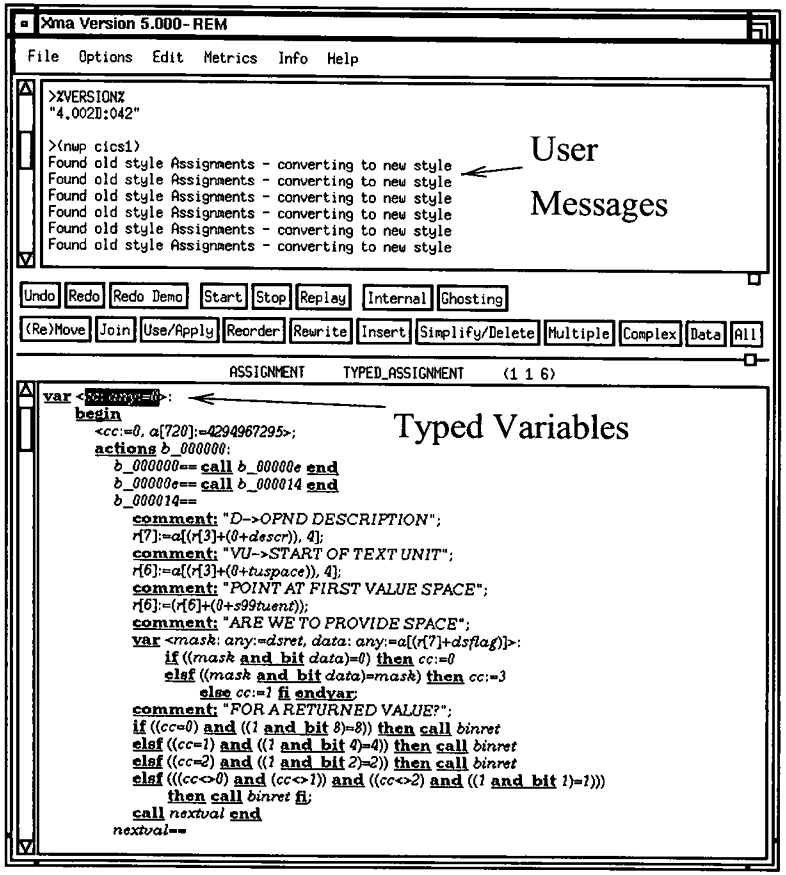

Xma Version 4.002e
File Options Edit Metrics Info Help
Undo Redo Redo Demo Start Stop Replay Internal Ghosting
(Re)Move Join Use/Apply Reorder Rewrite Insert Simplify/Delete Multiple Complex Data All

```
>%VERSION%
"4.002D:042"

>(nwp cicsl)
NIL

>(goto '(1 1 6))
(1 1 6)

>
```

```
ASSIGNMENT    ASSIGNMENT    (1 1 6)
```

```
var <cc:=0>:
    begin
      <cc:=0, a[720]:=4294967295>;
      actions b_000000:
        b_000000== call b_00000e end
        b_00000e== call b_000014 end
        b_000014==
          comment: "D->OPND DESCRIPTION";
          r[7]:=a[(r[3]+(0+descr)), 4];
          comment: "VU->START OF TEXT UNIT";
          r[6]:=a[(r[3]+(0+tuspace)), 4];
          comment: "POINT AT FIRST VALUE SPACE";
          r[6]:=(r[6]+(0+s99tuent));
          comment: "ARE WE TO PROVIDE SPACE";
          var <mask:=dsret, data:=a[(r[7]+dsflag)]>:
              if ((mask and bit data)=0) then cc:=0
              elsf ((mask and bit data)=mask) then cc:=3
                  else cc:=1 fi endvar;
          comment: "FOR A RETURNED VALUE?";
          if ((cc=0) and ((1 and bit 8)=8)) then call binret
          elsf ((cc=1) and ((1 and bit 4)=4)) then call binret
          elsf ((cc=2) and ((1 and bit 2)=2)) then call binret
          elsf (((cc<>0) and (cc<>1)) and ((cc<>2) and ((1 and bit 1)=1)))
              then call binret fi;
          call nextval end
        nextval==
```

Untyped Version


Xma Version 5.000-REM
File Options Edit Metrics Info Help
User
Messages
Undo Redo Redo Demo Start Stop Replay Internal Ghosting
(Re)Move Join Use/Apply Reorder Rewrite Insert Simplify/Delete Multiple Complex Data All
Typed Variables

```
>%VERSION%
"4.002D:042"

>(nwp cicsl)
Found old style Assignments - converting to new style
Found old style Assignments - converting to new style
Found old style Assignments - converting to new style
Found old style Assignments - converting to new style
Found old style Assignments - converting to new style
Found old style Assignments - converting to new style
```

```
ASSIGNMENT    TYPED_ASSIGNMENT    (1 1 6)
```

```
var <cc: any:=0>:
    begin
      <cc:=0, a[720]:=4294967295>;
      actions b_000000:
        b_000000== call b_00000e end
        b_00000e== call b_000014 end
        b_000014==
          comment: "D->OPND DESCRIPTION";
          r[7]:=a[(r[3]+(0+descr)), 4];
          comment: "VU->START OF TEXT UNIT";
          r[6]:=a[(r[3]+(0+tuspace)), 4];
          comment: "POINT AT FIRST VALUE SPACE";
          r[6]:=(r[6]+(0+s99tuent));
          comment: "ARE WE TO PROVIDE SPACE";
          var <mask: any:=dsret, data: any:=a[(r[7]+dsflag)]>:
              if ((mask and bit data)=0) then cc:=0
              elsf ((mask and bit data)=mask) then cc:=3
                  else cc:=1 fi endvar;
          comment: "FOR A RETURNED VALUE?";
          if ((cc=0) and ((1 and bit 8)=8)) then call binret
          elsf ((cc=1) and ((1 and bit 4)=4)) then call binret
          elsf ((cc=2) and ((1 and bit 2)=2)) then call binret
          elsf (((cc<>0) and (cc<>1)) and ((cc<>2) and ((1 and bit 1)=1)))
              then call binret fi;
          call nextval end
        nextval==
```

Typed Version (with Backward Compatibility)

Figure 6.6: Backward Compatibility for Language Parsing

the pretty printed representation of the program.

This presents a problem during the dynamic loading of new data types because the user interface requires re-compilation to introduce new components. This is not practical during a transformation session and needs some rework in future versions of the program.

6.3.2 Testing the Changes

At this stage the transformation engine has been extended to allow representation of typed WSL programs and all of the control flow transformations have been modified to work as before. The system does not yet include the module structure to allow addition of new types/type equivalences (although hooks for expression simplification are in place) and the DREAM (ghosting) module has not been added.

The operation of the transformation engine needs testing to ensure that the changes described above have been performed successfully. This testing was performed in two stages:

1. **Small, worked examples** — the Maintainer's Assistant is supplied with a number of sample WSL programs whose transformation is described in the user manual [53]. These examples were performed as per the manual to ensure that the transformations worked as documented. Whenever a problem was found its root cause was investigated and corrections were made to the transformation engine. Similar transformations were also examined to ensure that the same problem was not also present in those.

2. **Assembler restructuring** — once the simple examples were working as documented a larger example was attempted. This was actually the assembly code module which is used for case study two in chapter 8. During the test the raw translated code (from assembler to WSL) was pre-processed, prior to data transformation, using the "`fix_assembler`" compound transformation (see section 8.2.1). This transformation was developed as part of the initial Maintainer's Assistant research to aid in the removal of many of the idosyncracies found in assembly code. Execution of the transformation takes a number of hours and involves the use of many different transformations.

This test highlighted a number of problems which had not been caught in the previous stages and also highlighted two bugs in the original version of the Maintainer's Assistant. The first was due to incorrect simplification heuristics which cause unbounded expansion of an expression. This made the transformation execute for a long time (approximately 1 day!) and it eventually failed when Lisp's internal stack space was exhausted. The second problem was with the transformation of action systems (they allow gotos to be represented). The transformations were not recognising calls to other actions which occurred within certain constructs, e.g. loops, and were incorrectly transforming these. This type of construct was a feature of the case study code and had not been encountered in previous code.

Once the transformation was executing to completion arbitrary parts of the transformed program were compared with the original assembler code to confirm that the restructured program was functionally equivalent. This often involved some restructuring on paper to confirm equivalence of the two.

These tests highlighted a number of problems with the addition of data typing into the language. It is believed that the current version of the control flow transformations now operate in the same manner as the original except for the bug fixes described above. Future results will, however, be monitored to ensure this.

6.3.3 Adding Modules

The introduction of data type and type equivalence modules to the transformation engine involved the implementation of new code. These modules are only used by the DREAM module (except for expression simplification which is not yet fully implemented) and hence they have very little impact upon the rest of the system. As a result these modules do not require the same degree of integration testing as the code described above. The development of the type and equivalence modules was mainly done using a rapid prototyping model where functionality was tested against the desired result almost immediately.

A prime objective of the type and equivalence modules was to allow dynamic loading on demand thus making extension of the system easy. This was done using the Lisp dynamic program model which allows new program fragments to be loaded dynamically. Sections 6.2.2 and 6.2.3 described the interfaces for each of the routines which make up a type module and an equivalence module. The code which represents these routines was contained in a number of Lisp functions each of which corresponds to a particular interface definition. When a type/equivalence module is loaded into the system a small initialisation routine is called which adds this data into a global list of modules. The unique name of the module, as described in sections 6.2.2 and 6.2.3, is then used to retrieve the module's code when necessary.

```
;; Join two assertions together to form the union of the two
;; Parameters
;;   A,B - Internal Representations of the assertion
(defun GHOST_Discr-Assert_Union (A B Type)
  (Range_Add A B))

;; Derive a range assertion from the item and it's type
(defun GHOST_Discr-Get_Assert (Item Type Name)
  (Let ((Table ([_Match_] Expression
                          ((~>?~ Fn) (~>?~ E1) (~>?~ E2))
                          Empty)))
       (Cond ((Eq ([_Get_] Fn Table)
                  '+)
             (Range_Add ...)
              )
             (; Other operators
              ...
             )
             (T
              'Nil))))

...

;; The initialisation routine to install the integer type module.
(defun GHOST_Integer-Init ()
  (GHOST-Types_Add_Type 'Integer
                        #'GHOST_Integer-Blank_Assert
                        #'GHOST_Integer-Assert_Union
                        #'GHOST_Integer-Get_Assert
                        #'GHOST_Integer-Match_Assert
                        #'GHOST_Integer-Simplify_Expression))
```

Example 6.3: Module Implementation in Lisp

Example 6.3 shows a cut down version of the integer type module with outline code for some interfaces. At the bottom of the example is the initialisation function which adds the module into the global module lists.

```
(defun GHOST_Get_Assert (Item &key My_Name My_Type)
  (funcall (GHOST-Types-Get_Assert (Get (Get_Type_Category My_Type)
                                        GHOST-Types_Type-List))
           Item My_Type My_Name))

(GHOST_Get_Assert
 (Second (Args %Item%))
 :My_Name ([_Get_] Src_Name Table)
 :My_Type ([_Get_] Src_Type Table))
```

Example 6.4: Module Calling Interface

DREAM accesses a modules routines by calling a global function (shown in example 6.4) which corresponds to the desired interface. One of the parameters that is passed to this is the unique identifier for the module. These allow extraction of the module's code from the global list of modules.

6.3.4 Ghosting (DREAM Module)

The ghosting algorithm was described in detail in chapter 5 and the design of its implementation was given in section 6.2.1. In this section we describe the implementation process itself and show how the final ghosting transformation was developed.

The first stage of development was concentrated upon the main ghosting routine `do-ghosting`. This implements the main phase (see section 6.2.1) of the algorithm and care was taken to ensure that any testing performed at this stage did not rely upon any errors being caught in the initial phase.

The `do-ghosting` routine takes five parameters as shown in example 6.5. The first two of these are the names of the variables which are being transformed; the third is the type equivalence theory which is being used and the final two are information about the variables, e.g. assertions etc.

The routine transforms the currently selected program fragment and is designed to be re-entrant. It can therefore be called to ghost sequences of statements, e.g. in a **where** construct, which are subcomponents of the currently selected statements. After transforming a sequence of statements the `do-ghosting` routine returns up-

```
(defun do-ghosting (source-var
                    ghost-var
                    equiv-theory
                    source-goodies
                    ghost-goodies)
   ...
)
```

Example 6.5: The do-ghosting Interface

dated source and ghost goodies values which can be combined with corresponding values at entry to the routine.

The action performed by do-ghosting for each statement within its scope is dependent upon each particular statement. Development of this routine started with support for the assignments-to/uses-of the source/ghost variables. This was followed by support for basic control flow constructs, e.g. conditionals and progressed to support for more complex statements such as loops and action systems[6].

Some of the semantically complex statements such as loops and action systems proved difficult to implement because the input to the area is a combination of the initial input and the result of every iteration of the loop/action system. In the current implementation this processing has been restricted to finding the output from the first iteration and if it is a subset of original input then that original input is used. Otherwise the worst case input for that type is used. This saves processing time at a potential loss of accuracy.

This processing becomes more complicated when considering statements which may cause a change of control flow mid-way through a particular loop iteration or action. The loop exit, action call and procedure call statements are all examples of this and may cause a temporary, or permanent change to the control flow. These are handled by decomposing the blocks which contain them into a number of sub-blocks which are bounded by the statements which cause the control flow transfer. The ghosting operation is performed individually on these sub-blocks and the results of this is combined by the parent block, i.e. loop, action system, or subroutine definition.

[6] Action systems are used to represent code containing goto statements.

In general the parent block must propagate the output of a particular sub-block to the input of any block where the transfer of control may be passed. This is made difficult because procedure and action calls may be mutually recursive and it is generally not possible to determine statically the extent of these calls.

The current implementation takes the most basic approach to this and assumes that all calls are mutually recursive and therefore each sub-block is treated as though it may be preceded by any other sub-block and followed by any other. This is in line with Ward's [88] semantic definition of action systems and subroutine blocks which uses non-deterministic choice and recursion[7].

This approach means that individual sub-blocks must be shown to preserve assign-use relationships between variable assignments and uses. In our prototype work this is acceptable because valid assign-use relationships which are not handled by this method can be handled by applying Ward's control flow transformations to restructure the program into a form which can be handled. For an example of this problem and its solution see section 8.2.2 on page 215.

Figure 6.7 shows the control flow of the main constructs which are handled by the `do-ghosting` routine. Changes to the flow of control are denoted by dotted lines. The solid lines represent the statements which are executed. Assign-use relationships are checked between the points where control flow branches occur.

Once the main `do-ghosting` routine had been implemented the code which performs the initial and final phases was added. This was built into the **@Ghost** MetaWSL construct (example 6.6) which provides the maintainer's interface to data transformations. This routine is invoked with parameters specifying the source and ghost variables and the name of the equivalence theory which is to be used.

Construct	**Description**
@Ghost(*Source*, *Ghost*, *Equivalence_Theory_Id*)	Invokes Transformations.
[Ghost?](*Source*, *Ghost*)	Returns a list of applicable transformations.

Example 6.6: MetaWSL Ghosting Constructs

[7] Remember that tail recursion can be converted trivially into an iterative system.

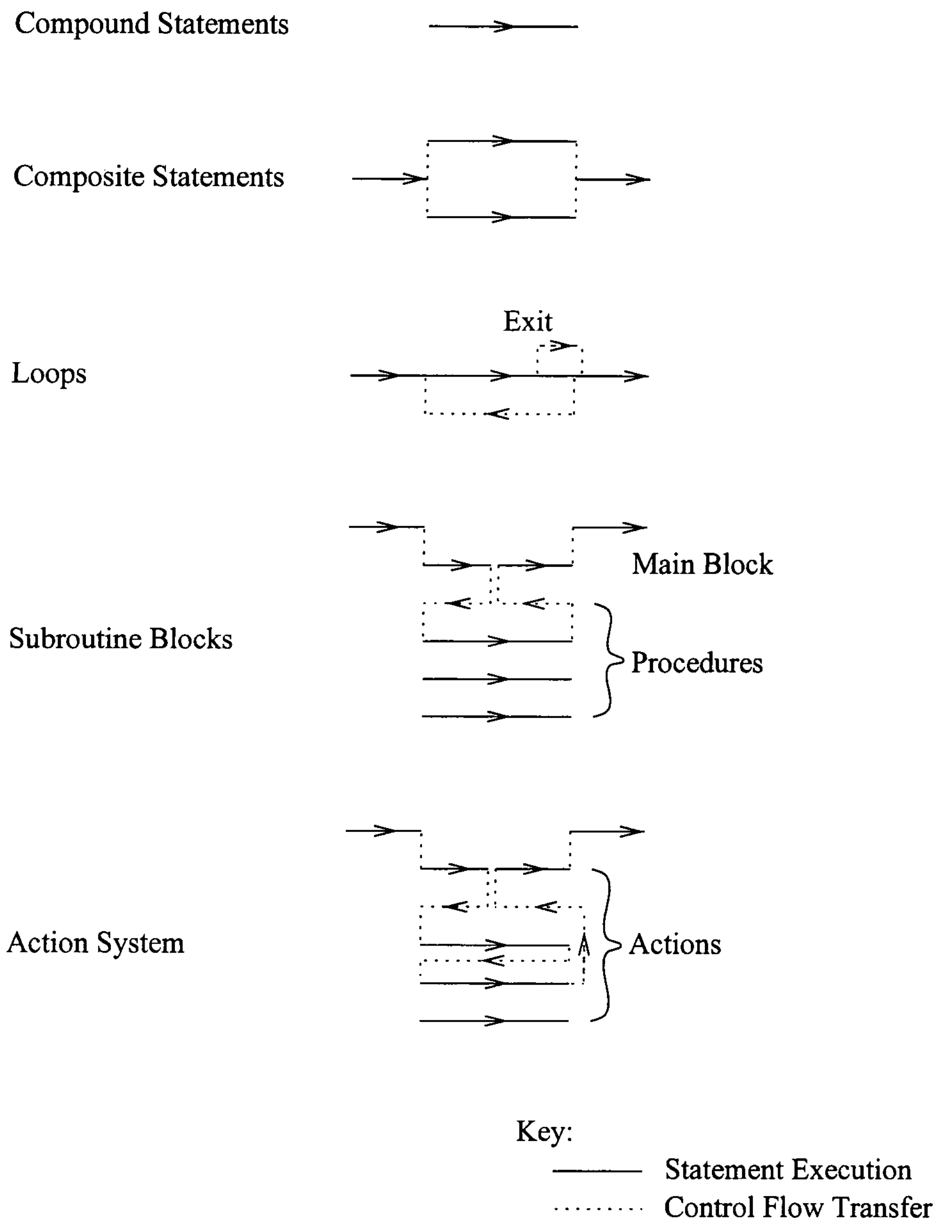


Figure 6.7: The Control Flow of WSL Statements

At many places during the ghosting routine calls are made to the type and type equivalence modules. These were done in the manner described in the previous section using either the type of the current variable or the *equivalence_theory_id* to select the appropriate module.

6.3.5 Ghosting User Interface

The final stage of the implementation was to provide the ghosting dialogue box in the user interface. This required a small amount of Xwindows code to display the box (shown in figure 6.4 on page 129) and also required implementation of the **[Ghost?]** $\mathcal{M}\varepsilon\mathcal{TA}$WSL statement. This calls the `Types_Compatible?` interface of individual type equivalence modules to determine whether it is applicable. It then passes this information to the transformation engine using part of the control flow transformation dialogue code. At this point a list of applicable transformation is presented to the user and then a corresponding **@Ghost** call is made when the user selects a transformation.

6.3.6 Code Summary

Data typing and data transformation capabilities have been added into the Maintainer's Assistant transformation engine. Table 6.1 summarises the changes which have been made to the original source code in order to perform this. The table shows the number of lines in the source files including comments. A more precise method of computation is not feasible due to the structure of Lisp code which uses brackets as statement/expression terminators.

Subsystem	**Lines of Code**		
	Added	**Changed**	**Total**
Transformation Engine (MA)	1400	3500	153709
Transformations (MA)	1000	1100	127486
User Interface (Xma)	300	300	56973
Ghosting (MA) — new	14789	n/a	14789
Total	17489	4900	352957
(percentage)	4.9%	1.4%	

Table 6.1: Source Code Summary

Table 6.2 details the hardware and software environment which was used during the extension of the tool. The development and testing was performed using the UNIX operating system.

Aspect		Description
Lisp Interpreter/ Compiler		Gnu Common Lisp (gcl), version 2.2 with locally developed patches for dynamic program image dumping on Solaris 2.4.
C Compiler		gcc, version 2.7.2.
OSF/Motif		Sun Common Desktop Environment (CDE) Motif.
X Windows		X11R6.1
Hardware		Sun SPARCstation-20 with 256 Mb memory (multi-user).
Operating System		Solaris 2.4.
Compile Time —	MA	38 minutes (typical).
	Xma	1 minute 40 seconds (typical).
Executable Size —	MA	10.59 Mb.
	Xma	1.13 Mb.

Table 6.2: Hardware and Software Environment

Typical data transformation timings are shown in table 6.3. These were taken while performing the transformations which are described in section 8.2. The transformations were performed over varying scopes to show how the size of code which is being transformed affects execution time.

The table shows the results of performing two data transformations (ghostings) over the entire scope of the program. To simulate the effect of transforming a smaller program a number of actions/procedures were successively removed from the program. In each case the number of assignments/uses which remained in the program is shown. The transformation time grows rapidly as the size of the program grows. This is due to the extra analysis which must be performed on the larger program. The Xma timings do not vary significantly. This is because the user interface does not take part in the ghosting itself. The memory size does not tend to vary significantly. It is believed that this is due to the operating systems memory allocation characteristics which tends not to recover significant amounts of memory from the heap.

Transformed Program Size	**MA**		**Xma**	
	CPU time (mm:ss)	**Memory** (Mb)	**CPU time** (mm:ss)	**Memory** (Mb)
Initialisation	0:02	11	0:01	6
Loading File	1:15	18	0:09	14
7250 lines				
37 changes	5:14	18	0:01	14
12 changes	5:01	18	0:02	14
4100 lines				
20 changes	1:54	18	0:01	14
9 changes	1:49	18	0:02	14
2294 lines				
16 changes	0:47	18	0:01	14
9 changes	0:44	18	0:02	14
1145 lines				
16 changes	0:21	18	0:02	14
0 changes	0:20	18	0:01	14

Table 6.3: Execution Times of Transformations

The implementation of typed WSL and ghosting is still in the prototype stage. The core functionality has been provided but some ancillary functions, such as expression simplification, have been left for future work. The use of the prototype tool for practical data transformation is examined in chapter 8.

6.4 Summary

DREAM data transformations are incorporated into the Maintainer's Assistant in a manner which complements the original modular design and user interface look and feel. The major factors in this prototype are:

- **DREAM transformation module** — this module concentrates knowledge about ghosting transformations and ensures that DREAM transformations are applied correctly and do not change the semantics of a program.

- **Modular data type and equivalence theories** — the use of a standard interface definition for data type modules and data equivalence modules makes it easy to add new data transformation functionality to the system. These modules assure correctness of the data transformations.

- **User interface** — the user interface removes the need for the maintainer to understand the inner workings of the transformation engine.

The implementation of these changes to the transformation engine has been discussed. It has shown that the addition of data transformation facilities to the Maintainer's Assistant is possible although there were some issues which had to be overcome before the typed version of the transformation engine would perform the same transformations that were available in the untyped version.

The next chapter examines the type and type equivalence theories which are necessary to perform data transformation using the DREAM. It shows how these are used within the tool and evaluates the operation of the tool during data transformation.

