Acquiring data designs from existing data-intensive programs

Yang, Hongji

ACQUIRING DATA DESIGNS FROM EXISTING DATA-INTENSIVE PROGRAMS

Ph.D Thesis

University of Durham
Department of Computer Science

Hongji Yang

1994

Abstract

The problem area addressed in this thesis is extraction of a data design from existing data intensive program code. The purpose of this is to help a software maintainer to understand a software system more easily because a view of a software system at a high abstraction level can be obtained.

Acquiring a data design from existing data intensive program code is an important part of reverse engineering in software maintenance. A large proportion of software systems currently needing maintenance is data intensive. The research results in this thesis can be directly used in a reverse engineering tool.

A method has been developed for acquiring data designs from existing data intensive programs, COBOL programs in particular. Program transformation is used as the main tool. Abstraction techniques and the method of crossing levels of abstraction are also studied for acquiring data designs.

A prototype system has been implemented based on the method developed. This involved implementing a number of program transformations for data abstraction, and thus contributing to the production of a tool. Several case studies, including one case study using a real program with 7000 lines of source code, are presented. The experiment results show that the Entity-Relationship Attribute Diagrams derived from the prototype can represent the data designs of the original data intensive programs.

The original contribution of the thesis is that the approach presented in this thesis can identify and extract data relationships from the existing code by combining analysis of data with analysis of code. The approach is believed to be able to provide better capabilities than other work in the field.

The method has indicated that acquiring a data design from existing data intensive program code by program transformation with human assistance is an effective method in software maintenance. Future work is suggested at the end of the thesis including extending the method to build an industrial strength tool.

Acknowledgements

I wish to acknowledge the Department of Trade and Industry (of the United Kingdom), the Science and Engineering Research Council (of the United Kingdom) and IBM (UK) for the financial support of the research.

I wish to thank Prof. K. H. Bennett sincerely for agreeing to be my supervisor, for his invaluable advice and encouragement and for his skillful supervision and administration throughout this research project.

I would like to thank Dr. Martin Ward, Dr. Tim Bull, Mr. Nigel Scriven, Mr. Brendan Hodgson, Dr. Roger Hutty, Dr. Brian Bramer and Mr. Tom Routon for their useful advice in research techniques, support in working environment, encouragement in completing the thesis and help in improving my English language.

I would also like to thank my wife Xiaodong Zhang for her unselfish support and to admit that I am in debt to my son Tianxiu Yang for being unable to spend enough time with him which I should whilst undertaking this study.

The thesis is prepared with LaTeX [97,99].

Copyright

Statement

The material offered in this thesis has not been submitted wholly or in part for any academic award or qualification other than that for which it is now submitted. During the period of registered study in which this thesis was prepared, the author has not been registered for any other academic award or qualification.

Contents

List of Figures

Chapter 1

Introduction

1.1 Purpose of the Research and Overview of Problem

The research described in this thesis was motivated by the increasing industrial demand to carry out software maintenance more efficiently, because software maintenance has become the most costly stage of the software lifecycle [21,22]. Study shows that reverse engineering is the first step in understanding the software to be maintained. Reverse engineering consists of two main activities - redocumentation and design recovery, and design recovery is the more challenging subset of reverse engineering [25,52]. Data design recovery of software is an important part of general design recovery. The purpose of the research is to establish the feasibility of using data abstraction techniques within a transformational programming approach by acquiring data designs from existing legacy programs.

In the typical waterfall software life cycle, the stage of *design* exists after the stage of *requirements analysis* and before the stage of *implementation.* Requirements analysis establishes what the system should do and under what circumstances it is to be done, whereas design establishes how it is to be done. Data design establishes what data should be held by the software system and how that data should be organised and used.

Data design recovery is the process in reverse. It should produce the in-

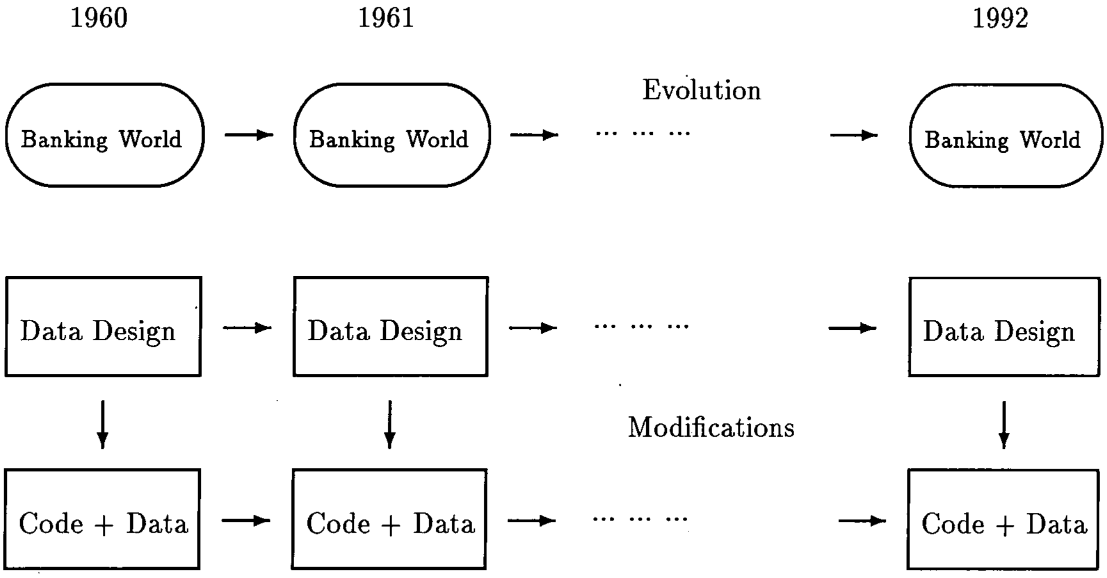


Figure 1.1: Evolution of Data Design and Code

formation required for a software maintainer to understand what software does, how it does it and why it does it, and so forth, via the data which the software uses. Usually, data design recovery recreates data abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains.

In practice, most software is heavily modified. The actual data design existing now will typically be very different from the original data design. Therefore, it may be thought very difficult and meaningless to extract the original data design. But data usually models the real world which we understand. There may be an opportunity to extract the current data design from the current version of the code, as the current data design should represent the reality. For example, a banking system might have been in use since (say) 1960. As the real banking world changes, the data/code of the banking system has to be modified. By 1992, it is probably not necessary to recover the data design of the system for 1960. Nevertheless, it is possible to recover the data design for 1992 from the code and the general knowledge of the banking world of 1992 (Figure 1.1). This suggests major understanding of data and/or code will be needed along the way.

Data plays a significant role in software. A good understanding of data used in a software system will assist a software maintainer in understanding the software system, so as to improve the maintenance activity.

The terms used in the chapter, such as software maintenance, reverse engineering, etc., will be defined in the following chapters.

1.2 Scope of the Thesis and Original Contribution

The primary goal of reverse engineering a software system is to increase the overall comprehensibility of the system in the software maintenance process. Five key objectives encompassed by the goal of software understanding are identified by [52]: to cope with complexity, to generate alternate views, to recover lost information, to detect side effects and to synthesise higher abstractions. According to these five general objectives, five concrete objectives were identified correspondingly for the research in this thesis:

1. Building A Tool — A key to tackling the complexity and volume of software is automated support. In reverse engineering an effective method is only commercially viable when it is backed up by tools. So a prototype system should be build as a demonstrator of this research.

2. Generating Entity-Relationship Attribute Diagrams — Graphical representations are good aids to understanding. Entity-Relational Attribute Diagrams have long been used as representations for data design. To derive data designs presented as Entity-Relational Attribute Diagrams is another objective of this research.

3. Recovering Lost Information — Information useful to software maintainers may be lost in the process of software evolution, in particular that software being heavily maintained. For example, modifications are often not recorded in the documentation, especially at a level higher than the code itself. Data

design recovery can assist us to salvage from the existing code potentially useful information.

4. Detecting Possible Faults — Mistakes could have been made when a software was first built or when a modification was carried out in the software's history. Data design recovery can be directly helpful in detecting faults in the software, because the obtained data design can show the true structure of the code and it is easier to spot faults in data design than in code.

5. Viewing Software at a Higher Abstraction Level — Software can be viewed at different abstraction levels. In particular, data, which is the main component of software, can be viewed at the *code level* and *conceptual level* (to be discussed in detail in later chapters). Obtaining data designs which belong to a conceptual data level makes it possible to view software at a higher abstraction level.

The scope of the thesis includes:

- Development of a data design recovery method: a method is established for discovering possible data designs from existing data-intensive programs for the purpose of reverse engineering.

- Implementation of a prototype system: a system is constructed for demonstrating the success of the above method.

- Experimentation with example programs: a number of programs are used for experiments with the prototype system and common problems in the programs, such foreign keys, are examined.

The original contribution of the thesis is to combine formal methods, transformational programming, data abstraction techniques and crossing levels of abstraction techniques for acquiring data designs from the existing programs. A method is proposed in the thesis and the result of applying the method shows that data relationships which exist only in the code can be identified and extracted. A prototype is built to demonstrate the research results.

The literature survey in Chapter 2 and 3 shows that there are no reverse engineering tools currently existing which are able to acquire an Entity-Relational Attribute Diagram directly from program code in this way.

Proving the correctness of transformations is not a focus of this thesis because it has been addressed in other research such as in [155]. It is also not intended to build an industrial-strength tool but it is pointed out that future research can be fruitful if carried out along the direction which is proposed in this thesis.

1.3 Criteria for Success

The success of the research described in this thesis is determined by the following criteria:

- If we start with old, heavily modified code which has never been developed using a formal or informal method, how viable is it to extract a program data design or specification from the code?
- If it is possible under certain restrictions, what are these restrictions? What exactly does the user need to supply?
- What is the method for extracting data design from the existing data intensive code?
- Obviously the process involves crossing levels of abstraction. How do we cope with this in order to obtain a data design from existing data intensive code?
- Can we build a tool to demonstrate the approach developed in this thesis?
- What is the metric to measure the resultant code (or data design) which has been reverse engineered by this method?

1.4 Thesis Structure

The structure of the thesis is as follows:

Chapter 2 provides an overview of the software engineering process and shows how reverse engineering fits into the process.

Chapter 3 gives an overview of related research in reverse engineering and current research in the area of reverse engineering.

The remaining chapters describe the research undertaken within the project.

Chapter 4 introduces data abstraction techniques, data abstraction levels and data design representations used in forward engineering and discusses the way in which they are used in reverse engineering. It also describes the problems of acquiring data designs from existing data-intensive programs.

Chapter 5 introduces the environment in which the prototype components of the Maintainer's Assistant for acquiring data designs from existing code are to be developed, and proposes a design recovery method.

Chapter 6 gives the reason why and how the existing WSL has to be extended and how the approach of transformational programming is used in the Maintainer's Assistant.

Chapter 7 discusses the use of crossing levels of abstraction to acquire data designs.

Chapter 8 describes the prototype components of Maintainer's Assistant which apply the result of the research in this thesis and the implementation of the prototype components.

Chapter 9 describes the details of experimenting with real program examples with the prototype, and shows the results obtained from the use of the prototype.

Chapter 10 summarises the thesis, assessing the research carried out in this thesis against the proposed criteria and suggesting areas for future research.

Statement: Although this research has been carried out in a collaborative project, REFORM, all work in this thesis is by the author (an original member of the REFORM project), except where explicitly stated.

Chapter 2

Software Engineering

2.1 Software and Software Engineering

2.1.1 Computer System Evolution

Although the technological revolution of computing is just a few decades old, a number of significant subrevolutions have taken place. During this time, software development has been closely coupled to computer system evolution.

Computer system evolution can be divided into four eras [132]. The first era was from the late 1940s to the mid-1960s. During this period, hardware underwent continual change and software development was largely unmanaged. Also batch orientation was used for most systems; software was customer designed for each application, often implemented by a single person and seen as a "craft", and had a relatively limited distribution.

The second era of computer system evolution spanned the decade from the mid-1960s to the late 1970s. Multiprogramming and multi-user systems introduced new concepts of human-machine interaction. Real-time systems could collect, analyse, and transform data from multiple sources, thereby controlling processes and producing output in milliseconds rather than minutes. Advances in on-line storage devices led to the first generation of data base management systems. At the same time, people started to use product software and software was developed for widespread distribution in a multidisciplinary market. Software purchased from outside the organisation was extended by the addition of new program statements to meet new needs. All of these software products had to be corrected when faults were detected, modified as user requirements changed, or adapted to new hardware. These activities were collectively called *software maintenance*. Efforts spent on software maintenance began to absorb resources at an alarming rate, and the personalised nature of programs made them very difficult to maintain. A software crisis had begun.

The third era of computer system evolution began in the mid-1970s. Distributed systems greatly increased the size complexity of computer-based systems. Global and local area networks, high band-width digital communications and increasing demands for data access put heavy demands on software developers. Microprocessors and personal computers were widely used. The personal computer has been the catalyst for the growth of many software companies. While the software companies of the second era sold hundreds or thousands of copies of their programs, the software companies of the third era sell tens and even hundreds of thousands of copies. Personal computer hardware is rapidly becoming a commodity, with software providing the differentiating characteristics. Many people in industry and at home spent more money on software than they did to purchase the computer on which the software runs.

The fourth era in software is just beginning. *Fourth generation techniques* (4GT) for software development are changing the manner in which some segments of the software community build computer programs. Object-oriented technologies are rapidly displacing more conventional software development approaches in many application ares. Expert systems and artificial intelligence software have finally moved from laboratory into practical application for wide-ranging problems in the real world. Artificial neural network software has opened exciting possibilities for pattern recognition and human-like information processing abilities. As we move into the fourth era, the software crisis continues to intensify.

The software crisis alludes to a set of problems encountered in the development of computer software. The problems are not limited to software that "does not function properly" according to required criteria. Rather, the software crisis encompasses problems associated with how we develop software, how we maintain a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software.

Problems associated with the software crisis have been caused by the character of software itself. Software is a logical rather than physical system element; therefore, success is measured by the standard of a single entity rather than many manufactured entities. Software does not wear out. If faults are encountered, there is a high probability that each was inadvertently introduced during development and went undetected during testing. We replace "defective parts" during software maintenance, but we have few, if any, spare parts; i.e., maintenance often includes correction or modification to design.

Recognising problems and their causes is the first step towards finding solutions. Then the solutions themselves must provide practical assistance to the software developer, improve software quality, and allow the "software world" to keep pace with the hardware world.

There is no single best approach to a solution for the software crisis. However, by combining comprehensive methods for all phases in software development: better tools for automating these methods; more powerful building blocks for software implementations; better techniques for software quality assurance; and an overriding philosophy for coordination, control, and management, we can achieve a discipline for software development — a discipline called *software engineering.*

2.1.2 Software Engineering and its Paradigms

Use of the term "software engineering" can be traced back at least as far as a 1968 NATO conference held in Garmisch, West Germany and the follow-up conference held near Rome, Italy, in 1969. The following definition is from Naur [125].

Software engineering *is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

This was partly prompted by the problems encounted in developing the Operating System OS360 for the IBM-360 computer.

Software engineering encompasses a set of three key elements — *methods, tools*, and *processes.* Software engineering methods provide the techniques for building software. Methods encompass a broad array of tasks that include: design of data structures, program architecture, and algorithmic procedure; coding; testing; and maintenance. Software engineering tools provide automated or semi-automated support for methods. Tools exist to support each of the methods noted above. Software engineering processes are the glue that holds the methods and tools together and enables rational and timely development of computer software. Processes define the sequence in which methods would be applied, the deliverables (documents, reports, forms, etc.) that are required, the controls that help assure quality and coordinate change, and the milestones that enable software managers to assess progress.

The above three components of software engineering are often referred to as **software engineering paradigms**. A paradigm for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. Three paradigms have been widely discussed and debated [132]. They are "the classic life cycle", "prototyping" and "fourth generation techniques".

The *classic life cycle* paradigm is sometimes called the "waterfall model", because there is no iteration in the process from the beginning to the end of a project. It demands a systematic sequential approach to software development. The life cycle paradigm encompasses the following activities:

- software requirements analysis,
- design,
- coding,
- testing, and
- maintenance.

Prototyping is a process that enables the developer to create a model of the software to be built and this process allows problems and requirements to be seen quickly [46]. Prototyping begins with requirements gathering. Developer and customer meet and define the overall objects for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software visible to the user. The quick design leads to the construction of a prototype. The prototype is evaluated by the customer/user and is used to refine requirements for the software to be developed. A process of iteration occurs as the prototype is "tuned" to satisfy the need of the customer, while at the same time enabling the developer to understand better what needs to be done.

Fourth generation techniques encompasses a broad array of software tools that have one thing in common: each enables the software developer to specify some characteristic of software at a high level [71]. The tool then automatically generates source code based on the developer's specification. The 4GT paradigm for software engineering focuses on the ability to specify software to a machine at a level that is close to natural language or in a notation that imparts significant function, but it tends to be used in a single, well defined application domain. Also the 4GT reuses existing packages, databases etc. rather than reinvents them.

2.1.3 Advantages and Disadvantages of Three Software Engineering Paradigms

The classic life cycle is the oldest and the most widely used paradigm for software engineering, and has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and maintenance can be placed. It has weaknesses as well: real projects rarely follow the sequential flow that the model proposes, i.e., iteration always occurs and creates problems in the application of the program; it is often difficult in the beginning for the customer to state all requirements explicitly, while the classic life cycle requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects; and the customer must be patient — a working version of the program(s) will not be available until late in the project time span [132].

Prototyping is an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is to be discarded (at least in part), and the actual software engineered with an eye towards quality and maintainability. The problems with this paradigm are: the customer sees what appears to be a working version of the software, unaware that in the rush to get it working overall software quality or long term maintainability have not been considered; the developer often makes implementation compromises (e.g., using inappropriate operating system or programming language, and inefficient algorithms) in order to get a prototype working quickly; etc.

Though it has been claimed that the fourth generation techniques are likely to become an increasingly important part of software development during the next decade because of the dramatic reductions in software development time and greatly improved productivity for people who build software, current 4GT tools are not much easier to use than programming languages because the source code produced by such tools is "inefficient" and the maintainability of large software systems developed using 4GT is open to question. Problems existing are: implementation using a 4GT enables the software developer to describe desired results which are translated automatically into source code to produce those results, but a data structure with relevant information must exist and be readily accessible by the 4GT; to transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other "transition activities" required in other software engineering paradigms; the 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.

2.1.4 A Generic View of Software Engineering

A generic view of software engineering can be obtained by examining the process of software development [132]. The software development process contains three generic phases regardless of the software engineering paradigm chosen. The three phases, *definition, development*, and *maintenance*, are encountered in all software development, regardless of application area, project size, or complexity.

The definition phase focuses on *what*. That is, during definition, the software developer attempts to identify what information is to be processed, what function and performance are desired, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. Thus, the key requirements of the system and the software are identified. Three specific subprocesses occur in this phase:

System analysis defines the role of each element in a computer-based system, ultimately allocating the role software will play.

Software project planning allocates resources, estimates costs, defines work tasks and schedules, and sets quality plans (and identifies risks).

Requirements analysis defines a more detailed information domain and software function before work can begin.

The development phase focuses on *how*. That is, during development, the software developer attempts to describe how data structure and software architecture are to be designed, how procedural details are to be implemented, how the design will be translated into a programming language, and how testing will be performed. Three specific steps also occur in this phase:

Software design. Design translates the requirements for the software into a set of representations (some graphical, other tabular or language based) that describe data structure, architecture, and algorithmic procedure.

Coding. Design representations must be translated into an artificial language that results in instructions executable by the computer. The coding step performs this translation.

Software testing. Once the software is implemented in machine-executable form, it must be tested to uncover defects in function, in logic, and in implementation.

The maintenance phase focuses on *change* that is associated with error correction, adaptations required as the software's environment evolves, and modifications due to enhancements brought about by changing customer requirements. The maintenance phase reapplies the steps of the definition and development phases, but does so in the context of existing software. Three types of change are encountered during the maintenance phase:

Correction. Even with the best quality assurance activities, it is likely that the customer will discover defects in the software.

Adaptation. Over time the original environment (e.g., CPU, operating system, peripherals) for which the software was developed is likely to change.

Enhancement. As software is used, the customer/user will recognise additional functions that would provide benefit.

2.1.5 Software Quality and Software Quality Assurance

Software engineering works toward a single goal: *to produce high-quality software.* It is therefore useful to clarify the terms "quality" and "software quality assurance" (SQA).

Software quality is defined as: conformance to explicitly stated functional and performance requirements, explicitly documented development standards, implicit characteristics that are expected of all professionally developed software [132].

Software quality factors include [117]: *correctness* (the extent to which a program satisfies its specification and fulfills the customer's mission objectives), *reliability* (the extent to which a program can be expected to perform its intended function with the required precision), *efficiency* (the amount of computing resources and code required by a program to perform its function), *integrity* (the extent to which access to software or data by unauthorised persons can be controlled), *usability* (the effort require to learn, operate, prepare input, and interpret the output of a program), *maintainability* (the effort required to locate and fix an error or other change in a program) (the maintainability of a software will be addressed later), *flexibility* (the effort required to modify an operational program), *testability* (the effort required to test a program to ensure that it performs its intended function), *portability* (the effort required to transfer the program from one hardware and/or software system environment to another), *reusability* (the extent to which a program can (or part of a program) be reused in other applications), and *interoperability* (the effort required to couple one system to another).