Chapter 7

Data Types and Type Equivalence

Chapters 4, 5, and 6 have shown how data transformation is performed in typed WSL using DREAM. In this chapter specific data type theories and data type equivalence theories are presented. These theories are not part of the original work that is central to this thesis but serve to illustrate the requirements of theories for data transformation. In particular they demonstrate the properties of a theory which are required for the data to be transformed.

This chapter concentrates upon two data types: integers and records. The former is an elementary data type whose values are indivisible (this theory of integers has no concept of bitwise manipulation). The latter is a common form of composite type which has a number of statically named components. Composite values can be referenced either as a whole or as individual components. This behaviour is allowed by the semantics of typed WSL which are described in section 4.2.2.

For each data type an outline of an axiomatic definition of its properties is given. This shows how the type is defined and how the properties of some of the type's operators are represented. For both of these types a brief outline is given which shows how the type theories are represented in the transformation engine's data type modules.

Section 7.2 shows how the integer and record data type theories are used in the generation of data type equivalence theories. An equivalence theory uses the semantics of the source and ghost variable's types (i.e. records and integers) to show that individual data expression refinement relations are valid. These data expression refinement relations are a central part of the ghosting transformation operation and provide the justification for the correctness of the transformation. The proof of these relations is demonstrated using two examples: one for integers and one for records.

To illustrate how these data type and equivalence theories are used in practice an example is presented which shows how the transformation engine transforms a single integer into a record which contains two integers. In this example the original integer represents an elapsed number of minutes and the record represents the same value as elapsed hours and minutes.

Finally section 7.3 outlines how other common data types can be represented and transformed in typed WSL. Examples from each of the four data type categories (elementary, composite, structural and dynamic) are presented. These do not go into the same amount of detail as is presented in the first half of the chapter but illustrate how the same techniques can be used for other data types.

7.1 Data Type Theories

The data type model which has been added into WSL is based around a shallow semantic embedding. This allows a data type theory to be introduced as a number of axioms which allow the properties of that data type to be described and reasoned about. These axioms must be written as formulae of infinitary logic[1] to allow integration with the WSL semantics and transformation theories.

This model of a data type represents values as atomic units which cannot be decomposed by the transformation theory. This means that all of Ward's [88] original transformations are still valid and also leaves scope for the proof of new transformations which use the axiomatic definition of the data type to prove transformations which combine control flow and data manipulation. These transformations are not, however, examined further within this thesis.

In addition to this model of data types the semantics of typed WSL have been extended to allow direct representation of composite data types. This makes the statically named components of these visible to the WSL control flow transformations. It, therefore, enables statements which use/assign-to individual components of these types to be transformed using Ward's transformations.

[1] Infinitary logic is used to describe the semantics of WSL.

Other type categories which do not have static component names, i.e. structural and dynamic types, cannot use this semantic extension. Therefore accesses to the individual components of these cannot be transformed using the original control flow transformations.

The general description of an assignment in typed WSL is

$$x := f(\dots)$$

where $f(\dots)$ is an expression which returns a value which has an appropriate type for variable x. This value may be arbitrarily complex and for data types which contain a number of distinct components it would contain an appropriate grouping of values. For assignments to individual components of a compound data type, e.g. to an entry in an array, the expression will generally take the original value of variable x as one of its parameters and modify that value in such a way that only an appropriate component is changed.

7.1.1 Integers

The integer is a common form of elementary data type. It is available in most programming languages and is used where exact values are required, e.g. for counting loop iterations. This is in contrast to real numbers which provide inexact representations although the range of values that real numbers can represent is much wider. Different programming languages/computer architectures provide many different variations of the integer data type. These variations reflect differences in size (number of bits) and differences in the semantics of the operators that act upon the integers. In many cases the machine architecture/processor that the programs run upon (especially for assembly code) can make a large contribution to the semantics of these data types.

Transformation of integer variables requires care to ensure that subtle properties of the variables and their associated operators do not cause the semantics of the program to be changed. In particular, care has to be taken with overflow conditions; some implementations just wrap around when this occurs, while others remain at the most positive, or most negative, value. In many situations the implicit properties of the program ensure that overflow cases do not occur but this is often not checked explicitly except in languages such as Ada where exceptions are raised if integer values go out of range. Note that exceptions are not considered further in this thesis because the WSL language does not contain an exception construct.

This section presents the syntax and semantics of integers as developed for use within typed WSL. These are used to develop transformation theories for integers and an example is given to show how the theories are used in practice. This theory is defined in an axiomatic manner and proofs which demonstrate the consistency of all integer constructs are not shown. This approach has been taken because the data type theories are not the central part of the thesis[2] and time constraints precluded development/use of a more rigorous theory.

Describing the data type

Integer types are described by the range of values that can be held within a specific instance of the data type. These ranges are then used to describe the behaviour of the data type and its operators. The theory presented below does not take the physical representation of the values into account and as such it cannot be used to reason about integers in terms of bit representations.

Integers can be conveniently represented by enumerating all of the possible values that can be contained within an instance of the type. This is done using a set enumeration which is written as

$$\text{discrete\_typedef} ::= \{v_0, v_1, \ldots, v_{n-1}, v_n\}$$

A shorthand notation is defined which allows a description of these ranges to be abbreviated. This is written as

$$\text{discrete\_type\_group}(v_s, v_e) ::= \{v_s, \ldots, v_e\} \text{ where } v_s < v_e$$

Example 7.1 shows how an integer type declaration is written in typed WSL.

Ranges are accessed, manipulated and compared by the new $\mathcal{META}$WSL constructs shown in table 7.1. Most of the definitions shown in the table are just com-

[2] the DREAM data transformation technique in chapter 5 is the central part of the thesis.

$$\underline{\textbf{type}}\ \textit{This-Type} \equiv [v_s - v_e].$$

where v_s represents the starting value and

v_e represents the final value in the range.

Example 7.1: Integer Type Declaration

mon set theoretic operators but the range constructors extract information from the program as follows:

- The **type_range** constructor returns the range which is specified within its parameter's type declaration.
- **range_of** computes the value held within the specified variable at the current point within the program. In theory this involves finding the ranges of all of the previous assignments to the variable and returning their combined range. The computation required to do this is non-trivial due to the $\mathcal{M}\varepsilon\tau\mathcal{A}$WSL internal program representation's structure which does not contain any direct links to the previous assignment. To work around this problem the DREAM module keeps track of the assignments to the source variable (see section 6.2.1), thus providing information for this variable directly. If the values for any other variables are required the current implementation does not search for them it merely returns the entire range of the variable's type. Potential solutions to this problem are discussed in section 9.2.
- The **make_range** operator is defined to be $discrete\_type\_group(s, e)$. In other words it returns the range which contains all of the values which are specified by the discrete type group.

These discrete type range operators (table 7.1) will be used in the following pages to describe the semantics of the integer data type.

Construct	**Definition**	**Description**
Constructors — return a range		
type_range(t)		The defined range of the type.
range_of(x)		The range of values in the specified variable.
make_range(s, e)		Make a range containing all of the integer values from s to e.
Selectors — return an integer		
start(a)	min(a)	Return the first (lowest) value in the range a.
finish(a)	max(a)	Return the (highest) last value in the range a.
Operators — return a range		
range_add(a, b)	$a \cup b$	Add range a to range b.
range_subtract(a, b)	$a \setminus b$	Subtract range b from range a.
range_intersection(a, b)	$a \cap b$	Return the intersection of ranges a and b.
Boolean Operators — return a boolean value		
range_includes(a, b)	$a \subseteq b$	Whether range a includes range b.
range_excludes(a, b)	$a \cap b = \emptyset$	Whether range a excludes range b.
range_intersects(a, b)	$a \cap b \neq \emptyset$	Whether range a intersects with range b.
range_equals(a, b)	$a = b$	Whether range a equals range b.
range_empty(a)	$a = \emptyset$	Whether range a is empty.

where a and b are discrete type ranges,
t is a type name,
x is a variable name and
s and e are integers.

Table 7.1: Operators upon Discrete Type Ranges

Integer Operators

A number of operators for integer types are included in the theory. These do not provide an exhaustive set of values but cover some of the more frequent operators. The following are defined:

Operator	Description	Operator	Description
$+$	Addition	$-$	Subtraction
$*$	Multiplication	$/$	Integer Division
mod	Modulus	$**$	Exponentiation
$<$	Less than	$>$	Greater than
$=$	Equals	$<>$	Not equals

Table 7.2: Integer Operators

Operator Semantics

Table 7.3 shows how the semantics of individual operators are defined. Specifically, it shows the semantics of the addition operator, "+", where the semantics of overflow are defined as truncation to the value nearest to the actual result, i.e. the most negative or positive. This means that the result of the addition always lies within the range of the data type. These properties of the operation are described using pre- and post-conditions.

Expression	$\underline{\textbf{funct}}$ "+"$(X_a : t, X_b : t) : t$
Input Ranges:	$R_a = \underline{\textbf{range\_of}}(X_a)$ $R_b = \underline{\textbf{range\_of}}(X_b)$ $R_t = \underline{\textbf{type\_range}}(t)$
Precondition:	$\underline{\textbf{range\_includes}}(R_a, R_t) \wedge$ $\underline{\textbf{range\_includes}}(R_b, R_t)$
Output Range (Postcondition)	$(R_o = (\underline{\textbf{make\_range}}((\underline{\textbf{start}}(R_a) + \underline{\textbf{start}}(R_b)),$ $(\underline{\textbf{finish}}(R_a) + \underline{\textbf{finish}}(R_b))) \cap R_t))$

Table 7.3: The Semantics of Integer Addition

The output range (or postcondition), R_o, shows how the output relates to the input. In this case the output range lies within the possible sums of the target's range. The characteristics of this implementation force the result to lie within the range of the functions return type and hence the output range is the intersection of the widest possible range and the range of values for the type.

If the definition of the plus operator treated overflow as having wrap-around semantics then the postcondition formula would need amending to account for this. The overflowed values would be converted to values in the lower (or upper) part of the range.

Exception semantics could also be defined for this operator but this is currently not possible in WSL because the language does not provide an exception mechanism.

The other integer operators have similar semantic definitions. These definitions are used to prove common properties of integer expressions such as commutativity and associativity. Note that each of these properties has a given precondition about the input's range over which it is valid. The semantics are also used by the type equivalence theories to show that particular transformations are valid. An example of this use is given in section 7.1.2.

Integer Type Module

The integer data type module is implemented using the interface described in section 6.2.2 (and shown in figure 6.2 on page 123). The type module represents the properties of the integer operators which have been semantically defined above. It does not, however, attempt to act as a theorem prover and derive new properties from the existing ones. This limits the type module's operation to those pre-proven properties but this is not seen as a major limitation because the same technique was used successfully in the untyped WSL implementation of the Maintainer's Assistant.

The module is conceptually separated into two groups of routines:

- `Blank_Assertion`, `Assert_Union`, `Get_Assert` and `Match_Assert` — these manipulate assertions about integer ranges which are extracted from the program. The assertions are used to represent the sets of values which may be stored within a particular variable. This information is used by the data type equivalence modules to show that equivalence relations, and hence transformations are valid.

  For example, the `get_assert` routine may be called to generate an assertion for an expression which consists of an addition operator. The routine would use the semantics of "+", which were defined earlier, to calculate the output range for the operator. This value would then be returned to the caller (i.e. the DREAM module) which would use this value appropriately.

The use of these assertion interfaces for each data type theory allows the theory to define an assertion format which is relevant to it. This means that the transformation engine can learn the format of the assertions for new types without re-implementation.

- `simplify_expression` — this provides the replacement for the functionality of the symbolic mathematics and logic modules which were used in the original Maintainer's Assistant. The expression which is passed to it as a parameter is simplified using the semantic properties of integers. This simplification is guided by heuristics which choose a specific simplification if more than one rule matches.

The implementation of these modules does not have a direct correspondence with the data type theory but there is sufficient similarity in the implementation to allow the correctness of the implementation to be verified by inspection.

7.1.2 Records

A record is a common method of grouping which allows a number of related objects to be referenced as a single unit. Each object within the record can also be referenced as an individual entity which allows operations to be performed upon the components of these types. The semantics of composite types are therefore formed by combination of the semantics of its components. This section shows how the semantics of records are represented in typed WSL.

Describing the data type

The primary function of a composite type is to store its individual components' values and to allow them to be retrieved from the data object. To allow this each component must be uniquely identified within the record. The type definition defines this as an ordered list of *component_name-type* pairs. Note that a composite type must have a static name. Types with dynamic names fall into the structural type category. Records fall into the composite type category and can take advantage of the typed WSL semantics (as defined in chapter 4) to map each component onto a unique WSL variable. The semantic definition is written as

$$\text{record\_type} ::= \langle \langle c_1, t_1 \rangle, \langle c_2, t_2 \rangle, \ldots, \langle c_n, t_n \rangle \rangle.$$

A record is defined (in a WSL program) using a corresponding ordered list of component names and types (see example 7.2). This information maps directly onto the semantics described above.

$$\underline{\textbf{type}}\ \textit{This-Type}\ \equiv\ (c_1 : a_1, c_2 : t_2, c_3 : t_3).$$

where c_1, c_2 and c_3 are component names,
t_1, t_2 and t_3 are type names.

Example 7.2: Record Type Declaration

One limitation upon the components of record types is that their constituent components may not include the current type which is being defined. If this was allowed then the record would be recursive and the static nature of composite types would mean that objects of that type would be infinite in extent. This restriction is enforced by most programming languages which allow static composite types. An exception to this would be a lazy functional language which only evaluates enough of the data structure to allow it to perform a particular operation.

Record Operators

In general there are very few operators which are specific to all record subtypes. This is because records are not generally meaningful in their own right but instead are usually associated with a particular context, e.g. abstract data types. The operators which act upon these abstract type records are usually defined as subroutines within the program.

The theory of record types which is presented here only defines the operators which are shown in table 7.4. This is consistent with the operators provided in many programming languages. The theory does, however, differ from some languages, e.g.

C, because the typed WSL assignment operator assigns to each individual component of a record whereas the equivalent operation in C must be done using a bitwise copy. Each of the record constructs is described below and we show how they are defined in terms of typed WSL's semantic framework.

Construct	Abbrev.	Description
Constructors — return a record value		
aggregate$(v_1, \ldots, v_n)$		A value for the specific type generated by aggregation.
Operators — return the component's l-/r-value		
l-select(c_n, x)	$x.c_n$	Retrieve the l-value of component c_n from object x to allow assignment to it.
r-select(c_n, x)	$x.c_n$	Retrieve the r-value of component c_n from object x.
Statements — no return type		
Assign(x, v)	$x := v$	Assign to all components of x from record v
Boolean Operators — return a boolean value		
Equals(x, y)	$(x = y)$	The corresponding components of two records are equal.
Others are usually defined in specific cases.		

where x and y are record variables,
$c_1 \ldots c_n$ are component names,
v is a record value and
$v_1 \ldots v_n$ are component values.

Table 7.4: Operators for Record Types

Aggregation — aggregation allows the construction of a record value. It is described in terms of the record's individual components and enforces the rule that all components must have a corresponding value. Semantically the aggregation operator creates an ordered list of expressions each component of which corresponds to a component in the record definition.

$$\text{aggregate} ::= \langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$$

When the typed WSL semantic transformations are applied (as shown in section 4.2.5), the list will be expanded into individual value-type pairs which will be used appropriately.

L-Select — this construct allows direct assignment to a particular component of the record. It has a direct semantic mapping onto the typed WSL semantic operator of the same name.

R-Select — this performs a complementary operation to l-select. It allows the value of a particular record component to be used directly within a program.

Assign — assign is not strictly a record construct; it is actually the primitive typed WSL construct which expands semantically into a parallel assignment to each component of the record as follows

$$x := v \;\equiv\; \langle x.c_0 := v.c_0, x.c_1 := v.c_1, \ldots, x.c_n := v.c_n \rangle$$

In terms of l-select and r-select is written as

$$\begin{aligned} < \underline{\textbf{l-select}}(c_0, x) &:= \underline{\textbf{r-select}}(c_0, v), \\ \underline{\textbf{l-select}}(c_1, x) &:= \underline{\textbf{r-select}}(c_1, v), \\ \ldots & \\ \underline{\textbf{l-select}}(c_n, x) &:= \underline{\textbf{r-select}}(c_n, v) > \end{aligned}$$

The semantics of typed WSL enforce the rule that the types of the left-hand side and right-hand side must be the same. The value on the right-hand side could therefore be generated from another variable or from an aggregation construct.

Equals — The equals operator specifies that all components of one record are equal to those of another record. The semantics of this are defined as:

$$x = y \;\equiv\; (x.c_0 = y.c_0) \wedge (x.c_1 = y.c_1) \wedge \ldots \wedge (x.c_n = y.c_n)$$

This assumes that all of the component types of the record have equals operators defined.

Record Type Module

The record type module differs from the integer type module (an elementary type) because it must be capable of handling information about all of the record's components. This makes the record type module dependent upon the type module of its individual components and it calls upon these modules to provide assertions and to simplify expressions.

- `Blank_Assertion`, `Assert_Union`, `Get_Assert` and `Match_Assert` — the assertions which are produced/manipulated by these routines consist of lists of assertions which describe the current properties of each component of the record. Each sub-assertion is generated by calling an appropriate routine from the component's type module.

  The `Get_Assert` routine uses the definitions of any record operators. These describe how the assertion is to be formed in more detail. This may cause the routine to call the `Get_Assert` routine for a subexpression of the main expression.

- `simplify_expression` — this performs the simplification of any record type operators and may invoke the simplify routines of component type routines.

The implementation of the record type module is particularly straightforward because of the relatively small number of operators which are defined for it.

So far in this chapter we have demonstrated how type theories can be constructed in typed WSL and have outlined how these can be used in the implementation of type modules. We will not go into more detail about the description of data type theories in this thesis but we will use the theories to show how type equivalence theories are developed and how they are implemented.

7.2 Data Type Equivalence Theories

Data type equivalence theories are used to demonstrate that a data expression refinement relation (see section 5.5.2) is valid at a specific point within the program.

The validity of this relation allows the ghosting transformation to be performed at that point. This, in turn, allows the representation of the data to be transformed and therefore allows a program to be re-engineered.

DREAM separates the data transformation into three stages:

1. Generating assignments to the ghost variable which are equivalent to the existing assignments to a source variable;
2. Propagation of the assertions which show the equivalence of the assignments to the places where the variables are used and
3. Use of this assertion information to generate an equivalent expression which uses the ghost variable rather than the source variable. If this expression can be generated then the data expression refinement relation is shown to be true and the ghosting succeeds.

The relation between the source and ghost variables is governed by an invariant. This invariant is used to generate the ghost variable assignments from the source assignment and is then combined together with an assertion about the range of values which could have been assigned in the source assignment. The assertion is propagated to the places where the source variable is used (with a potential merging of assertions whenever two control flow paths join). The invariant part of the assertion is then used to generate a suitable use for the ghost variable and the data expression refinement relation is proven for this specific relation. In this section we show how the truth of the data expression refinement relation can be demonstrated for both integer and record types. The final section of this chapter shows how these theories are applied in the transformation engine.

7.2.1 Integer Equivalence Theory

To demonstrate how an integer equivalence theory is proven we will use a small example where the representation of a year value between 0 and 99 is to be transformed to values between 1900 and 1999. This is a likely scenario given the current problems regarding the year 2000.

Let us say that we wish to replace the integer variable "*oldyear*" with the integer variable "*newyear*" using an invariant of

$$newyear = oldyear + 1900$$

This would correspond to the data transformation shown in example 7.3 assuming that we know that the function *input*() returns values from 0 to 99.

```
begin                                          begin
  var < oldyear : value                          var < newyear : value
                 := input() >:                                   := (input() + 1900) >:
    output(oldyear + 1900);               ≡        output(newyear);
  end                                            end
where                                          where
  type value ≡ [0 - 10000].                      type value ≡ [0 - 10000].
end                                            end
```

Example 7.3: Year 2000 Date Transformation

The transformation must guarantee that the transformed program provides the same set of possible values to the *output*() routine. The first stage of doing this is to replace the assignment to *newyear* with an assignment to *oldyear* according to the invariant. The semantics of integers are used to show that this can be done by proving that the *newyear* variable can hold all of the possible values which correspond to *oldyear*'s value.

At this point an assertion is generated which states that

$$(oldyear \in \{0, \ldots, 99\}) \wedge (newyear = oldyear + 1900)$$

This assertion is then moved using Ward's transformations to the point in the program where *oldyear* is used.

A new expression which corresponds to a use of *oldyear* is generated by rearranging the invariant to produce the original value of *oldyear* from the *newyear* variable. In this case the expression is

$$newyear - 1900$$

This will eventually be simplified using the integer type theory rules to produce the simple expression presented in example 7.3.

To show that this ghost expression can replace the source expression we must show that a corresponding data expression refinement relation is true. In this case the relation is

$$\begin{array}{l}\exists h : integer \leftrightarrow integer \bullet \\ \quad \forall oldyear, newyear \bullet (oldyear, newyear) \in h \implies \\ \qquad \big((\bar{g}(newyear) \subseteq \bar{f}(oldyear)\big) \wedge \big((\bar{g}(newyear) = \emptyset) \equiv \big(\bar{f}(oldyear) = \emptyset\big)\big)\big)\end{array}$$

where $\bar{f}(oldyear)$ is $\{p \mid p = oldyear\}$,

$\bar{g}(newyear)$ is $\{q \mid q = newyear - 1900\}$ and

$h = \{\langle 0, 1900\rangle, \langle 1, 1901\rangle, \ldots, \langle 99, 1999\rangle\}$.

The relation h specifies the range over which the ghosting must be true and is determined from the assertion about the source variable's possible values and the invariant. In this case it is the set of equivalent years.

The truth of the data expression refinement relation is shown using the semantics of the integer data type and standard mathematical reasoning techniques. The truth could be proven by case analysis but in this case it is easier to use the invariant to rewrite $\bar{g}(newyear)$ to be

$$\{q \mid q = (oldyear + 1900) - 1900\}$$

This simplifies to *oldyear* and hence the relation can be shown to be true.

In some cases the transformation may fail because there may not be a suitable inverse of the invariant relation over the possible range of source values. For instance, if we were transforming example 7.3 in the reverse direction, e.g. from a four digit year to a two digit year, and we defined an invariant of

$$newyear = oldyear \bmod 100$$

then it would not be possible to generate an inverse relation if the range of the *oldyear* variable spanned more than 100 years.

The DREAM theory of data transformation ensures that the proof of the data expression refinement relation is sufficient for the data transformation that it describes to be performed.

Integer Type Equivalence Module

Type equivalence modules are implemented in a similar manner to the data type modules. They provide only two interfaces: `assignment_body` and `use_body`. These use the supplied invariant to select an applicable equivalence relationship; this in turn allows an appropriate assignment-to/use-of the ghost variable to be generated. If no equivalence relationships are applicable then the equivalence routine signals to the DREAM module that the transformation has failed.

Invariant	$X_g = X_s + C$
Input Ranges:	$R_g =$ **type_range**(X_g) $R_s =$ **range_of**(X_s)
Assignment:	$X_g := X_s + C$
Expression:	$X_g - C$
Validity Condition:	**range_includes**(**make_range**(**start**$(R_s) + C$, **finish**$(R_s) + C), R_g)$

Table 7.5: Integer Transformation Relation

Proof of the data expression refinement relation by the transformation engine is generally non-computable. For this reason we adopt a similar approach to that used to describe the properties of individual data type operators. A general invariant is used to describe a particular representation change. This is used to produce an applicability condition for the use of this relation and if the condition is satisfied the specified assignments and uses can be used to replace the source variable. For example, the transformation from a two to a four digit year, which has been presented in this section, uses the invariant

$$newyear = oldyear + 1900$$

and its equivalence relation can be generalised as shown in table 7.5.

This states that the ghosting transformation is valid if the range of the ghost variable's type (R_g) includes the entire range of the source variables possible input values (shifted along by C). The rule also states what the ghost assignments and expressions will be in this case.

7.2.2 Record Equivalence Theory

Proof of record type equivalences is performed in a similar manner to that used for elementary types. It is slightly complicated however because the value of a record variable is made up of a number of individual components. This means that the source and target expressions $f(x)$ and $g(y)$ are described in terms of the whole record. Proof of the equivalence relation is shown by demonstrating that it is valid for all possible combinations of individual values.

The source and target assignments/expressions do not necessarily have to use all of the components of the records which are being transformed. For instance, it is likely that one component of the source record is assigned-to or used within the scope of the ghosting. In this case the equivalent expression, $g(y)$, would be described in terms of a corresponding component in the target expression.

Common transformations for record types include those which add and remove components from the record. Adding a component could use an invariant of

$$(x.c_1 = y.c_a) \wedge (x.c_2 = y.c_b) \wedge (x.c_3 = y.c_c)$$

where x has as a record type with components c_1, c_2, c_3 and

y has as a record type with components c_a, c_b, c_c, c_d.

This operation would be valid in all cases because the new component will not be used initially and therefore it will not affect the output of the program.

The converse, removing a component, is only possible if that component is not used within the scope of the ghosting. It is not immediately obvious how this equivalence is proven because it seems that there is no mechanism in the theory for showing that components are missing. The answer to this is, however, simple because the invariant does not have an inverse if the value of the missing component is used. This means that there is no suitable equivalence relation to prove and hence the ghosting transformation is not valid.

The proof of a record type equivalence relation/record transformation is not shown in the thesis because it is performed in a similar manner to that for integer transformation. For the same reasons record type equivalence modules are not described here either.

7.2.3 Ghosting Example

The data type theories and type equivalence theories demonstrated above are used to allow data transformation using the ghosting technique. In this section an example is presented which shows how the theories are combined in practice to perform data transformation. The example shows how the representation of elapsed time can be changed from a number of minutes into a record consisting of hours and minutes. During the description of this transformation a number of features are identified which shows how the formal transformation theory can help to avoid potential re-engineering errors.

The initial code is shown in example 7.4 and has been carefully designed to show only the essential components for this transformation. In particular, it does not show why the time value is used or any intermediate steps between assignments and uses. The transformation examples are presented using the six steps of ghosting given in chapter 5 on page 97.

The example transformation will replace the "*time*" variable by a "*daytime*" record which contains "*hours*" and "*minutes*" fields. The structure of these is shown in step 1 (example 7.5) where the replacement type and variable have been introduced. The relationship (invariant) between the old and new data representations is

$$\mathit{time} = (\mathit{newtime.hours} \times 60) + \mathit{newtime.minutes}$$

The data expression refinement relation which describes this invariant is shown in table 7.6. This has been derived in a similar manner to the theories described in

```
begin
  external $< h : int_{32}, m : int_{min} >$:
    var $< time : int_{32} := 0 >$:
      $time := ((h \times 60) + m)$;
      $print($“hours:  ”,
            $(time \div 60)$,
            “, minutes:  ”,
            $(time \text{ mod } 60))$
    end
  end
where
  type $int_{32} \equiv [0 - (2^{32} - 1)]$.
  type $int_{min} \equiv [0 - 59]$.
end
```

Example 7.4: Time — The Initial Code

```
begin
  external $< h : int_{32}, m : int_{min} >$:
    var $< time : int_{32} := 0$,
           $newtime : daytime := [0, 0] >$:
      $time := ((h \times 60) + m)$;
      $print($“hours:  ”,
            $(time \div 60)$,
            “, minutes:  ”,
            $(time \text{ mod } 60))$
    end
  end
where
  type $daytime \equiv (hours : int_{32}, minutes : int_{min})$.
  type $int_{32} \equiv [0 - (2^{32} - 1)]$.
  type $int_{min} \equiv [0 - 59]$.
end
```

Example 7.5: Time — Step 1: New Types and Variables Added

Invariant	$X_s = (X_{ga} \times C) + X_{gb}$
Input Ranges:	$R_s = \underline{\textbf{range\_of}}(X_s)$ $R_{ga} = \underline{\textbf{type\_range}}(X_{ga})$ $R_{gb} = \underline{\textbf{type\_range}}(X_{gb})$
Assignment:	$<X_{ga} := X_s/C;$ $X_{gb} := X_s \bmod C$
Expression:	$(X_{ga} \times C) + X_{gb}$
Validity Condition:	$\underline{\textbf{range\_includes}}(\underline{\textbf{make\_range}}(0, C), R_{gb}) \wedge$ $\underline{\textbf{range\_includes}}(\underline{\textbf{make\_range}}(0, ((\underline{\textbf{finish}}(R_s)/C) \times C)), R_{ga})$

Table 7.6: Time Transformation Relation

table 7.5; further proof is not shown here.

The invariant is then used to introduce assignments to the ghost variable which are equivalent to the source variable's assignments. Example 7.6 shows the program after these have been introduced. Note that the assignments to *newtime.hours* and *newtime.minutes* have been simplified in the example from the raw substitution described by the transformation relation. For instance, *newtime.hours* has been simplified from

$$\textit{newtime.hours} := ((h \times 60) + m) \div 60;$$

to

$$\textit{newtime.hours} := h;$$

This simplification relies heavily upon the properties of both h and m because their ranges determine whether the arithmetic simplifications are possible. In the case above the division can be rewritten, using the law of integer distributivity over "+", into two division operations whose results are then summed together. This is written as

$$\textit{newtime.hours} := ((h \times 60) \div 60) + (m \div 60)$$

The first part of the resulting addition can be simplified to h using the laws of identity, i.e. $(X \times C)/C \equiv X$. The simplification of the second part depends upon

```
begin
  external < $h : int_{32}, m : int_{min}$ >:
    var <$time : int_{32} := 0$,
           $newtime : daytime := [0, 0]$ >:
      $time := ((h \times 60) + m)$;
      $newtime.hours := h$;
      $newtime.minutes := m$;
      $print$("hours: ",
           ($time \div 60$),
           ", minutes: ",
           ($time$ mod 60))
    end
  end
where
  type $daytime \equiv (hours : int_{32}, minutes : int_{min})$.
  type $int_{32} \equiv [0 - (2^{32} - 1)]$.
  type $int_{min} \equiv [0 - 59]$.
end
```

Example 7.6: Time — Step 2: Assignments to Ghost Variable Introduced

the range of m. In our example the m variable is limited to values between 0 and 59 which means that the result of $m \div 60$ is always 0. If the range of the variable extended beyond 59 then the division could not be simplified further.

This situation highlights the benefit of using a formal tranformation technique because manual introduction of the new representation may have resulted in this complication being overlooked and the expression being simplified anyway. In general the transformation engine will tend to err on the side of caution and will not simplify expressions unless it is sure that it is possible.

Example 7.7 shows the program with assertions added which state the range of the source variable and state that the source and ghost variables are equivalent. This assertion is a by-product of the assignment introduction because the equivalence theory guarantees that this is correct for the ghost assignment.

The assertion is then moved to the places where the source variable is used. In this case the operation is trivial but it could involve moving the assertion along many possible control flow paths until a use is found. The assertion movement could be interrupted in a number of ways:

1. if the source variable is re-assigned at a point between an assignment and use

```
begin
  external $< h : int_{32}, m : int_{min} >$:
    var $<time : int_{32} := 0,$
          $newtime : daytime := [0, 0] >$:
      $time := ((h \times 60) + m);$
      $newtime.hours := h;$
      $newtime.minutes := m;$
      {range_includes(range_of(time), type_range(int32))$\wedge$
       $(time = (newtime.hours \times 60) + newtime.minutes)$};
      print("hours:  ",
            $(time \div 60)$,
            ", minutes:  ",
            $(time \bmod 60)$)
    end
  end
where
  type $daytime \equiv (hours : int_{32}, minutes : int_{min})$.
  type $int_{32} \equiv [0 - (2^{32} - 1)]$.
  type $int_{min} \equiv [0 - 59]$.
end
```

Example 7.7: Time — Step 3: Equivalence Assertions Introduced

then the current assertion cannot be propagated any further and a new ghost assignment/assertion is generated;

2. if the ghost variable is assigned-to then the equivalence assertion cannot be propagated further and the transformation fails.

At this stage the ghosting transformation is completed by replacing uses of the source variable with equivalent uses of the ghost variable. Example 7.8 shows the program after the uses have been transformed. These uses have been simplified in a similar way to the assignments leaving just uses of the *hours* and *minutes* components.

Example 7.9 shows the final result of the transformation when the source variable, its assignments and the assertions have been removed. This represents both steps 5 and 6 as shown in the description of ghosting in chapter 5.

```
begin
  external < $h : int_{32}, m : int_{min}$ >:
    var < $time : int_{32} := 0$,
            $newtime : daytime := [0, 0]$ >:
      $time := ((h \times 60) + m)$;
      $newtime.hours := h$;
      $newtime.minutes := m$;
      {range_includes(range_of($time$), type_range($int_{32}$))$\wedge$
       ($time = (newtime.hours \times 60) + newtime.minutes$)};
      $print$("hours:  ",
              $newtime.hours$,
              ", minutes:  ",
              $newtime.minutes$)
    end
  end
where
  type $daytime \equiv (hours : int_{32}, minutes : int_{min})$.
  type $int_{32} \equiv [0 - (2^{32} - 1)]$.
  type $int_{min} \equiv [0 - 59]$.
end
```

Example 7.8: Time — Step 4: Source Variable Uses Ghosted

```
begin
  external < $h : int_{32}, m : int_{min}$ >:
    var < $newtime : daytime := 0$ >:
      $newtime.hours := h$;
      $newtime.minutes := m$;
      $print$("hours:  ",
              $newtime.hours$,
              ", minutes:  ",
              $newtime.minutes$)
    end
  end
where
  type $daytime \equiv (hours : int_{32}, minutes : int_{min})$.
  type $int_{32} \equiv [0 - (2^{32} - 1)]$.
  type $int_{min} \equiv [0 - 59]$.
end
```

Example 7.9: Time — Transformation Complete

Issues Raised

Study of the example presented above has highlighted a number of important issues which characterise the DREAM data transformation method. These are:

1. **Highlighting errors/code dependencies** — the introduction of an assignment to the *newtime.hours* variable highlighted the potential for a dependency upon the m variable. If m (input minutes) could introduce values which are outside of the range of minutes in an hour (i.e. 60 or greater) then the assignment to *newtime.hours* would be drastically different. This feature of the code could easily be a coding error or it could be a deliberate dependency which a maintainer may have missed and consequently introduced an error into the program. The use of formal data transformations automatically takes this into account by removing the potential for these subtle mistakes.

2. **Refinement is also possible** — the time transformation example used an equivalence relationship between the old and new representation. In some cases it would be possible to use refinement relations which take away repeated information from the data. For instance, an angle measurement may originally have represented the total distance turned (i.e. there could be more than one rotation) but the variable representing this may only be used to measure the current angle travelled around a circle. The transformed variable could then hold the original value modulo 360 degrees.

3. **Different complexities of transformation theory are possible** — a transformation theory does not have to handle every possible relationship between source and ghost variables. It is acceptable to use simple theories for basic cases such as direct integer to integer conversion but more complex theories are needed if there is a more complex relationship between new and old. The transformation theories must, however, be able to recognise the limits of their capabilities.

4. **User input is important for meaningful restructuring** — if the invariant used in the ghosting example above had been slightly different, e.g.

$$time = (newtime.hours \times \mathbf{59}) + newtime.minutes$$

then the final representation of the data would have been different. The main change is that the ghosted assignments/uses would not simplify as they did in the example. This would have resulted in a semantically equivalent program but the program may not be any more meaningful.

The integer and record data type and type equivalence theories which have been presented here will be used in section 8.2 as the basis for evaluating the automated application of DREAM transformations.

7.3 Other Data Types

In the remainder of this chapter we will consider a number of other data types which are commonly found in programming languages. An overview is given which shows how their semantics would be described in WSL and which shows some of the transformations which may be performed upon these types. No attempt is made to fully specify these types. This is left for future research.

7.3.1 Discrete Types

Discrete types are ones whose values can be precisely enumerated (although there may be infinitely many of these values). Section 7.1.1 presented a theory for the integer data type which is a common discrete type. There are many other discrete types which share similar properties and which can, therefore, be described and transformed in a similar manner, i.e. in terms of a set of possible values. Common discrete types are shown in table 7.7.

Equivalences between discrete types are described in terms of mappings between individual values in the source and ghost types. These mappings do not have to map to/from every possible value but do have to provide a mapping for all of the values which may be stored within the source variable.

Data Type	Description
Integers	Described in section 7.1.1.
Bits	A subclass of integers which contain only the values zero and one. These are commonly referred to in terms of groups of them which make up the machine representation of data, e.g. bytes.
Characters	These are very common in programming languages and a number of different groupings/encodings exist. Examples include: ASCII and EBCDIC.
Enumerations	Enumerations are used to represent program specific discrete types. They allow names to be defined which represent individual states. There may often be an ordering associated with the members of an enumeration. This allows comparison of members using relational operators, e.g. "<" and ">". Many languages allow the implementation of an enumeration to be specified explicitly by defining a mapping between names and numbers.
Boolean	A boolean value represents the true and false logical values which are used in boolean algebra.

Table 7.7: Common Discrete Types

In many cases the equivalence mapping will be one-to-one. For example, the ASCII character encoding maps each character onto an integer. Part of this encoding is shown pictorially in figure 7.1. A transformation could use this equivalence relation to convert the storage of a character to an integer, or vice-versa. Another example may involve a transformation from the EBCDIC encoding into the ASCII encoding. In this case not every EBCDIC character has an ASCII equivalent and any variable which stores one of the unmapped characters would cause the transformation to fail. The transformation would also fail if the source program performs an integer comparison of the codes for the two characters. The ghost program would not be able to do this operation because the ordering of the characters in the two encodings is different.

It is possible to have a many-to-one mapping as shown in figure 7.2. In this example the source program encodes a boolean value as an integer: zero represents a false condition and non-zero represents a true condition. A transformation may be used to change the integer representation into a boolean representation. This mapping is valid provided that the original program does not perform any operation which distinguishes between any of the non-zero values.

A many-to-one transformation is actually a data abstraction operation. The

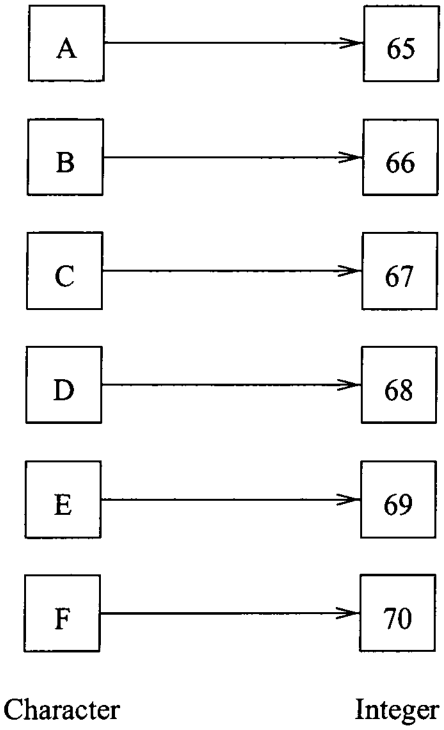


Figure 7.1: Mapping ASCII Characters onto Integers

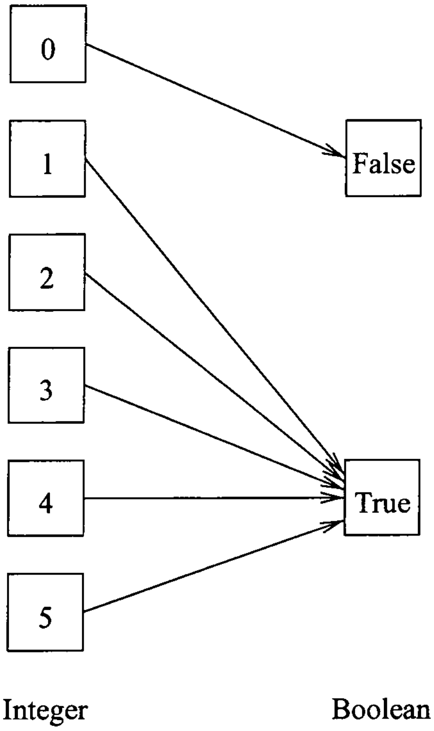


Figure 7.2: Mapping from Integers to Boolean Values

transformation is hiding some of the implementation detail of the program. Note, however, that in terms of program output the transformation is actually an equivalence relation because there is no observable difference in the final output state.

One-to-many transformations can be performed in a similar manner to those described above. These correspond to the reverse of the many-to-one transformations and are used to refine the implementation of data. A typical example of this would be a refinement of false into a non-zero number whose exact value reflects the reasons for failure of a particular operation. A transformation which is used to perform this type of operation will need some form of manual/heuristic guidance to help it to determine what the correct value should be.

Finally many-to-many equivalence relations are possible. These allow arbitrary mappings between source and ghost types. Note that in a mapping of this nature the equivalence relation is actually mapping a group of source values onto a group of ghost values. Any one of the values in a particular group is equivalent to any other within that group.

Table 7.8 lists some of the type equivalences which could be used between discrete type variables. Equivalences between discrete types and other categories of type will be discussed when the appropriate types are introduced.

Type	**Transformation**
Integer	Subtype to different integer ranges.
	Change the representation of an integer, e.g. 00, ... ,99 is transformed to 1900, ... ,1999.
Bits	Convert bits into integers.
	Convert to boolean values.
Characters	Convert to integer representation and vice-versa.
	Convert between ASCII, EBCDIC and other encodings.
Enumeration	Convert the use of integer constants into enumeration types.
Boolean	Convert an integer boolean, e.g. in C, into a logical boolean.
	Encode an error code into a boolean which signals a reason for failure.

Table 7.8: Discrete Type Transformations

In each case the ghosted program must be type correct. That is a ghost expression must have the same type as the corresponding source expression. For instance, if a character is replaced by an integer representation and the original program uses the original character as input to a character printing routine then the ghosting must replace the original character with an expression which converts from the integer into a character.

7.3.2 Real Numbers

Another commonly occuring elementary data type is the real number. This allows the representation of inexact quantities as opposed to the exact quantities which are represented by discrete types.

There are two main representations of real numbers: fixed point and floating point. The former represents numbers with a fixed exponent and the latter has a variable exponent which means that the type may hold a wider range of values. Typically, however, this means that the implementation of floating point arithmetic is much more complex than that of fixed point.

Property	**Description**
Range of values	The maximum and minimum values which could be stored within the variable.
Precision	The accuracy of a particular representation.
Error	The maximum difference between actual (real world) and stored numbers.

Table 7.9: The Properties of Real Numbers

Table 7.9 shows the properties which may be used to describe a real number. These place a bounds upon the range of values which may be stored within the data type and describe the accuracy of the representation. Note that error is represented separately because the error value is cumulative over multiple arithmetic operations whereas precision is not.

The semantics of real number operators are described in terms of these properties and describe the relationship between input and output values.

Transformation of real numbers may involve conversion between differing representations of the same type, e.g. single and double precision values. It may also involve conversion between one form of real number and another. In some situations it may also be possible to convert between real numbers and integers.

These transformations involve checks to ensure that a new representation does not change the bounds of errors to such an extent that the output from the program changes. For instance, checking for a value of zero is usually done by checking that the value lies within certain bounds. A decrease in precision may cause more error to be introduced into a calculation which may cause the result to be recorded as zero when it would previously have been non-zero. These types of situation could have possibly disasterous consequences and should be detected by the equivalence theories.

Transformation of the real numbers is inherently difficult due to their complex semantics. Any transformation which is performed will generally require a significant amount of information about the values which are stored within it. This places limits upon the transformation of real number types because of the stringent criteria which are involved. But it does not necessarily prohibit transformation.

7.3.3 Sets

The set is another primitive data type which is used to represent groups of values. A set is defined in terms of the individual values which may be held within it. The properties of a set are described in terms of the values that the set may, or may not contain at a particular moment in time. This then allows reasoning about the properties of set operators.

Transformation of sets may involve a number of changes including:

- changing the values which are contained within the set into equivalent values;
- splitting a set into two distinct subsets and vice-versa;
- replacing the set representation with an implementation of it.

Each of these is discussed below.

The values which are contained within a set may be replaced with other values which represent the same state. This is analogous to the transformations which were described for discrete types and involves the same steps of reasoning. For example, a set of characters, "a" to "z", may be transformed into a set of integers, "1" to "26". This differs from the discrete type transformations, however, because the transformation must be a one-to-one equivalence due to the fact that sets cannot contain multiple occurrences of the same value.

A set may also be split into two distinct sets whose combined content is equivalent to those of the original. This is done by defining which values belong to each set in the new representation. In general any set can be transformed in this way because the contents of the original set can be recovered by computing the union of the new sets.

Transformation of sets may involve the conversion between set data types and their implementation. A set implementation may be represented in a number of different ways: the set may be represented as an array of boolean values where each element of the array corresponds to a particular value which may be contained within the set. If the boolean value which corresponds to a particular member of the set is true then that value is contained within the set.

Figure 7.3 shows the mapping between a set and an array of boolean values. This mapping is described by the invariant

$$(x \in S) \;\equiv\; (A[x] = 1)$$

where x is a member of the set S and

A is the array which corresponds to set S.

When using this invariant an assignment which uses a set union operation would be replaced as follows:

$$S := S \cup \{x_1, \ldots, x_n\} \;\equiv\; \begin{array}{l} < A[x_1] := 1, \\ \quad \ldots, \\ \; A[x_n] := 1 > \end{array}$$

Conversely an array assignment would be transformed into a set operation as follows:

$$A[x] := 0 \;\equiv\; S := S \cap \{x\}$$

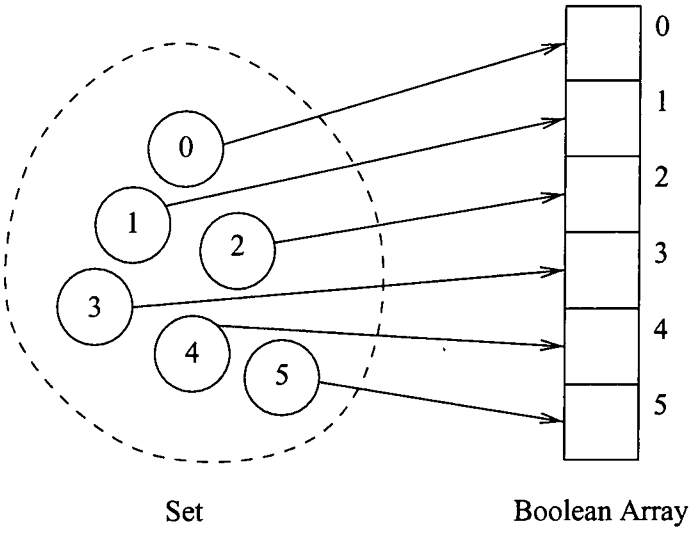


Figure 7.3: Converting a Set into an Array of Boolean Values

This transformation may not be appropriate if a set may contain a wide range of values. In this case another possible representation is to store the actual set values in an array.