Software quality assurance is an activity that is applied at each step in the software engineering process. Software quality assurance encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

2.1.6 Current State of Software Engineering

An important consideration in the development of a software system is the entire development environment. In its most general sense, the development environment includes the technical methods, the management procedures, the computing equipment, the mode of computer use (batch or interactive, centralised or distributed), the automated tools to support development, the software development staff, and the physical work space. An ideal development environment should enhance the productivity of the information system developers and provide a set of tools (both manual and automated) that simplifies the process of software production. The environment should contain facilities both for the individual member of a development group and for the overall management of the project [157].

Now, *software engineering* has become a well defined, constantly evolving discipline. Software production is much different now from that prior to the year 1968 when the concept of software engineering was first introduced. The state of the art of software production then can be seen by the problem being discussed when the two NATO conferences on software engineering of 1968 and 1969 were held. For example, questions were [133,134]:

- problems of scale,
- in what orders to do things,
- strategies and techniques to use,

- how to specify software systems,
- projects planning and control,
- proliferation of unreliable software, etc.

Though some of these are still problems today, areas where progress has especially been made are :

- Modelling: requirements, systems.
- Formalisation: specification, verification.
- Computer science: languages, software concepts such as modularity and abstract data types.
- Method/design paradigm: structured programming, object oriented design, etc.
- Support: database, tool, software development environments.
- Human factors: user participation, project management, human-computer interface.
- Metrics: quality, reliability, costing.

Nevertheless, despite such advances, there are still many problems unsolved in the following areas:

- Formal methods: further development of specification and verification and their scaling up to cope with large 'real-life' problems, particularly with tool support.
- Metrics: improved methods for assessing and predicting cost, and software quality and reliability, maintainability, and other quality attributes.
- Reuse: software reusability will potentially represent a major way of effecting desperately needed increases in productivity, if software practice is going to have any chance of coping with the demand for software products.

- Maintenance: improved and new methods to reduce the cost and to increase the maintainability.
- Management: more reliable, more effective techniques for managing the life cycle in all aspects.
- Coping with *existing* systems (that were written using *old* technology).
- Tool support: increased provision of automated software tools for supporting all activities of software engineering, both on an individual basis and as integrated support environment.
- Applied technologies: application of other techniques, e.g., AI, to the general enhancement of software enginering .

It can be seen from the above analysis that software maintenance is a very important part in software engineering. Further issues of software maintenance will be reviewed in the next section.

2.2 Software Maintenance

In the early days of computing (1950s and early 1960s), software maintenance comprised a very small part of the software lifecycle. In the late 1960s and the 1970s, as more and more software was produced, people began to realise that old software does not simply die, and at that point software maintenance started to be recognised as a major activity. By the late 1970s, industry was suffering major problems with the applications backlog, and software maintenance was now taking more effort than initial development in some sectors. In the 1980s, it was becoming evident that old architectures were severely constraining new design [21]. All of these were placing demands that the changes to the software were performed. Changes include, for instance, fixing errors, adding enhancements and making optimisations. Besides the problems whose solutions required the changes in the first place, the implementation of the changes themselves create additional problems.

One of the five Lehman's laws of the evolution of a software system directly addresses the modification of software. It states that "a program that is used in a real world environment must change or become less and less useful in that environment" [105]. So mechanisms must be developed for evaluating, controlling, and making changes.

Software Maintenance is defined as *the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment* [2].

Software maintenance is required to meet the needs of three principal "change" types described in the previous section. So maintenance activities can be divided into these categories correspondingly [149].

The first category is called **corrective maintenance**. There may be a *fault* in the software, so that its behaviour does not conform to its specification. This fault may contradict the specification, or it may demonstrate that the specification is incomplete (or possibly inconsistent), so that the user's assumed specification is not sustained. Corrective maintenance involves removing these faults.

Even if a software system is fault-free, the environment in which it operates will often be subject to change, e.g., the upgrade of computer hardware or moving a system from a mainframe to a PC. Modifications performed as a result of changes to the external environment are categorised as **adaptive maintenance**, e.g., the manufacturer may introduce new versions of the operating system, or remove support for existing facilities, and the software may be ported to a new environment, or to different hardware.

The third category of maintenance is call **perfective maintenance**. This is undertaken as a consequence of a change in user requirements of the software. For example, a payroll suite may need to be altered to reflect new taxation laws; a real-time power station control system may need upgrading to meet new safety standards.

Finally, **preventive maintenance** may be undertaken on a system in order to anticipate future problems and make subsequent maintenance easier [20]. For example, a particular part of a large suite may have been found to require sustained corrective maintenance over a period of time. It could be sensible to re-implement this part, using modern software engineering technology, in the expectation that subsequent errors will be much reduced.

The large cost associated with software maintenance is the result of the fact that software has proved difficult to maintain. Early systems tended to be unstructured and *ad hoc*. This makes it hard to understand their underlying logic. System documentation is often incomplete, or out of date. With current methods it is often difficult to retest or verify a system after a change has been made. Successful software will inevitably evolve, but the process of evolution will lead to degraded structure and increasing complexity [23,69,105].

Now it is well established that software maintenance is the most costly stage of the software lifecycle for most projects. In 1970s, 30 - 40 % of the budget was used on software maintenance, and 40 -60 % in 1980s. Now the budget for software maintenance is up to 70 - 80 % [29,106,128,132,173].

Software maintenance has its own life cycle and its own features. Over the years, several software task models have been proposed, while the model by Bennett [21] is used here. Software maintenance can occur due to changing user needs, to errors which must be fixed, and to a changing environment. Although these types are different at the detailed level, at a high level they can be described by an iterative three stage process:

1. *request control*: the information about the request is collected; the change is analysed using impact analysis to assess cost/benefit; and a priority is assigned to each request.

2. *change control*: the next request is taken from the top of the priority list; the problem is reproduced (if there is one); the code (and design and the specifications if available) are analysed; the changes are designed and documented and tests produced; the code modifications are written; and quality assurance is implemented.

3. *release control*: the new release is determined; the release is built; confidence testing is undertaken; the release is distributed; and acceptance testing by

the customer takes place.

Currently, these three steps are almost always undertaken in terms of source code. Design information and even adequate documentation often do not exist. Thus software maintenance is thought of predominantly as a source code activity. Understanding the functions and behaviour of a system from the code is hence a vital part of the maintenance programmer's task [136]. Approaches to program comprehension will be described in later chapters.

Most of the effort for software maintenance research has focused upon the methods, techniques and tools which support the maintenance process.

When maintenance activities are carried out, an essential characteristic of all software — maintainability — must be considered. Maintainability is a key goal that guides the steps of a software maintenance method, as well as software engineering method. **Software maintainability** is the ease with which software can be understood, corrected, adapted, and/or enhanced [132].

The maintainability of software is affected by many factors. It is difficult to quantify the software maintainability (no adequate, widely accepted, quantitative definition exists). However, many efforts have been made to tackle this problem from different angles. Here are three of them.

Kopetz [98] defined a number of factors that are related to the development environment: availability of qualified software staff, understandable system structure, ease of system handling, use of standardised programming languages, use of standardised operating systems, standardised structure of documentation, availability of test cases, built-in debugging facilities and availability of a proper computer to conduct maintenance.

Gilb [76] provided the maintainability metrics by measuring the effort expended during maintenance in terms problem recognition time, administrative time, maintenance tools collection time, problem analysis time, change specification time, active correction time, local testing time, global testing time, maintenance review time and total recovery time.

Sneed [143] measured the maintainability in terms of the original development expenditure. The smaller the expenditure on maintaining the system —

relative to the expenditure on development — the greater the maintainability. The expenditure includes:

- *modularity* — an operating measure of the extent to which a system can be broken down into small independent building blocks;
- *flexibility* — an operating measure of a software system's independence from any specific application;
- *portability* — an operating measure of a software system's independence from its technical environment;
- *complexity* — an operating measure of a software system's aggregation and distribution of components/complexes [62].

Attributes of software can be divided into two types, *internal* and *external.* Internal attributes are a property of the software itself, e.g., complexity, size, data structure, coupling, cohesion, quality, reliability, etc. External attributes are a property of the environment, e.g., availability of debugging tools, skill and training, repository, management, etc.

Possibly the most important factor that affects maintainability is planning for maintainability. If software is viewed as a system element that will inevitably undergo change, the changes that maintainable software will be produced are likely to increase substantially [132]. However, maintainability is also dependent on the process as well as the software itself [98]. A major problem with maintenance is changes which were not even conceived of when the software was first designed and this cannot be planned for. Nevertheless, because maintainability is an essential characteristic of software, at each stage of the software engineering process, maintainability must be considered. For example, during the requirements stage, areas of future enhancement and potential revision should be noted, software portability issues discussed, and system interfaces that might impact software maintenance considered; during design stage, data design, architecture design, and procedural design should be evaluated for ease of modification, modularity, and functional independence; during the coding stage, style and internal documentation, two factors that have an influence on maintainability should be stressed; etc.

Also, maintenance activities should be carried in a careful way, because modification of software is dangerous in the sense that errors and other undesirable behaviours may occur as the result of software modification. The term *side effects* are used to refer these errors or undesirable behaviours [132]. Software maintenance side effects include *coding side effects, data side effects and documentation side effects.* Coding side effects are caused by the change of code. Data side effects occur when data changes in the software design may no longer fit the data, and when modifications are made to software information structures. Documentation side effects occur when changes to source code are not reflected in design documentation or user-oriented manuals.

Although progress in software maintenance research has been achieved recently, there are still many problems to be solved. One of the research topics identified by [21] for software maintenance is *Reverse Engineering*, which is the topic the thesis will address.

2.3 Reverse Engineering

Reverse Engineering *is the process of analysing a subject system to identify the system's components and their inter relationships, and to create representations of the system in another form or at a higher level of abstraction* [48].

Reverse engineering involves the identification or "recovery" of program requirements and/or design specifications that can aid in understanding and modifying the program. The main objective is to discover the underlying features of a software system including requirements, specification, design and implementation. In other words, it is to recover and record high level information about the system including:

- the system structure in terms of its components and their interrelationships expressed by interface,
- functionality in terms of what operations are performed on what compo-

nents,

- the dynamic behaviour of the system in understanding how input is transformed to output,
- rationale — design involves deciding between a number of alternatives at each design step,
- construction — modules, documentation, test suites etc.

There are several purposes for undertaking reverse engineering listed in [21]. They can be separated into the *quality issues* (e.g., to simplify complex software, to improve the quality of software which contains errors, to remove side effects from software, etc.), *management issues* (e.g., to enforce a programming standard, to enable better software maintenance management techniques, etc.) and *technical issues* (e.g., to allow major changes in a software to be implemented, to discover and record the design of the system, and to discover and represent the underlying business model implicit in the software, etc.).

It is seen that reverse engineering is an activity which neither changes the subject system, nor creates a new system based on the reversed engineered subject system. It is a process of examination and understanding of the object system, and of recording the results of that examination and understanding. On the other hand, reverse engineering is a key to the rest of the process of software maintenance, because it enables us to take an existing software system which is being maintained (e.g., in terms of its source code), and recover an abstract representation which can be used for subsequent maintenance or even reimplementation.

Because the techniques and methods of reverse engineering are still immature, the following precautions must be considered when reverse engineering is carried out:

1. the code may be specific, and not generic, so that few advantages are gained when the system is reengineered,
2. the code may have errors and it is not clear if it is useful to reverse engineer error filled code,

3. reverse engineering itself may introduce errors, and revalidation will be essential in the project plan,

4. reverse engineering can be very expensive and the returns are not clear and a cost benefit analysis will be needed,

5. there are no standards or standard methods for reverse engineering,

6. there are no well established measures of maintainability.

In most cases, reverse engineering is the first step of software maintenance. The analysis of the object software is crucially important to accomplish the request control stage of software maintenance.

One typical reverse engineering objective is to extract the program design or specification from the program code. There are two reasons for this. The first reason is that in order to achieve major productivity gains, software maintenance must be undertaken at a higher abstraction level than code, i.e., at the design level or specification level, because [21]:

1. The representation of a system at higher levels of abstraction is more compact than at lower levels, so the system is easier to understand as a whole.

2. The objects which represent the system at high levels of abstraction (e.g., modules, requirements, specifications etc.) are structures which encourage highly maintainable systems. Furthermore, they are closer to the application domain in terms of which many changes are expressed.

3. The documentation for systems maintained in this way can be clearly specified.

4. Modification can be better controlled leading to less structural degradation.

5. Modern software engineering techniques become available to the software engineer, leading to high quality in the software maintenance phase.

6. The high abstraction level objects are appropriate vehicles in which to express the testing plan.

From another point of view, software expressed at a higher level of abstraction is more maintainable than at a lower level of abstraction.

The second reason is that the need is often encountered in software maintenance projects. Firstly, the documents and relevant reference materials are not complete, and the personnel who may have relevant knowledge have already forgotten about it or have left. Secondly, there might be even some documents available, but the software may not be implemented consistently the documents. Thirdly, the original documents and reference materials were not written in a modern specification language and they can not be used in a modern software maintenance environment, not even be machine readable.

This means that the extraction of the program design or specification of an old program code is a vital step especially when the program is the only available documentation or is the only source on which to rely. The purpose of this kind of reverse engineering is (a) to reimplement the system or (b) to help understand the existing software. Someone may challenge by asking "Why do not you simply reimplement the software instead of carrying out reverse engineering of the old software even with no documentation?". It is because that there has been considerable investment in existing software, and it is often not cost effective to throw away existing software and rewrite it with the latest development technique. The significance of reverse engineering can be seen by the setting up many reverse engineering projects, for example [7,9,41,63,144].

2.4 Summary

Due to the rapid development of computer hardware and software, the demands and costs of software maintenance are increasing continuously. Software maintenance now comprises a major part in the software engineering lifecycle costs. The first step for conducting software maintenance is to understand the software to be maintained and the abstraction of the program design and specification from the existing source code is one of the methods which helps us to understand software systems. Reverse engineering is to carry out this task. Reverse engineering is particularly important when the source code is the only source with which to work. That is why the REFORM project was set up (more details on the REFORM project will be provided later). The final aim of the REFORM project is to discover designs and eventually specifications given only source program code. The REFORM project is based on a transformation approach and its aim is to find out the formal relation between program code and its design and eventually specification. There is not an existing method completed yet to be used in the project for acquiring a program design or specification from program code. This problem is tackled in this thesis.

Related research results are reviewed in the next chapter.

Chapter 3

Work Related to Reverse Engineering

3.1 Introduction

In the second chapter, the principles of each step in the software engineering lifecycle were described. This chapter will describe some existing software development approaches which are relevant to the thesis, together with several existing reverse engineering projects. This will help to clarify the research problem to be solved in this thesis.

In this context, it is necessary to restrict the scope so as to avoid discussion of a great many approaches. Although some of the ideas discussed below could be applied to the hardware development and the development of concurrent or real-time systems, they are out of the scope of this thesis. In this sense, a specification only refers to the **functional specification**, which describes the effect of software on its external environment, not to **performance specification**, which describes constraints on the speed and resource utilisation of the software, etc.

Let us start with the software system development. The most widely used method is to derive the final program from a specification. We use SP to represent a specification of requirement which the software system is expected to fulfill, expressed in some specification language SL (if any); and P to represent the ultimate object program which satisfies the specification in SP, written in some given programming language PL.

The usual way to proceed is to construct P by whatever means are available, making informal reference to SP in the progress, and then verify in some way that P does indeed satisfy SP. The only practical verification method available at present is to test P, checking that in certain selected cases that the input/output relation it computes satisfies the constraints imposed by SP. This has the obvious disadvantage that (except for trivial programs) correctness of P is never guaranteed by this process, even if the correct output is produced in all test cases. An alternative to testing is a formal proof that the program P is correct with respect to specification SP.

Most recent work in this area has focused on methods for developing programs from specification in such a way that the resulting program is guaranteed to be correct by construction. The main idea is to develop P from SP via a series of small refinement steps, inspired by the programming discipline of stepwise refinement [163]. Each refinement step captures a single design decision, for instance a choice between several algorithms which implement the same function or between several ways of efficiently representing a given data type. This yields the following diagram (Figure 3.1) (SP_0 represents the initial specification; those steps in between SP_0 and P are represented by SP_1, SP_2, and etc.).

Let SL represent the corresponding specification language for the specification and PL the programming language. Thus, languages needed for software development are shown in Figure 3.2.

$$\begin{tikzcd} SP_0 \arrow[r] & SP_1 \arrow[r] & SP_2 \arrow[r] & \ldots \arrow[r] & P \end{tikzcd}$$

Figure 3.1: Stages of Program Development

If each individual refinement step ($SP_0 \longrightarrow SP_1, SP_1 \longrightarrow SP_2$ and so on) can

$$SL_0 \longrightarrow SL_1 \longrightarrow SL_2 \longrightarrow \ldots \longrightarrow PL$$

Figure 3.2: Languages for Software Development

be proved correct, the P itself is guaranteed to be correct. Each of these proofs is orders of magnitude easier than a proof that P itself is correct since each refinement step is small.

As described above, we are interested in answers to questions such as "How to represent the process?"; "What does refinement mean and under what circumstances is a refinement step correct?" and "What methods are available for proving the correctness of refinement steps?". In the remaining parts of the chapter, formal specification, program transformation and program verification tools are reviewed in the context of the above questions. We then address the relevance of these approaches to reverse engineering. Finally, several software maintenance projects involving reverse engineering will be introduced.

3.2 Formal Specification

3.2.1 Specifications

A specification of a software system may serve different purposes [90].

- Specifications are used for program documentation.
- Specifications serve as a mechanism for generating questions. The construction of specifications forces the designers to think about the requirements definition and the intrinsic properties and functionalities of the software system to be designed.
- A specification can be considered as a kind of contract between the designers of a program and its customers (in the commercial world, vendors and customers).

- Specifications are a powerful tool in the development of a program during its software life cycle. The presence of a good specification helps not only designers, but also implementors and maintainers.
- With regard to program validation, specifications may be very helpful to collect test cases to form a validation suite for the software system.

Important properties of a specification are:

- completeness — the specification must cover the functionality of the requirement, and
- consistent — the specification does not contain internal contradiction. A specification which is to be implemented must not be inconsistent (or else it cannot be implemented).

Any specification must be expressed in some specification language. Usually language sheds considerable light on a system's abilities. Although some systems are conceptually independent of a particular language, each implementation is in the end tied to a particular language.

There are compilers for *low-level languages* (e.g., assembly language) and *high-level languages* (e.g., C, BASIC, PASCAL, LISP, etc.) [120]. They allow us to write components of "programs", which suggest *how* the desired result is to be computed [75]. This is contrasted with a *specification language* which is a description giving details of *what* is required and no more. A specification language is mainly used to write the specification of the requirements of the software system.

Specification languages may be classified into two major classes: formal specification languages and informal specification languages [17,72,86,87,127].

Formal specifications have a mathematical (usually formal logic) basis and employ a formal notation to model system requirements.

The advantages of using formal specification [146] are as follows:

- The development of a formal specification provides insights into and understanding of the software requirements and the software design.

- Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specification. Thus, the absence of certain classes of system error may be demonstrated.

- Formal specifications may be automatically processed. Software tools can be built to assist with their development, understanding and debugging.

- Depending on the formal specification language used, it may be possible to animate a formal system specification to provide a prototype system.

- Formal software specifications are mathematical entities and may be studied and analysed using mathematical methods. In particular, can the system even be implemented adequately.

- Formal specification may be used as a guide to the tester of a component in identifying appropriate test cases.

Informal specification languages, on the other hand, use a combination of graphics and semiformal textual grammars to describe and specify system requirements. Given the graphical and "English-like" nature of these languages, they provide a vehicle for eliciting user requirements and communicating the analyst's understanding of the requirements back to the user for verification.

The two approaches have complementary strengths and weaknesses. Whereas informal specifications have advantages for requirements elicitation, ease of learning and communication, formal languages provide conciseness, clarity and precision, and are more suitable for analysis and verification. Therefore, formal and informal specifications must not be regarded as competitive but rather as complementary.

However, the use of formal specifications is the most distinguishing feature of a formal method. The term *formal methods* is used to cover the use of mathematics in software development. The main activities are [84,95]:

- writing a formal specification,

- proving properties about the specification, e.g., its consistency,
- constructing a program by mathematically manipulating the specification, and
- verifying a program by mathematical argument.

In fact, formal methods are all about specifications. Formal methods are used in the thesis for undertaking reverse engineering, so that the key issue of using formal methods — formal specification languages — will be reviewed in the next section.

3.2.2 Algebraic Specification Languages

There exist three basic families of specification approaches:. the algebraic, the state-machine, and the abstract model [24,108].