7.3.4 Abstract Data Types

The record data type was presented in detail in section 7.1.2. It has a static number of components which are accessed via static component names. Many programming languages provide some form of record data type and often allow the production of abstract data types using the record as a base for the storage of the type's components. To accompany this a number of subroutines are often defined to provide operations upon the abstract data types.

Abstract data types can be represented in typed WSL in two ways:

1. in an implemented form where subroutines are used to manipulate the data and

2. in an abstract form where the abstract data type is a WSL data type in its own right.

The former way of describing the data type uses the record data type's semantics and the semantics of the individual components of that record. Individual operations upon the data values are represented using subroutines which perform specific operations.

The latter method encapsulates the semantics of the data in a data type theory and allows reasoning about the data type as a whole rather than as individual components. It also allows the semantics to be described in an abstract way without reference to the control flow semantics of WSL.

A typical abstract data type is the complex number. This has two components, the real and imaginary parts and has a number of associated arithmetic operators such as "+" and "×". The semantics of these are defined in terms of arithmetic which involves both the real and imaginary components.

These two types of representation of the complex number provide a way to re-engineer a program to remove the specific implementation of the data type and to replace it with an abstract version which has well known properties. This will generally make the program much easier to understand because the complexity of the data manipulation is contained within the data type definition rather than in program code. It also provides more opportunities for transformation of the data types into other formats because type equivalence theories can be generated using the semantics of the data type.

7.3.5 Static Arrays

The static array is a common structural data type. It contains a fixed number of components which all have the same data type. Each component of the array is accessed via a unique identifier which may be computed dynamically during program execution. The semantics of an array are described in a similar way to those of records although the dynamic computation of component names means that WSL control flow transformations cannot determine whether two accesses to array components are referencing the same component.

The properties of a particular array at any one moment in time are described in terms of the contents of individual components. Typically this information will not be precise because it is not generally possible to statically determine the index value which is used in any particular assignment. In many situations, however, it will be possible to show that the value of the index expression has not changed between an assignment and use.

Many transformations of the representation of specific arrays do not require that the contents of the array are known precisely. They simply require that one storage location is mapped onto another location in the ghost program. This ghost location must not be assigned-to from any other part of the program.

The simplest transformations of arrays involve extension or restriction of the length of the array. The former case is trivial in most cases because the addition of the extra entries does not affect the entries which are already present. One foreseeable situation where this may not be possible is if the array has an operator which calculates a value, e.g. a checksum, based on all of the values in the array. If the operator is used on an extended array then a different checksum may be returned. For this reason the transformation would fail.

Restriction of the array's length involves showing that the removed components are not assigned-to/used in the source program. If this can be shown then an equivalent array with less components can be introduced.

In both the extension and restriction cases the invariant used to define the transformation would have the form

$$\forall \mathrm{x} : t_x \bullet (x \in R_g) \implies (a[\mathrm{x}] = b[\mathrm{x}])$$

where a and b are the source and ghost arrays,

t_x is the type of the array subscript and

R_g is the range of subscripts where the ghosting is valid.

In most programming languages it would only be possible to add/remove elements at the end of an array because the subscripts are natural numbers, i.e. the integers from 0 upwards. If addition/removal of elements from the start or middle of an array is necessary then this can be performed using a slightly different invariant which has the form

$$\forall x : t_x \bullet (x \in R_g) \implies (a[x] = b[f(x)])$$

where $f(x)$ is a function which maps the source array's elements onto the ghost array's elements.

Figure 7.4 shows the mapping between array components where

$$f(x) = x + 3$$

is used as the invariant. This adds three elements to the beginning of the array shifting the original values further up the array. Of course the ghost array must have an extra three entries for the transformation to succeed.

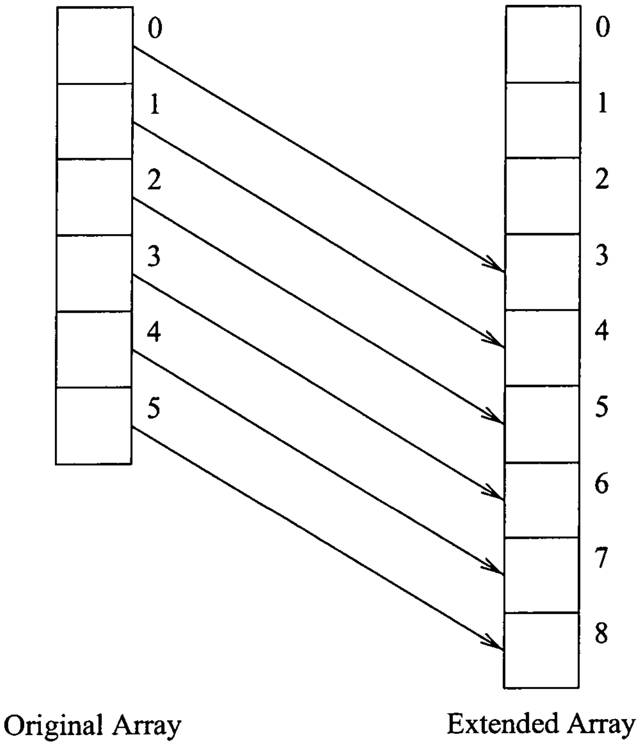


Figure 7.4: Adding Three Elements to the Beginning of an Array

It is also possible to add extra entries into the middle of the array using the same technique. In this case the function must have a discontinuous range which does not include the subscript values which are being added.

Another possible transformation is to reverse the order of the entries in the array. This can be done using an invariant of

$$f(x) = c - x$$

where c is the number of elements in the array

assuming that the array subscripts begin at 0.

So far we have only examined the mapping of arrays onto similar arrays. It is also possible to map a single dimensional array onto a two dimensional array and vice-versa. Figure 7.5 shows an example of this mapping where the invariant

$$x[(i \times c) + j] = y[i][j]$$

is used. This transformation is only possible if array b has at most c entries in the dimension which is indexed by variable j. Both arrays must also begin at index 0. If this is not the case then the multiple elements in the source array will be mapped onto one element of the ghost array causing the transformation to fail.

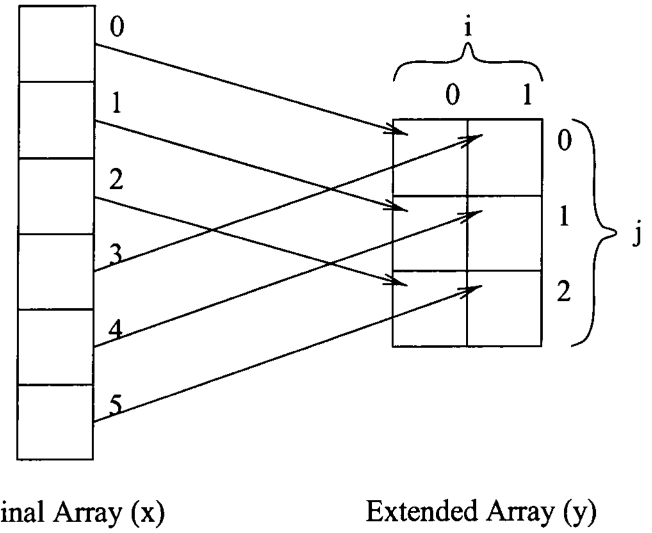


Figure 7.5: Transforming a Single Dimensional Array into a Two Dimensional Array

Another transformation which may be desirable is to transform an array of records into a record of arrays. This changes the emphasis of the variable groupings and allows each array to be transformed individually to take advantage of any structure which is inherent in one particular collection of values. An inverse transformation is also possible.

7.3.6 Dynamic Types

Dynamic types differ from composite and structural types by virtue of having variable numbers of components. This means that the data type must provide mechanisms for creating and destroying components at arbitrary places within the program. It must also provide a means by which links between individual values can be represented. These links can be changed at any time to reflect the changing state of the data.

Representation of this type category is not easy in the WSL language (both untyped and typed versions). This is because the language is carefully defined to provide static variable scope and to ensure that an assignment to one variable does not result in a change to any other.

The approach which is adopted is to represent the whole of the dynamic data as a single value in a typed WSL program. Any change to a particular part of the data is then described as a change to the whole data value. An assignment of the whole, or a part of the value to a variable, results in a new copy of the entire data structure. This is distinct from the original value and any change to the new structure does not result in a change to the original.

The definition of a dynamic data type may be performed in any suitable way. One method which is applicable in a number of situations is to use a recursive definition where a data value contains other instances of the same data type. One example would be

$$\underline{\textbf{type}}\ T \equiv \underline{\textbf{recursive}}(x : \mathit{integer}, y : T).$$

where T is a recursive type,

x is a value which is contained within the type instance and

y is a recursive instance of the type.

The component "*y*" is the central part of this recursive definition and may contain another instance of the type or may contain a "null" entry which represents the end of the recursive structure.

This thesis does not go into detail about the transformation of these data types but we present one example which will give the reader an idea of how a program which contains dynamic data types may be restructured.

Example 7.10 shows the definition of a record which contains an integer component and a static array. The integer is used to provide the index into the array. A possible transformation of this data structure would be to transform it into a dynamic list

```
begin
  ...
where
  type contents  ≡  array (1...100) of integer.
  type group  ≡  record(items : contents,
                        index : integer,
                        last : integer).
end
```

Example 7.10: An Array which is Used to Represent a Dynamic List

Figure 7.6 shows the mapping between the record and dynamic list structure. In this the first element of the array forms the head of the list and the rest of the array is contained within the tail of the list. This is repeated for successive elements of the list.

The structure of the list does not allow access to elements which occur earlier than the current point within the list, i.e. each sublist does not contain a reverse link to its parent. In order for a transformation from an array representation into a list representation to be valid the accesses to the array must be performed in a manner which maps onto this. One way in which this may be done is shown in table 7.10.

In the array representation the *index* component represents the head of the list and the *last* component denotes the last value in the list. The program must enforce the rule that *S.index* <= *S.last* because if this were not the case then it would be possible to access values which are beyond the end of the list.

In general, the array accesses must be shown to preserve an invariant which

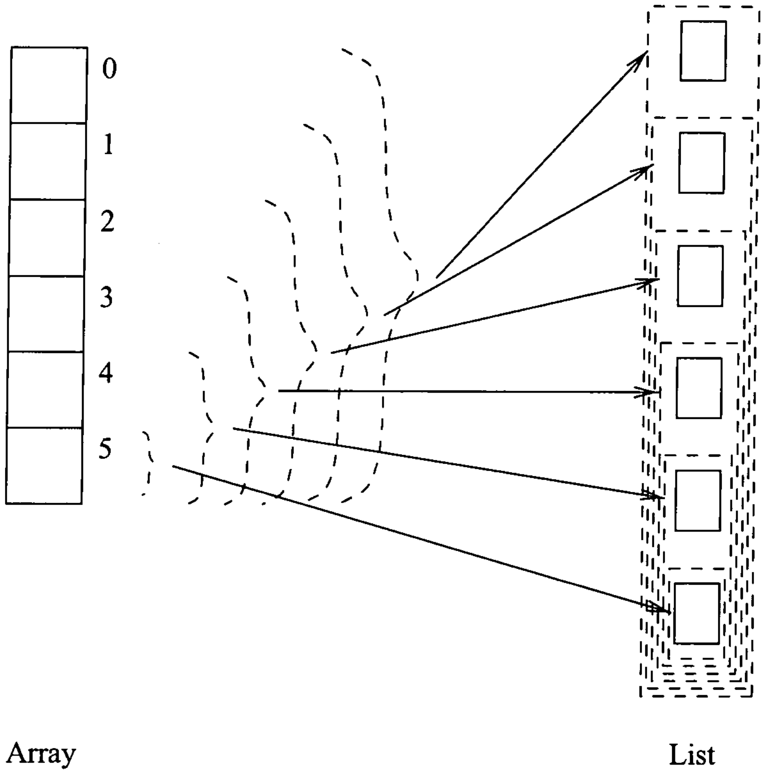


Figure 7.6: From an Array to a Dynamic List

Array Access	**List Access**	**Description**
Assignments		
$S.index := S.index + 1$	$G := tail(G)$	Move to the next item in the list.
$< S.item[S.last] := v;$ $S.last := S.last + 1 >$	$G := append(G, v)$	Append a new item onto the end of the list.
Expressions		
$S.items[S.index]$	$head(G)$	The value at the head of the list.
$S.items[S.index + 1]$	$head(tail(G))$	The second value in the list.
Conditions		
$S.index = S.last$	$is\_empty(G)$	Is the list empty (null).

where S is the source variable of type "*group*",
G is the ghost variable of type "T",
v is an integer value and
in all cases $S.index <= S.last$.

Table 7.10: Array and List Accesses

describes the list properties. This invariant is derived directly from the recursive type definition which states that a valid list can be either "null" (empty) or is a concatenation of an element and another valid list.

Similar transformations may be performed for other dynamic data types. Table 7.11 summarises these.

Data Type	**Description**
Tree	A tree differs from a list because each node in the tree has two siblings. This means that the tree invariant must be demonstrated for both subtrees.
Graph	A graph is much more complex than a list or a tree because there is no restriction as to the links between different items in the graph. This makes it much harder to identify any structure in the graph. It is, however, possible to use an invariant to recognise a graph providing that all updates to the graph can be shown to either preserve the structure of the graph or to be a base case which trivially satisfies the invariant.
Relation	A relation represents mappings between two sets of values. There are a number of different sub-categories of relations such as: functions, partial functions, onto relations and surjective functions. Programs may represent these in a number of different ways; common methods include: arrays of records which contain the mappings' source and target elements; arrays where the subscript is the source element and lists which contain the individual mappings.
Dynamic Array	Dynamic arrays are similar to static arrays although the size of a dynamic array may change during program execution. There is a definite potential for transformation between these data types although the transformation from a dynamic to a static array must ensure that there are sufficient components in the latter.
Strings	Many different varieties of strings are found in programming languages. Strings have many properties in common with both static and dynamic arrays. In some cases it may also be possible to treat a string as an elementary data type where the string represents an abstract name.

Table 7.11: Transformation of Dynamic Data Types

The most important aspect of dynamic data type transformations involves ensuring that any accesses to the data do not extend past the limits of the dynamic data structure. In general, this reasoning is not trivial and the examination of these issues is left for future research.

7.4 Summary

This chapter has examined data type and type equivalence theories. These are used to perform data transformation in DREAM and their availability is, therefore, essential for data re-engineering work. Table 7.12 summarises the data types which have been examined. These are grouped according to their type categories and a check against table 4.4 on page 59 reveals that most of the common data types which were introduced there have been examined.

Type Category	Data Type	
Elementary	Discrete	Bit
		Integer
		Boolean
		Enumeration
		Character
	Real Number	Fixed Point
		Floating Point
	Set	
Composite	Record	
	Abstract Data Type	
	Tuple	
Structural	Static Array	
Dynamic	List	
	Tree	
	Graph	
	Relation	
	Dynamic Array	
	String[3]	

Table 7.12: Data Types and Transformation Theories

The types which have not been examined: c-unions, c-pointers, first-class functions and object-oriented classes were all identified in chapter 4 as being unsuitable for transformation using DREAM.

The main focus within this chapter was on the integer and record data types. Examples have been presented which show how the semantics of these can be represented within typed WSL. Type equivalence theories have also been presented which show how integers and records can be transformed both individually and together.

[3] A string may also by a structural type.

These theories form a basis for the experimental work which is presented in the next chapter.

Chapter 8

Results

In this chapter we present experiments which have been performed to show that DREAM data transformations can be used in practical re-engineering situations.

The first part of the chapter presents the results of a case study which examines the use of DREAM for reverse engineering commercial legacy code. This case study shows the steps which are necessary to transform the source code into a higher level representation. The reverse engineered code is compared with the original design documents for the code and is shown to correspond to the original design. These results have been verified by members of IBM's staff who have confirmed that DREAM data transformations would provide a useful tool during software maintenance.

The second part of the chapter looks at the automation aspects of the transformation work and describes the experiences which were encountered during use of the extended Maintainer's Assistant. These extensions to the tool have been evaluated in a second case study where the tool was used to re-engineer aspects of the data in a large assembly code module. This case study highlighted some of the practical aspects of applying the ghosting transformations.

The final part of the chapter re-examines the criteria for success which was presented in chapter 1. It draws upon the work presented in this and other chapters to answer the questions which were posed.

8.1 Case Study One — Data Reverse Engineering

In this section we take a step back from the theoretical aspects of DREAM data transformations and look at re-engineering from the maintainer's perspective. The reverse engineering of a substantial amount of source code is examined to identify typical aspects of data that need re-engineering and to identify the transformations which are useful for performing this. The case study was performed by hand without the aid of a transformation engine. No attempt has been made to give rigorous formal arguments for the correctness of the transformations used here but an outline of the reasoning that is required is given where appropriate.

The case study experiments were performed upon a substantial section of code which forms part of a large commercial software product. It is written in the PL/X language and for reasons of confidentiality specific examples cannot be given. The analysed code is a complete subsystem of the software product and has been subject to maintenance over a number of years. The subsystem was re-developed from a previous version as part of a major re-engineering initiative some years ago. This re-developed code was originally specified in Z and a document exists which describes the refinement/development performed to convert between the specification and executable source code. This makes the code an excellent subject for this case study because the results can be checked back against the original design.

To make the case study realistic it was performed in relative isolation from the developers of the product and used the source code, in PL/X, as the basis for experimentation. The study examined the data present within the program and the operations which are applied to that data. A number of data objects and associated operators were analysed in more detail and their implementation was reverse engineered back towards the original specification.

8.1.1 Analysing the Code

The code which was analysed during the case study covers 370 pages of listing, specification and text. This contains approximately 4000 lines[1] of source code which is made up of a number of modules. These modules interact with each other through

[1] This is an approximate figure because an electronic copy of the code was not available.

shared data structures which are protected by read/write locks[2]. There are five major examples of these shared data structures each of which is composed of a number of smaller structures. PL/X uses a layered form of data structuring which forms a hierarchy of data objects providing a limited amount of data encapsulation. The data structures are therefore explicitly defined and this provides many clues to the uses of each data item. The data structure hierarchies are summarised in table 8.1 which shows the number of sub-structures and data objects at each level in the hierarchy.

Level	**Number**	**Description**
1	5	Subsystem data structures
2	11	Data structures
	13	Data objects
3	2	Data structures
	46	Data objects
4	2	Data objects
Total	**18**	**Data Structures**
	61	**Data objects**

Table 8.1: Case Study One — Data Structure Summary

At a number of places throughout the data structure definitions, certain fields are inherited from elsewhere and, therefore, have a common format across the whole of the system. These are inherited textually (from header files) within the code providing standard components for use by external routines. The hiding of the context of these data objects would be desirable because they add no useful information (they provide audit and debugging information) to the subsystem being re-engineered and a global type definition would make it easier to control the data items.

The data structures are instantiated many times throughout the lifetime of the system at a number of different points within the code. One main (static) data structure acts as the base of these dynamic data instantiations and holds pointers to the relevant parts of the dynamic structures. In subroutines and blocks a number of local variables are used to hold temporary values or to reference parts of the dynamic data structures.

[2] The system has a number of concurrently executing threads.

The dynamic data structures are changed in response to stimuli from the system environment. The code ensures that the system uses the data values in a synchronised manner and that the structures remain intact. These represent examples of a number of common data paradigms such as lists, trees and hash tables.

Individual data elements are strongly typed and hold a number of different types of values. These data types include:

- integers (binary and fixed point),
- characters,
- bit fields and
- pointers[3].

There is no use of floating point arithmetic within the program.

The language allows data items to be “respecified” within local blocks. This temporarily gives a variable a different type which allows access to the same machine representation in a number of different ways. This is undesirable because it makes the code harder to understand and makes the semantics of the data value more difficult to define. In practice this did not cause a problem within this module because the data items in question are only accessed as one format. When examined globally the data items are found to provide a small amount of storage for the routine which creates (and owns) the data item. This would pose a problem for re-engineering of the whole program.

The code contains 84 constants which are used within the code to define limits for data ranges and to enumerate the values of particular data objects. Fifty nine of these constants are used to enumerate status codes and errors. They are grouped into five classes and would be better implemented as enumeration types whose names are the only attribute which are visible to the programmer. The other constants are mainly strings although there are also a few integer values.

[3] Remember that these are not being covered in detail in this thesis.

8.1.2 Reverse Engineering the Code

Detailed reverse engineering using data transformations was performed by hand upon a small segment of the code (approximately 80 lines). This showed how transformations would be used in practice and allowed assessment of the ease with which data could be transformed. The target set for the reverse engineering of this code was to abstract the data structures and therefore provide a more meaningful representation.

The code was selected because it is very self-contained (it does not use many external routines) and it uses a wide variety of different data types. Its main purpose is to search a dynamic list structure to find a specific entry.

This fragment of code is executed very frequently within the system and its control flow structure has been heavily optimised. This involves ensuring that the most frequently used control flow paths do not perform any operations which are not necessary. Before the data could be restructured these control flow optimisations were removed to make the program's effect upon the data explicit.

For example, when the code begins to examine the list it first checks to see if the entry at the head of the list is the one that it is looking for. If so it does not bother to initialise a number of variables which are used during traversal of the list. When this optimisation is present the values of the variables cannot be determined if the optimised path is taken. This makes reasoning about the data difficult and hinders transformation.

Once these optimisations had been removed the data structures were transformed with the aim of reverse engineering them to produce abstract versions of the original representation. This was performed in the following steps:

1. **Create explicit structures** — the PL/X data structures needed restructuring to produce a number of logical groupings which reflect the meaning of the data. These groupings still form the same basic data structures but can then be individually manipulated by transformations.

   Example 8.1 shows this type of transformation for an employee information record whose components can be separated into two distinct groups: personal information about the employee and information about the employees length of service with the company. This latter information is separated into a subrecord which would allow it to be abstracted at a later date.

```
begin
  ...
where
  type employee ≡
    record(emp_id : integer,
                name : string,
                sex : char,
                joined : time,
                last_payrise : time,
                ...).
end
```

≡

```
begin
  ...
where
  type employee ≡
    record(emp_id : integer,
                name : string,
                sex : char,
                info : emp_history,
                ...).
  type emp_history ≡
    record(joined : time,
                last_payrise : time).
end
```

Example 8.1: Creating Explicit Data Structures

2. **Subtype variables** — variables were examined to identify their use within the program and subtypes appropriate to these uses were introduced.

```
begin
  var < my_emp_id
          : integer := 0 >:
    my_emp_id
     := get_emp_num();
    ...
    print(my_emp_id);
    ...
  end
where
  ...
end
```

≡

```
begin
  var < my_emp_id : id := 0 >:
    my_emp_id := get_emp_num();
    ...
    print(my_emp_id);
    ...
  end
where
  ...
  type id ≡ [0 – 60000].
end
```

Example 8.2: Subtyping Variables

A common form of this type of transformation is shown in example 8.2. Here the transformation has analysed the assignments to the *my_emp_id* variable and determined that only values between 0 and 60000 are assigned to it. In this example the variable is assigned either 0 or the return value of function *get_emp_num*. The latter is known to return values between 1 and 60000 (from its definition).

Another example where subtyping can be performed is on the values which represent string lengths. These can be constrained to represent the possible string lengths which may be used.

3. **Create abstract data types** — the structures and subtyped entities can now be transformed into new representations involving primitive abstract data structures. Linked lists were very common within the code; these were characterised by the operations which manipulated the links between individual items. The abstract types use operators which reflect the manipulations that can be performed upon them, i.e. adding an item to the end of a linked list, and these are handled in a similar way to the addition and subtraction operators for integer types.

   Example 8.3 shows a typical transformation. Here a list which was implemented using pointers has been converted into an abstract list type which has a number of procedures/functions associated with it. In this example the list retains the concept of the current element that is being examined. As such the *emp_id*() and *name*() function implicitly return values associated with the current element.

   The theory needed to perform this type of transformation has not been covered in this thesis. Preliminary ideas for the theory can be found in section 9.2 which describes possible further work.

4. **Convert to specifications** — a final stage in the reverse engineering of the code involves converting from implementable types into specifications of the relationships which these types represent. This involves recognition that a linked list, whose contents are identified by a single key value, is actually just an implementation of a mapping relation.

   Example 8.4 shows the final result when the list abstract data type which is shown in example 8.3 is converted into a partial function between employee identifiers and employee details. In the example the loop which finds the desired value in the list has been replaced by a function call to print out the name for the appropriate employee.

   This type of transformation has not been covered in detail in this thesis because it requires combined transformation of both control flow and data structures.

```
begin
  var <my_emp_id : id := 0,
         this : emp_ptr := get_emp_list() >:
    my_emp_id := my_emp_num();
    while (this.emp_id ≠ my_emp_id) do
            this := this.next;
    od
    print(my_emp_id);
    print(this.name);
  end
where
  type emp_ptr ≡ pointer employee.
  type employee ≡ record(emp_id : id,
                                name : string,
                                sex : char,
                                info : emp_history,
                                next : emp_ptr).
  type emp_history ≡ record(joined : time,
                                   last_payrise : time).
  type id ≡ [0 – 60000].
end
```

≡

```
begin
  var <my_emp_id : id := 0,
         this : emp_list := get_emp_list() >:
    my_emp_id := my_emp_num();
    while (emp_id(this) ≠ my_emp_id) do
            this := next(this);
    od
    print(my_emp_id);
    print(name(this));
  end
where
  type emp_list ≡ list employee.
  type employee ≡ record(emp_id : id,
                                name : string,
                                sex : char,
                                info : emp_history).
  type emp_history ≡ record(joined : time,
                                   last_payrise : time).
  type id ≡ [0 – 60000].
end
```

Example 8.3: Introducing Abstract Data Types

```
begin
  var <my_emp_id : id := 0,
         employees : emp_rel := get_emp_rel() >:
    my_emp_id := my_emp_num();
    print(my_emp_id);
    print(employees(my_emp_id).name);
  end
where
  type emp_rel  ≡  emp_id ⇸ employee.
  type employee  ≡  record(emp_id : id,
                            name : string,
                            sex : char,
                            info : emp_history).
  type emp_history  ≡  record(joined : time,
                               last_payrise : time).
  type id  ≡  [0  –  60000].
end
```

Example 8.4: Conversion into a Specification

This combined reasoning is beyond the scope of DREAM but is a logical progression for future work.

These transformations were performed without the aid of the transformation engine. Each transformation performed has been carefully examined to ensure that a suitable implementation of each data type theory would allow the re-engineering to be performed with automated assistance (except for those noted as further work).

8.1.3 Reverse Engineering Summary

This case study has examined the possibilities for reverse engineering the data contained in a substantial portion of code from a large commercial software system. Investigation of the code was performed without initial knowledge of the code's function and with only a little knowledge of the underlying system. This provided a maintenance environment which is similar to that which may be encountered by a maintainer who has just started working upon a particular software product.

The results of the reverse engineering were checked by cross comparison with original design documentation. This showed that they were consistent although there were a few differences in the grouping of some data items which were caused by a misunderstanding of the purpose of the data. Some other differences were found but these were traced to modifications made to the code after the design had been performed. The design document had not been updated to include these.

During the reverse engineering a number of aspects of the code/data were identified as being candidates for data re-engineering. These included:

1. The data types used within programs are typically implementation types, such as integers, rather than application derived types (subtyping);
2. Constants are used to represent logical states rather than using enumeration types;
3. Data is often split up in source code; logically it should be aggregated;
4. Properties of data are often implicit, for example, that the program relies on integers represented in 16 bits;
5. Low level data types, such as pointers, are used to implement high level types, e.g. lists and trees;
6. Data and the code operating upon it are disjoint (however, we are not addressing the problem of recognising modules here);
7. Data is often used for several purposes in the same program;
8. Data is often badly used in terms of its scope. Typically scope is not limited to the area were data is applicable.

Staff at IBM's laboratories were consulted to confirm that the analysis of the code and results of transformations were consistent with their experiences with the code. They confirmed this and stated that the types of operations which are available in DREAM would be useful for their practical re-engineering projects. They did, however, express a desire for a more abstract form of transformation application than is currently provided in DREAM. In particular they were keen to be able to apply a transformation directly upon a variable without having to introduce ghost variables to hold the final result. This aspect is discussed further in section 9.2.

8.2 Case Study Two — Automated DREAM

In this section the application of data transformations using the enhanced transformation engine is examined. The tool is used to transform the data which is present in a substantial piece of legacy code from a large software system. The aim of the case study is to examine the practical use of the DREAM data transformation and to demonstrate that the current implementation gathers sufficient information from the program to allow transformations to be completed. Note that the case study is not aimed at testing the completeness/flexibility of the implementation of data type and type equivalence theories.

This case study was performed in conjunction with IBM (UK) Ltd. who provided access to their code and whose employees helped in the selection of a suitable piece of code. For reasons of confidentiality this section does not provide any specific examples from the code. It does, however, provide similar examples when appropriate.

Selection of the source code was made towards the end of the three year period of study and involved visits to IBM's site to talk with the personnel responsible for maintaining individual sections of code. After initial discussions four code modules were selected as potential candidates for re-engineering. All of these had been heavily modified over a number of years and are destined for substantial re-engineering in future releases of the software products. Two modules were written in IBM/370 assembly language and two in the PL/X language (they were also designed to run on IBM/370 based machines). After examination of these modules it was decided to rule out the PL/X code because a suitable PL/X to WSL translator was not available[4]. The two remaining assembly code modules both used a large number of different data structures and had a number of interesting features. Only one of these was fairly self-contained (the other was responsible for transferring data between modules without manipulating it) and was chosen for the case-study.

The remainder of this section is separated into four parts:

- **Translation and pre-processing** — where the translation of the source code from assembly language into WSL is described. This includes a description of the assumptions which were used to make the re-engineering possible.

[4] There was not enough time to develop a translator.

- **Transformation** — a summary of the re-engineering which was performed including specific examples of the transformations which were performed.
- **Search techniques** — the data transformations could only be applied after the code had been examined to find suitable transformation possibilities. This section describes the search techniques which were used during the case study.
- **Summary** — a summary of the observations about the operation of the transformation engine and transformation techniques.

8.2.1 Translation and Pre-processing

Before any transformations could be performed the assembly code needed translation into WSL and a number of aspects of the code needed manual adjustment to make the data self-contained within the module rather than it being part of a larger system.

The translation from IBM/370 assembly language into WSL was performed using a translator which is part of the FermaT [81] transformation tool[5]. This translator was chosen because it is a mature tool which has been used to translate a large number of assembly modules as part of FermaT's use for commercial software migration activities. This translator does not transfer explicit data typing information into the WSL program (remember FermaT uses untyped WSL); section 8.2.2 shows how this information can be introduced using suitable data type equivalence theories.

Translation is an inherently informal procedure because there is no formal definition of the semantics of the IBM/370 family of processors. For this reason the translator is designed to perform a direct translation between the source language and WSL. This allows the correctness of each step to be validated by manual inspection. The output of the translator is therefore generally three or four times larger[6] than the original source code and contains many extraneous constructs (see below). These are removed during an initial transformation stage using the "`fix_assembler`" transformation. This transformation embodies heuristic knowledge about transformations which are known to be effective at simplifying the translator output. In

[5] FermaT is the commercial equivalent of the Maintainer's Assistant.

[6] This information was provided during informal discussions with the FermaT developers.

particular the transformation aims to remove assignments-to/uses-of the cpu's internal registers and to introduce the control flow constructs, e.g. conditionals/loops, which are an implicit part of the code structure.

```
                                    actions a_000000 :
                                    ...
                                    a_000146 ≡
                                      if (a[!xf address_of (var_1)] = imm_1)
                                        then cc := 0
                                      elsf (a[!xf address_of (var_1)] < imm_1)
                                          then cc := 1
000146 CLI var_1, imm_1                    else cc := 2 fi
                                ≡     call a_00014a.
00014a BE  label_1                  a_00014a ≡
                                      if (cc = 0) then call label_1 fi
                                      call a_00014e.
                                    ...
                                    label_1 ≡
                                      ....
                                    ...
                                    endactions
```

Assembly Code Raw Translated WSL

Example 8.5: Translation of IBM/370 Code into WSL

Example 8.5 shows the result of the translation of a number of typical IBM/370 instructions. These instructions are described in terms of the internal registers of the cpu and of accesses to the system's memory. In the example a compare instruction has been converted into an explicit assignment to the condition codes register, "*cc*". The value of the register is then used in the next instruction to determine if the control flow should branch to the named location.

Note that each assembly language instruction is represented as an action within an action system. Most actions are given a name which corresponds to their address in the assembler output (shown as 000146 and *a_000146* in the example). The flow of control is stated explicitly as an action call after each instruction has been executed. The `fix_assembler` transformation uses Ward's [88] transformations to transform the individual actions into conditionals and loops; for more information see section A.1.1.

The translator treats data accesses as references to an array of data values, "*a*", which represent the entire memory of the machine. It makes the assumption that these accesses are well-behaved and that a named memory location is only accessed via that name rather than by the dereferencing of an arbitrary pointer at some place within the program. Addresses may still be manipulated within the program but these accesses are assumed to refer to other memory locations than those which are explicitly named. This technique is used within this case study because it has been proven to be effective in previous transformation work [23]. The use of a more formal technique would require complex reasoning about pointers (which is beyond the scope of this thesis) and would require the implementation of a new translator.

During translation instructions which load the address of a variable into a register are converted into a format of

$$r0 := \underline{\textbf{!xf}}\ \textit{address\_of}\ (\textit{variable\_name})$$

where $r0$ is the register variable, and $\underline{\textbf{!xf}}\ \textit{address\_of}\ ()$ is a pseudo function which returns the variable's address. The `fix_assembler` transformation uses this to transform an access to

$$a[\underline{\textbf{!xf}}\ \textit{address\_of}\ (\textit{variable\_name})]$$

into a direct access to the variable "*variable_name*". The `fix_assembler` transformation will not translate any memory accesses which involve arithmetic with these addresses because it is not generally possible to determine the exact variable that these are referring to.

The assembler code has a number of instructions/operands (see table 8.2) which have no equivalent in untyped WSL. The translator represents these in the program as external functions/procedures. These do not have formally defined semantics although they may only affect the values which are stored in specified variables. In particular the external procedure may only modify the variables which are explicitly named as parameters in a call to it. An external function is not allowed to have any side-effects but may return any value. In typed WSL many of these functions/procedures would become operators in specific data type theories such as those for packed decimal, bitwise arithmetic and characters.

Instruction	**WSL Representation**	**Description**
`oi v_int, c_int`	!xf *oi* (v_{int}, c_{int})	Bitwise OR with immediate operand.
`ni v_int, c_int`	!xf *bit_and* (v_{int}, c_{int})	Bitwise AND with immediate operand.
`tm v_int, c_int`	!xf *tm* (v_{int}, c_{int})	Bitwise test under mask.
`oc v_char, c_char`	!xf *oc* (v_{char}, c_{char})	Logical OR with character constant.
`icm v_char, c_char`	!xf *icm* (v_{char}, c_{char})	Insert character under mask.
`cvb r`	!xf *cvb* (r)	Convert from packed decimal into binary.
`cvd r`	!xf *cvd* (r)	Convert from binary into packed decimal.
`ap v_dec, v_dec`	!xf *ap* (v_{dec}, v_{dec})	Add packed decimal numbers together.
`mp v_dec, v_dec`	!xf *mp* (v_{dec}, v_{dec})	Multiply packed decimal numbers together.
`sll v_dbl, c_shift`	!xf *sll* (v_{dbl}, c_{shift})	Shift left logical.
`srl v_dbl, c_shift`	!xf *srl* (v_{dbl}, c_{shift})	Shift right logical.

where v_{int} is an integer variable,
v_{char} is a character variable,
v_{dbl} is a double word variable,
v_{dec} is a packed decimal variable,
r is a CPU register,
c_{int} is an integer constant,
c_{char} is a fixed length character sequence constant and
c_{shift} is a shift value.

Table 8.2: External Functions/Procedures for IBM/370 Assembly Code

After translation and initial restructuring the program now references a number of untyped named variables. There are still accesses to the memory array which could not be resolved by the translator. Manual inspection of the code revealed that most of these involved some form of dynamic address calculation. No further attempt was made to convert these into variable accesses because these variables were not found to be necessary for the purposes of the case-study.

Each of the variables in the program has an external scope due to the fact that the case study code forms part of a larger system. WSL's semantics do not allow transformation of the representation of these external data items because that would constitute a change to the output of the program. Therefore, for the purposes of this case study any variable which is to be transformed is converted into a local variable. This local variable has a scope which extends across the whole program as shown in example 8.6. The change to the program is made by manually editing the WSL code. This operation would not be acceptable in general if the aim was to re-engineer the case study code and then to replace the original code with the new version. If the data representation had been transformed then the implementation of new and old would be inconsistent. For the purposes of this case study, however, this editing is acceptable because we are interested in showing the feasibility of transformation.

```
begin
  var <cc : any := 0,
         flags : bits := 0 >:
    begin
      actions start_ :
      start_ ≡
        ....
      ...
      endactions
    where
      proc $name_1$( var ) ≡ ....
      ...
    end
  end
where
  type bits ≡ [0 – 255].
end
```

Example 8.6: Local Variables for Transformation Evaluation

Example 8.6 shows the general structure of the code which has been translated and pre-processed. The outermost statement in the program is a **where** block which holds the type definitions which are used in the declaration of the typed variables which are to be transformed. These variables are declared in the **var** block along with the variable for the condition codes register. Inside this block is another **where** which contains an action system and a number of procedures which have been extracted by the "fix_assembler" transformation. The action system contains the actions which represent the main flow of control within the program.

Examination of the assembly output listing reveals that a number of the WSL variables are actually constants which are defined in other modules. This has not been reflected in the WSL code because the translator did not have access to the other modules or the output listing. A number of these constants are important for some of the transformations described in the next section. In particular they are often used as bit masks for masking/setting flags which are contained within integer words. These constant values were manually expanded in the source program when necessary to allow transformations to be performed.

8.2.2 Transformation

At this point the program is ready to undergo data transformation. The initial 179 pages (9390 lines) of assembly code have been converted into 3510 lines of WSL code. This code contains over 700 variables (including constants) as shown in table 8.3. The table shows that a number of variables are only assigned-to or used. These variables are exclusively used to interface with the other parts of the system.

Variable	**Number**
Assigned to	310
(but not used)	60
Used	550
(but not assigned to)	300
— of which constants	160
Total Data Items	610

Table 8.3: Variable Summary in the Case Study

Transformation of the code is based upon the integer and record types described in section 7.1. These are supplemented by the "*any*" type which is used for the untyped variables in the translated WSL code. Table 8.4 shows the type equivalence theories which were used during the case study and gives a brief description of their meaning/uses. Further details of these theories will be given at appropriate places within the following pages.

Theory Name	Description
`integer-2-integer`	Subtype integers into a new range.
`any-2-integer`	Convert a variable of type "*any*" into a variable of type "*integer*".
`any-2-member`	Move a variable of type "*any*" into a record.
`integer-2-member`	Move a variable of type "*integer*" into a record.
`integer-2-bitrec`	Convert an integer which holds a number of flags into a record of bit values.

Table 8.4: Type Equivalence Theories used during the Case Study

In particular note the `any-2-integer` theory which allows conversion of untyped variables into integer variables. This theory only allows transformation of variables which are exclusively assigned-to using integer constants/expressions. Use of the theory allows the initial untyped program to be converted into a program which uses specific data types.

The examples below demonstrate how the operation of DREAM transformations was evaluated. The first examples show how the operation of various aspects of the DREAM module were tested to confirm that they were performing in the manner described in chapter 6. The other examples show more specific applications of transformations within the program.

In general the implemented data type and type equivalence modules were developed to a stage where they were good enough to perform the individual transformations which were applied. In particular the integer type theory does not attempt to provide reasoning about every possible integer expression. The equivalence theories which involve records were hard-coded for transforming into specific fields because the prototype DREAM module/user interface does not currently allow details of specific invariants/record fields to be supplied by the user.

Testing the DREAM Module

Section 6.2.1 described the main phase of the DREAM module as consisting of four distinct operations upon the code. These were:

1. To gather assertions about the contents of the source variable;
2. To insert (ghosted) assignments to the ghost variable;

3. To perform the ghosting transformation on the uses of the source variable and

4. To check that assign-use relationships are valid.

These operations are performed in parallel during traversal of the control flow paths of the program. The module's traversal algorithm was described in section 6.3.4 and is designed to handle the following categories of statement:

- Terminal statements (which cannot be separated into component statements);
- Compound statements;
- Choice constructs and
- Recursive (and iterative) constructs.

The implementation does not handle all possible examples of these statements; it concentrates upon those which are typically found in implemented code rather than in specifications. In particular it does not handle non-deterministic choice and general expression/conditions[7]. Each of the constructs which are supported had been tested individually during its coding but in this case study combinations of these were tested to ensure that they are treated correctly when combined. The case study also aims to show that useful transformations can be performed given the limitations which are placed by the implementation.

To test the operation of the DREAM module it was desirable to test the transformations using a variable which is assigned-to/used at many places during the code. This makes it more likely that errors/limitations will be found. The condition codes variable, "*cc*", proved to be a very good choice for this purpose. It is used widely throughout the code (because the `fix_assembler` transformation was not able to remove 220 occurrences of this variable) and it has very well understood properties (i.e. the values and assigning expressions are well known).

The approach taken was to test the transformation of this variable from type *any*

[7] A general expression/condition is one which may have side effects.

to a new variable, *ncc*, of type *integer*. These tests were performed in two parts:

1. **testing the interfaces to the type and type equivalence modules** — by varying the parameters of the transformation to induce failures and check that different versions of valid/invalid transformations work as expected.

2. **testing the checking of assign-use relationships** — this involved introducing assignments-to/uses-of the ghost variable into the code prior to transformation. Checks were made to ensure that the transformation correctly reported errors in the assign-use relationships during ghosting. This test also involved validation of scope checking to ensure that scope was handled correctly.

Details of these tests are given below and the results are summarised in table 8.5 on page 212.

Type and Type Equivalence Interface Testing

This test shows that the DREAM module is correctly interfacing with the data type and type equivalence modules. The purpose is to ensure that the correct calls are made to type modules to collect and manipulate assertions and that this information is correctly passed on to the type equivalence modules. The type equivalence module uses the information to check that the production of a ghosted assignment/use is possible. Finally this assignment/use is added to the program at the appropriate position.

The integer data type module was used during this transformation. It generates assertions for the expressions shown in table 7.2 on page 153 plus the IBM/370 specific instructions shown in table 8.2 on page 204.

The type equivalence module is used to replace any assignments to the source variable (*cc* in this case), which use one of the above expressions, with the same expression (but this has type "*integer*"). This replacement is returned to the ghosting module for insertion into the program. Uses of the source variable are then replaced by uses of the ghost variable. Both assignments and uses are only carried out if the assertions show that the replacement values are representable in the target variable.

To ensure that the system was performing this transformation correctly a number of different scenarios were tried. These included:

- **range restriction of "*ncc*" to** 0 **to** 3 — this is the basic case which corresponds to the values which may be stored in the IBM/370 condition codes register. All assignments to "*cc*" are known to satisfy this transformation.

- **excessive restriction of range (i.e.** 0 **to** 2**)** — in this case the transformation (correctly) failed reporting that some values, which would be assigned to "*ncc*", were outside of its range.

- **changing the values that are assigned to "*cc*" (i.e. assign** 4**)** — in this case the program was edited, as shown in example 8.7, to include assignments which include values out of the range 0 to 3. In this case the transformation fails because the new variable cannot represent the original values.

```
if (var1 = 0)                  if (var1 = 0)
  then cc := 0         ≡         then cc := 0
   else cc := 3                   else cc := 4
fi;                            fi;

Original Program               Edited Program
```

Example 8.7: Assigning an Out-of-Range Value to *cc*

- **using a less restrictive range for the new variable (i.e.** 1 **to** 65535**)** — in this case the new variable can hold all possible condition code values and should succeed.

All of these tests operated as expected except when the erroneous statement was written within a loop which contains exits from multiple levels of nesting. This is a limitation of the implementation and is discussed later.

Assign-Use Testing

Assign-use testing was performed in a similar way but involved the introduction of assignments-to/uses-of the "*ncc*" variable into the original program. This caused failures, as described in section 5.5.3, due to interleaving of the original and ghosted values which are stored in *ncc*. Example 8.8 shows a possible situation which may occur. This is typical of the situations that were being tested.

```
if (x = y)
  then cc := 0;
            ncc := 10;
   else cc := 3 fi
if (cc = 0)
  then print(x) fi
```

Source Program

≠

```
if (x = y)
  then < ncc := 0, cc := 0 >;
            ncc := 10;
   else < ncc := 3, cc := 3 > fi
if (ncc = 0)
  then print(x) fi
```

Invalid Ghosting

Example 8.8: Assign-Use Testing

Assign-use testing also includes checks to ensure that the scope of variables is handled correctly when local variables/formal procedure parameters are defined. These tests are difficult to evaluate on assembly code because there is no use of local variables and very little use of parameters in procedures. Testing was performed, however, by introducing local variable blocks which redefined the *cc* and *ncc* variables. The transformations were checked to ensure that these were handled correctly.

These tests were carried out over a wide variety of different transformation scopes within the program as shown in table 8.5. Only one problem was found during the tests which was due to an unforeseen limitation of the DREAM module's implementation. The implementation does not correctly take account of exits from multiple levels of nesting, e.g. from an **exit**(2) statement which leaves the two innermost loops. The DREAM algorithm incorrectly propagates the assertions at this point to the end of the innermost loop rather than to the end of the outermost loop.

Once these tests had validated the tool's implementation a number of data structures were transformed to verify that the DREAM technique is capable of transforming common data structures which are found in the code.

Creating Record Structures

The raw assembly code has no concept of explicit data structuring (although the code's design does). A very useful technique was to introduce data structures into

Scope of Transformation	Result	
	Modules	Assign-Use
Compound Statements	✓	✓
Conditionals	✓	✓
Loops	✓[a]	✓[a]
Local Variable Blocks	✓	✓
Where Blocks (Subroutines)	✓	✓
Action Systems	✓	✓

[a] — Problems with exits from multiple levels of nesting.

Table 8.5: Summary of DREAM Module Testing

the program using the record data type.

Performing the transformation into static data structures involved two steps:

1. **creating a new structure with appropriate members** — the format of the new structure was found by analysing the program (as described in section 8.2.3) to determine possible groupings. This was time consuming but information was generally found in the places where data is defined. It was also found in individual chunks of code where a number of variables were assigned-to.

2. **successively transforming individual values into the record** — each variable which corresponds to a record component was transformed using the `any-2-member` and `integer-2-member` transformations. During this procedure the transformation engine checked the scope of the old and new variables to ensure that they are compatible with each other. This checking was generally not necessary in the assembly code because there are very few local variables.

The `any-2-member` and `integer-2-member` transformations allow a variable to be ghosted into a record component of the same type. The semantics of this transformation are trivial because the source and ghost data types are equivalent.