The approach of algebraic specification languages is based on the concept of *abstract data type* (ADT) [85,90]. The idea (originated by Guttag [82]) is that for specification purposes a functional program can be modelled as a *many-sorted algebra*, i.e., as a number of sets of data values (one set of values for each data type) together with a number of operations on those sets corresponding to the functions in the program. The many-sorted algebra is needed because many interesting operations in computing involve more than one sort, e.g., *equals*: Int × Int → Boolean. An **abstract data type** is a class of many-sorted algebras with *same* signature and same specified common properties. An *algebraic data type* is a definition of an abstract data type by a *signature* and some axioms. This abstracts away from the algorithms used to compute the functions and how those algorithms are expressed in a given programming language, focusing instead on the representation of data and the input/output behaviour of functions. It is possible to extend this paradigm to handle imperative programs as well by modelling imperative programs as functional programs or else by using a different notion of algebra [137]. The original motivation for this work was to provide a formal basis for the use of data abstraction in program development.

In this approach, a specification consists of a *signature* — a set of *sorts* (data type names) and a set of function names with their types — together with a set of equational axioms expressing constraints which the functions must satisfy. For example [24], Figure 3.3 is an algebraic specification of a bounded stack with a bounded size of three.

The *sort* part lists the abstract data types being described. In this example there is only one type, namely Stack. The *operators* part lists the services available on instances of the type Stack and syntactically describes how they have to be called. These parts are called the *signature* of the algebraic specification. The *axioms* part formally describes the semantic properties of the algebraic specification.

The basic idea of algebras is to write down a set of *key properties* of the ADT in terms of *equations* (equational logic). We want a minimum set of such properties (no duplication etc.). These are the axioms, and this allows an *axiomatisation* of the theory. From the *axioms* and rules of inference we can generate any valid formula.

There are two basic approaches to semantics, *initial algebra* (e.g., OBJ) and *loose algebra* (e.g., CIP-L).

Algebraic specification techniques can also be used in a *wide-spectrum language* [18,90,137,139], which is viewed as a specification language when used to write software specifications. A *wide-spectrum language* incorporates a variety of constructs, from high-level specification constructs down to low-level machine-oriented ones, to permit expression of broad range of the stages in the development of a program. Furthermore, in the intermediate steps of transformation, which is done incrementally, it is natural for specification constructs to be mixed freely with programming constructs because of the way that high-level specification are gradually refined to programs. This also avoids various problems which arise when separate specification and programming languages are used: there is no essential difference between refinement of programs and refinement of specifications; the same modularisation constructs can be used to construct specification as well as programs; there is no sudden leap from one notation to another but rather a

```
1. obj Stack  { basic object in OBJ, which corresponds to an abstract data type. }
2. sort Stack/integer, boolean; { new type definition, old type used following the "/".}
3. ok-ops
4.    push: Stack, integer -> Stack; { "underline sign" denotes a key word of the language or a known type.}
5.    pop: Stack -> Stack;
6.    top: Stack -> integer;
7.    empty: Stack -> boolean;
8.    newstack: -> stack;
9.    depth:stack -> integer; hidden; { hidden function is not accessible to an abstract program.}
10. error-ops
11.    underflow -> stack;
12.    no_more -> integer;
13.    overflow -> Stack;
14. ok-eqn's    {these are axioms.}
15.    pop(push(s, item)) = item;
16.    top(push(s, item)) = item;
17.    empty(newstack) = true;
18.    empty(push(s, n)) = false;
19.    depth(newstack) = 0;
20.    depth(push(s, item)) = 1 + depth(s);
21. error-eqn's
22.    pop(newstack) = underflow;
23.    top(newstack) = no_more;
24.    push (s, item) = overflow if depth(s) > 2;
25. jbo
```

Figure 3.3: A Stack Specification in an Algebraic Specification Language (OBJ)

gradual transition from high-level specification to efficient program.

A brief review of the main existing algebraic specification languages is now given:

CLEAR The specification language CLEAR [40,137] provides a small number of specification-building operations which allow large and complicated specifications to be built in a structured way from small, understandable and reusable pieces. The operations provide ways of combining two specifications, of enriching a specification by some new sorts, function and axioms, of renaming and/or forgetting some of the sorts and functions of a specification, and of constructing and applying parameterised specifications.

The semantics of CLEAR allows it to be used with different kinds of axioms (not just equations) to specify different kinds of algebras. This allows appropriate treatment of exceptions, non-terminating functions and imperative programs, among other things.

CIP-L CIP-L [18] is the language on which the CIP project was based. CIP-L is a wide-spectrum language which includes constructs for writing high-level specifications, functional programs, imperative programs and unstructured programs with gotos.

The language provides constructs for the specification and implementation of data structures as well as constructs for the specification and implementation of control structures. Algebraic (data) types provide a means for giving the algebraic specification of data. They can be implemented by computation structures combining data and algorithms. Modes are described by specific types for which computation structures can be provided automatically. Based on algebraic types and/or computation structures, program can be specified using predicate logic, description, comprehensive choice, and fully typed set operations.

Larch The Larch [83,137] family of specification languages was developed at MIT and Xerox PARC to support the productive use of formal specifications in programming. Each Larch language is composed of two components: the *interface* *language* which is specific to the particular programming language under consideration and the *shared language* which is common to all programming languages. The interface language is used to specify program modules using predicate logic with equality and constructs to deal with side effects, exception handling and other aspects of the given programming language. The shared language is an algebraic specification language used to describe programming-language independent abstractions using equational axioms which may be referred to by interface language specifications. The role of a specification in the shared language is to define the concepts in terms of which program modules may be specified.

Other algebraic specification languages [90,137,138] are ACT ONE, OBJ family [78], HOPE, etc.

To summarise, the strengths of the algebraic approach are:

- theory now well established,
- structuring techniques for large systems introduced, e.g., parameterisation,
- seemingly promising for transformation systems, refinement and reuse,
- some tools becoming also available,
- some theorem provers also available;

and the weaknesses:

- still at laboratory stage,
- needing considerable mathematical ability,
- not accessible to practitioner, and
- difficult to scale.

3.2.3 State-Machine Specification Languages

A state-machine specification defines a set of functions that specify transformations on inputs. The set of functions may be viewed (depending on the particular

```
module Stack;
declarations integer index, item; boolean; flag
functions
   vfun h_depth -> index;
      hidden;
      initially index = 0;
   vfun h_set_of_items (index) -> item;
      hidden;
      initially item = ?;   { “?” means “undefined”. }
   ofun push (item);
      exceptions h_depth > 2;
      effects 'h_set_of_items (h_depth) = item;
              'h_depth = h_depth + 1;
   ofun pop;
      exceptions h_depth < 0;
      effects 'h_depth = h_depth - 1;
   vfun top () -> item;
      exceptions h_depth < 0;
      derivation item = h_set_of_items (h_depth - 1);
   vfun empty () -> flag;
      derivation flag = (h_depth = 0);
end module;
```

Figure 3.4: A Stack Specification in a State-machine Specification Language

specification) as defining the nature of an abstract data type or describing the behaviour of an abstract machine. A state-machine specification is given in terms of states and transitions. Its functions are divided into two classes: *V-functions* allow an element of the state to be observed but do not define any aspect of transitions; *O-functions* define transitions by means of *effects*. The effect of an O-function is to change the state, which is done by denoting a V-function and altering the value it will return. Specification languages based on this approach include SPECIAL and INA JO [24]. The example in Figure 3.4 is in SPECIAL.

3.2.4 Abstract Model Specification Languages

The model based approach describes the key objects in terms of *foundational objects* which we assume exist (are given). We use the foundational objects plus construction operations to build compound objects which model our system. A *model* (here) is a mathematical theory expressing the aspects of the system we wish to describe and analyse. A model will often be at quite a high level of abstraction, ignoring much of the unnecessary detail.

A model (for specification) comprises:

1. a state space

2. operations, functions on this space

3. state invariant — defines *valid states.*

The basic types are typically *sets, cartesian products, sequences, schemas* (in Z), etc. We should (like any mathematical theory) be able to deduce interesting *theorems* about the system. The abstract model technique [24,108] differs in both syntax and semantics from the techniques of previous two method. For syntax it uses the basic precondition/postcondition format. It defines its function in terms of an underlying abstraction that is selected by the specifier. The specifier can use any abstraction (sets, lists, arrays, and so on) about which it is possible to reason formally. The usefulness of a given abstract model specification depends greatly upon the appropriateness of the selected underlying abstraction to the functions being specified. In order to illustrate the relationship between an abstract model specification and the underlying abstraction, a bounded integer stack is specified in terms of arrays (Figure 3.5 and Figure 3.6).

Two widely used specification languages, VDM and Z, belong to this approach.

VDM VDM [93] (the Vienna Development Method) is a method for *rigorous* (not formal) program development, and also a modelling notation. The objective is to produce programs by a process similar to the procedure in which the individual refinement steps are shown to be correct using arguments which are formalisable rather than formal, thus approximating the level of rigour used in mathematics. This is supposed to yield most of the advantages of formal program development by ensuring that sloppiness is avoided without the foundational and notational overhead of full formality.

```
obj iarray
sort iarray/integer;
ok-ops
   new: -> iarray;
   assign: integer, iarray, integer -> iarray;
   read: iarray, integer -> integer;
error-ops
   empty -> integer;
ok-eqn's
   read (assign (val, array, index 1), index 2)
      if index 1 = index 2
        then val
        else read (array, index 2);
error-eqn's
   read (new, index) = empty;
jbo
```

Figure 3.5: An Array Specification in an Algebraic Specification Language

```
type stack;
stack is modeled as iarray and (depth: integer);
invariant 0 ≤ depth ≤ 2;
initially stack = new and depth = 0;
functions
   push (item: integer)
      pre 0 ≤ 2;
      post stack′ = assign (item, stack, depth) and depth′ - depth + 1;
   pop
      pre stack ≠ new;
      post depth′ = depth - 1;
   top returns (item: integer)
      pre stack ≠ new;
      post item = read (stack, depth - 1);
   empty returns (flag: boolean)
      post flag = (stack = new);
end;
```

Figure 3.6: A Stack Specification in an Abstract Model Specification Language (based on pre- and post-conditions)

VDM uses a model-oriented approach to describing data types. Models are built using functions, relations and sets. A simple example is the following specification of dates:

Date :: Year : Nat MONTH: {Jan, Feb, ..., Dec} DAY: { 1:31 }

This models dates as triples, but does not require that dates be represented as triples in the final program. Not all of the values of type Date are valid; the legal ones are characterised by the following *data type invariant*:

$$
\begin{aligned}
&\text{inv-date} (< y, m, d >) =_{def} \\
&\quad (m \in \{\text{Jan, Mar, May, Jul, Aug, Oct, Dec}\} \Rightarrow 1 \leq d \leq 31) \wedge \\
&\quad (m \in \{\text{Apr, Jun, Sep, Nov}\} \Rightarrow 1 \leq d \leq 30) \wedge \\
&\quad (m = \text{Feb} \wedge \neg\ \text{is-leap-year}(y) \Rightarrow 1 \leq d \leq 28) \wedge \\
&\quad (m = \text{Feb} \wedge \text{is-leap-year}(y) \Rightarrow 1 \leq d \leq 29)
\end{aligned}
$$

Problems with VDM are that it is easy to overspecify a system and that it may bring side effects when pre- and post-conditions are used to specify procedures [126].

Z Z [119,147,148] is a specification language based on the principle that program and data can be described using set theory just as all of mathematics can be built on a set theoretic basis. Thus, Z is no more that a formal notation for ordinary naive set theory. The first version of Z used a rather clumsy and verbose notation but the current version adopts a more concise and elegant notation based on the idea of a **schema** which generalises the sort of thing behind mathematical notations.

Schemas are used to describe a system by both static aspects (the states it can occupy; and the invariant relationships that are maintained as the system moves from state to state) and dynamic aspects (the operations that are possible; the relationship between their inputs and outputs; and the changes of state that happen) [148].

A simple example of Z specification, *A Birthday Book*, is cited here. It is a system for recording people's birthdays, and is able to issue a reminder when the day comes. A schema is used to describe the *state space* of the system.

BirthdayBook

$Known : \mathbb{P}\ NAME$

$birthday : NAME \rightarrowtail\!\!\!\!\!\to DATE$

$Known = \operatorname{dom} birthday$

One of the operations on the system, *adding a new birthday*, is described by another schema.

AddBirthday

$\Delta BirthdayBook$

$n? : NAME$

$d? : DATE$

$n? \notin known$

$birthday' = birthday \cup \{n? \mapsto d?\}$

Z has been used with success in UK industry to specify real systems. The main problems are, for example, that the Z language is hard to be used for the purpose of theorem proving and program refinement.

3.2.5 A Comparison of the Approaches

The similarities among the three approaches are that they are all nonprocedural and use sets of function definitions to specify the effect of operations in terms of known mathematical objects [24]. Each approach also has its own feature. For instance, in an algebraic approach, it is difficult to get the equations right for complex systems; and in the modelling approach, basic and well understood concepts (e.g., sets) are used so that it is more natural to specify objects in terms of such well understood mathematical objects.

State-machine and abstract model techniques, which both rely on explicit state descriptions, can be transformed into each other; a set of functions defined algebraically (and therefore side-effect-free) can always be transformed one-for-one into functions in either of the other two that permit side effects, but a function with side effects may have to be split into several visible and hidden functions when the algebraic approach is taken.

It was interesting that at the Refinement conference in January 1992 [54], almost all the papers were on the modelling approach, while none was on the algebraic approach.

3.3 Program Transformation Systems

3.3.1 Refinement and Transformational Programming

The term *refinement* has been referred earlier in this thesis. We now discuss more details of refinement. We describe **refinement** as a technique to produce correct implementations from specifications [161,163]. From this, we know that specification and implementation are two essential elements in the refinement process.

For most approaches of program development (whether formal or informal), a design stage is involved. Refinement takes the notion of a rigorous treatment of design a stage further. Each time a design decision is taken, a new version of the specification, incorporating this new information, is produced. We can now check that the new specification is acceptable with respect to the previous one, or more formally, that it "satisfies" it. Additionally, this new specification provides the basis for further refinement, so we can also ensure that our separate decision interact correctly. This notion of refinement also gives us a framework in which to consider the enhancement of the functional constraints discussed in requirements document. As we make decisions we can then check that this new information is consistent with satisfying the constraints, and finally we can check that the implementation does indeed satisfy them.

Refinement can be carried out informally or formally. Figure 3.1 presented a general picture of formal program development in which programs were evolved from specifications in a gradual fashion via a series of refinement steps. Probably the most useful potential application of formal specifications is to the formal development of programs by gradual refinement from a high-level specification to a low-level "program" or "executable specification" [70,94,140]. Actually, some refinement steps are more or less routine. Such refinement steps can typically be described schematically as transformational rules. The process of changing a program (specification) to a different program (specification) with the same semantics as the original program (specification) is called **program transformation**.

Any refinement obtained by instantiating a transformation rule will be correct. Rather than proving correctness separately for each instantiation, the rule itself can be proved correct and then applied as desired without further proof. Sometimes such a rule will be correct only provided certain conditions are met by the program fragments matching the schematic variables or by the context in which the rule is applied; in this case the proof obligation is reduced to checking that these conditions are satisfied.

This led to a method of program construction — **transformational programming**, i.e., to construct program by successive application of transformation rules. Usually this process starts with a formal specification and ends with an executable program.

Much recent work has been focused on the program transformation as one kind of programming paradigm in which the development from specification to implementation is a formal, mechanically supported process. Research on program transformation aims at developing appropriate formalisms and notations, building computer-based systems for handling the bookkeeping involved in applying transformation rules, compiling libraries of useful transformation rules, developing strategies for conducting the transformation process automatically or semi-automatically. The long range objective of this paradigm is dramatically to improve the construction, reliability, maintenance, and extensibility of software.

An implemented system for supporting transformational programming is called a **program transformation system**. Those languages designed with some of the techniques used in expressing transformations and developments are called **transformation support languages**. Researchers have built a number of systems for transformational programming.

3.3.2 Features of Transformation Systems

To compare the various systems [67,129,130], we consider the following aspects:

- Purpose

  Generally speaking, transformation systems are built to experiment with the mechanically assisted development of a broad range of programs.

  A first goal of program transformation is *program synthesis*: the generation of an equivalent, executable, and efficient program from a (formal) description of the problem. Program synthesis may start from specifications in (restricted) natural language, or from mathematical specification. It is correct with respect to the specification.

  The second goal is *general support for program modification*. This includes: optimisation of control structures, efficient implementation of data structures, and the adaptation of data structures and given programs to particular styles of programming (e.g., applicative, procedural, machine oriented).

  A third goal is that of *program adaptation* to particular environments. For example, a program written in one language may need to be adapted to a related language with different primitives.

- Functions

  1. Transformation data base

     The system consists of a facility for keeping the (predefined) collection of transformations for use by the end user.

  2. User Guidance

     Nearly all transformation systems are *interactive*. Even the "fully automatic" ones require an initial user input and rely (interactively) on the user to resolve unexpected events. The system's reaction to input may include automatic checks on the "reasonableness" of given commands, as well as incremental interactive parsing using correction mechanisms.

3. History recording

   Most systems also have some facility for *documenting the development process* — one of the promising aspects of the transformational approach. These facilities include internal preservation of the source program, of final output, and of all intermediate versions. The documentation itself ranges from a simple sequential log of the terminal session (bookkeeping) to rather sophisticated database mechanisms.

4. Assessment of Programs

   Assessment of programs can be supported in qualitatively different ways: the system may incorporate some execution facility, such as an interpreter or a compiler to some target level, or it may utilise aids for "testing", such as symbolic evaluators. Occasionally the system will also have tools for program analysis, either for aiding in the selection of transformation rules or simply for "measuring" the effect of some transformation.

- Working Mode

  1. A *"manual" system* makes the user responsible for selecting and applying every single transformation step. It is the simplest implementation and the system must provide some means for building up compact and powerful transformation rules. System checks application of use.

  2. A *fully automatic system* enables the selection and appropriate rules to be determined completely by the system using built-in heuristics, machine evaluation of different possibilities, or other strategic consideration.

  3. A *semi-automatic system* works both autonomously for predefined subtasks and manually for unsolvable problems.

- Type of transformation

Basically, there are two different methods for keeping transformations in the system: the *catalogue approach* and the *generative set approach.*

A *catalogue of rules* is a linearly or hierarchically structured collection of transformation rules relevant for a particular aspect of development process. Catalogues may contain, for example, rules about *programming knowledge*, optimisations based on *language features*, or rules reflecting *data domain knowledge.* A user can select certain transformation rules from the catalogue and apply the selected transformation.

By a *generative set* we mean a small set of powerful elementary transformations to be used as a basis for constructing new rules. A user can decide what transformation rules are to be constructed from the generative set.

To judge whether a transformation system is good eventually depends on the extent to which it can fulfill the goal — transforming a specification to a running program. However, it is not the only purpose of this review, and a more important aspect is to learn what can be used in undertaking reverse engineering.

3.3.3 Program Transformation Systems

In this section, the features of transformation system listed in last section are used to comment on existing transformation systems used in forward engineering according to available information. Other information may include, e.g., the year that a work was done, the specification and programming languages used, the result, etc.

Optimising Compilers Program transformation techniques have been used for many years in optimising compilers, because inefficient programs can be transformed into efficient programs (e.g., loop induction, strength reduction, expression reordering, symbolic evaluation, constant propagation, loop jamming).

Burstall and Darlington's Work The work on program transformation by Burstall and Darlington was done in the mid-1970's [39,130]. Their system was based on schema-driven method for transforming applicative recursive program into imperative ones with improving efficiency as the ultimate goal. The system worked largely automatically, according to a set of built-in rules, with only a small amount of user control. The rule set contained only seven simple rules and the system could only work on simple programs.

Balzer's Work Balzer built an implementation system for program transformation [14,15,16]. This system was designed mechanically to transform formal program specifications into efficient implementations under interactive user control. He expressed the problem by a formal specification language GIST, which was operational (i.e., having an executable semantics). He used this system to solve a small (but nontrivial) example, the "eight queens" problem. The result was that the optimisation and the conversion of the program into conventional form remained incomplete. His system depended too much on user guidance, and the specification was also not at a high level.

ZAP Feather's ZAP system [68] is based on the Burstall/Darlington system with a special emphasis on software development by supporting large-program transformation. The input/target language of the system is NPL (an applicative language for first-order recursion equations). The system provides the user with a means for expressing guidance. An overall transformation strategy is hand-expanded by the user into a set of transformation tactics such as *combining, tupling, generalisation.*

It is claimed that ZAP system can deal with example programs ranging from "toy" to small but realistic ones. Unfortunately, the system has to operate partially informally and even entirely by hand.

DEDALUS System The DEDALUS system (DEDuctive Algorithm Ur-Synthesiser) by Manna and Waldinger was implemented in QLISP [111]. Its goal was to derive LISP programs automatically and deductively from high-level input-output specifications in a LISP-like representation of mathematical-logical notation.

The system incorporates an automatic theorem prover and includes a number of strategies designed to direct it away from rule applications unlikely to lead success. The system is considered by its designers to be a laboratory tool rather a practical tool. The examples being treated by DEDALUS system were toy examples, like the greatest common divisor of two numbers.