Integers Containing Flags

Analysis of the code revealed some variables which each held a number of boolean flags. These typically represented specific system states or options which had been requested by a calling routine. The values in these variables were usually set/cleared using the "*bit_and*" and "*oi*" (or-immediate) external functions. Individual values were tested using the "*tm*" (test-under-mask) external function which sets the condition codes register appropriately.

A type equivalence theory, "`integer-2-bitrec`", was developed specifically for this type of use of the integer data type. The theory converts an integer containing an eight bit value into a record which contains a separate component for each bit. This uses the invariant

$$S = (G.bit_7 \times 128) + (G.bit_6 \times 64) + (G.bit_5 \times 32) + (G.bit_4 \times 16) + \\ (G.bit_3 \times 8) + (G.bit_2 \times 4) + (G.bit_1 \times 2) + G.bit_0$$

where S is the source variable and

$G.bit_n$ is the component which is used to store the target bit.

This is used to describe the relationship between the source and ghost. The theory behind this is similar to that for the example which is given in section 7.2.3 although in this case the ghost type has eight components. Each component in it represents a single bit in the byte which is manipulated by the bitwise operators.

Table 8.6 shows the assignments/expressions which may be produced by this equivalence theory when a source assignment/use needs ghosting. Note that the *bit_and* and *oi* assignments may involve constants which have more that one bit set. In this case the ghost assignment will involve parallel assignment to each ghost variable whose corresponding bit is being set/cleared. This is not shown in the table for reasons of clarity.

The type equivalence theory was applied to the program using ghosting to transform the flags into appropriate record structures. Example 8.9 shows the results of ghosting a typical example from the code. In the ghosted code the bitwise operations have been replaced by direct assignments to the ghost variables. Note that the test-under-mask comparison is still present in the code. This is because the result of the ghosting has not yet been simplified. In this case simplification of the

Source Construct	**Ghost Equivalent**	**Description**
Assignments		
$S :=$ **!xf** $oi\ (S, C)$	$G.bit_n := 1$	Convert a bit set into an assignment of 1 into the appropriate bit(s) specified by the constant.
$S :=$ **!xf** $bit\_and\ (S, (255 - C))$	$G.bit_n := 0$	Convert a bit clear into an assignment of 0 into the appropriate bit(s) specified by the constant.
$S := 0$	$G.bit_0 := 0, \ldots, G.bit_7 := 0$	Clear all bits/flags.
Uses		
!xf $tm\ (S, C)$	**!xf** $tm\ (G.bit_n \times (2 * *n), C)$	Convert a test of the whole variable into a test of only the bits mentioned in the constant.
S	$G.bit_0 + (G.bit_1 \times 2) + \cdots + (G.bit_7 \times 128)$	A general use of the source becomes the right-hand side of the invariant.

Preconditions for each equivalence	
Source Variable (S)	**range_includes**(**range_of**(S), **make_range**(0, 255))
Ghost Variables ($G.bit_n$)	**range_includes**(**type_range**($G.bit_n$), **make_range**(0, 1))
Constant (C)	**range_includes**(**range_of**(C), **make_range**(0, 255))

where S is the source variable,

$G.bit_n$ is a component of the bit record and

C is an eight bit integer constant.

Table 8.6: Operations on Integers which Represent Boolean Flags

comparison

$$\underline{\textbf{!xf}}\ tm\ \ ((G.bit_1 \times 2) + G.bit_0, 2) \neq 3$$

results in the new comparison

$$G.bit_1 \neq 1$$

The prototype implementation of this theory is limited to working with hard-coded component names. This limitation can be removed in the future but is not important for the purposes of this case-study.

This use of the integer data type occurs a number of times within the program. Three of these were transformed using the tool and the others were analysed by hand to ensure that they did not have any different assignments/uses which would cause the ghosting to fail. In a number of these other cases the flag representation differed. Some flags were contained in multiple bits to allow multiple flag values (as opposed to boolean values). In one case the top four bits of the value were used to hold a "transaction type" code and the lower four bits were encoded as boolean flags. A similar equivalence theory could be constructed to allow this type of variable usage to be ghosted — the top four bits would be placed in one variable rather than in four separate variables.

Logical Subtyping

During analysis of the code it was noted that a number of variables were used for similar purposes within the program. These were candidates for logical subtyping where the type of the variables is changed to an equivalent subtype which has a different name. This does not cause a change in representation but serves to make the variable's purpose explicit.

The `integer-2-integer` and `any-2-any` equivalence theories were used to ghost these variables into a new subtype. This subtype has the same properties as the source variable's type. This means that the equivalence theories are always able to generate ghost assignments/uses. The transformation could, however, fail due to invalid assign-use relationships within the scope of the ghosting.

```
begin
  var <cc : any := 0, S : bits := 0,
         G : bitarray := [0, 0] >:
    S := !xf oi (S, 1);
    if ((!xf tm (S, 2)) = 3)
      then S := !xf bit_and (S, (255 − 2)) fi;
    if (S = 0)
      then cc := 0
       else cc := 1 fi
  end
where
  type bits ≡ [0 − 255].
  type bitarray ≡ record(bit1 : bit,
                                   bit2 : bit).
  type bit ≡ [0 − 1].
end
```

Source Program

```
begin
  var <cc : any := 0, S : bits := 0,
         G : bitarray := [0, 0] >:
    G.bit1 := 1;
    if ((!xf tm (((G.bit2 × 2) + G.bit1), 2)) = 3)
      then G.bit1 := 0 fi;
    if (((G.bit2 × 2) + G.bit1) = 0)
      then cc := 0
       else cc := 1 fi
  end
where
  type bits ≡ [0 − 255].
  type bitarray ≡ record(bit1 : bit,
                                   bit2 : bit).
  type bit ≡ [0 − 1].
end
```

Transformed Program

where S is the source variable and

G is the ghost variable.

Example 8.9: Ghosting Flag Variables

Invalid assign-use relationships did occur on a number of occasions because the source variable was not assigned to before it was used in a particular action[8]. The assignment to the variable was typically in another action and was followed by a call to the first action. In this scenario the assign-use relationship is actually valid but is not detected by the DREAM transformation implementation. Example 8.10 shows a typical case where this occurs. The variable "a" is used in action "x" which is called by action "y" after an assignment to "a".

```
actions y :
x ≡                          actions y :
  print(a).                  y ≡
y ≡                    ≡       a := 10; print(a).
  a := 10; call x.           endactions
endactions
```

Example 8.10: An Undetected Valid Assign-Use Relationship in an Action System.

The implementation of the transformation engine restricts the checking of assign-use relationships to sequences of statements which cannot exit prematurely due to calls to other routines or loop exits (see section 6.3.4). In an action system this means that each action, i.e. x or y in the example, must have correct assign-use relationships. In addition to this an action which contains a call to another action must have correct assign-use relationships in the sequences of statements at either side of the action. This is because the call causes a transfer of control flow outside of the action.

In the example shown it is possible to allow the ghosting to succeed by replacing the call to action x with the code in that action and then by deleting the action. This is done using the "`expand_action_call`" control flow transformation followed by use of the "`delete-unused-action`" transformation. The process is complicated if action x is called from many other places within the action system because these calls to it must be expanded before the action can be removed.

An alternative solution to the one shown above is to ensure that the variable, a, has a valid value at entry to the action system. When this is the case all assign-use

[8] Remember that an action is contained within an action system and is used to represent goto statements.

relationships within the action system will be valid. In this case study an assignment to the source variable can be added before entry to the action system if the variable is actually providing input from another module. This cannot be done automatically because it involves manual examination of the assembler source code to ensure that this is the case.

While performing this case study these problems were encountered a number of times. Similar problems were also encountered with loops and procedure calls. The problems were not limited to only this type of transformation but its occurrence is not mentioned in the other examples to make their explanations clearer.

8.2.3 Analysing the Program

The transformations which are described above would not have been possible without considerable effort being placed into analysis of the code. This analysis helped to identify suitable transformation options and to verify that the transformation system had correctly performed the transformations.

During analysis of the code four main sources of information were used:

1. **assembler source code listing** — this provided a reference against which to compare both the pre-processed WSL version of the code and the transformed data. The comments in the source code were very useful because they provided information about the meaning of individual variables and constants.

2. **assembler output listing** — the assembler output provided a slightly different view of the source code. In particular it contained cross reference information which related assembler labels (variables and constants) to the lines on which they are used. The listing also showed the values of constants which was very useful for identifying the variables which represented bit flags.

3. **pre-processed WSL code** — the pre-processed WSL code was useful because the control flow was much more structured. This made visual inspection of the code much easier because the uses of particular variables could be traced more easily than in the equivalent assembler code.

4. **IBM/370 texts** — a number of text books which describe the IBM/370 series machine were very useful. They provided help in understanding the effects of

specific instructions. The books which were consulted include: Chapin [31], Tuggle [86] and Yarmish [96].

This information was examined manually by inspecting printed and electronic copies of it. Examination of the electronic information was made easier by the use of search utilities which are found as standard on Unix systems, e.g. grep. The XEmacs editor was also very useful because it allowed the search utilities to be used in conjunction with an editor to visit the areas highlighted during searches.

The MetaWSL statements provided by the transformation engine were also very useful during the analysis of the program because they allowed information to be retrieved from the program very quickly. In particular the `[_Variables_]`, `[_Used_]` and `[_Assigned_]` family of commands allowed variable usage information to be extracted from the program. These commands return lists of variables which are present, used or assigned-to within a specified area of the code — further information about these can be found in Bull's thesis [23].

The names of variables provided clues about the meaning of a variable or about the variables which were related to it. Most variables had a short (less than eight characters) mnemonic name which was not meaningful in itself but by examination of the program's comments the purpose of the variable could usually be determined. Many variables which had similar meanings had similar names. For example, the flag variables which are described on page 212 usually had a name similar to "sysflg" where "sys" is the meaning of the variable and "flg" identifies that it is a flag. The constants which are related to this flag would typically have a name which began with "sys". This was then followed by four or five characters which identified the meaning of the flag.

Records could often be identified in a similar manner. The first part of the name would reflect the name of the record instance and the latter part would reflect the particular field that the variable represented.

This use of variable naming conventions during program analysis is very program specific. In this program the developers/maintainers have been very disciplined in the use of consistent names. The search techniques which were employed in this case study may not be as effective on code which is developed by less disciplined engineers.

Another technique which proved useful was a search for all uses of a particular language construct. Many instances of the same variable usage involved assignments and expressions which used similar constructs. A search for all "test-under-mask" expressions would often find many of the uses of the flag concept. Variable subtyping was also highlighted in this way.

8.2.4 Transformation Summary

In addition to validating the operation of the transformation tool in a well understood environment (the uses of the *cc* variable) the transformation of different features of the data has been examined. Table 8.7 summarises the transformations which have been performed. The main emphasis of this transformation has been on the integer and record data types. It is noted that there was very little use of integer arithmetic in the code except for logical bitwise operations and the manipulation of pointers. This reflects the fact that the code forms part of an information management system rather than a scientific computation program. Many of the uses of integers in the code were to hold values passed from other parts of the system. It was not generally possible to restrict the ranges of these variables' types because the range of input from the other parts of the system could not be determined.

Introduction of records and logical subtypes was easier to perform, however, because these do not affect the values stored within the variables. They do, however, make the re-engineered code easier to understand because items which are related to each other are explicitly marked. This makes the grouping information easily available to future maintainers.

Successes

- The transformations can restrict the range of a variable's type to reflect the values which are actually stored within it. This has been shown by restriction of the type of the condition codes register.

- Monolithic assembler data structures can also be re-engineered to produce structures which represent the logical entities present within the program. This does, however, require a considerable degree of user intervention to search for

Transformation	Number		Description
	Identified	Transformed	
Flag Variables	31	3	Converting from a bitwise representation of flags into a record of bit values.
Records	13 × 6 entries 14 × 4 entries	8	Transforming individual variables into records which contain groups of variables.
Subtypes	15 groups	4	Introducing logical subtypes which reflect the variable's use within the program.
Condition Codes	unknown	unknown	Evaluating the operation of the transformation module.

Table 8.7: Case Study Two — Summary

suitable groupings.

- Addition of new type equivalence theories was found to be easy although the theories used in the case study are very primitive and can only cope with a limited number of cases. In particular the `integer-2-bitrec` theory must be edited to allow use of different component names for the bit record.

- It was easy to extend the type theories to handle new expressions which were not originally covered. In particular the properties of the IBM/370 bit manipulation operators were added easily.

Deficiencies

- The main prerequisite for DREAM transformations is knowledge about the inputs to the code that is being re-engineered. At times it proved difficult to re-engineer something towards the end of a routine because the variables which are assigned before it needed re-engineering first.

- Action systems also proved to be problematic, especially when there were large numbers of them which could call each other. This made assign-use relation checking fail unnecessarily because of restrictions about where valid assign-use relations could occur. Similar problems also occurred for loops and procedures.

- Procedure parameters which alias other parameters proved to be a slight problem. This can be worked around in many cases by expanding the procedure call using its definition. Note, however, that these problems were not encountered in the assembly code because procedure parameters are not normally used in this code.
- The internal representation of the program which is being transformed does not represent the flow of data values very well. It was originally designed for control flow transformations and needs extending to allow more efficient data transformation. Possible ways of doing this will be discussed in section 9.2.
- The time taken for transformation application increases considerably as the size of the program increases. This is in part due to the transformation engine's internal program representation and due to inefficiencies in the prototype tool.

The analysis of the results of this automated data transformation case study show that DREAM is capable of performing constructive re-engineering although a number of aspects require further work.

8.2.5 Case Study Summary

This case study has evaluated the use of the prototype DREAM data transformation tool with the aim of showing that the implementation performs correctly upon practical code and that data re-engineering can be performed using these transformations.

The operation of the data transformations was verified by repeated transformation of the condition code variable, "*cc*", under differing circumstances. These tests showed positive results although one situation was identified where the implementation failed to handle exits from multiple levels of loop nesting.

The data in the code was re-engineered using six basic type equivalence theories which allowed data types to be introduced; variables to be grouped into logical subtypes; data structures to be introduced and allowed variables which contain a number of flags to be converted into a record containing individual variables for each flag.

The main aim of the case study was to highlight the good and bad features of the automated application of DREAM transformations. These cover different aspects of the technique and range from the ability to perform useful restructuring through to the efficiency of the transformation engine.

8.3 Criteria for Success Revisited

At this point in the thesis we have presented the DREAM technique for performing data transformation and have presented the results of two case studies which have examined the use of DREAM for re-engineering the data which is present in commercial software. In addition to this chapter 7 introduced a number of data types and showed how they may be represented and transformed.

This section analyses the results and presents answers to the questions which were posed in chapter 1 regarding the criteria for success.

8.3.1 Data Types in WSL

Typed WSL has been defined to allow the explicit representation of data typing within a program that is being transformed. It is important that this language allows the representation of a wide variety of data types and that the typed WSL constructs do not have a detrimental effect upon Ward's [88] original control flow transformations.

Can typed WSL represent a full range of data types which may be encountered in common programming languages?

The ability to represent a full range of common data types is an important factor which governs the suitability of typed WSL for data transformation. If the language cannot represent a data type then it is not possible for that type to be transformed using DREAM. Chapter 7 examined the representation of data types within the language. In particular, the integer and record data types were examined in detail to show the steps involved in adding specific types to the language. In addition to this the chapter also described how other types could be represented in typed WSL. These included examples from each of the four data type categories (elementary, composite, structural and dynamic) which were identified in chapter 4.

The main feature of data values in typed WSL is that they are indivisible entities which cannot be decomposed by WSL transformations. This isolates control flow from data semantics and, therefore, allows data types to be developed in isolation from the semantics of WSL. Unfortunately, this also means that the language cannot represent data types which require any aspect of control flow to describe them. In particular, this means that first-class functions (those which can be passed as data values) cannot be represented in typed WSL. For similar reasons it is difficult to represent data types which allow inheritance of methods (subroutines).

Data types whose operators may raise exceptions, e.g. those in the Ada [2] programming language, are also precluded from representation in typed WSL. When an exception is raised the program transfers its control flow to a suitable error handler which performs an appropriate action. This behaviour could be captured by adding checks around each use of an operator but these cannot be transformed in conjunction with the data representation.

Two other data types: c-unions and c-pointers have not been represented in typed WSL. This is not due to limitations in the typed WSL data model but is due to the complex semantics of these data types.

In summary, typed WSL can represent most common data types although those which involve interaction between control flow and data values cannot be represented.

Is the set of data types that may be represented in typed WSL extensible?

A key requirement for the successful transformation of a number of different programming languages is the ability to extend the range of data types which may be represented in typed WSL. This allows representation of the specific semantics of a data type that is used in a particular language/machine.

Typed WSL allows data types to be added without requiring any changes to the transformation theory/engine. This is made possible by the use of a shallow embedding of data type semantics within WSL. It allows a data type to be introduced as a set of axioms which describe the properties of the data. These axioms do not conflict with the semantics of WSL and, therefore, cannot affect the semantics of control flow transformations.

Addition of these data types into the transformation engine is correspondingly simple. The well-defined interface to data type modules means that a new data type module need only provide routines which implement these interfaces. Once this has been performed the data types may be used within typed WSL programs.

Does typed WSL have a detrimental effect upon Ward's transformations?

Ward's transformations [88] are the original untyped WSL transformations which allow control flow restructuring. The Maintainer's Assistant was developed to provide automated support for these transformations. One of the aims of this thesis is to allow data transformations to be performed in the same environment as control flow transformations.

Typed WSL has a minimal effect upon the semantics of the original transformations because it is defined in terms of the original WSL language. This means that every typed WSL program maps onto a program which is composed entirely of untyped WSL constructs. Any transformation which requires expression simplification is, however, affected because this is now performed using the simplification rules of specific data types rather than using the general rules which were originally used. This does, however, turn out to be beneficial because the use of specific data type properties allows the simplification of features which are specific to one particular data type.

The typed WSL semantic extensions also allow composite types to be transformed as both individual variables and as a whole group. This adds to the capabilities of the control flow transformations because it is now possible to transform and simplify individual components, when appropriate, as well as transforming the whole structure as if it were an abstract data type.

The effect of these language changes upon the transformation engine (the Maintainer's Assistant) were examined in chapter 6. This showed that the extensions to the language do affect the implementation of transformations in a number of ways. While these changes are conceptually simple they have a significant impact due to the amount of code which must be modified and re-verified. These changes have been made successfully and the updated control flow transformations have been used during case study work.

8.3.2 Data Transformations using DREAM

DREAM provides a transformation mechanism which allows program data to be re-engineered. These questions cover the direct use of the mechanism without reference to the possible applications of the transformations.

Is data transformation possible for a full range of data types?

Earlier we demonstrated that a complete range of data types can be represented within typed WSL. It is desirable to be able to transform variables which are instances of these data types during re-engineering. Chapter 7 examined this issue by concentrating upon the transformation of both integer and record data types. The chapter demonstrated how the proof of these transformations is performed within the DREAM environment.

The chapter also examined some transformations which may be performed on other data types in typed WSL. The proof of these other transformations was not examined in detail but similar techniques to those used for integers and records are needed. In general, the transformation is based around an invariant which describes the relationship between source and ghost representations. This is then used to produce the ghosted assignments/expressions and to prove that these are a correct transformation of the data.

How easy is it to add new data transformation theories?

Transformation theories are represented in the transformation engine as type equivalence modules which encapsulate the theory in two routines which generate new, equivalent (or refined) versions of the uses-of/assignments-to the source variable. The new versions are produced by an appropriate, often heuristic, method but ghosted uses of the source variable must be shown to satisfy the data expression equivalence relation presented in section 5.3.

Addition of a new data transformation theory is separated into two parts:

1. **Developing the theory** — using the theories of source and target types to prove the desired relationship between the two. This is a highly skilled task requiring knowledge of both the data types and proof techniques.

   It is possible to develop a new type equivalence theory without excessive formal proof. In these cases the theory developer should take care to ensure that the relationship between source and ghost types is reasonably simple. This makes it possible to validate the relationship by inspection and informal argument.

2. **Implementing type equivalence modules** — equivalence modules turn the theory into a form which is usable within the transformation system. This stage requires knowledge of the intended use of the transformations and a number of different type equivalence modules may be implemented for one theory. These would provide transformations which use different aspects of the equivalence theory. Some may provide support for only the simple cases of a particular theory.

Once the modules have been created it is a simple task to load the modules into the transformation engine (in the same way that type theories are loaded). Thus, development of new transformation theories is a task which must be undertaken by a person with suitable skills. However, once the theories and equivalence modules have been developed it is easy to load the module into the transformation engine to allow the transformations to be applied.

Do data transformations rely upon control flow transformations?

The use of ghosting and the data expression refinement relation divorces data transformation away from control flow transformation. It allows changes in data representation to be described without reference to the semantics of WSL. This simplifies the proof of both the ghosting transformation and data changes. It also provides an opportunity to reuse existing data type theories which have been developed by other members of the formal methods community.

Application of data transformations does, however, rely upon the control flow of the program to identify the flow of data through the program. This makes it possible to determine which values may be held in a variable at a particular point in time. This information is then used to demonstrate the validity of the expression refinement relation before the ghosting transformation can be applied.

During the case studies a number of problems were encountered due to these control flow dependencies. The transformation engine is not capable of reasoning fully about the control flow of constructs which cause changes to the sequential flow of instruction execution. If assignments-to/uses-of data are within these constructs then the tool may not be able to analyse the program properly and will not, therefore, be able to perform the ghosting transformation. The solution to this is to restructure the program, using control flow transformations, into a form which can be handled by the ghosting transformation.

This highlights an important feature of DREAM because control flow and data transformations can be freely mixed within a re-engineering session without any change of transformation environment.

In summary, the theory of data transformations does not rely upon control flow transformations but the application of transformations does involve some use of control flow transformations. This makes it possible to perform automated reasoning about the validity of a ghosting transformation in specific cases.

8.3.3 Data Re-engineering using Formal Transformations

DREAM data transformations have been developed for use during re-engineering. It is important that they allow this to be performed.

Can DREAM perform all of the classes of transformation which are necessary for data re-engineering?

Chapter 3 identified refinement, abstraction and restructuring as the three types of change which may be made to a program during re-engineering. These describe the effect that a transformation has upon the representation of data in terms of its relationship with the implementation of the program.

In chapter 7 we examined a number of different transformations upon data types. The chapter showed examples of refinement, abstraction and restructuring for a number of common data types. This demonstrates that it is indeed possible to perform each of these three activities using DREAM. No attempt was made to show that every possible combination of data type/re-engineering operation can be performed. It is, however, reasonable to conclude that other combinations are possible because similar techniques for describing the mappings between source and ghost data types have been used throughout chapter 7.

Transformation Operation	**Description**
Refinement	A boolean error code is converted into an integer which contains a reason for failure.
Abstraction	An integer is converted into an array of flags.
Restructuring	Change the representation of an integer, e.g. 00, ... , 99 is transformed to 1900, ... , 1999.
Relationship between Objects	A time value is converted into hours and minutes (and vice-versa).
Scope of Data	Scope has been manipulated as part of the DREAM implementation verification.
Introducing Subtypes	The IBM/370 condition codes variable has been subtyped to a range which is suitable for its possible values.

Table 8.8: Re-engineering of Integer Variables

In addition to these transformations, which change the representation of data, chapter 5 identified another three types of data transformation. These capture changes to the relationship between program code and data. The implementation of the ghosting algorithm has been specifically designed to allow these transformations to be performed in conjunction with changes to the data representation. Table 8.8 shows examples of each transformation type which have been performed on integer variables during the case studies.