The DRACO System The DRACO System [130] is a general mechanism for software construction based on the paradigm of "reusable software". "Reusable" here means that the analysis and design of some library program can be reused, but not its code. DRACO is an interactive system that enables a user to refine a problem, stated in a high level problem domain specific language, into an efficient LISP program. The DRACO ideas have been implemented in a prototype system running under TOPS-10 on a DEC PDP-10 computer. Results show that only small programs (tens of lines) can be created using this prototype. The main reason of this considered by the designer was restricted by memory size.

CIP-S CIP-S is the approach of the Project CIP (computer-aided, intuition-guided programming) [19], which is to develop along the idea of transformational programming within an integrated environment, including methodology, language, and system for the construction of "correct" software. The system uses a wide-spectrum language, CIP-L (introduced in section 3.2.2).

A prototype system has been implemented. The system is interactive and the development process is guided by the programmer who has to choose appropriate transformation rules. The system is language-independent and is based on the algebraic view of language definition; any algebraically defined language is suited for manipulation, provided respective facilities for translating between external and internal representations are available.

It is claimed that the system not only allows the treatment of concrete programs, but also the formal derivation of new, complex rules within the system. The CIP project has developed several theories for program transformation, such as well-founded theories of nondeterminism, abstract data type, algebraic language definition, and correctness of transformation rules and the CIP-L language turned out to be successful both as an educational vehicle in teaching beginners, and as a tool in developing software. It is also noted that the prototype served as the essential software tool in developing CIP-S itself. However, the prototype system can only deal with programs of a small scale and the proposed system (CIP-S) itself is under construction.

Other transformation-related system include the "SETL System" [59], the "TAMPR System" [32], the "FOCUS System" [135], and Morgan's work on the Refinement Calculus [121,122], the "Programmer's Apprentice" [158], etc.

3.3.4 Summary

There is widespread demand for safe, verified, and reliable software. This demand arises from economic considerations, ethical reasons, safety requirements, and strategic demands. Transformational programming can clearly make a valuable contribution toward this goal. It already covers several phases of the classic software engineering lifecycle and shows promise of covering the remaining ones. But, after near twenty year's research, existing transformation systems are still experimental and the problems they are capable of coping with are still more or less toy problems. To make practical use of transformation systems is no doubt the key problem to be solved in transformational programming.

3.4 Program Verification

Program verification is used when the program already exists and has been developed by informal development methods. It is contrasted with the correctness of software developed by transformational programming, which is guaranteed by the process itself (assuming that the transformations are correct).

3.4.1 Concept of Proof (Program Proving)

The concept of "correctness" is a relative notion. When we refer to a program being correct we mean relative to some given *specification*. Generally, the specification of a programming problem consists of a **precondition** describing the properties of the supplied data and a **postcondition** describing the desired effect of the computation. There may also be a state invariant defining valid states.

A proof of *conditional* or *partial correctness* assumes that the execution of a process terminates and concentrates on establishing that its specification is met. A complete proof also includes a proof of termination, e.g., by showing that some variable decrements on each loop down to a test on zero.

When the word "proof" is used it can generally be understood in two different ways. An **informal proof**, the sort most commonly used by mathematicians, consists of an outline of, or a strategy for constructing, a formal proof. A **formal proof** is a sequence of statements, each of which is a well-established theorem or which follows from earlier statements by a process (an inference or axiom). A formal proof is conducted in an artificial (or "formal") language consisting entirely of signs and symbols; a mathematician's proof, on the other hand, will make significant use of natural language (such as English) as well as sign and symbols where they are considered appropriate. Both types of proof have their own characteristic type of complexity.

There are two basic approaches to program verification, one using inference rules originally developed by C.A.R. Hoare [88] and the other using so-called "predicate transformers" developed by E.W. Dijkstra [60]. The two approaches are related, although different.

Proofs are a central part of the program development method. One property of a formal specification is that proofs can be written which clarify its consequences. In order for proofs to be useful, they must possess a number of properties. One of these requirements is that the proofs should be natural and that they should ensure certainty.

It is difficult to be precise about what constitutes a natural proof. The concept of informal proof is to indicate how a proof could be constructed: the major steps are given in the knowledge that further details can be provided if these major steps are in doubt.

Another aspect of what constitutes a natural proof concerns the crucial distinction between the discovery and presentation of a proof. A proof is often found by working back from the goal; sub-goals are created and discharged until the sub-goals correspond to known facts. In order to show how the steps relate, it is normal to present an argument working forwards from the known facts towards the goal. This forward presentation is more natural to read. But when readers become writers, they must learn to discover proofs one way and present their steps in a different order.

It should be clear that the claim that something has been proved must eliminate doubt. Unfortunately, informal arguments cannot create certainty. In order to achieve the same level of certainty with a proof, it is necessary to reduce proof construction to a "game with symbols": each proof step must depend only on known (i.e., proven) theorems or axioms and be justified by one of a fixed set of inference rules. The inference rules themselves must require only the mechanical rearrangement of symbols. Such proofs are called **formal proofs**.

A formal proof uses formal semantics of a programming language. The formal semantics of a programming language maps every syntactically correct language construct into a metalanguage that is based on a well-understood mathematical notation. Consequently, formal semantics can be specified as a set of translation rules from the domain of language constructs to the range of well-formed formulas of the formalism.

The most important benefit of formal semantics is that it produces the basis for correctness proof of implementation and basis for program correctness proofs. The formal semantics has the potential of providing mechanical support to correctness proofs. The only way for a computer to aid in verification of a language implementation or the correctness of a program is to start from a precise, formal language definition.

There are two kinds of formal semantics: axiomatic semantics and denotational semantics. Axiomatic semantics describes the meaning of each syntactically correct program by associating it to properties of variables (in terms of predicate calculus) that hold before execution starts and after the program halts. Axiomatic semantics is based on mathematical logic. In the axiomatic semantics approach [24] of program verification, the metalanguage used is a logic language, such as predicate calculus. Please refer to the example in VDM section (Section 3.2.4).

Denotational semantics [4,10,80] defines the meaning of a program written in a language $\mathcal{L}$ by a mapping from the syntax of $\mathcal{L}$ to functions denoted. Denotational semantics of programming constructs of a programming language are defined by so-called *semantics valuation functions.* Semantics valuation functions map programming constructs to values (numbers, truth values, functions, and so on) that they denote. These valuation functions are usually defined recursively: the value denoted by a construct is given in terms of the values of its constituent parts, and an emphasis on the value denoted by the constituent parts gives the approach its name. In the denotational semantics of program verification, the metalanguage used is that of functional calculus (i.e., lambda calculus). For example, the following is the denotational semantics of decimal numbers of a simple language:

Syntax:

<number> ::= <number><digit>|<digit>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

We now define a valuation function V from the syntax to the functions denoted. We do this for each syntax rule:

V: Num -> Integer (Number is the language defined by the syntax above)

So

V[<number><digit>] = 10 * V[<number>] + V[<digit>]

V[0] = 0

V[1] = 1

...

V[9] = 9

For each sentence in this simple language, the above valuation defines the meaning. For instance, the number is 724:

V[724] = V[<number><4>]

= 10 * V[<number>] + V[<4>]

= 10 * (10 * V[7] + V[<2>]) + V[4]

$$= 10 * (10 * 7 + 2) + 4$$
$$= 724$$

which gives us the answer we expect.

Formal verification is the application of reasoning expressed in a mathematical formalism, i.e., a formal system such as *first order predicate logic.* Because formal reasoning programs involves a large amount of tedious symbol manipulation which is a perfect job for a machine; tools, such as theorem provers and program verifiers, have been built for this purpose.

3.4.2 Examples of Existing Program Verification Tools

Program verification tools give rise to a variety of different approaches to formal reasoning about a variety of different tasks, especially the programs. It is difficult to compare program verification systems directly [107], because the problems they have been designed to tackle are often quite different. The important properties of existing systems are listed below and attention is drawn to the following aspects.

- Object language — The logical language in which propositions are expressed and reasoned about will be called the *object language* of the system. The class of object languages supported is usually a major factor in determining the usefulness of a program verification tool to formal reasoning tasks.

- Theories — When using a theorem prover systematically and over a long period of time, it is important to be able to build up "theories", going from simple properties of data structures to deeper and stronger results about their relationship.

  A "theory" of a program verification tool is the fundamental principle on which the tool is based. A theory is specified by giving its "signature", a set of axioms and rules of inference. Usually, the basic specification of a theory module consists of: its name, the modules on which it is built, the new sorts it will use, the new operators it will use, a collection of axioms and derived facts (theorems, lemmas, etc.). Taking the Boyer-Moore theorem prover as an example, the theory of it is to prove theorems by induction, mainly in the style of proofs in elementary number theory.

- Automated deduction and user interaction — Once the object language and a theory are fixed and a conjecture is stated, the search for a proof begins. If a machine can be programmed to recognise the truth (i.e., provability) of certain conjectures, it is needed to set a *test* which, when applied to a conjecture, returns one of the following answers: "established", "open" or "contradictory". The test should be consistent with the object theory. There are two types of systems: automated and interactive. In an automated system, the test is generated by the system; in an interactive system, on the other hand, the test is usually generated with user involvement. Eventually, it is known that in principle there can be no way of determining whether or not a conjecture is provable. It will be up to the user, with the machine's help, to discover a proof.

A number of program verification tools are listed below as examples:

The Stanford Pascal Verifier The Stanford Pascal Verifier (SPV) [107,118], is a program which basically checks the correctness of the proof of a Pascal program. It was written in a version of LISP for use on the PDP 10 range of computers. The SPV basically attempts to automate the inductive assertions method. The first action of the verifier is to generate sets of verification conditions which have to be satisfied. In the second phase the verifier employs a theorem prover which takes as input the verification conditions which have to be established and also a set of rules which can be used by the theorem prover. The SPV system is basically interactive.

Unless some precautions are taken the system will eventually run into the problem of being unable to decide on the equivalence of two equivalent arithmetic expressions (this problem is in general undecidable).

Verifier's Assistant The Designer/Verifier's Assistant is a system that parses programs and specifications and generates and proves verification conditions. In addition it can provide an understanding of the kinds of structures that can be changed and added and the ways in which these interact [123]. To do this the verifier employs the use of the wide spectrum language Gypsy [79] in connection with knowledge-based techniques. This system pays more attention to reasoning about changes at the implementation level in terms of pre and post assertions on program modules, rather than addressing the maintenance of specifications and development histories. It is by concentrating on these latter aspects that we might gain a better insight into the consequences of maintaining both changes to documentation (expressed as a specification) and implementation.

LCF (Logic for Computable Functions) There are several version of LCF [107] (at Stanford, Edinburgh, Cambridge, etc.) and the Edinburgh version is probably the best-known and most used of them all. The fundamental principle of LCF is that new theorems can only be formed by applying inference rules to axioms and already-formed theorems. LCF is implemented as a cluster of abstract data types in the ML language: 'term' and 'form' for terms and formulae of the object language. Edinburgh LCF has virtually no proof management facilities: proof trees are not built; validations must be done by hand; and partial proofs are not stored. Multiple proof attempts are possible, but no help is given for organising them.

The Gypsy Verification Environment (GVE) GVE [107] is a highly integrated specification and verification environment, originally targeted at communications processing systems of 1000-2000 lines of code. It is one of the few development systems that can handle concurrency (which it does by message passing), and it is perhaps the only one that maintains dependencies between proof, (parts of) specifications and (parts of) programs. A particularly strong point is the high degree of unification of specification and programming constructs in the Gypsy language. The user can invoke a proof strategy which will apply rules automatically according to built-in heuristics. Very simple proofs can be finished without user intervention. This facility is a particularly heavy user of resources, and suffers from most of the problems of fully automatic theorem provers, without being particularly strong.

The Boyer-Moore Theorem Prover The Boyer-Moore Theorem Prover (BMTP) [30,31] takes programs written in pure LISP. Properties of these programs are also stated as expression which themselves are expressed in LISP. The theorem prover then attempts to show that the program possesses the desired properties. It does this by applying simple heuristics and also structural induction - LISP programs tend to be heavily recursive. The theorem prover could be used as a verification tool to establish properties of programs about as complex as a sorting procedure but not much more.

Though the Boyer-Moore Theorem Prover is viewed as one of the successful systems for mechanical program verification, there are still some problems. The user supplies a conjecture with which BMTP tries to prove from axioms and already-proven results, but without direct assistance from the user. The object language is very expressive, but its type structure is very limiting and probably not suited to many applications. It incorporates some very effective heuristics for inductive proofs; unfortunately they seem to be inextricably bound up with the object language, and it would probably be quite difficult to use them directly. The Boyer-Moore Theorem Prover is a heavy user of resources, e.g., memory and CPU time.

The Boyer-Moore Theorem Prover has been used in the REFORM Project [167] (see Chapter 8).

Other program verification tools include [96,107], the "Interactive Proof Editor", "AFFIRM", "Interactive Proof Editor", "NuPRL", "B Tool [5,115]", etc.

3.4.3 Summary

Generally speaking, the following points are weaknesses: for the languages, a wide range of symbols (especially mathematical symbols) should be available, and mixed operators should be allowed, so that programs at higher level of abstraction (specifications) can be taken into the verification process; to the theories, they should also include many other things such as tactics, decision procedures, simplification methods, normal forms, and so on, but none of above systems have done this; to other aspects, the nature of finding proofs is to experiment, therefore, speed of response, help facilities, comprehensibility (especially for a novice) and so on are also crucial factors to be stressed. This is also the problem of writing correct mechanical theorem provers.

3.5 An Overview of the Main Existing Reverse Engineering Approaches Used in Software Maintenance Projects

The review in the previous section addressed obtaining programs from specifications. At present, there is increasing interest in the reverse direction — obtaining specifications from programs (i.e., reverse engineering). Reverse engineering is often one early part of a software maintenance project. This area is being researched in many projects in seeking a good method to achieve the goal — obtaining specifications from programs. This section will review several existing reverse engineering approaches in software maintenance projects. This will help to determine the advantages as well as disadvantages of these approaches and will help the further development of the REFORM project. The REFORM project is introduced very briefly in this section and will be described in detail in a later section.

Reverse engineering techniques are of two kinds: a maintainer-driven design recovery and a knowledge-based design recovery. In the first approach, the maintainer is facing a set of tools without any assistant guidance; in the second approach, tools rely only on syntactic level code analysis. Two kinds of assistance are provided: (1) assistance in determining the functional intent behind a piece of code (automatic meaning recovery), (2) assistance in maintenance process guidance. In both techniques, the tools can be integrated or not integrated.

3.5.1 MACS

MACS (Maintenance Assistance Capability for Software) has as its objective the definition and implementation of a software maintenance assistance system [57, 73]. The basic premise of MACS is "maintenance through understanding". The main design objective of MACS is to aid the acquiring, ordering and exploring a structure representing facts and assumptions about the application to be maintained. In particular, MACS offers aid in the following areas:

- comprehension of the application design and development process: the WHY of the application,
- understanding and capture of the existent: the WHAT of the application, and
- assistance in maintenance actions: the HOW of maintenance actions in the context of the application.

The MACS project is developing an integrated tool-set which is built on a common repository. The "What" tool consists of two major parts called the Change Management World (managing the changes during the maintenance process) and the Abstraction Recovery World (providing filtered or abstracted views of the code using Dimensional Design). The "Why" tool is called the Reasoning World (for understanding the domain and design decisions) with a Domain Knowledge Base as its main component. Another major part of the tool-set is called the Interconnection World (for understanding inter-world relations and code semantic understanding).

According to available literature [58], the major achievements of the first two years (by January 1992) have been the design of the MACS architecture, the complete realisation of the Change Management system, the Reasoning World and the "syntactical" Abstraction Recovery World. The MACS project still uses informal analysis, though the need for merging informal and formal approaches is identified as a long-term plan.

MACS uses a conventional Change Management System for configuration management. Its representation for abstraction is a diagrammatic representation called "Dimensional Design". This is an enhanced flow chart on which sequencing is represented vertically, the components of loops and conditions horizontally, and procedural abstraction at 45°. A similar notation is used for data structures. Therefore, the abstract representation is at a relatively low level.

Of more interested is the representation of the "Reasoning World". This amounts to a very flexible data base in which the user can add information incrementally. For example, consider the situation occurs in which an error is found. Once the module is located, the user can, via a graphical WIMP interface, add an extra node giving free text information about the error, and then link it to the module. A new version of the module can also be created, and linked via the CMS.

Note that a great deal is left up to the user to enter information. The attraction of MACS lies in the power of the links; the user can navigate from a module to its source, to its dimensional design, to its version records etc. This is the integration mechanism. The tools on their own provide very little semantic interpretation — that is left to the user.

MACS is built on the IPSYS ECLIPSE system, which provides the repository mechanism and common front end user interface.

3.5.2 Reverse Engineering in REDO

REDO (Restructuring, Maintenance, Validation, and Documentation of Software Systems) is an ESPRIT II project, which started in 1989, and is concerned with "rejuvenating" existing applications into more maintainable forms by improving documentation, by restructuring code, and by validating the code against the original intentions. As one part of REDO project, reverse engineering (reverse-engineering COBOL programs into Z specifications) was carried out at Oxford University [33,34,35,100,101,102,103]. There are three stages in their process:

- clean — Translation to the intermediate language UNIFORM, eliminating redundant language constructs (for restricting the original language to a small subset of permissible constructs, because UNIFORM is more compact and COBOL semantics are woolly).

- specify — Using data-flow diagrams for guidance, associated variables are grouped together to create prototype *objects*, but as yet containing no list of associated *operators*. The code is also split into *phases* at this point. Equational descriptions of the functionality associated with these phases are obtained automatically and transcribed into the intermediate functional language, simplifying transforms being automatically applied in order to reduce the equational presentation to a *normal form*.
- simplify — The abstracted functional descriptions are incorporated into the outline objects as descriptions of their *operations*, thus filling in the semantics of the prototype objects identified at the first stage. A full specification (in the language Z or Z^{++}) is then printed out using the object-orientated abstraction as a basis, along with associated textual documentation.

The project was to develop a set of tools to serve as a useful aid in the comprehension of raw code, and also to transform it into a concise mathematical representation which can support further development work.

The strategy here is to perform abstraction first, and then perform transformation on the high level language. This will no doubt increase the degree of difficulty in the second stage, because the original code might not be structured and hence understandable at all. So far, only a few toy examples have been done by hand (about 40% of the COBOL language has been treated [102]) and have been presented by this project, and support for providing automated tools for program comprehension and the generation of technical documentation from software systems as part of maintenance still need to be done.

3.5.3 Sneed's Work

Sneed and Jandrasics use automated tools to support the retranslation of software code in COBOL back into an application specification by the process of reverse engineering [144]. Two steps are needed, to recover a program design from the source code and to recover a program specification from the program design.

In the first step, the code of COBOL programs is translated into an interme-

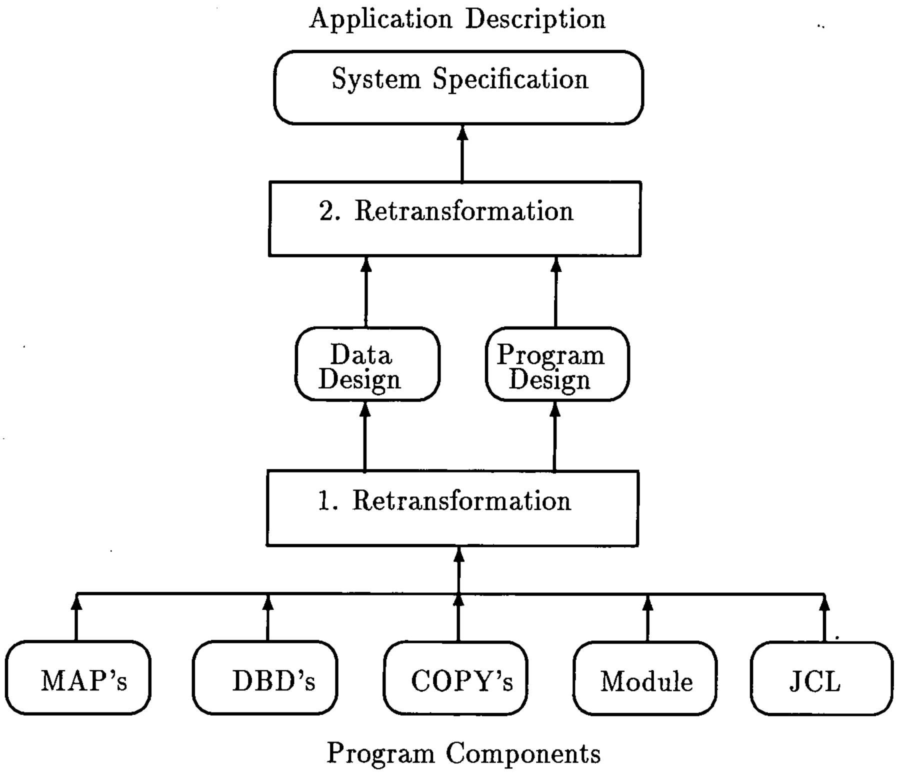


Figure 3.7: Inverse Transformation of Software from Code to Specification

diate design schema based on a set of normalised relational tables for the modules, data capsules, and interfaces extracted from the source programs (see Figure 3.7).

Secondly, two activities are carried out jointly in this step, i.e., data design recovery and program design recovery. The data design part contains five design elements: database structure design, file design, data communication design, data capsule design, and data constant design. The program design part also contains five parts: process structure design, component design, data flow design, module interface design, and module design.

A set of transformation rules for mapping COBOL source code back into the design schema is obtained by inverting those rules used to generate COBOL programs from the design. The programs are modularised and restructured as a by-product of the reverse transformation process.

In the second step, the intermediate design representation is retransformed into a specification schema based on the entity/relationship model. The detail of how the authors did this is not available.

Though the authors claimed "it is not only possible to retranslate programs into a program design but that it is also possible to retranslate a set of program designs into a system specification", the experiments that they carried out were mainly limited to a low level of abstraction and there is still work to do to reach high level of abstraction. It seems the authors did most of work by hand and have not developed a full system in accomplishing their ideas.

3.5.4 A CASE Tool for Reverse Engineering

Bachman introduced a CASE tool, DOCMAN, for reverse engineering COBOL programs [12]. The Re-Engineering Cycle chart (Figure 3.8) provides an architectural view of this CASE tool, which features both forward and reverse engineering. Particularly, reverse engineering begins at the bottom left with the definition of existing applications and raises the applications to successively higher levels of abstraction. At the top, the design objects created by the reverse engineering steps are enhanced and validated to become the revised design objects used in the forward engineering process. At the bottom, a new applications system becomes an existing applications system at the moment that it goes into production.

The following points are stressed by this philosophy:

- reverse engineering enables the CASE tool to extract business rules from old applications and use them as the basis for refurbishing and maintaining those applications,

- reverse engineering also involves the removal of optimisation mechanisms and implementation artifacts that were introduced in an earlier implementation of the application,

- it is impossible to reverse engineer a file, database definition, or program automatically, because some of the information essential to the task is not present in existing COBOL programs, and

- a reverse engineering product built as an expert system can work interactively with the professional user and identify the missing information, de-

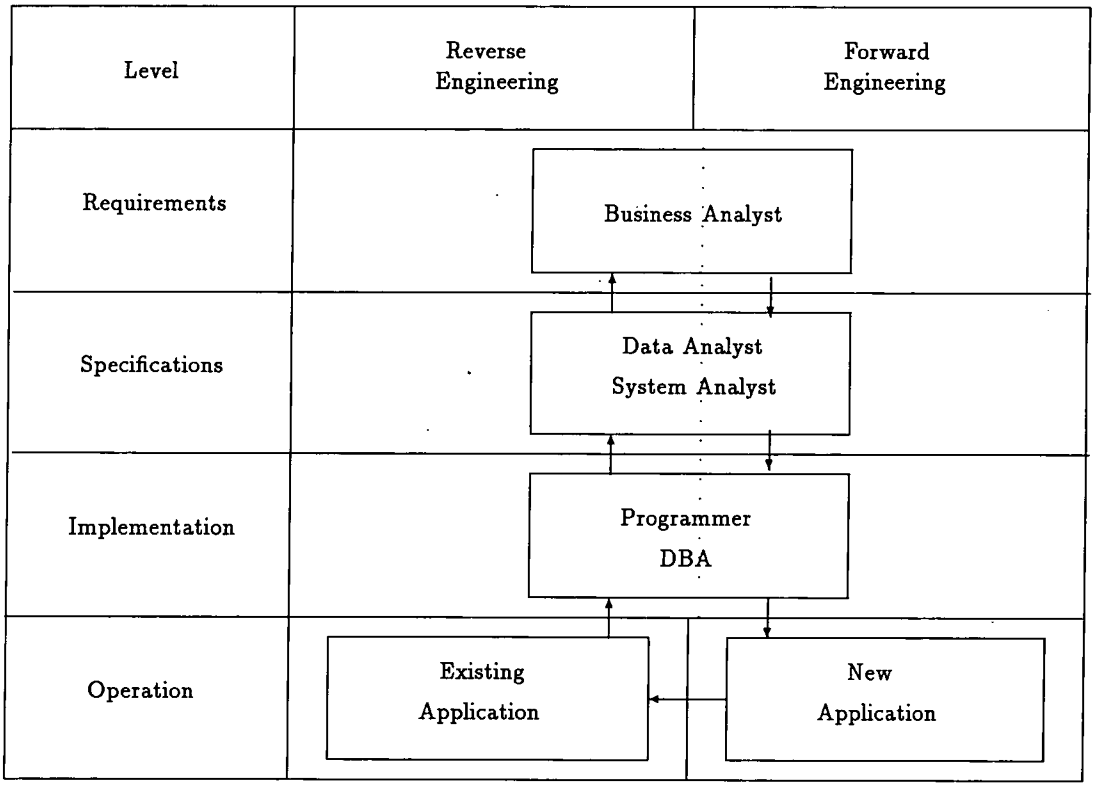


Figure 3.8: Re-Engineering Cycle

termine its nature, propose alternatives, and insert the user's choice where required to complete the process.

Because the CASE tool is commercial, which is not available at Durham, several questions about the usability of the tool remain unknown.

3.5.5 TMM

A method was proposed in [9] for recovering abstractions and design decisions that were made during implementation. This method is called *Transformation-based Maintenance Model* (TMM). The purpose of this system is to reimplement a system in order to adapt it to a new environment through reuse. The abstractions and design decisions of software must be recovered first before the software is reimplemented. The recovery work in TMM paradigm is done by *maintenance by abstraction* (MBA). Apart from working on the assumption that the documentation of the program exists, the TMM will also work assuming that the specification and refinement history of the program are not available, but a systematic approach must be used to recapture implementation knowledge before the TMM can be applied. Unfortunately the abstraction recovery was carried out manually and human experience plays a vital role. Tools need to be developed to aid this approach.

3.5.6 A Concept Recognition-Based Program Transformation System

This is an approach that applies a transformation paradigm to automate software maintenance activities [63]. The characteristic of this approach is its use of concept recognition, the understanding and abstraction of high-level programming and domain entities in programs, as the basis for transformations. Four understanding levels are defined: the text level, the syntactic level, the semantic level, and the concept level. The program transformation system depends on its program understanding capabilities up to the concept level. The key component is a concept library which contains the knowledge about programming and application domain concepts, and the knowledge about how these concepts are to be transformed. Concept recognition is done by pattern matching. This work is based on *program plans* (please also see Section 7.2.2). A program transformation tool has been developed to support the migration of a large manufacturing control system written in COBOL.

At present, the maintainer has to write correct and complete transformations. In the experiment, the system only contains 60 concept recognition and transformation rules. The result shows that the speed of the system need to be improved. Another problem is that the system does not provide a facility for the maintainer to guide the transformation, but this is necessary because complete automation of maintenance modifications is not always possible. It is also found out that the system still needs a browser to support the *high-level editing.*

3.5.7 REFORM

REFORM - Reverse Engineering using FORmal Methods - is a joint project between University of Durham, CSM Ltd. and IBM (UK) to develop a tool called the Maintainer's Assistant. The main objective of the tool is to develop a formal specification from old code. It will also reduce the costs of maintenance by the application of new technology and increase quality so producing improved customer satisfaction. The old code in this project is the IBM CICS. The aims of the Maintainer's Assistant are to provide a tool to assist the human maintainer, handling assembler and Z in an easy to use way. The Maintainer's Assistant system will be discussed in detail in later chapters.

Other systems may be seen in [162].

3.6 Summary

The purpose of the chapter is to discuss the state of the art in the area of acquiring a design/specification from program code in two aspects: the experience of forward engineering which is useful for reverse engineering (because acquiring a specification from an existing program covers many stages of software development), and the latest developments in reverse engineering.

Problems and lessons learned from forward engineering In the specification phase, formal specifications can be found in many applications and they have shown many advantages over informal specification languages. Some of the languages in previous section have been relatively widely used, such as VDM and Z, but they still have same problems. For example, they cannot meet the needs of representing all levels of abstraction in acquiring a specification from the program code; the specifications written in them cannot be easily verified formally, because it is not easy to integrate them with program verification tools. Probably, a wide spectrum language is better than other types of language when undertaking reverse engineering.

In the development phase, existing transformation systems have shown their potential power. These systems suffer from various problems. For instance, the CIP is one of the most representative projects of transformation system. It started in 1975. By 1989 when [19] was published, the conclusion is still ".. experiences ... strengthened our belief that transformational programming will become an important factor in software engineering." This implies that it will take a long time for the paradigm of transformational programming to become practical. In a general sense, the reviewed systems in the previous section almost all fall into at least one of the following categories: theoretical problems to be solved; only "toy" or comparatively simple example programs being experimented; and operations being carried out informally or even entirely by hand.

In the verification phase, we can identify a number of verification systems, but they are not built for the purpose of reverse engineering. The author has carried out experiments with the Boyer-Moore Theorem Prover [167] for building a supporting tool of a program transformer for the REFORM project. Experiences show that the Boyer-Moore Theorem Prover is a powerful verification tool. The disadvantages can also be seen: a user has to spend time trying to find the right intermediate lemmas for the prover; and the prover is a major consumer of resources. One idea is to use one of the program tools for reverse engineering at some stage of the research.

This suggests that the research of the thesis be carried out by using a formal or rigorous method, particularly using a formal language to represent both specification and program; developing a transformation system to transform specifications (or programs) into equivalent specifications (or programs), and using program verification tools to obtain transformation rules (including transformations for crossing levels of abstraction).

Problems and lessons learned from reverse engineering From the reviewed systems, we know that a great effort is still needed to put the paradigm of reverse engineering into practical use. It is particularly a hard job to reverse an existing program back to its design or specification. For instance, one of the problems with reviewed systems is that the availability and accuracy of the design information are both assumed. Actually, such information is typically obsolete or lacking in systems which have gone through years of maintenance. For such systems, source code is the only reliable source of information. Another problem is that there is not a method for coping crossing levels of abstraction covering all abstraction levels in these systems.

The state of the art in reverse engineering may be summarised as follows. Most existing commercial tools are basically restructurers, and these operate at the same level of abstraction.. Even module recovery tools, such as those in MACS or Sneed's work, operate at the syntactical level, e.g., grouping variables and operations on them. Where genuine crossing of levels of abstraction occurs, this is done manually, e.g., in Sneed's system for COBOL, or in redocumentation systems such as DOCMAN. The recent Refinement conference [54] is also a reference of the state of the art.

The most relevant work to this thesis is that of Breuer and Lano, and Bachman. Work is also being done on the business use of reverse engineering but this is only indirectly of relevance to this thesis.

It should pointed out that reverse engineering is still an activity of high risk and high cost from the management's point of view. It has been argued that for large systems deriving formally correct designs or specifications from existing source code is impracticable [36], because the importance of a design or specification as a model of the application domain is ignored, as well as a description of the code itself. This argument claimed that the reasons for unsuccessful reverse engineering include that the original design or specification might not exist at all and that the implementer of the software did not observe the design or specification (if existed). Nevertheless, these problems had been addressed by the project before the paper [36] was published. The aim of the project also includes finding out the possibility of dealing with large scaled software.

The author of this thesis would see acquiring designs and specifications for data-intensive programs as redesigning the original programs rather than seeking the (possible) original designs.

Chapter 4

Proposed Research Problem

As we have seen in previous chapters, acquiring a program design or specification from program code is important and significant in reverse engineering. The thesis takes this topic as its subject.

After studying the subject, problems which are related to this issue are identified in this chapter. The general area chosen is that of crossing levels of data abstraction to extract data design from existing code. The thesis focuses on data design recovery for *data intensive* programs — those whose computational complexity is low, but whose data complexity is high. Many COBOL programs are of this type.

Program design often starts with data and there are many data intensive programs existing now in the form of COBOL programs. The Entity-Relationship Attribute Diagram is one of the good forms of data design for a data intensive program; for example, Entity-Relationship Attribute Diagrams are used as the tool for representing data designs in Structured Systems Analysis and Design Method (SSADM).

The aim of the research is to tackle not just "toy" program code, but also real program code, including heavily maintained industry scale program code. The code is of modest scale, e.g., a few hundred lines or even up to a few thousand lines. Coping with a large scale program also needs further research, which has to be studied in the next step and is beyond the scope of the thesis. Real-time and concurrent programs are not considered, because the theoretical and foundation work is still in progress.

The features of data-intensive programs are first introduced in this chapter.

4.1 Features of Data-Intensive Programs

Design processes (SSADM in particular) in forward engineering are briefly reviewed in this section in order to understand the reverse direction and to understand features of data-intensive programs.

4.1.1 Software Design Process

Design is the process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realisation. It is the first step in the development phase for any engineering product or system [132].

Usually the design phase starts once software requirements have been established, regardless of the software engineering paradigm applied. From the technical point of view, design comprises three activities: data design, architectural design and procedural design.

The main goal of *data design* is to select logical representations of data objects (data structures) identified during the requirements definition phase. Data design is the most important design activity for some program classes among the three design activities, because well-designed data can lead to better program structure, modularity, and reduced procedural complexity, no matter which design technique was used.

The main goal of *architectural design* is to develop a modular program structure and represent the control relationship between modules. Furthermore, architectural design melds program structure and data structure and defines interfaces enabling data to flow throughout the program. The main goal of *procedural design* is to define algorithmic details after data and program structure have been established.

Effective software design is best accomplished by using a consistent design method. There have been a vast number of design methods developed and used in different application during last four decades. Essentially, most of these methodologies can be grouped into one of three categories [132,146]:

- *Data structure-oriented design.* This method is to transform a representation of data structure (information structure) into a representation of software. The idea behind it is that input data, internally stored information and output data may each have a unique structure and these structures can be used as a foundation for the development of software. In addition there is an intimate relationship between software and data — the original concept behind the stored program computer is that programs could be viewed as data and data interpreted as programs. It has been shown that data structure has an important impact on the complexity and efficiency of algorithms designed to process information. The approach may be successfully applied in applications that have a well-defined, hierarchical structure of information, e.g., business information systems applications and systems applications.

- *Data flow-oriented design.* This method is to provide a systematic approach for the derivation of program structure. The idea behind it is that information may be represented as a continuous flow that undergoes a series of processes as it evolves from input to output. The *data flow diagram* is used as a graphical tool to depict information flow. The approach is particularly useful when information is processed sequentially and no formal hierarchical data structure exists.

- *Object-oriented design.* This method is to create a representation of the real-world problem domain and map it into a solution domain that is software. It results in a design that interconnects data objects and processing operations in a way that modularises in formation and processing rather than processing alone. This approach has been developed more recently than the other design methods.

Compared with the data flow-oriented design, data structure-oriented design is more suitable for data-intensive programs, because data structure-oriented design is more powerful in coping with more complicated data structures. Data-intensive programs currently in need of maintenance were usually developed before the object-oriented design method existed. In summary quite a number of data-intensive programs to be maintained today were developed by the data structure-oriented design method.

In the next section, a widely used data structure-oriented design method is described.

4.1.2 Structured Systems Analysis and Design Method

Structured Systems Analysis and Design Method (SSADM) [11,53,61] is one of family of systems development methods which has led the methods field in Britain during the 1980s. The method was accepted by the UK government's Central Computer and Telecommunications Agency (CCTA) and became mandatory for systems analysis and design in the UK in January 1983. It is constantly being updated, and version 3 was released in July 1986.

The importance of software design is to help producing "quality" software. It explains why SSADM is one of the most mature and widely used structured methods in the UK though it requires a significant investment in training — it provides good quality to software product which was developed by SSADM.

SSADM prescribes how a systems development effort should be conducted. The prescription is adjusted to suit individual needs. It breaks down a project into phases which are then divided into stages. Each stage is subdivided into steps. Each step has a list of tasks, inputs and outputs. SSADM provides structural and procedural standards.

SSADM consists of three phases: feasibility, analysis and design. The feasibility phase is optional. The analysis and design phases are divided into three stages each. The three analysis stages of the analysis phase and the three stages of the design phases are:

1. Analysis of system operations and current problems: to investigate the current system

2. Specification of requirements: to redraw the current system view built up in stage 1 to extract *what* system does without any indication of *how* this is achieved; to complete Business System Operations; and to build up and check a detailed specification of the required system.

3. Selection of technical options: to cost out the purchase of new computer equipment, etc., if required and to weigh the benefits against the costs to give the user some help in choosing the final solution, e.g., selecting the final system hardware.

4. Detailed data design: to build up the logical data design so that all the required data will be included.

5. Detailed process design: to expand the definition developed in stage 2 to a very high level of detail so that the constructor can be given all the detail necessary to build the system.

6. Physical design control: to convert the complete logical design — both data and processing — into a design that will run on the target environment.

SSADM is a data driven approach. Within SSADM several different views of data are employed. The analysis and design of processes is a part of SSAMD, but the context within which these are done is determined by the data. SSADM takes three basic views of an information system:

- Logical Data Structures — showing what information is stored and how it is interrelated;
- Data Flow Diagrams — showing how information is passed around;
- Entity Life Histories — showing how information is changed during its lifetime.

The first view of the data structure is developed as a model of the organisation's information base using the logical data structuring technique (LDST). This technique tries to capture a picture of the underlying and stable information on which the organisation and its information system and based.

The method of data modelling is based on an *entity relationship model* [47]. The model recognises two different classes of relations, *entity relation* and *relationship relation*. An entity relation has data on all entities of the same type, and has one tuple per entity, including a key to identify the particular entity. All other fields should be functionally dependent on this key. A relationship relation links the keys of two or more entity relations. It may also have attributes that are functionally dependent on this relationship [81]. The major, real world entities in which the organisation is interested are represented on a diagram. The relationships between entities are examined to ensure that all, and only the useful ones are included in the model. The nature of the relationships are also explored and details included.

This technique is a top-down approach to data modelling and relies upon the modeller's perception of the information being modelled. It is quite simple, and moderate-sized diagrams can be quickly produced. These can be intelligible to users, who may thereby contribute their understanding to the development process.

A second view of the data is represented by dataflow diagrams (DFDs). These show the data which flow into, out of, and around an information system, as well as the processes which transform it, the entities which are external to the system but which communicate with it, and the stores of data within a system.

A third view of the data is represented by the use of entity life histories (ELHs) and their associated ELH metrics. LDST takes a static view of the data, DFDs look at the movement of data and the dependency of processes upon certain flows. ELHs complement these perspectives by looking at how entities change over time.

Each of these views is developed through system analysis (stage 1-3) and logical design (stage 4 and 5) before conversion to an executable physical design. In each part of SSADM all three of these views are used and interrelated. There are several steps in one stage. The task belonging to each of the three views can

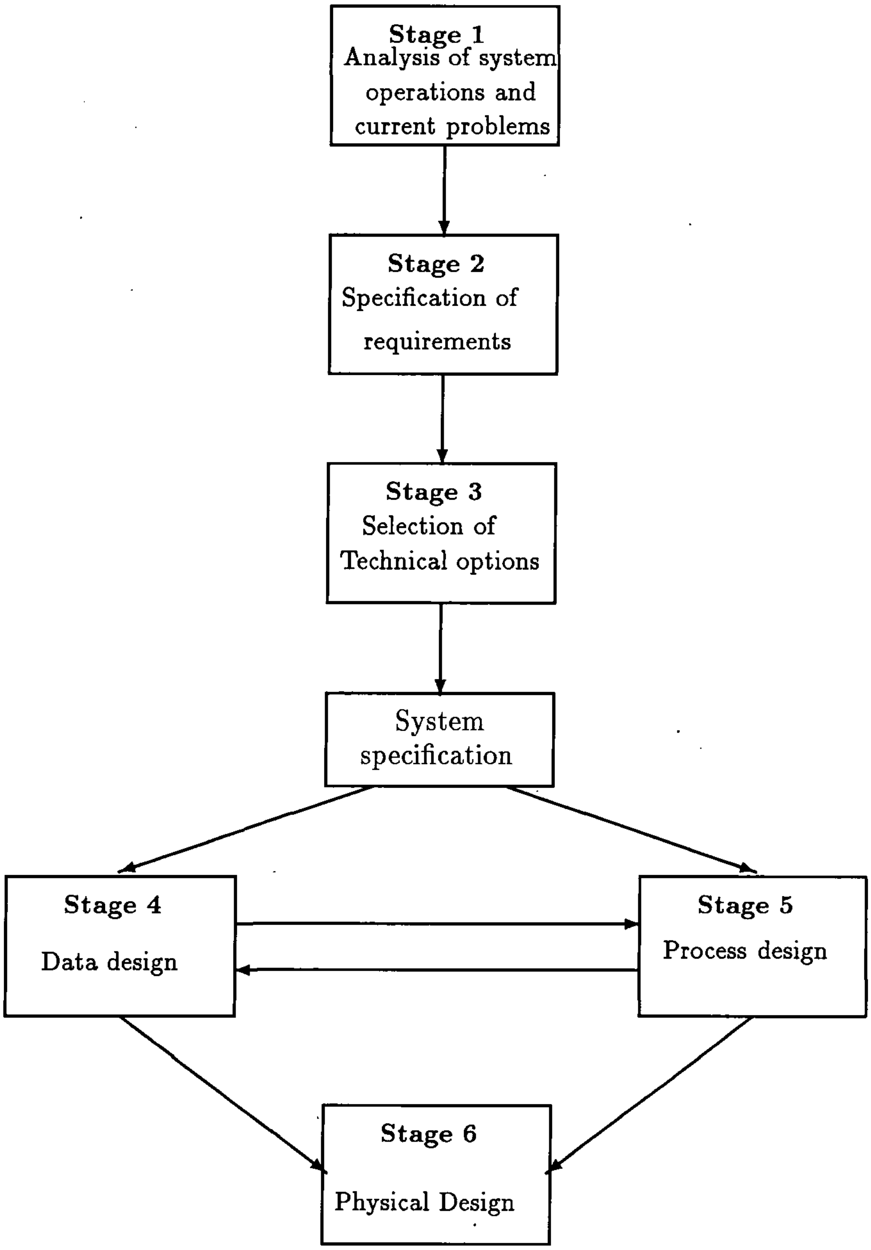


Figure 4.1: SSADM Stages

be done in one or more steps in a stage, or even does not appear in one stage. The convention of naming a step is to use a 3-digit number, first number indicating the stage where the step is in. For instance, step 110 is in stage 1 and step 260 is in stage 2.