This shows that it is possible to perform examples of each data transformation category. It does not guarantee that each of these will be applicable for every data type but chapter 7 showed that similar operations could be performed on other data types. It is therefore reasonable to conclude that DREAM can perform many of the transformations which are necessary for the re-engineering of data.

Does DREAM scale to practical re-engineering tasks?

The data type theories presented in chapter 7 examined data transformation in small, controlled environments. The case studies presented in sections 8.1 and 8.2 have taken the same basic data type theories and applied them to medium-sized examples of code which were taken from a large, commercial software product.

The results of the case studies show that the data transformations can perform similar re-engineering operations to those shown for small scale code. The case studies did require some additions to the type equivalence theories to allow extra operators which are present in the case study code to be handled. This highlights the need for more research into the basic data type and type equivalence theories but does not represent a limitation of the work presented within the thesis.

One problem with the use of DREAM transformations was the difficulties which arise when trying to determine the flow of data within a program. Ghosting requires that whenever a change in representation is made all of the possible sources of assignment to the variable which is being ghosted must have been examined. Constructs with complicated control flow semantics, i.e. loops, procedures and action systems, present a challenge when trying to perform this. The current implementation adopts a very simplistic view of these and performs very little analysis of the program. With more development the transformation could be made to recognise a number of common scenarios and, therefore, require less use of control flow transformation prior to data transformation.

Despite these limiting factors in the current implementation DREAM does allow practical data re-engineering to be performed.

Does DREAM complement existing software maintenance activities?

Chapter 2 identified three aspects of software maintenance: management, process and technical. These are all essential for successful program re-engineering. This thesis has examined the technical aspects of data re-engineering using formal transformations. It concentrates upon the primitive transformation operations that are necessary to bring about changes to data representation.

Code migration, reverse engineering and program understanding were all identified as different forms of re-engineering. Previous research, e.g. Ward et al. [91, 43], has used formal transformations as an aid to these activities. DREAM extends their work by allowing data to be manipulated in a similar manner.

The case studies have highlighted the need for the development of compound data transformations which combine primitive ghosting operations to perform the changes that the maintainer may want to make. For instance, the maintainer may want to change the type of a variable and does not want to have to deal with the introduction of ghost variables into the program.

DREAM does require some external assistance to identify suitable transformations to apply in particular situations. The case studies confirmed this although it may be possible to automate a substantial amount of re-engineering in well known situations such as initial transformation of assembly code.

In summary, we see that DREAM does complement software maintenance activities although further work is required to put DREAM into a form which provides the transformations required for these.

8.4 Summary

This chapter has examined the practical application of DREAM transformations and has used the results of this to examine the criteria for success which was proposed in chapter 1.

The practical application of DREAM was examined in two case studies:

1. The first examined the overall use of DREAM in a full reverse engineering situation. It examined the use of data transformations to recover a high level description of the program and its data. The results of the case study showed that this is possible using DREAM.

2. The second case study examined the practical application of data transformations using the extended transformation tool. This showed that the data type theories presented in chapter 7 do scale to practical re-engineering although program control flow complexity does make application of the transformations difficult.

The criteria for success examined the achievements of the thesis. It took into account the results from this chapter and the work presented in previous chapters.

The next chapter takes these results and draws conclusions about the success of the thesis.

Chapter 9

Conclusions

This thesis has focussed upon the maintenance of computer software specialising in the problem of re-engineering program data. DREAM has been developed to integrate theories of data type equivalence into a formal program transformation environment. DREAM allows variables to be replaced by others which differ from the original in one or more aspects while still ensuring that the program produces the same output.

WSL, the transformation language, has been extended to form "typed WSL". This is achieved by the use of a shallow semantic embedding which allows the semantics of the data type to be separated from those of WSL. A data expression refinement relation is defined which allows the description of the relationship between source and ghost expressions. This relates the values produced by both expressions, under given input conditions, and is used to show that the ghost expression is semantically equivalent-to or a refinement-of the source expression.

The transformation theory is put into practice by extensions to the Maintainer's Assistant. These provide a modular approach to the implementation of data transformations. The DREAM module provides a generic data transformation which uses the semantic knowledge provided by type and type equivalence theories. This allows new data types and transformation theories to be added with minimal effort (apart from proving them!). Heuristic knowledge can be built into the implementation of the theories to provide guidance in the selection of appropriate transformation activities. This reduces the need for user intervention. The style of DREAM data transformation complements existing control flow transformations by taking care of the small details while allowing the maintainer to concentrate upon the tactical issues of re-engineering.

Chapter 8 evaluated the use of the DREAM data transformation technique in a re-engineering environment and examined the criteria for success. Table 9.1 summarises the answers to this which were judged against the results of the case studies and the work presented earlier in the thesis. In the remainder of this chapter conclusions are drawn based upon these results and ideas for further work are discussed.

9.1 Meeting the Criteria

The most important question to consider when judging the success of the thesis is whether it has achieved the aims of allowing "data re-engineering using formal transformations". The criteria for success has judged this by examining the results of case studies. These were performed with the aim of showing that DREAM data transformations will scale to use on medium sized programs. The results show that the transformations do allow useful re-engineering to be performed although there are problems due to the difficulties involved in analysing complex control flow. This analysis is required to demonstrate the validity of data expression refinement relations.

The criteria for success shows that the WSL language extensions have proved remarkably successful. They allow a wide variety of types to be added to the language without the difficulty involved in reproving the correctness of existing WSL transformations. This adds a great deal of flexibility into the data transformation environment and provides a separation between the theories of WSL and individual data types. This separation makes it possible for theories about data types to be developed by individuals who know very little about the program transformation domain.

If the semantics of data types had been specified directly in infinitary logic[1] the introduction of individual data types would have been more difficult. It would have been necessary to specify each data type in terms of infinitary logic rather than importing a data type from another semantic environment. Use of infinitary logic

[1] Infinitary logic is used to describe the semantics of WSL.

Criteria	**Summary**
Can typed WSL represent a full range of data types which may be encountered in common programming languages?	Most common data types can be represented but those which involve use of control flow semantics cannot be represented.
Is the set of data types that may be represented in typed WSL extensible?	New data types can be added into the transformation engine easily provided that suitable data type theories are available.
Does typed WSL have a detrimental effect upon Ward's transformations?	Typed WSL does not affect Ward's transformations and allows extra reasoning about specific data types' semantics.
Is data transformation possible for a full range of data types?	Transformation of integer and record data types has been examined in detail. Other data types have been examined to demonstrate that transformation is possible.
How easy is it to add new data transformation theories?	New transformation theories must be properly developed but adding them into the transformation system is simple.
Do data transformations rely upon control flow transformations?	Control flow transformations are necessary to restructure complex control flow. This allows ghosting to be performed.
Can DREAM perform all of the classes of transformation which are necessary for re-engineering?	Data refinement, abstraction and restructuring transformations have been demonstrated for WSL data types.
Does DREAM scale to practical re-engineering tasks?	The case studies have shown that DREAM can be successfully applied in practical re-engineering situations.
Does DREAM complement existing software maintenance activities?	DREAM provides an underlying mechanism by which software maintenance techniques can make changes to program data.

Table 9.1: Summary of the Criteria for Success

would, however, have made it possible to reason about both data types and control flow simultaneously.

It is unfortunate that more time was not available for development (or importation) of theories for a wider variety of data types. This is obviously beneficial for performing complete data re-engineering tasks but is not justified given that the aim of this thesis was to investigate "the integration of data type equivalence and refinement relationships into existing theories for the specification of program control flow behaviour". The main contribution provided to this is the theoretical link between the two rather than the individual data type theories themselves.

The data type examples and case studies have shown that DREAM can be used to perform data refinement, abstraction and restructuring transformations. These transformations can be performed upon types from each of the four type categories which were identified in chapter 4. The reasoning which is required in order to show the validity of data transformations is likely to become more complex when transformation of dynamic data types is studied in depth.

The use of a static data type binding mechanism was one of the major decisions in chapter 4. This makes it possible to determine the type of a specific variable with minimal analysis of the program. A dynamic binding would have required analysis which is much more complex for even the simplest of data transformations.

The ease with which data typing was introduced into WSL reflects upon the actual power of the language and its suitability for use in formal environments. The Maintainer's Assistant provides an excellent base upon which to implement DREAM and data typing. Most of the infrastructure required for data transformation was already present within the system and new features were often just implemented as extra aspects of the original support routines. This does not mean that the prototype implementation was without its problems. Often subtle assumptions in the original implementation would cause considerable difficulty when adding new features.

One of the biggest problems which has still not been resolved is the difficulty in finding the previous/next accesses to a variable. The current implementation of DREAM requires that the control flow of the program is followed to find the appropriate points in the program where the variable is accessed. This cannot be fully solved by keeping track of the values in the source and target variables because the values assigned to these may be created from other variables whose value is not known. Subtyping does help to alleviate some of these problems but does not provide all of the information about specific values when necessary.

This problem is caused by the control flow centred internal representation of a program which is used by the Maintainer's Assistant. At times it seemed that it would have been better to abandon the Maintainer's Assistant framework in favour of a new one specifically targeted at data transformation. This would have made the implementation of data transformations easier but would have removed the support for control flow transformation which has proved to be essential in the evaluation of the work. The integration of control flow and data transformations is also one of the prime aims of the thesis and a division in the practical aspects of control and data transformation would not have been productive.

The use of the ghosting technique to provide a basis for data transformation has made proof of the transformation theories much easier. It concentrates the proof upon the essential change to the program while still allowing more complex changes to be made. These changes are built up using repeated application of individual ghosting transformations. The result of this is flexibility in transformation at the user's level — as well as performing restructuring the transformation can also allow scope to be changed and a variable's logical uses to be separated. On the downside the changes to the program are composed of a number of individual data transformations and detailed analysis of the groupings which can be performed has not been carried out (see "further work" below).

In summary, this thesis has presented and demonstrated the use of a powerful addition to the WSL transformation environment. It allows data to be transformed in a similar way to control flow thus extending the range of transformations significantly. The thesis has not examined every possible category of data type and there are a number of limiting factors in the implementation of DREAM data transformations. These are all aspects which can be addressed in future research.

9.2 Further Work

The possible directions for further work which are presented below are split into three areas: data types and equivalence theories; transformation groupings and the transformation engine. The possible research for each area is examined with reference to other relevant work where appropriate.

Data Types and Equivalence Theories

It is desirable to introduce a number of new data equivalence theories to allow transformation of many extra data types. There are a number of ways to pursue this area of research. The most basic is to develop new theories specifically for WSL taking existing research into account and identifying the most important aspects for re-engineering transformation work.

One possibility would be to develop theories for transformation of dynamic data structures (e.g. using pointers or an equivalent mechanism) by modelling the state space of the dynamic structure. This state space would be wholly contained (at least theoretically) within the value stored in a WSL variable. Assignments to these variables would consist of an entire description of the state space. This description would most likely be produced by modifying an input to the expression in a well defined manner. This modification is very similar to the concept of the "schema" in Z. Re-engineering of these structures would involve recognising invariants in a variable's values, i.e. that the pointers are always manipulated in a way which maintains a linked list. Recent work by Möller [70] provides a suitable starting point for this research.

Instead of developing new theories for DREAM transformations another approach would be to reuse the work which has been performed by a number of groups around the world. The theorem proving community has developed a number of theories for individual data types such as integers, real numbers and recursive structures. This work on the HOL [44, 50], PVS [30] and other theorem provers could be integrated directly into the transformation engine to provide access to suitable theories. Work by Pratten [75] is currently investigating methods for providing common interfaces between theorem provers and systems which make use of their theories.

A third approach is to utilise and adapt the data engineering facilities provided in many of the formal program development methods, e.g. the B Method [61], Z [74, 52] and RAISE [47]. This would provide access to a large number of theories for high level and abstract data types. DREAM transformations could be used to help to recognise individual instances of these and to re-engineer the legacy code into suitable representations which could then be re-implemented using an appropriate formal development method.

Another vein of research would be to investigate the integration of control flow and data type semantics. This could be done by extensions to the control flow semantics which allow data types to have specific effects upon control flow, e.g. exception handling. An exception could be handled in a similar way to a loop **exit** statement where control flow automatically jumps to the end of a block. It may be possible to represent this extension using a similar method to that used to add composite types into WSL (see section 4.2.2). This would allow the data type to represent a small aspect of control flow (the exception mechanism) while still allowing continued use of a shallow semantic embedding.

Transformation Groupings

DREAM data transformations perform low level changes to the structure of a program's data. The transformation captures the central part of the manipulation, e.g. changing the representation, but the maintainer is expected to perform any extra operations. For example, if the maintainer wishes to change the type of a variable a new variable must be introduced into the program; DREAM is then used to change the representation of the data by ghosting the variable into the new variable; as a final stage the original variable is removed and the new variable is renamed to have the same name as the original. Further research could investigate these typical operations and develop "super-transformations" which provide complete restructuring operations. This is a similar approach to the idea of "compound transformations" which was used by Bull [23] in the development of transformations which perform a large amount of initial restructuring of code which has just been translated from other languages.

Transformation Engine

The transformation engine requires further work to provide a more efficient method for tracing accesses to individual variables. This would make it easier to find the expressions which are used to produce the values which are assigned to the source and ghost variables. The current implementation makes this operation very difficult to perform and in most cases the entire range (the worst case scenario) of the input variable's value is used rather than values which are specific to individual cases.

$\mathcal{M}\mathcal{ETA}$WSL could be extended to provide new constructs which find the appropriate ranges/values and, therefore, make description of individual transformations much easier. This alone would not make the transformations any more efficient because the $\mathcal{M}\mathcal{ETA}$WSL constructs would only perform the same actions as would be performed in the transformation body. A suitable approach for research would be to adapt the transformation engine's internal abstract data representation to introduce links from one use/assignment to all of the possible subsequent/previous ones which could be affected-by/affect that use/assignment. This could involve significant additional overheads in the memory used by the transformation engine. Ideas presented by Baxter [9] and the techniques used in compiler code optimisation algorithms may be useful to help alleviate these problems.

Appendix A

A Review of the Maintainer's Assistant

This appendix presents a summary of the accomplishments and deficiencies of the Maintainer's Assistant tool and theories. These have been identified from practical experience and from comparison with other systems and tools.

A.1 Accomplishments

The development of the Maintainer's Assistant has involved the development of a number of solutions to key problems. Many of these problems cause considerable difficulty in the re-engineering process and the solutions discussed in the following pages represent a significant advance in the available technology.

A.1.1 Code Restructuring

An important aspect of the Maintainer's Assistant is its code restructuring capabilities. **Code restructuring** involves the change of code structure into another form which satisfies some criteria as defined by the maintainer. This could involve abstraction of code to make it resemble the real world objects that it represents or a change from an iterative to a recursive algorithm.

To-date the Maintainer's Assistant has been mainly used for restructuring code written in low level programming languages such as assembly languages. Therefore, development of the tool has been directed towards the provision of facilities that are necessary for work with that code. This development has been demand driven according to the needs of potential users which provides a good basis for ensuring that the tools are useful and tackle real-world problems. One problem with this approach, however, is that there has been no attempt to follow an overall strategy for development to coordinate the various options and maintain a clear set of aims.

Machine language code typically uses a very basic set of data types and has a number of instructions which act upon data items or control the flow of execution. The low level nature of this code means that the overall effect of code is often not apparent, especially when coupled with the need to refer to data objects and code fragments by memory or register addresses rather than by meaningful names.

The control flow of code is often very difficult to follow because jumps are made to locations in other parts of the program. Code can also be repeated a number of times if it has been inlined by the programmer to improve efficiency. Looping structure is often not apparent and appears as a number of jumps which criss-cross backwards and forwards past each other.

The solutions which have been developed to aid the removal of the problems described above can be summarised under the following headings:

- Action Systems,
- Loops,
- Separation of Unrelated Code,
- Creating Procedures,
- Adding Parameters to Procedures,
- Program Editing (Error Correction).

The restructuring is performed using the WSL language (see section 4.1) which was developed specifically for program transformation work. This requires translation of code from the source language, e.g. assembly code, into WSL. For more details about this see sections 8.2.1 and A.1.2.

Action Systems

A program written using labels and jumps can be translated directly into an action system which allows emulation of goto statements[1]. Machine code is represented as a series of actions each of which equates to the code between jumps to different parts of the program. This is just as incomprehensible as the original machine code transformations allow the action system to be converted into a series of nested loops and conditions. The control flow of the code is then much easier to understand.

Loops

Loop restructuring is important because it allows the basic structure of an algorithm to be changed. Sometimes there is a need to convert between recursive and iterative versions of a code fragment. These conversions and their inverse cannot only affect the clarity of the algorithm but can also affect the efficiency of its execution.

An example of loop restructuring is provided in the situation where the first iteration of a loop is handled differently to the others because it does not have an initial context upon which to base subsequent operation, see example A.1. A loop which searches an array to find the minimum value contained within the data structure can be used to demonstrate this. The first time that the loop is executed the minimum value is undefined and if not treated specially could cause an invalid result.

```
$x := 0$;
while $(x < 6)$ do
    if $((x = 0) \vee (data[x] > max))$
      then $max := data[x]$ fi;
    $x := (x + 1)$ od;

$\equiv$

$max := data[0]$;
$x := 1$;
while $(x < 6)$ do
    if $(data[x] > max)$
      then $max := data[x]$ fi;
    $x := (x + 1)$ od
```

Example A.1: Calculation of the Maximum Value in an Array

[1] The semantics of an action system are very similar to the use of continuations in other languages' semantic definitions.

Separation of Unrelated Code

When reverse engineering code it is typical to find that code which performs one task has been interleaved with code for another unrelated task. The statements for each task may not effect the result of the other although at some point in the future the result of a code fragment may depend upon the combined result of these two. It is desirable to separate the code for each fragment from that of the other.

$$\begin{array}{lcl}
\{(x \geq 0)\}; & & \\
x := (x+7); & & \{(x \geq 0)\}; \\
\underline{\textbf{if}}\ (x > 5) & \equiv & x := (x+7); \\
\quad \underline{\textbf{then}}\ \mathit{result} := \mathit{item1} & & \mathit{result} := \mathit{item1} \\
\quad\ \underline{\textbf{else}}\ \mathit{result} := \mathit{item2}\ \underline{\textbf{fi}}; & &
\end{array}$$

Example A.2: Removal of Dead Code

A special case of this is the removal of dead code from a program, example A.2. Dead code does not affect the result of the program. This is usually because the result of a computation may be overwritten by that of a later one. The code can therefore be safely removed without affecting the semantics of the program.

A number of transformations allow movement of statements around the program. The most basic ones allow the execution order of two statements to be swapped provided that one statement does not effect the result of the other. A more complex example is the swapping of two assignments when one uses the result of the other in the calculation of its own result. To do this the value of the first assignment must be accounted for and replaced in the right hand side of the other assignment.

$$\begin{array}{lcl}
x := (y+4); & \equiv & z := (4 + (2 \times y)); \\
z := (x+y); & & x := (y+4)
\end{array}$$

Example A.3: Movement of Statements

The Maintainer's Assistant also makes it possible to move statements into and out of conditionals and loops. This is slightly more complex because of the need to ensure that the semantics of the program are not affected along all possible execution paths. Ward's theory [88] allows this to be performed with minimal effort.

Creating Procedures

Procedures are a basic concept in modern programming languages. Producing them involves the grouping of statements, which perform a desired operation, into a unit which can be called by its name. Whenever this name is encountered in the code the statements making up the procedure are executed. This makes a program easier to comprehend but in general makes it less efficient to execute because of the overhead of passing parameters and storing return addresses. The technique of inlining is used to replace the occurrences of a procedure call with the statements making up the body. In many of the older languages this was done using macro pre-processing.

```
while ($x \neq 50$) do
        $x := (x + 1)$;
        if (($x$ mod 7) = 0)
          then $x := (x + 1)$ fi;
        calculate($x$ var result);
        $y := (y + 1)$ od;
```

$\equiv$

```
begin
  while ($x \neq 50$) do
        inc( var );
        if (($x$ mod 7) = 0)
          then inc( var ) fi;
        calculate($x$ var result);
        $y := (y + 1)$ od
where
  proc inc( var ) $\equiv$ $x := (x + 1)$.
end;
```

Example A.4: Introducing Procedures to Code

When source code is maintained it is common to find that the bodies of procedures are interspersed around the program and it is desirable to collect these back together into named procedures which aid program understanding. The Maintainer's Assistant allows identification of groups of statements that may be converted into procedures. These can then be attached to a block of code thus giving the procedure a scope. At any point thereafter it is possible to replace any other occurrences of the body of the procedure with a call to the procedure.

Adding Parameters to Procedures

To make the use of a procedure more generic parameters are often added. The procedure then operates on the parameters specified in the call rather than on the global variables which relate to the named variables within the procedure.

The use of a simple transformation allows a procedure to be parameterised making its use more general. For example a procedure which added one to the variable x would now be able to add one to any variable.

```
begin
  while $(x \neq 50)$ do
        inc( var );
        if $((x \; mod \; 7) = 0)$
          then inc( var ) fi;
        calculate(x var result);
        $y := (y + 1)$ od
where
  proc inc( var ) $\equiv$ $x := (x + 1)$.
end;
```

$$\equiv$$

```
begin
  while $(x \neq 50)$ do
        inc( var x);
        if $((x \; mod \; 7) = 0)$
          then inc( var x) fi;
        calculate(x var result);
        inc( var y) od
where
  proc inc( var z) $\equiv$ $z := (z + 1)$.
end
```

Example A.5: Adding Parameters to Procedures

Program Editing (Error Correction)

An essential part of software maintenance is the editing of the source code to correct errors or implement new functionality. The Maintainer's Assistant provides a full program editing tool which allows individual statements to be changed or deleted and new code to be added. The tool shows which constructs are valid at any particular position. To add a construct to the program the user simply has to highlight the insertion position and then select the construct which is to be inserted. The Maintainer's Assistant automatically places position markers wherever extra parameters are required, see example A.6.

```
$var$ := $expn$
```

Example A.6: An Assignment Statement as Added by the Maintainer's Assistant

A.1.2 Multiple Source Languages

One of the major advantages of the Maintainer's Assistant is its ability to represent programs written in a number of different source languages. Because the Maintainer's Assistant is based around the WSL language this is done by translating the code from the source language into WSL. The tool can then be used to maintain the code. At the end of the maintenance process the WSL code is translated back to the source language or into another programming language. Currently translators exist for a number of languages including IBM/360 assembly language, COBOL, C and Jovial.

WSL is capable of representing both executable and specification languages thus making the process of transformation easier. Use of many formal methods is hindered at the stage where the formal specification must be converted into an executable language or vice-versa. At this point the specification typically describes the algorithm and needs relatively minor changes for it to become executable. During the process of conversion from specification language to executable language there is the possibility that errors may be introduced. The Maintainer's Assistant avoids this problem as a result of the use of the wide spectrum language WSL.

A.1.3 Formally Defined Semantics

WSL has formally defined semantics which are based around general specifications and an imperative kernel language. Transformations are proven using weakest preconditions expressed as formulae in infinitary logic. This provides assurances that the transformations that are applied to the program guarantee that the semantics remain unaltered.

A number of programming languages, e.g. Pascal and the Spark Ada subset, also have formally defined semantics. WSL has a major advantage over these languages when used for program transformations; it was designed to make the proof of transformations relatively easy. Therefore, development of transformations is much quicker in WSL. Other languages can be transformed using these transformations by translating the original source code into WSL. This removes the need for a large amount of redevelopment effort which would be necessary to redevelop transformations for these other languages.