The first view of data is corresponding to one of the three design activities — data design. Since data design is the emphasis of the research, only those steps relating to the first view (Logical Data Structures) are discussed here. Steps directly related to Logical Data Structures at the logical design phase are step 125, step 245, step 410 and 420.

In step 125, *Investigating system data structure*, the Logical Data Structure Diagram and its supporting documentation are produced. Development of Logical Data Structure involves identifying entities, identifying direct relationships between entities, creating a diagram representing the entities and their relationships, producing supporting documentation to the diagram, validating against the processing requirements and validating with the user.

In step 240, *Creating required system data structure*, the Current System Logical Data Structure and supporting documentation are expanded to define the required system data structure. This involves defining new requirements which the Logical Data Structure must support by the chosen Business System Option and the Problems/Requirements List, amending the Problems/Requirements List to describe any solutions adopted, and validating the Required System Logical Data Structure against the required system processing and extending the Entity/Data Store Cross Reference.

In step 410, *Relational data analysis*, a set of normalised relations which minimise redundancy of data and avoid consistency problems are produced.

In step 420, *Composite Logical Data Design*, the Logical Data Structure, and the relations derived from relational data analysis are combined to form the Composite Logical Data Design. This involves representing the relations as a data structure diagram to aid comparison with the Logical Data Structure, merging two diagrams and resolving differences with reference to the processing requirements and to the user, extending the Entity Description to show the full data content defined by the relations, completing any remaining documentation of the system data and consolidating all volumetric information on the data.

4.1.3 Features of Data-Intensive Programs

Data-intensive programs and computation-intensive programs are comparative notions. There is no clear distinction between these two sorts of programs. *Data-intensive programs* mean programs which are written in data-intensive programming languages that provide complex data structuring mechanisms and high-level composite operations to manipulate them. *Computational-intensive programs* mean programs which are written in computational-intensive languages that provide ways to express computations using relatively simple operations on elementary objects [74]. COBOL is a typical data-intensive programming language.

Since examples of data-intensive program are needed in the research, COBOL has been selected as the data-intensive language. The features of COBOL programs are studied as an example of data-intensive program in general. In this text, all examples of data-intensive program are in COBOL. It is believed that the generality of the research will not be limited by this assumption. On the other hand, there are claimed to be 800 billion lines of COBOL programs existing in the world [104] and the result of the research can be easily applied to maintain COBOL software.

The COBOL language was first developed in 1959. The CODASYL committee (Conference on Data Systems Languages) produced the initial specification of COBOL in 1960, and a revised version appeared in 1961. The first ANSI (American National Standard Institute) specification of the COBOL language was published in 1968. Later standards were the ANSI 1974 and the ANSI 1985 Standard. COBOL offers the following advantages within the standard language [92], which are related to the research:

1. Uniform treatment of all data as records.

2. Extensive capabilities for defining and handling files.

3. Incorporation of many functions which in other contexts would be regarded as the province of system utilities.

4. The ability to construct large programs from independently compiled modules which communicate with each other by passing parameters or by using common files.

The COBOL language used in this research not only is unrestricted to any dialect of COBOL but also covers features written in ANSI COBOL Standard 1985. More importantly, this research will be not only of benefit to COBOL programs but also to other data intensive programs written in other languages. However, programs (such as those written in COBOL) with built-in calls to data base management packages will not be addressed in this thesis and this surely is a good area for future research.

Programs written in COBOL have characteristics which are different to those of typical computation-intensive programs, and these are important constraints in reverse engineering such systems, e.g.:

- Important data is represented in the form of records and operations on data are therefore heavily record based.

- COBOL programs are often designed using Entity-Relationship Attribute Diagrams, rather than process based design methods.

- COBOL allows the programmer to specify that two different records (with different structures) may share the same memory location. This is known as the *aliasing problem* and is found in many COBOL programs.

- COBOL programs usually have external calls to the operating system and database management system.

- COBOL programs may use many *foreign keys* to represent complex data structures which in other languages would use pointers.

4.2 Representing Data Designs Using Entity-Relationship Attribute Diagrams

It can be seen that the SSADM's first view of data, i.e., Logic Data Structures, may be used in reverse engineering, i.e., to represent data design by Logical Data Structures when extracting a data design from existing code. How Logical Data Structures can be used is a problem to be solved in the thesis.

4.2.1 Entity Models

Entity models provide a system view of the data structures and data relationships within the system [47,53,55,114]. All systems possess an underlying generic entity model which remains fairly static in time. The entity model reflects the logic of the system data, not the physical implementation.

Entity models provide an excellent graphical representation of the generic data structures and relationships. They provide a clear view of the logical structure of data within the boundary of interest and allow the analyst to model the data without considering its physical form. Entity modelling provides a system view independent of current processing; it is a system-wide view not a functionally decomposed view.

An **entity** is something, real or abstract, about which we store data [114]. The name of each type entity type should be a noun, sometimes with a modifier word. An entity type may be thought of as having the properties of a noun. An entity has various *attributes* that we wish to record. An *entity type* is a named class of entities that have the same set of *attribute* types. An *entity instance* is one specific occurrence of an entity type.

We can describe data in terms of entity types and attributes by using *Entity-Relationship Attribute Diagrams.* On an Entity-Relationship Attribute Diagram the boxes are interconnected by links that represent associations between entity types. In Figure 4.2 there are four entities: CUSTOMER, PART, CONTRACT, and CONTRACT-ITEM. This diagram shows that a customer can have multiple contracts. A contract is for one customer and can be for more than one contract

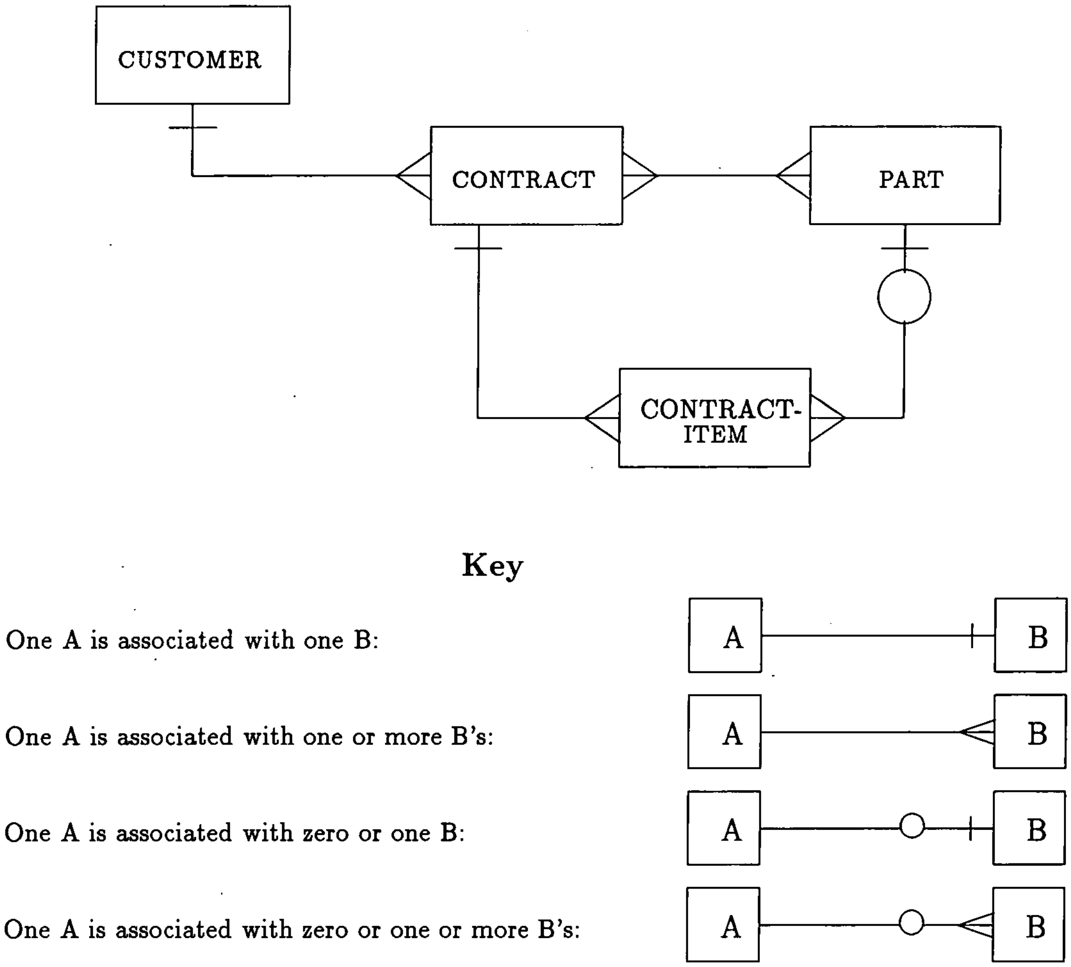

CUSTOMER
CONTRACT
PART
CONTRACT-ITEM
Key
One A is associated with one B:
One A is associated with one or more B's:
One A is associated with zero or one B:
One A is associated with zero or one or more B's:
A
B

Figure 4.2: An Entity-Relationship Attribute Diagram

item. There are zero, one, or many contracts for each part. A contract item relates to one contract and zero or one part.

4.2.2 Entity-Relationship Attribute Diagrams in SSADM

Entity models are sometimes called logical data structures, as before we use SSADM as our examples. Entity-Relationship Attribute Diagrams used in SSADM are revised versions of generally used Entity-Relationship Attribute Diagrams shown in Figure 4.2.

Entities, Data Items and Identifiers

An *entity* is something of significance to the system about which information is held [11]. An *entity type* represents a number of *entity occurrences.* According to convention, entity is always used to refer to the entity type and entity occurrence is always used to refer to a specific entity occurrence.

A data item is the smallest discrete component of the system information that is meanful (in other approaches a data item defined here is usually called an *attribute*).

Each entity is made up of a number of data items.

A data item which can be used to identify uniquely each entity occurrence is called the *primary key.*

An entity is represented as a box in a Logical Data Structure Diagram.

Relations

A *relation* is a group of non-repeating data items identified by a unique key [11]. Mathematically a relation is defined as a set of tuples and that this set is a subset of the cartesian product of a fixed number of domains [81].

Relationships

A *relationship* is a logical association between two entities on a data structure that is important to the system. Relationships are normally described as verbs.

The degree of relationships Between two entities A and B there are four possible degrees of relationships:

1. One A can be related to many B's.

2. Many A's can be related to one B.

3. Many A's can be related to many B's

4. One A can be related to one B.

1 and 2 are examples of one-to-many relationships. It is assumed that both one-to-one and many-to-many relationships rarely exist. So only one-to-many relationships are drawn on Logical Data Structure Diagrams.

Exclusive relationships This is when the existence of one relationship precludes the existence of another (see Case B in Figure 4.3).

Recursive relationships This is when entity occurrences have direct relationships with other entity occurrences of the same type (see Case C in Figure 4.3).

Logical Data Structure Diagrams

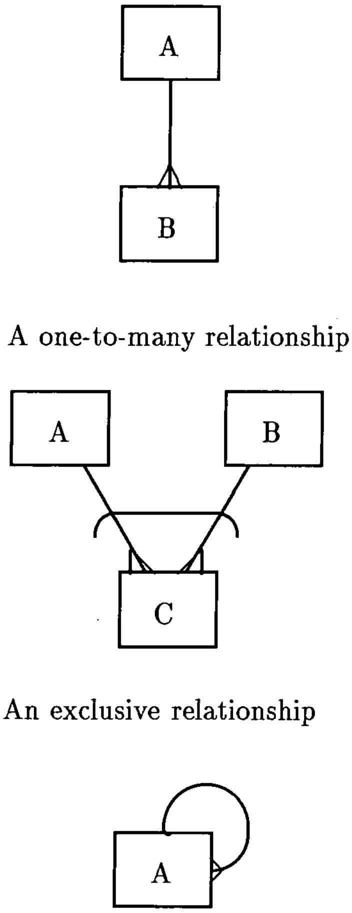
Case A in Figure 4.3 shows a one-to-many relationship. The line with the crow's foot describes the relationship. The crow's foot is always shown at the 'many' end. The entity at the one end is often referred as the *master entity* and the entity at the 'many' end referred to as the *detailed entity.*

The components of the Logical Data Structure are entities and relationships. The Logical Data Structure deals with entity and relationship types only rather than their occurrences. Relationships relate one entity to another and indicate access from one entity occurrence to all the related ones. The Logical Data Structure Diagram is supported by entity descriptions and sometimes relationship descriptions.

4.3 Reverse Engineering Through Data Abstraction

4.3.1 Abstraction Techniques in Programming

Abstraction techniques are widely used in "forward engineering" [49,51,56,109]. There are two advantages by using abstraction: firstly, the abstractions allow large systems to be broken into smaller parts with logical interfaces based upon the data being handled. These interfaces stand alone as the specification of the system with



(A) A one-to-many relationship

(B) An exclusive relationship

(C) A recursive relationship

Figure 4.3: Relationships in Entity-Relationship Attribute Diagrams

the actual implementation being hidden and flexible; secondly, the abstractions can be defined in a rigorous mathematical fashion, which means that the data type itself is a well defined mathematical system. A systematic development of a body of knowledge is thus made possible.

Abstraction is the process of ignoring certain details in order to simplify the problem and so facilitates the specification, design and implementation of a system to proceed in a step-wise fashion.

There are three stages in this process. The first requirement in designing a program in forward engineering is to concentrate on relevant features of the system, and to ignore factors which are believed irrelevant. The next stage in program design is the decision of the manner in which the abstracted information is to be represented in the computer. Finally there comes the task of programming the computer to get it to carry out these manipulations at the representation of the data that corresponds to the manipulations in the real world in which we are interested.

Three basic abstractions have been used in programming [109,110,141]:

Procedural abstraction combines the methods of abstraction by parameterisation and specification in a way that allows us to abstract a single operation or event. A procedure provides a mapping from input arguments to output arguments. In another words, it is a mapping from a set of input arguments to a set of output results. Desirable properties of a procedure include simplicity and generality.

Data abstraction allows us to extend the base type level with new types of data. Data abstraction is the most important method in program design. Choosing the right data structures is crucial to achieve an efficient program. In the absence of data abstraction, data structures must be defined too early , i.e., they must be specified before the implementations of modules which use them can be designed.

Iteration abstraction, or **iterator** for short, is a generalisation of the iteration methods available in most programming languages. They permit users to iterate over arbitrary types of data in a convenient and efficient way. In other words, iterators are a mechanism that solve the problem in the adequacy of data types that are collections of objects.

4.3.2 Data Abstraction

Data is essential to programming. One of the most important objectives of programming is to process data or to achieve certain goal through processing data. The program development process can be described in terms of data. Data abstraction has following important aspects [50]:

- Data abstractions separate the use of a data type from the implementation of a data type.

- Data abstractions simplify issues of correctness.

- Data abstractions permit the exchange of (correct) implementations. Performance is ideally the only criterion for choosing an implementation.

- Data abstraction is a software design technique that promotes modularity and independent development of data abstraction implementation and the application program.

4.3.3 Data Type

A *data type* defines a set of valid values and the operations on these values. Another way to state this is that a data type is a language mechanism to enforce authentication and security [28,124]. A summary of some of the important points data type was given by C. A. R. Hoare [89]:

1. A type determines the set of values which may be assumed by a value or expression.

2. Every value belongs to one type only.

3. The type of a value denoted by any constant, variable, or expression may be deduced from its form or context, without any knowledge of its value as computed at run-time.

4. Each operator expects operands of fixed type, and delivers a result of some fixed type. Where the same symbol is applied to several different types, this symbol may be regarded as ambiguous, denoting several different actual operators. The resolution of such systematic ambiguity can always be made at compile-time.

5. The properties of the values of a type and of the primitive operations defined over them are specified by means of a set of axioms.

6. Type information is used in a high-level language both to prevent or detect meaningless construction in a program, and to determine the method of representing and manipulating data on a computer.

7. Types can be constructed from a number of primitive types, i.e., by constructors. These include Cartesian products, discriminated unions, sets, functions, sequences and recursive structures.

Each type has a range of basic operations associated with it. Usually these are the operations provided by the basic hardware of the computer system and they will apply directly to the basic types [151]. Further operations will be defined in terms of this basic set. More generally, the operations associated with a type are:

- Assignment and test for equality will be required for both primitive and structured types.

- Transfer or conversion functions are required to convert values of one type to another.

- Constructors are necessary to donate the construction of a new type from component types.

- Selectors are required to access the component values of a structured type.

All types are constructed from further types, which ultimately must be either primitive (probably supported directly by the computer's hardware) or are defined by the programmer by using type constructors. The constructors (fundamental data types) for types are as follows [89,151]:

- Unstructured data types — All structured data must be built up from unstructured components, belonging to a primitive or unstructured type. Some of these unstructured types (for example, reals and integers) may be taken as given by a programming language. Although these primitive types are theoretically adequate for all purpose, there are strong practical reasons for encouraging a programmer to define his or her own unstructured types. This can be done by an *enumeration*, i.e., indicating the set of values that can be taken by an unstructured type. For instance, type Day is defined as {MON, TUE, WED, THU, FRI, SAT, SUN}. Other examples include the *boolean* type (with only two values, false and true) and the *char* type (with all characters in the ASCII table, ranging from ASCII value 32 to 126).

- The Cartesian Product — The Cartesian Product is a data structuring method which gives the space of possible values of a composite type. Such structures usually have a fixed size and are called records or structures in programming languages, where their components can be named, e.g.:

```
type DATE is
  record
    D: DAYS;
    M: MONTHS;
    Y: YEARS;
  end record;
```

It must be possible to refer to the individual components of a Cartesian product.

- The Discriminated Union — A discriminated union is a type which is the union of two or more sets of values, each of which may have components in common. It is usually specified by listing the components that the sets have in common, followed by the components which differ.

```
type PERSON is
  record
    NAME: STRINGS;
    SEX: (M or F);
    IDENTIFICATION:
            case: STUDENT: STUDENT CARD NUMBER;
                  EMPLOYED: NI NUMBER
            end case;
      end record;
```

- The Array — It is available in almost every language. Viewing abstractly it is a mapping between the subscript values and the elements of the array.
- The Power Set — The powerset of a given set is defined as the set of all subsets of that set; and a powerset type is a type whose values are sets of values selected from some other type known as the *base* of the powerset.
- The Sequence — The cardinality of a sequence cannot be decided at compile time, because a sequence of values may be indefinitely long and, more practically, may vary as the program executes. A sequence can be regarded as an arbitrary number of items of given type placed in a particular order. Such sequences include strings, stacks, queues and so on. The sequence is the abstraction notion and there are various representations of this abstract notion.
- Sparse Data Structures — If the set potentially contains a very large number of elements or if the range of possible subscript values is very large, then the data type is said to be sparse if only a small proportion of possible values are present. A particular example is an array which represents a dictionary and is indexed by character strings corresponding to words. This array can be declared as:

  type DICTIONARY = sparse array WORD of DEFINITION.

  Sparse data structures can be represented by keeping tables to map the index values into either main store addresses or positions in a file. Sparse data structures, in general sense, are not recognised in programming languages.
- Pointers — A pointer can be thought of as the name of the place where an object is kept; usually it will be implemented as an address. Pointers are

usually used to build data structures whose size is, in general, unknown at compile time.

4.3.4 Abstract Data Types

The major conceptual idea of *abstract data types* is to separate the use of a type from the representation and implementation of a type. The use of a type should depend only on the set of values and operations. It should not depend on either its representation or its implementation.

Data abstraction allows the description of abstract data types. Usually, a data abstraction consists of "objects" and "operations". To implement the data abstraction, we implement the operations in terms of the chosen representation, and we must reimplement the operations if we change the representation. However, we do not need to reimplement the program by using this abstraction, because the program depends only on the operations and not on the representation.

It is important to understand this. For example, a stack can be viewed as an abstracted data type. A stack is defined in terms of a representation (such as an array) and several operations (such as NEW, PUSH, POP, READ and EMPTY). If these operations are met in a program, they can be abstracted to a "stack".

4.3.5 Abstraction Levels of Data and Software

The ANSI-SPARC Standard has established three distinct abstraction levels in viewing data. These are the *physical level*, *logical level* and *conceptual level* [1].

Data at the physical level At the physical level, data is viewed as a set of records connected through pointer arrays, inverted lists, etc., depending on the type of physical implementation, e.g., hierarchical, network, or relational. The forms of data are:

- raw data — consisting of bits, bytes and words.

- scalar types — such as natural numbers, integers, reals, characters and boolean variables together with pointer variables (representing memory addresses).

- data structures — referring to compound structures which are directly supported by programming languages such as arrays, heterogeneous structures (e.g., the record in COBOL), sets, lists, simple user definable types and persistent structures such as files (random, sequential, etc.). Associated with the data structures are the operations used to manipulate them.

Data at the logical level At the logical level, data is viewed as tables of normalised relations or tuples of data elements linked by keys, which are independent of any particular physical implementation, e.g., the data manipulation language.

The basic form of data here is *data items*. Data items of an abstract class, as opposed to values of an abstract data type, encapsulate the definition of an internal state and are associated with a set of accessing operations, which may update and/or query the object state. The assumption is that the classes of abstract items will be initiated as independent subsystems in different contexts. The item classes include stack, queue, sets, bags, etc. An abstract data type is at this level.

Data at the conceptual level At the conceptual level, data is viewed as a network of entities with attributes and relationships among one another, e.g., the entity/relationship model. At this level, data is existing in the form of application objects and it is more close to the concept of real world rather than to the software.

Abstraction levels of software In principal, the three data levels also apply to software [51,144,174]. At the physical level software exists as a set of discrete units of code of various types — modules, maps, data descriptions, access paths, and command procedures — link editors and loaders of the operating system. At the logical level, software exists in the form of a meta language, which describes processing units — modules, data capsules, and interfaces of any particular implementation language. At the conceptual level, software, like data , can be viewed as a set of abstract entities, such as data objects, data elements, processes and relationships among one another. At the conceptual software models some real world application.

The software at the physical and logical level is typically executable and hence it can be referred as that the **software** is **at** the **code level**. Whilst the **software at** the **conceptual level** is typically not executable and can be viewed as the design and specification.

4.4 Definition of Proposed Research Problem

In seeking solutions to the problems described in Chapter 1 (Section 1.3), Chapter 2 and Chapter 3 reviewed the work (done in software engineering and in reverse engineering in particular) related to this thesis, and the previous sections in this chapter addressed features of data intensive programs, Entity-Relationship Attribute Diagrams and data abstraction techniques. This enables the research problems proposed in Section 1.3 to be defined more precisely as follows:

- Can data intensive programs be reverse engineered to Entity-Relationship Attribute Diagrams? What are the difficulties? How do we use *code* and *data structure* in the sources?
- How do we cope with source code techniques such as
    1. foreign keys
    2. aliasing
    3. input/output
    4. abstract data type, and
- Is it possible to reverse engineer data intensive programs which have been heavily maintained and hence whose structure has become heavily degraded?
- What is the method to extract program data designs (represented in Entity-Relationship Attribute Diagrams) from the existing data intensive code?
- With what size of code can this method cope?

- What information do we throw away when crossing abstraction levels?
- What can be automated and what can be done by humans?
- Suppose that this method is demonstrated by extending the Maintainer's Assistant. What changes have to be made to WSL? What extension has to be made to the transformation library? What other supporting components should be implemented?
- How do we know what we have done is correct? How do we measure our success?

It is claimed that answers to the above questions are making a contribution to research in computer science.

Because this research aims at developing a method (and a tool) to extract program data designs from the existing data intensive code, it should be pointed out that the method should only be used under certain circumstances where, for example, maintenance to a relative self-contained module needs carrying out. This means that the method would not be very helpful when a minor change (e.g., a change to one line of code) is needed. The reason is that an Entity-Relationship Attribute Diagram can only be extracted from a block of code (containing both control and data structures) rather than just one or two lines of code.

Chapter 5

Working Environment and Design Recovery Method

As the research described in this thesis is part of the REFORM project, the background of REFORM and the working environment are introduced in this chapter.

5.1 Working Environment

The REFORM project started in July 1989 [44,156]. As mentioned in Chapter 3, the aim of the project is to build a prototype tool — the Maintainer's Assistant — which will take existing software written in low-level procedural language (in particular, IBM CICS code written in IBM-370 assembler), through a process of successive transformation, turn it into an equivalent high-level abstract specification expressed in terms of non-procedural abstract specification language (in particular, Z). The theoretical foundation for the project was established by the work carried out at Oxford and Durham by M. Ward [155]. Naturally, as the process of applying program transformations cannot be totally automated, the Maintainer's Assistant is an interactive tool including an interactive interface.

5.1.1 Ward's Work and its Application in the REFORM Project

The REFORM Project has its roots in Ward's work [155], in which he developed methods of proving refinements and transformations of programs. Although he used the popular approach of defining a core "kernel" language with denotational semantics, and permitting definitional extensions in terms of the basic constructs, he did not use a purely applicative kernel; instead, the concept of states is included, using a *specification statement* which also allows specification expressed in first order logic as part of the language (thus providing a genuine wide spectrum language).

In contrast to other work, Ward used infinitary first order logic (an extension of first order logic which allows infinitely long formulae) both to express the weakest preconditions of programs [60] and to define assertions and guards in the kernel language. Engeler [64] was the first to use infinitary logic to describe properties of programs and Back [13] used such a logic to express the weakest precondition of a program as a logic formula but his kernel language was limited to simple iterative programs. Ward used a different kernel language which includes recursion and guards, and he showed that the introduction of infinitary logic as part of the language (rather than just the metalanguage of weakest preconditions), together with a combination of proof methods using both denotational semantics and weakest precondition, is a powerful theoretical tool which allows some general transformations and representations theorems to be proved.

In Ward's approach [155], it is possible to prove that two versions of a program are equivalent. Programs are defined to be equivalent if they have the same semantic function. Hence equivalent programs are identical in terms of their input-output behaviour, although they may have different running times and use different internal data structures. A refinement of a program, or specification, is another program which will terminate on each initial state for which the first program terminates, and will terminate in one of the possible final states for the first program. In other words a refinement of a specification is an acceptable implementation of the specification and a refinement of a program is an acceptable substitute for the program.

Here is a very brief look at Ward's approach to proving the equivalence of two programs in terms of Dijkstra's weakest precondition. For a given program S and on the final state R the weakest precondition $WP(S, R)$ is the weakest condition on the initial state such that the program will terminate in a state satisfying condition R. It is possible to express the weakest precondition of any program or specification as a single formula in infinitary logic. The value of weakest preconditions lies in the fact that two programs are equivalent if and only if they have equivalent weakest preconditions [152,153,154,155].

Ward's work can be used not only in the program development but also in the software maintenance which has been overlooked traditionally by the people who were building transformation systems. The aim of the REFORM project is to develop a computer-based, semi-automated transformation system, founded on Ward's approach, for use in software maintenance and especially reverse engineering.

5.1.2 The Wide Spectrum Language

In the process (within the REFORM project) of acquiring a specification from the program code, a notation (or a language) is needed to represent the program and specification at all intermediate steps, especially as objects (program or specification) are changed from one form to another. As we have seen, a wide spectrum language is a suitable language for this, so that a wide spectrum language named WSL has been defined by Ward, which incorporates a variety of constructs, from low-level machine-oriented constructs up to high-level specification ones.

The WSL [37,155] consists of two types of construct: WSL constructs and Meta-WSL constructs. WSL constructs include statements, functions, expressions, logic and arithmetic operator and test, etc., for representing both program code and program specification; Meta-WSL constructs include Meta-WSL statements, Meta-WSL function, Meta-WSL pattern, Meta-WSL condition and etc., for representing program transformations. Both types of WSL constructs were originated from the kernel language.

The syntax and semantics of expressions and formulae are discussed here first and they are used to define the kernel WSL and the first level WSL in this chapter, and will be used to extend WSL and prove program transformations later in this thesis.

Syntax of Expressions

Numeric operators: $e_1 + e_2$, $e_1 - e_2$, e_1/e_2, $e_1^{e_2}$, $e_1{**}e_2$, e_1 *mod* e_2, e_1 *div* e_2, *frac*(e_1), *abs*(e_1), *sgn*(e_1), *max*$(e_1, e_2, ...)$, *min*$(e_1, e_2, ...)$, with the usual meanings.

Sequences: $s = \langle\ a_1, a_2, ..., a_n\ \rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i\ ..\ j]$ is the subsequence $\langle\ s[i], s[i+1], ..., s[j]\ \rangle$, where $s[i\ ..\ j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s. $s[i\ ..]$ is used as an abbreviation for $s[i\ ..\ \ell(s)]$. *reverse*$(s) = \langle\ a_n, a_{n-1}, ..., a_2, a_1\ \rangle$, *head*$(s)$ is the same as $s[1]$, *tail*(s) is $s[2\ ..]$, *last*(s) is $s[l(s)]$ and *butlast*(s) is $s[1\ ..\ s[\ell(s)]-1]$.

Sequence Concatenation: $s_1 +\!\!+\ s_2 = \langle s_1[1], ..., s_1[\ell(s_1)], s_2[1], ..., s_2[\ell(s_2)]\rangle$. The *append* function, *append*$(s_1, s_2, ..., s_n)$, is the same as $s_1 +\!\!+\ s_2 +\!\!+\ ... +\!\!+\ s_n$.

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

Stacks: Sequences are also used to implement stacks, for this purpose the following notation is used: For a sequence s and variable x: $x \xleftarrow{pop} s$ means $x := s[1]$; $s := s[\ 2..]$ which pops an element of the stack into variable x. To push the value of the expression e onto stack s: $s \xleftarrow{push} e$ is used to represent: $s := \langle e\rangle +\!\!+\ s$.

Sets: The usual set operations $\cup$ (union), $\cap$ (intersection), $-$ (set difference), $\subseteq$ (subset), $\in$ (element) and $\mathcal{P}$ (powerset) are used. $\{\ x \in A \mid P(x)\}$ is the set of all elements in A which satisfy predicate P. For the sequence s, *set*(s) is the set of elements of the sequence, i.e., *set*$(s) = \{\ s[i] \mid \mathrm{a} \leq i \leq \ell(s)\}$.

Relations and Functions: A relation is a (finite or infinite) set of pairs, a subset of $A \times B$ where A is the domain and B the range. A relation f is a function if $\forall x, y_1, y_2.(((x, y_1) \in f \wedge (x, y_2) \in f) \Rightarrow y_1 = y_2)$. In this case $f(x) = y$ is written when $(x, y) \in f$.

Map: The map operator $*$ returns the sequence obtained by applying a given function to each element of a given sequence: $(f * \langle a_1, a_2, ..., a_n \rangle) = \langle f(a_1), f(a_2), ..., f(a_n) \rangle$. The map operation can also be applied to set in the same way [26,27].

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

Syntax of Formulae

In the following, $\mathbf{Q}$, $\mathbf{Q}_1$, $\mathbf{Q}_2$, etc. represent arbitrary formulae and e_1, e_2, etc. arbitrary expressions:

Relations: $e_1 = e_2$, $e_1 \neq e_2$, $e_1 < e_2$, $e_1 \leq e_2$, $e_1 > e_2$, $e_1 \geq e_2$, $even?(e_1)$, $odd?(e_1)$;

Logical Operators: $\neg\mathbf{Q}$, *not* $\mathbf{Q}$, $\mathbf{Q}_1 \vee \mathbf{Q}_2$, $\mathbf{Q}_1 \wedge Q_2$;

Quantifiers: $\forall \mathrm{v}.\mathbf{Q}$, $\exists \mathrm{v}.\mathbf{Q}$.

Syntax of Statements

The kernel language in the REFORM approach has two primitive statements: the atomic specification and the guard statement. The atomic specification [155] is written $\mathrm{x}/\mathrm{y}.\mathbf{Q}$, where $\mathbf{Q}$ is a formula of first order logic and x and y are sequences of variables. Its effect is to add the variables in x to the state space, assign new values to them such that $\mathbf{Q}$ is satisfied, remove the variables in y from the state and terminate. If there is no assignment to the variables in x which satisfies $\mathbf{Q}$ then the atomic specification does not terminate. The guard statement is written $[\mathbf{P}]$, where $\mathbf{P}$ is a formula of first order logic. The statement $[\mathbf{P}]$ always terminates; it enforces $\mathbf{P}$ to be true at this point in the program; it has the effect of restricting previous nondeterminism to those cases which leave $\mathbf{P}$ true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore, all possible final states will satisfy any desired condition.

The kernel language is constructed from these two primitive statements, a set of *statement variables* (these are symbols which will be used to represent the recursive calls of recursive statements) and the following three compounds:

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

Semantics of Kernel WSL

In order to interpret statements as programs, the initial and final state spaces of the statements (i.e., the initial set of variables (input variables), and the final set of variables (output variables)) need to be known. These must be related properly according to the syntax of the statement. For a statement S and a finite non-empty set of variables $\mathcal{V}$ and $\mathcal{W}$, it is defined that the relation S: $\mathcal{V} \rightarrow \mathcal{W}$ to be true when $\mathcal{V}$ and $\mathcal{W}$ are suitable input and output state spaces for S. Thus:

1. x/y.Q: $\mathcal{V} \rightarrow \mathcal{W}$ iff $\mathcal{W} = (\mathcal{V} \cup \tilde{x}) - \tilde{y}$ (where $\tilde{x}$ is the set of variables in sequence x),
2. [**P**]: $\mathcal{V} \rightarrow \mathcal{W}$ iff $\mathcal{V} = \mathcal{W}$ and $\mathcal{V}$ contains all the variables in **P**,
3. ($\mathbf{S}_1$; $\mathbf{S}_2$): $\mathcal{V} \rightarrow \mathcal{W}$ iff $\exists \mathcal{V}'.(\mathbf{S}_1\colon \mathcal{V} \rightarrow \mathcal{V}' \wedge \mathbf{S}_2\colon \mathcal{V}' \rightarrow \mathcal{W})$,
4. ($\mathbf{S}_1 \sqcap \mathbf{S}_2$) $\mathcal{V} \rightarrow \mathcal{W}$ iff $\mathbf{S}_1$: $\mathcal{V} \rightarrow \mathcal{W}$ and $\mathbf{S}_2$: $\mathcal{V} \rightarrow \mathcal{W}$,
5. ($\mu\mathcal{X}.\mathbf{S}_1$) $\mathcal{V} \rightarrow \mathcal{W}$ iff $\mathcal{V} = \mathcal{W}$ and $\mathbf{S}_1$: $\mathcal{V} \rightarrow \mathcal{V}$.

For example, three fundamental statements can be defined immediately:

abort $=_{DF}$ ⟨⟩/⟨⟩.**false** **null** $=_{DF}$ [**false**] **skip** $=_{DF}$ ⟨⟩/⟨⟩.**true**

For any finite, non-empty set $\mathcal{V}$ of variables: **abort**: $\mathcal{V} \rightarrow \mathcal{V}$, **null**: $\mathcal{V} \rightarrow \mathcal{V}$ and **skip**: $\mathcal{V} \rightarrow \mathcal{V}$.

A weakest precondition (WP) for kernel language statements is defined as a formula of infinitary logic. WP is a function which takes a statement (a syntactic object) and a formula from $\mathcal{L}$ (another syntactic object) and returns another formula in $\mathcal{L}$. The WPs for those five statements in the kernel language are defined as follows:

1. $\mathrm{WP}(\mathrm{x}/\mathrm{y}.\mathbf{Q}, \mathbf{R}) =_{DF} (\exists \mathrm{x}.\mathbf{Q} \wedge \forall \mathrm{x}.(\mathbf{Q} \Rightarrow \mathbf{R}))$

2. $\mathrm{WP}([\mathbf{P}], \mathbf{R}) =_{DF} \mathbf{P} \Rightarrow \mathbf{R}$

3. $\mathrm{WP}(\mathrm{S}_1;\ \mathrm{S}_2, \mathbf{R}) =_{DF} \mathrm{WP}(\mathrm{S}_1, \mathrm{WP}(\mathrm{S}_2, \mathbf{R}))$

4. $\mathrm{WP}(\mathrm{S}_1 \sqcap \mathrm{S}_2, \mathbf{R}) =_{DF} \mathrm{WP}(\mathrm{S}_1, \mathbf{R}) \wedge \mathrm{WP}(\mathrm{S}_2, \mathbf{R})$

5. $\mathrm{WP}((\mu\mathcal{X}.\mathrm{S}), \mathbf{R}) =_{DF} \bigvee_{n<\omega}\mathrm{WP}((\mu\mathcal{X}.\mathrm{S})^n, \mathbf{R})$

For the three fundamental statements in the previous example, their WPs are: WP(**abort**, **R**) = **false**, WP(**skip**, **R**) = **R** and WP(**null**, **R**) = **true**.

The First Level Language

The kernel language just described is particularly elegant and tractable but is too primitive to form a useful wide spectrum language for the transformational development of programs. For this purpose the language needs to be extended by defining new constructs in terms of the existing ones using "definitional transformations". A series of new "language levels" is built up, with the language at each level being defined in terms of the previous level; the kernel language is the "level zero" language which forms the foundation for all the others. Each new language level automatically inherits the transformations proved at previous level and these form the basis of a new transformation catalogue. Transformations of new language construct are proved by appealing to the definitional transformation of the construct and carrying out the actual manipulation in the previous level language. This technique has proved extremely powerful in the development of a practical transformation system such as the Maintainer's Assistant.

The first set of language extensions are as follows.

1. Sequential composition: The sequencing operator is associative so the brackets can be eliminated:

   $\mathrm{S}_1;\ \mathrm{S}_2;\ \mathrm{S}_3;\ \ldots\ ;\ \mathrm{S}_n \quad =_{DF} \quad (\ldots((\mathrm{S}_1;\ \mathrm{S}_2);\ \mathrm{S}_3);\ \ldots\ ;\ \mathrm{S}_n)$

2. Deterministic Choice: Guards can be used to turn a nondeterministic choice into a deterministic choice:

$$\underline{\textbf{if}}\ \mathbf{B}\ \underline{\text{then}}\ \mathbf{S}_1\ \underline{\text{else}}\ \mathbf{S}_2\ \underline{\text{fi}}\ =_{DF}\ (([\mathbf{B}];\ \mathbf{S}_1) \sqcap ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

3. Assertion: An assertion is a partial **skip** statement; it aborts if the condition is false but does nothing if the condition is true. It can be defined using an atomic specification which changes no variables:

   $$\{\ \mathbf{B}\ \}\ =_{DF}\ \langle\rangle/\langle\rangle.\mathbf{B}$$

4. Assignment: A general assignment can be expressed using a pair of atomic specifications:

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

5. Simple Assignment: If **Q** is of the form $x' = t$ where t is a list of terms and x'' is a list of new variables, then:

   $$\mathrm{x}{:=}\ \mathrm{t}\ =_{DF}\ \mathrm{x}''/\langle\rangle.(\mathrm{x} = \mathrm{t});\ \mathrm{x}'/\mathrm{x}''.(\mathrm{x}' = \mathrm{x}'')$$

6. Deterministic Iteration: A $\underline{\textbf{while}}$ loop is defined using a new recursive procedure $\mathcal{X}$ which does not occur free in **S**:

   $$\underline{\textbf{while}}\ \mathbf{B}\ \underline{\textbf{do}}\ \mathbf{S}\ \underline{\textbf{od}}\ =_{DF}\ (\mu\mathcal{X}.(([\mathbf{B}];\ \mathbf{S};\ \mathcal{X}) \sqcap [\neg\mathbf{B}]))$$

7. Initialised Local Variables:

   $$\underline{\textbf{var}}\ \mathrm{x}{:=}\ \mathrm{t}{:}\ \mathbf{S}\ \underline{\textbf{end}}\ =_{DF}\ \mathrm{x}/\langle\rangle.(\mathrm{x}{=}\mathrm{t});\ \mathbf{S};\ \langle\rangle/\mathrm{x}.\mathbf{true}$$

8. Counted Iteration:

   $$\begin{aligned}\underline{\textbf{for}}\ \mathrm{i}{:=}\ \mathrm{b}\ \underline{\textbf{to}}\ \mathrm{f}\ \underline{\textbf{step}}\ \mathrm{s}\ \underline{\textbf{do}}\ \mathbf{S}\ \underline{\textbf{od}}\ =_{DF}\ &\underline{\textbf{var}}\ \mathrm{i}{:=}\ \mathrm{b}{:}\\ &\quad\underline{\textbf{while}}\ \mathrm{i} \le \mathrm{f}\ \underline{\textbf{do}}\\ &\quad\quad \mathbf{S};\ \mathrm{i} := \mathrm{i} + \mathrm{s}\ \underline{\textbf{od}}\ \underline{\textbf{end}}\end{aligned}$$

9. Procedure Call:

   $$\underline{\textbf{proc}}\ \mathcal{X} \equiv \mathbf{S}.\ =_{DF}\ (\mu\mathcal{X}.\mathbf{S})$$

10. Block with local procedure:

    $$\underline{\textbf{begin}}\ \mathbf{S}_1\ \underline{\textbf{where}}\ \underline{\textbf{proc}}\ \mathcal{X} \equiv \mathbf{S}_2.\ \underline{\textbf{end}}\ =_{DF}\ \mathbf{S}_1\ [\underline{\textbf{proc}}\ \mathcal{X} \equiv \mathbf{S}_2./\mathcal{X}]$$

11. Comments:

    $$\underline{\textbf{comment}}\text{: “any text string”}\ =_{DF}\ \mathbf{skip}$$

These go to make up the "first level" language. Subsequent extensions will be defined in terms of the first level language. For the purposes of this thesis only a subset of the first level language is described.