A.1.4 Practical Experience

The Maintainer's Assistant and its commercial counterpart FermaT have been used on a wide range of projects [23, 81] involving a number of different source and target languages. The results from these projects show that the transformation theory is valid and that real benefit can be gained from the use of program transformations. The main uses of the tools have been as an aid to program understanding and to perform migration from one language to another.

A.2 Deficiencies

The Maintainer's Assistant has a number of areas where further development work is needed. These fall into three categories: (1) the tasks which have not yet been performed due to time pressures; (2) the developing needs of the computing industry; and (3) problems with the skills needed to use the system. The main deficiencies are:

- Data Typing,
- Data Abstraction and Modularisation,
- Translation from/to source languages,
- Selection of appropriate transformation strategies,
- Backtracking facilities,
- Unsupported Language Constructs,
- The Laws of Arithmetic,
- Poor code modularisation support,
- Multi-Layer Software (libraries).

A.2.1 Data Typing

Data typing involves the classification of the data objects to reflect the possible values that can be represented by that object. The use of composite data types, e.g. arrays, allows the structure of data to be described and allows the representation of abstract objects which resemble the real-world items that they represent.

The use of data typing presents a number of advantages in different types of software engineering development work. Two categories which are easily identified are those of forward and reverse engineering. The main advantage of using data typing for forward engineering is that it helps to reduce the occurrences of errors. Data types make it easier to spot incorrect type usage and a considerable amount of automatic error checking can be done by compilers. Use of composite types allow logically related data items to be referred to as one item.

When reverse engineering a system the detection of errors becomes less important because the code will typically have been working correctly for a considerable amount of time. New code structure becomes one of the primary aims and this does not require the provision of explicit data typing.

The provision of data typing during reverse engineering is important however under the following conditions:

1. To allow reasoning about conditions within the language which allow control flow manipulation. For example to allow determination of data values which affect conditional branch instructions.

2. When the maintainer needs to consider data structure and produce abstractions of data it is vital that this can be represented concisely within the language. If the machine does not have a standard definition of data types it becomes very difficult to write transformations which will perform sensible operations upon the data.

3. It is also important that the maintainer is able to view the code in a form that is similar to other computer languages. This makes the process of program understanding easier.

The current typing system in the Maintainer's Assistant does not provide any specific types. Instead all variables have a universal type which allows a number of different types to be stored within that variable. These may contain primitive objects such as integers, characters or composite objects such as strings, sequences or trees. Any properties of these types must be explicitly stated within the program as a series of assertions. This makes the code look more complicated than it actually is and means that appropriate assertions must be supplied whenever necessary, rather than using implicit properties of objects.

There are also potential performance overheads associated with the existing method of representing types using assertions. The tool cannot be optimised to deal with the common types found in programs because there is no standard definition of their properties which can be appealed to.

A.2.2 Data Abstraction and Modularisation

When performing software maintenance it is desirable to be able to group data objects and the code that performs operations upon them into modules. This is a common practice when developing software and allows abstract thought about the real life objects that the data represents. Common concepts in the software engineering field are those of abstract data types and object orientation. Support for these is missing in the Maintainer's Assistant.

A.2.3 Translation from/to Source Languages

Work with the Maintainer's Assistant requires that the code which is being transformed is represented in the WSL language. This presents a problem for maintainers of code which has been developed in other languages. Before the Maintainer's Assistant can be used upon this code it must be translated into WSL. At the present moment the translators that are available have not been formally proven to be correct, therefore providing a potential source of errors in the conversion process. It will not, however, be possible to give a formal definition of the translation unless both source and target languages are formally specified. This presents a major problem because in general it is necessary to rely on conventional software validation techniques.

A.2.4 Selection of Appropriate Transformation Strategies

Before a maintainer can successfully start to apply transformations and make the code more understandable he must have an idea about his goals. The primary goal is to make the code more structured and to remove many of the implementation details while still retaining 100% functional equivalence. This revolves around standard software engineering techniques such as modularisation. The maintainer however needs to understand how transformations can be used to implement these techniques and to have a knowledge of which transformations are available.

When attempting to transform a program into a different form the maintainer must plan the general direction in which the transformation should be done. A series of steps must then be identified which will progress in that general direction. Some of the steps required are often counter-intuitive because they require that complexity is temporarily increased in order for subsequent steps to be able to remove underlying complexity.

The identification of these steps is therefore a complex task. The probable outcome of each step must be known along with an idea of what preconditions each step demands. This requires that the maintainer must be reasonably experienced before he can start doing productive work.

Bull [23, page 206] reports that case studies using the Maintainer's Assistant in practical case studies at IBM Hursley showed that most users could become proficient with basic transformations in under two weeks. However, the use of other transformations, predominantly those relating to loops, takes longer to master due to the need for greater understanding of the underlying mathematical theory.

In his thesis, Bull [23, page 212] also notes that the Maintainer's Assistant lacks facilities to assist the user in selecting appropriate transformations. The areas he identifies where help could be given are:

- The system could suggest a number of possible "next step" transformations.
- The use of a "jittering" mechanism similar to that used by the TI (Transformational Implementation) System [5]. The jittering system modifies a program to allow a transformation, which fails due to a technical detail, to be performed.
- An additional class of transformations which uses knowledge of programming goals. For example "divide and conquer", recursion removal and backtracking.
- A system to allow a user to build up his own catalogue of compound transformations which consist of frequently used combinations of existing transformations.

A.2.5 Backtracking Facilities

Linked with the problems of user knowledge is the problem of correcting minor flaws in a transformation strategy which occur at an early point in the development. These flaws often mean that a later transformation cannot be applied because a precondition is not met. The solution is to unwind the transformation steps, correct the flaw and reapply the subsequent transformations.

Currently the Maintainer's Assistant allows the engineer to undo transformation steps and maintains a history of the unwound steps. A redo facility is available to allow retracing of the steps if the sequence of transformations is unwound too far. Unfortunately if a transformation is applied at the place where the flaw occurred the history of the unwound steps is lost. Therefore the unwound steps have to be manually recorded and reapplied in order for the flaw to be corrected.

A.2.6 Unsupported Language Constructs

A problem that becomes apparent when trying to handle some of the more advanced languages available is the lack of support for certain language constructs. The major examples of this category include:

Exceptions — Error handling constructs which are associated with specific blocks within a program. They do not affect the logical control flow of a program unless an error condition occurs. These constructs are found in a number of languages such as Ada, Java and C++;

Pointers — Pointers are found in many languages and are used to allow the dynamic creation and destruction of objects at runtime;

Temporal Constraints — Many programs for real-time systems require that time constraints are met for successful execution. The ability to reason about these constraints is useful in these cases;

Concurrency — This is an important concept in multi-tasking environments and often allows a program to be split into a number of simple subsystems which communicate whenever necessary.

Self-modifying Code — It is very difficult to handle this type of code because the program is represented as data and it can be difficult to determine exactly when a part of a program is being changed[2].

A lack of these constructs can be tolerated in many cases, but the current popularity of Ada, Java and C++ suggests that exceptions in particular should be considered. Pointers are a particular problem because their semantics are difficult to formalise, but their use is very wide spread in programs. There are no plans to tackle pointers rigorously at the moment, an intermediate solution using human intervention is planned to solve the problem. The importance of correct execution in safety critical systems requires that any system designed to maintain these treats temporal constraints and concurrent aspects rigorously. Current research by Younger [25, 98] is addressing the latter two problems.

A.2.7 The Laws of Arithmetic

The Maintainer's Assistant uses a symbolic mathematics and logic module to provide reasoning about the arithmetic and logical conditions present in expressions within a program. These provide a very basic level of functionality which includes elementary cases only. More complex reasoning about complex mathematical functions is not easy. It often requires a large amount of user interaction and in many cases it is not feasible to produce the desired results.

Another problem with the mathematics module is that the expressions provided have not been formally specified and their correctness relies only on testing. This is obviously not acceptable when reasoning with correctness preserving transformations.

When examined closely it is apparent that the lack of strong data typing hinders the mathematics and symbolic logic package. The system cannot make judgements on the type of the values stored in data objects and, therefore, cannot reason properly about the result of the expression. This is another factor which makes the inclusion of typing within the language desirable.

[2] This problem is even difficult to overcome in microprocessors which have separate instruction and data caches. A write to the program code must be followed by cache flushing instructions to ensure that the next execution of the changed instructions will be performed properly.

The preferred way to add rigorous support for this logic is to use a theorem prover for reasoning instead of the logic package. This can take standard theories for various types and allows their use whenever necessary.

A.2.8 Poor Code Modularisation Support

WSL provides a limited amount of support for code modularisation. Procedures and functions can be attached to code blocks making them visible only within the block. There is no facility for collecting procedures and functions together into related groups which provide operations of a similar class or type. The functions which are attached to a block are visible to all sub-blocks and there is no concept of hiding within the program tree. Thus it is difficult to identify which components are needed in which block. The addition of these concepts to WSL will make it easier to represent the program in abstract terms which can be related to real world objects.

The collection of procedures and functions into a block is known, in Ada, as packages. Packages also allow the definition of types and variables to be incorporated. This allows packages to be used to represent abstractions of operations and objects. These packages are then explicitly included into code blocks when they are needed. This reduces the number of concepts which are available implicitly and therefore helps to make program understanding easier.

A related problem occurs in languages where different modules of the program can be stored in different files. This is not addressed in the Maintainer's Assistant where everything must be stored in the main file. The lack of support for this has the disadvantage that the Maintainer's Assistant cannot easily represent the relationships between files. To do so would require that all of the files are combined into one file before transformation work is done. This makes it difficult to perform re-engineering in the large.

A.2.9 Multi-Layer Software (libraries)

The approach to subprogram modularisation is made more complex when the introduction of standard libraries is considered. These appear in most systems and range from operating system facilities through mathematical libraries to vendor supplied software such as database engines. In almost all of these cases the source code will not be available for use during maintenance and if available would describe the operations in terms of low level operations. For example, when using file system primitives it is not necessary to know where individual bits are stored on the disk and how the disk controller can be used to access the data. Instead only the contents of the file are important.

The Maintainer's Assistant allows library subprograms to be introduced as external procedures and functions. These externals have an undefined effect on the state of the program and therefore their use is of limited benefit. For usable reasoning about external library subprograms the properties of the library units are needed. These could be described as a specification or as a relatively abstract piece of WSL code.

Bibliography

[1] J. R. Abrial, S. T. Davis, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. *The B Method.* BP Research, Sunbury Research Centre, U.K., 1991.

[2] Ada Joint Program Office. *Ada Reference Manual.* Ada Joint Program Office, ISO/IEC 8652:1995(E) edition, 1995.

[3] D. Andrews. *Data Reification and Program Decomposition*, volume 252 of *Lecture Notes in Computer Science*, pages 389 – 422. Springer-Verlag, 1987.

[4] R. J. R. Back. *Correctness Preserving Program Refinements*, volume 131 of *Mathematical Centre Tracts.* Mathematisch Centrum, 1980.

[5] R. Balzer, N. M. Goldman, and D. S. Wilde. On the transformational programming approach to programming. In *Proceedings of International Conference on Software Engineering.* IEEE Computer Society, 1976.

[6] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal construction by transformation—computer aided intuition guided programming. *IEEE Transactions on Software Engineering*, 15(2), 1989.

[7] F. L. Bauer and the CIP Language Group. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[8] F. L. Bauer and the CIP System Group. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[9] I.D. Baxter. An overview of a (transformational) design maintenance system. In *Proceedings of 1st UK Program Transformation Workshop*, Durham, 1996.

[10] K.H. Bennett. Automated support of software maintenance. *Information and Software Technology*, 33(1):74–85, 1991.

[11] K.H. Bennett, T. Bull, and H. Yang. A transformation system for maintenance - turning theory into practice. In *Proceedings of International Conference on Software Maintenance.* IEEE Computer Society, 1992.

[12] K.H. Bennett and N. Chapin, editors. *Journal of Software Maintenance : Research and Practice.* Wiley, 1997.

[13] P.J. Biggs. ReThree-C++ — a reverse engineering, redocumentation and reuse tool for C++. http://www.dur.ac.uk/~dcs3pjb/re3-cpp.html.

[14] D. Bjorner, C.B. Jones, M. Mac an Airchinnigh, and E.J. Neuhold, editors. *VDM '87, VDM - A Formal Method at Work*, volume 252 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[15] B.W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[16] C. Boldyreff, E.L. Burd, R.M. Hather, R.E. Mortimer, M. Munro, and E.J. Younger. The AMES approach to application understanding - a case-study. In *Proceedings of International Conference on Software Maintenance*, pages 182–191. IEEE Computer Society, 1995.

[17] R. Boulton, A. Gordon, M.J.C. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland/Elsevier, 1992.

[18] J.P. Bowen and P.T. Breuer. Decompilation. In H. Van Zuylen, editor, *The REDO compendium: reverse engineering for software maintenance*, pages 131–138. Wiley, 1993.

[19] J.P. Bowen and M.J.C. Gordon. Z and HOL. In *Proceedings of 8th Z User Meeting (ZUM '94)*, Workshops in Computing series. Springer-Verlag, June 1994.

[20] P.T. Breuer and J.P. Bowen. Decompilation — the enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems*, 16(5):1613–1647, 1994.

[21] P.T. Breuer, K.C. Lano, and J.P. Bowen. Understanding programs through formal methods. In H. Van Zuylen, editor, *The REDO compendium: reverse engineering for software maintenance*, pages 195–223. Wiley, 1993.

[22] W. Brew. *Reengineering Your Software for the Millennium.* Reasoning Systems Inc., 3260 Hillview Avenue, Palo Alto, CA 94304, 1996.

[23] T.M. Bull. *Software Maintenance by Program Transformation in a Wide Spectrum Language.* Ph.D. Thesis, University of Durham, 1994.

[24] T.M. Bull and K.H. Bennett. The work of the Durham Centre for Software Maintenance. Technical report, Department of Computer Science, University of Durham, 1995.

[25] T.M. Bull, K.H. Bennett, E.J. Younger, and Z. Luo. Bylands: Reverse engineering safety-critical systems. In *Proceedings of the International Conference on Software Maintenance.* IEEE Computer Society, 1995.

[26] E.L. Burd, M. Munro, and C. Wezeman. Analysing large COBOL programs: The extraction of reuseable modules. In *Proceedings of International Conference on Software Maintenance*, pages 238–243. IEEE Computer Society, 1996.

[27] G. Canfora, A. Cimitile, and A. De Lucia. Specifying code analysis tools. In *Proceedings of International Conference on Software Maintenance*, pages 95–103. IEEE Computer Society, 1996.

[28] M.A.M. Capretz. *A Software Maintenance Method Based on the Software Configuration Management Discipline.* Ph.D. Thesis, University of Durham, 1992.

[29] L. Cardelli. A semantics of multiple inheritance. *Information and Computing*, 76(2/3), 1988.

[30] V.A. Carreño and P.S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In J. Alves-Foss, editor, *Proceedings of HOL-95: International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 1–16, September 1995.

[31] N. Chapin. *360 programming in assembly language.* McGraw-Hill, 1968.

[32] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1), 1990.

[33] C. Cifuentes and V. Malhotra. Analysing code. In *Proceedings of International Conference on Software Maintenance*, pages 340–349. IEEE Computer Society, 1996.

[34] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: A case study. In *Proceedings of International Conference on Software Maintenance*, pages 124–133. IEEE Computer Society, 1995.

[35] M. Clint. Program proving: coroutines. *Acta Informatica*, 2:50–63, 1973.

[36] M. Clint and C. Vicent. The use of ghost variables and virtual programming in the documentation and verification of programs. *Software—Practice and Experience*, 14(8):711–734, 1984.

[37] E.F. Codd. Normalized database structure: A brief tutorial. In *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control*, 1971.

[38] B.J. Cornelius. *Programming with TopSpeed Modula-2.* Addison-Wesley, 1991.

[39] C.J. Date. *An Introduction to Database Systems, Fourth Edition.* Addison Wesley, 1986.

[40] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[41] M.S. Feather. Zap program transformation system: Primer and manual. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1978.

[42] M.S. Feather. *A Program Transformation System.* Ph.D. Thesis, University of Edinburgh, 1979.

[43] R.J. Gadd. ReForm — from assembler to Z using formal transformations. In *Proceedings of 4th European Software Maintenance Workshop*, Durham, 1990.

[44] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL — A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[45] D. Gries and S. Owicki. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[46] W.G. Griswold, M.I. Chen, R.W. Bowdidge, and J.D. Morgenthaler. Tool support for planning the restructuring of data abstractions in large systems. *Software Engineering Notes*, 21(5):33–45, November 1996.

[47] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioner Series. Prentice Hall, 1992.

[48] J. Grundy. A window inference tool for refinement. In C.B. Jones, B.T. Denvir, and R.C.F. Shaw, editors, *Proceedings of the 5th Refinement Workshop*, Workshops in Computing. Springer-Verlag, 1992.

[49] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[50] J. Harrison. Constructing the real numbers in HOL. *Formal Methods in System Design*, 5(1/2):35–59, July 1994.

[51] L. Hatton. *Safer C developing software for high-integrity and safety-critical systems.* McGraw-Hill, 1995.

[52] J.P. Hoare. Application of the B-method to CICS. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, Series in Computer Science, pages 97–124. Prentice-Hall, 1995.

[53] B.M. Hodgson. *The Maintainer's Assistant - User Guide - version 2.* Durham Software Engineering Ltd., August 1993.

[54] Proceedings of International Conference on Software Maintenance, 1997.

[55] Xinotech Research Inc. *Enterprise-Wide Automated Software and Data Reengineering with the Xinotech Technology.* Xinotech Research Inc., 1997. Information from http://www.xinotech.com/.

[56] C. B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 1986.

[57] C.B. Jones. *Systematic Software Development using VDM, 2nd Edition.* Prentice-Hall, 1990.

[58] D.E. Knuth. *Literate Programming.* Center for the Study of Language and Information, Stanford University, 1992.

[59] T. Lake and T. Blanchard. Reverse engineering of assembler programs using a TDF-based intermediate language. In *Proceedings of European Workshop on Software Maintenance*, Durham, 1995.

[60] T. Lake and T. Blanchard. Reverse engineering of assembler programs: A model-based approach and its logical basis. In *Proceedings of Working Conference on Reverse Engineering*, pages 67–75. IEEE Computer Society, 1996.

[61] K. Lano. *The B Language and Method - A Guide to Practical Formal Development.* Springer-Verlag, 1996.

[62] K.C. Lano. Z++, an object-oriented extension to Z. In J.E. Nicholls, editor, *Z User Workshop, Oxford*, pages 179–185. Springer-Verlag, 1990.

[63] K.C. Lano, P.T. Breuer, and H. Haughton. Reverse engineering COBOL via formal methods. In H. Van Zuylen, editor, *The REDO compendium: reverse engineering for software maintenance*, pages 225–248. Wiley, 1993.

[64] J. Leonard, J. Pardoe, and S. Wade. Software maintenance — cinderella is still not getting to the ball. In *Proceedings of IEE/BCS Conference on Software Engineering*, pages 104–106, 1988.

[65] B.P. Lientz and E.B. Swanson. *Software Maintenance Management.* Addison Wesley, Reading, MA, 1980.

[66] R. McCrindle. The reality of the virtual maintainer. In *Proceedings of European Software Maintenance Workshop*, Durham, 1996.

[67] J.A. McDermid and P. Rook. Software development process models. In J.A. McDermid, editor, *Software Engineer's Reference Book*, chapter 15. Butterworth Heinemann, 1991.

[68] T. Melham. Using recursive types to reason about hardware in higher order logic. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification—Proceedings of the IFIP WG10.2 Working Conference*, pages 27–50. North-Holland, 1988.

[69] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[70] B. Möller. Calculating with pointer structures. In R.S. Bird, editor, *Proceedings of IFIP TC2/WG2.1 Working Conference on Algorithmic Languages and Calculi.* Chapman and Hall, 1997.

[71] C. Morgan. *Programming from Specifications, 2nd Edition.* Prentice-Hall, 1994.

[72] R.E. Mortimer and K.H. Bennett. Maintenance and abstraction of program data using formal transformations. In *Proceedings of International Conference on Software Maintenance*, pages 301–310. IEEE Computer Society, 1996.

[73] M. Phillips. CICS/ESA 3.1 experience. In J.E. Nicholls, editor, *Proceedings of Z User Workshop*, pages 179–185. Springer-Verlag, 1990.

[74] B. Potter, J. Sinclair, and D. Till. *An introduction to formal specification and Z.* Prentice-Hall, 1991.

[75] C.H. Pratten. An introduction to proving AMN specifications with the HOL theorem prover. In H. Habrias, editor, *Proceedings of International Conference on: Putting into practice methods and tools for information system design, Nantes*, 1995.

[76] V. Rajlich and S.R. Adnapally. VIFOR 2: A tool for browsing and documentation. In *Proceedings of International Conference on Software Maintenance*, pages 296–300. IEEE Computer Society, 1996.

[77] N.F. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, 1987.

[78] H.M. Sneed. Object-oriented COBOL recycling. In *Proceedings of European Software Maintenance Workshop*, Durham, 1994.

[79] Black hole or buried treasure — 8th European Software Maintenance Workshop, Durham, 1994.

[80] The year 2000 and related issues — European Software Maintenance Workshop, Durham, 1997.

[81] Software Migrations Ltd. *An Introduction to FermaT.* Durham, 1995.

[82] J.M. Spivey. *The Z Notation: A Reference Manual, 2nd Edition.* Prentice-Hall, 1992.

[83] G.L. Steele. *Common Lisp: the language, 2nd edition.* Digital Press, 1990.

[84] Sun Microsystems. *SunOS 2.x to 1.x Binary Compatibility Package.* Sun Microsystems, 1996. Part of the Solaris(tm) documentation.

[85] R.E. Sward. Extracting functionally equivalent object-oriented designs from imperative legacy code. In *Proceedings of the NASA Reuse Workshop*, September 1996.

[86] S.K. Tuggle. *Assembler language programming: systems/360 and 370.* Science Research Associates, Inc., 1975.

[87] J. van Heijenoort. *From Frege to Gödel: a source book in mathematical logic, 1879—1932.* Harvard University Press, 1967.

[88] M.P. Ward. *Proving Program Refinements and Transformations.* D.Phil. Thesis, Oxford University, 1989.

[89] M.P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

[90] M.P. Ward. Reverse engineering through formal transformation. *The Computer Journal*, 37(9), 1994.

[91] M.P. Ward. Program analysis by formal transformation. *The Computer Journal*, 39(7), 1996.

[92] M.P. Ward. Derivation of data intensive algorithms by formal transformation - the schorr-waite graph marking algorithm. *IEEE Transactions on Software Engineering*, to appear.

[93] M.P. Ward, F. W. Calliss, and M. Munro. The use of transformations in "the maintainer's assistant". In *Proceedings of International Conference on Software Maintenance.* IEEE Computer Society, 1989.

[94] R. Wiener. *Software Development using Eiffel: There is life other than C++.* Prentice Hall, 1995.

[95] H.J. Yang and K.H. Bennett. Extension of a transformation system for maintenance - dealing with data-intensive programs. In *Proceedings of International Conference on Software Maintenance*, pages 344–353. IEEE Computer Society, 1994.

[96] R. Yarmish. *Assembly language fundamentals, 360/370 OS/VS DOS/VS.* Addison-Wesley, 1979.

[97] P. Young. Software visualisation in cyberspace. Ph.D. Thesis Proposal, University of Durham, 1996.

[98] E.J. Younger, Z. Luo, K.H. Bennett, and T.M. Bull. Reverse engineering concurrent programs using formal modelling and analysis. In *Proceedings of International Conference on Software Maintenance*, pages 255–264. IEEE Computer Society, 1996.