Meta-WSL

In order to express program transformations (i.e., in contrast to programs), a language is needed. In the Maintainer's Assistant, extending WSL to include suitable statements and expressions for writing transformations is chosen rather than writing a completely new language for this purpose. This language is called **Meta-WSL** [37,38], reflecting the fact that it is both an extension of WSL, and designed to manipulate WSL. This contrasts with the CIP project which uses several languages [18] for formulating program schemes, transformation algorithms and applicability tests.

It is assumed that the current program being transformed is stored as a tree in some global variable, so that all that a transformation needs to do is to modify the contents of this variable. In fact, WSL is a general language so it can be used to write transformations, since WSL can be used to operate on the variable which holds the program tree. The term "program tree" here refers to the internal representation of WSL programs in LISP. The main reason for designing Meta-WSL is because the pure WSL lacks statements and expressions for manipulating the program tree efficiently whereas Meta-WSL allows writing transformations easily and efficiently.

In addition to incorporating WSL, Meta-WSL includes the following main extensions to WSL:

Program Editing Statements These include "(@ Del)" (to delete current item), (@Change_To Thing) (to change the current item to "Thing" which is the new item to replace the old item), etc.

Pattern Matching and Template Filling This includes a function which matches a section of a WSL program tree against a given pattern and returns the result in a table, and a function which takes a pattern and a table and replaces the tokens in the pattern with values from the table. For example, "([_Match_] Type Pattern Table)" (to match the pattern with the current item when its generic type is "Type" and put the results in the "Table"), ([_Fill_In_] Type Pattern Table) (to fill in the current item with the contents in the Table when the pattern matched), etc.

Movement Statements These includes statements for moving to different parts of the program tree. Though the section of code on which the transformation is to be performed is first selected, it may be necessary for the transformation temporarily to select another section of the program in order to perform a transformation. For instance, "(@Up)" (to move to the father node in the tree), "(@Down)" (to move down to the first branch node of the tree), "(@<<)" (to move to the left-hand node of the tree), "(@>>)" (to move to the right-hand node of the tree), etc.

Movement Applicability Testing Functions These are functions which test the applicability of a particular form of movement within a program tree since a specific movement within a tree is not always possible. Examples of these functions include: "([_Up?_])" (to move to the father node in the tree), "([_Down?_])" (to move down to the first branch node of the tree), "([_<<?_])" (to move to the left-hand node of the tree), "([_>>?_])" (to move to the right-hand node of the tree), etc.

Repetition Statements It is often necessary within a transformation to test a condition or perform some operation at every subnode of the program tree, subsubnode and so on within the selected program item. The repetition statements allow this to be done easily, such as "(@Exit_When)" (to exit the current loop when some condition meets) and "(@No_Deeper)" (to go no deep than the current node when some condition meets).

Other Statements and Functions These include:

- statements for setting state flags — "(@Pass)" and "(@Fail)", etc.;
- a statement for calling other transformations — "(Trans Name");

- Function for other code checking — "([_S_Type?_])", "([_P_Type?_])", etc. ;
- A function for testing applicability — "([_Trans?_] Name)"; and etc.

5.1.3 Advantages of Using Program Transformation and WSL

The REFORM approach used program transformation and a wide spectrum language because there are a number of benefits from using them. The benefits of using program transformations are:

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

Apart from the general advantages of a wide spectrum language, the benefits of using a wide spectrum language in the Maintainer's Assistant are:

- There is flexibility for extending the system to build a wide set of transformations. The WSL can be extended by applying definitional transformations. This is particularly useful when new WSL constructs are needed to writing new program transformations.
- The WSL is an intermediate language. The advantage of using an intermediate language is that the system of acquiring the specification from the

program code needs only being developed once. Programs written in any language can be translated into the WSL as long as the WSL translator for that particular language has been built.

5.1.4 The Original Design of the Maintainer's Assistant

In the original design of the Maintainer's Assistant, the system supports the transformation of an existing source code to a specification in three phases [156,164].

In phase 1, a "Source-to-WSL" translator takes the assembler (or other language) and translate it into its equivalent WSL. The maintainer undertakes all operations through the Browser. The Browser then checks the program and uses the Program Slicer [6,43,160] to chop the program into smaller programs which are in manageable size. The maintainer may conduct the process more than once until satisfied that (s)he has split the code in such a way that it is ready for transformation. Eventually, this code is saved to the database in order to assemble specifications of those code modules (refer to Figure 5.1).

In the second phase, the maintainer will take one piece of code out from the database with which to work. The Browser allows the maintainer to look at and alter the code under strict conditions and the maintainer can also select transformations to apply to the code. The program transformer works in an interactive mode. It presents WSL on screen in pretty printed form format and searches a catalogue of proven transformations to find applicable transformations for any selected piece of code. These are displayed in the user interface's window system. When the Program Transformer is working, it also depends on the General Simplifier, the Program Structure Database and the Knowledge Base System (not yet implemented) [142] by sending them requests. The maintainer can apply these transformations or get help from the Knowledge Base as to which transformation are applicable. Once a transformation is selected it is automatically applied. These transformations can be used to simplify code and expose errors. Finally, the code is transformed to a form at higher level of abstraction, which can be translated into specifications in Z, and the code is saved back to the DataBase (Figure 5.2).

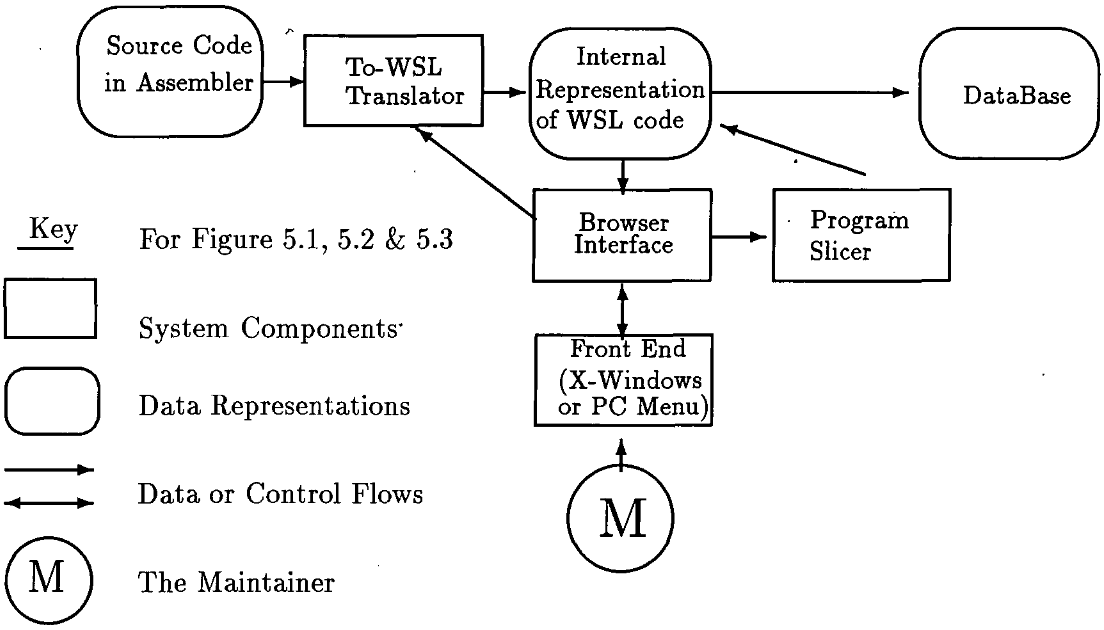


Figure 5.1: From Source Code to a Program in Low-Level WSL

The third phase comes when all the source code in the DataBase has been transformed. A Program Integrator is called to assemble the code or specifications into a single program in high-level WSL. A WSL to Z translator will translate this highly abstracted specification in WSL into specification in Z (Figure 5.3).

5.1.5 The State of the Maintainer's Assistant by 1991

By the summer of 1991, the Maintainer's Assistant[1] consists of:

- an Assembler to WSL translator,
- a History/Future Database
- a Structure Editor
- a Browser,

[1] The Program Structure Database and the General Simplifier were investigated by the author, and other components of the Maintainer's Assistant were studied by other three members of the REFORM research team.

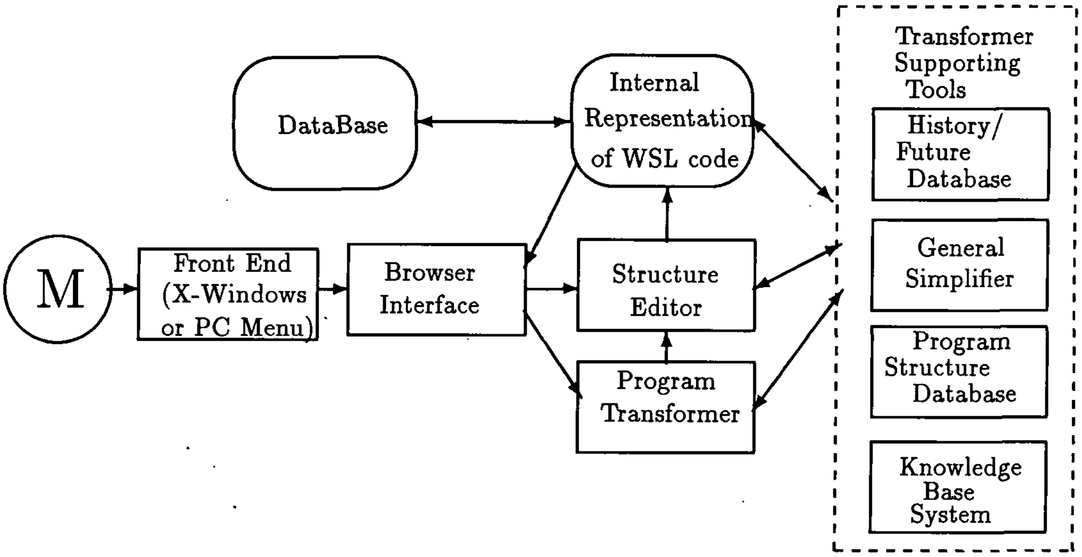


Figure 5.2: From a Program in Low-Level WSL to a Program in High-Level WSL

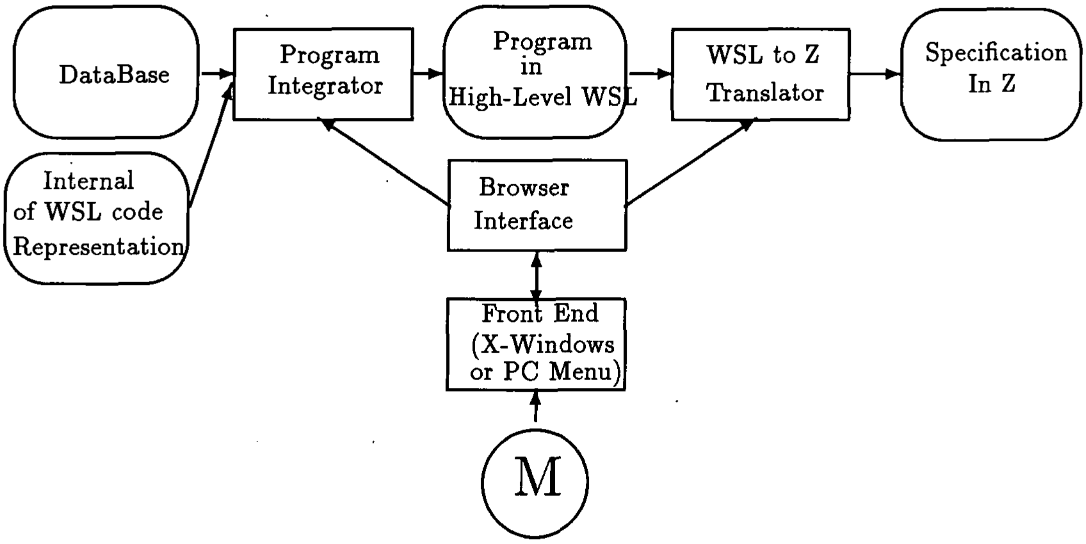


Figure 5.3: From a Program in High-Level WSL to Z Specification

- a Front End,
- a Program Structure Database,
- a General Simplifier, and
- a Program Transformer and a library of over 400 transformations.

IBM-Assembler-to-WSL Translator An IBM-assembler to WSL translator was implemented. This translation process use existing compiler writing technology.

History/Future Database This is included to allow the maintainer to go back to an older version of the program (s)he has transformed. It is usual for the maintainer to move forwards and backwards several times through a sequence of transformations in order to reach an optimal version of the program. Two commands "Undo" and "Redo" are provided. "Undo" is used to retrieve the previous version of the program, before the last program editing or transforming operation. The "Redo" command undoes the last "Undo" command.

Structure Editor The Structure Editor is usually used as a last resort to remove errors (in the code) found by the maintainer. The maintainer can select an edit command from the Front End (Figure 5.4). For example, if the "Change" button is clicked, the change menu is displayed. It allows the maintainer to change the currently selected item in the program with an item (s)he specifies or a default item of the same generic type.

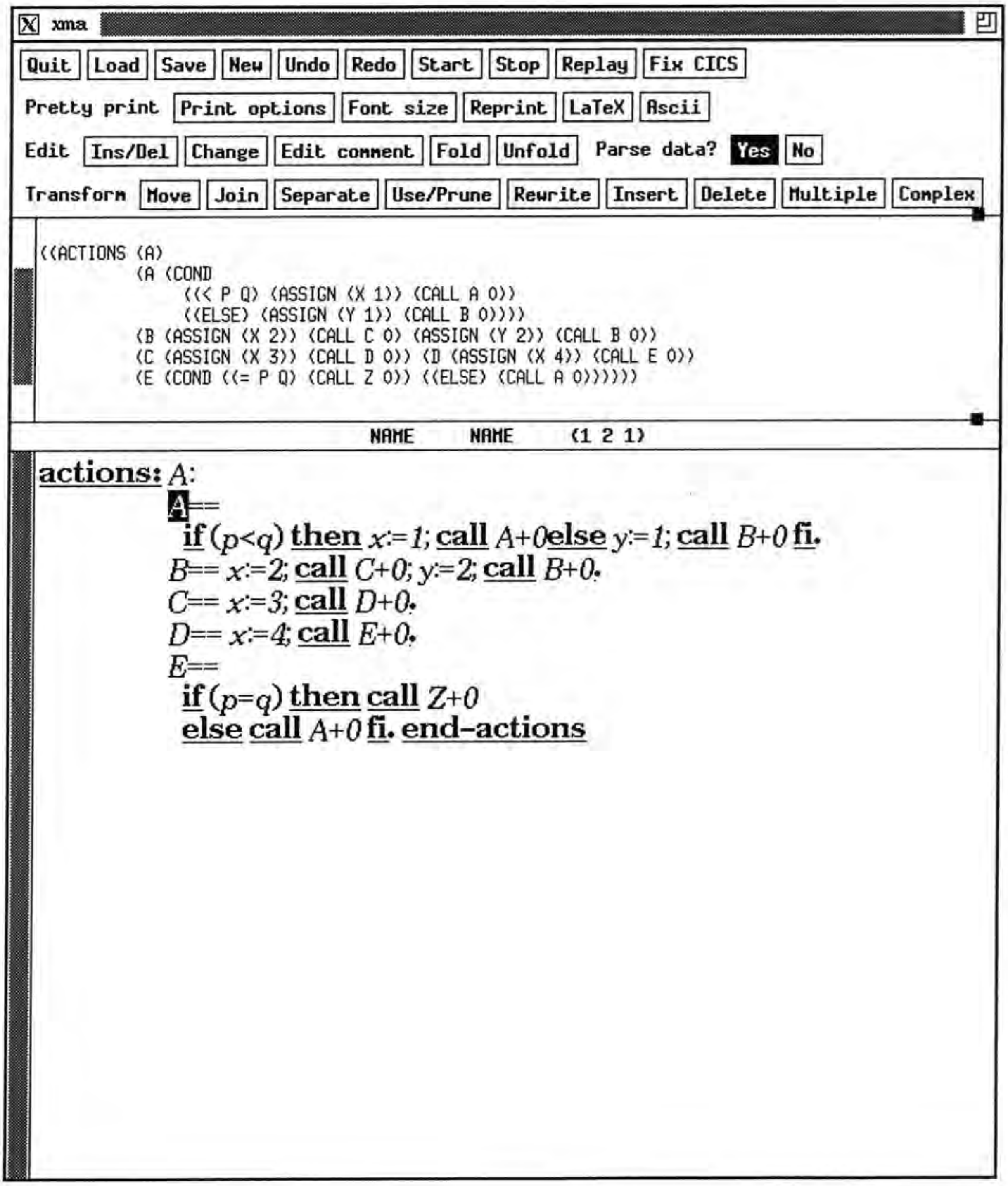
Browser and X-Window Front End The Browser and the X-Window Front End are implemented together as a graphical user interface to the other subsystems of the Maintainer's Assistant using the X-Windows System. It provides all the commands necessary to use other Maintainer's Assistant programs via buttons and pop-up menus and uses several windows to display the output from the system and to receive text input from the user. In particular it provides a browser to display the program being transformed by the Transformer, and has facilities not provided by the transformer such as pretty printing the program and a mechanism to fold or unfold sections of code. It displays a frame made up of three windows (Figure 5.4). The first window is a box containing several buttons and labels. By clicking these buttons the user can invoke various commands, change options, and pop-up other windows. The second window (the application window) is an interface to the transformer command driven user interface and the third window (the display window) is used to display the program being transformed by the user. A manual page for the front end is available for the novice user.

Program Structure Database The Program Transformer often needs to know the properties of current program item to be transformed. These properties may be used several times during the transformation process. The Program Structure Database facility can figure out the properties of the program item and save them in the database in case these properties need to be calculated several times.

The Program Structure Database is a dynamic database mainly serving the Program Transformer. The Program Transformer accesses the Database via the Database Manager. When the Program Transformer is transforming a section of program code, queries about the program are sent to the Database Manager. When a query is made for the first time, the Database Manager will go through the program structure and calculate the answer to that query. For instance, the Program Transformer may ask that "which variables are used in this section of the program?". The result will both be sent to the Program Transformer and saved in the database. Any extra information produced by the calculation, which may be used to answer other queries, will also be recorded in the database. When the question is asked again, the database manager will check the database and simply return the result.

When the Program Transformer changes the old program into a new version of the program it is necessary to create a new version of the database. The old data corresponding to the previous program is saved in the History/Future Database.

Furthermore, the Program Transformer may have also modified (or edited) the data (questions and answers) in the Database, since it performed the last



```
((ACTIONS (A)
        (A (COND
             (((< P Q) (ASSIGN (X 1)) (CALL A 0))
             ((ELSE) (ASSIGN (Y 1)) (CALL B 0))))
        (B (ASSIGN (X 2)) (CALL C 0) (ASSIGN (Y 2)) (CALL B 0))
        (C (ASSIGN (X 3)) (CALL D 0)) (D (ASSIGN (X 4)) (CALL E 0))
        (E (COND ((= P Q) (CALL Z 0)) ((ELSE) (CALL A 0))))))
```

```
                                NAME     NAME     (1 2 1)
```

```
actions: A:
          A==
           if (p<q) then x:=1; call A+0else y:=1; call B+0 fi.
          B== x:=2; call C+0; y:=2; call B+0.
          C== x:=3; call D+0.
          D== x:=4; call E+0.
          E==
           if (p=q) then call Z+0
           else call A+0 fi. end-actions
```

Figure 5.4: The X-Windows Front End

transformation, and the data related to the changed program will not be suitable for answering the same questions a second time. Thus, the database manager needs to manage the alteration in the database. LISP makes this easy to do.

The database manager will also serve the Knowledge Base System when this is implemented.

General Simplifier The Program Transformer often needs to carry out symbolic calculations in mathematics and logic. This is implemented by sending queries to the General Simplifier. It can help the Transformer by calculating the conditional statement in the program.

The Program Transformer and the Transformation Library The program transformer [37] works in an interactive mode. It goes through a piece of given code in WSL and prompts a catalogue of applicable transformations about the piece of code. These are displayed in the user interface's window system. The maintainer can instruct the system by clicking the mouse on a chosen WSL construct, and then clicking on a transformation class.

Proven transformations are stored in a library. Once a transformation is selected it is automatically applied. These transformations can be used to simplify code and expose errors. There are about 400 transformations in the library.

5.2 Review of the Maintainer's Assistant

The prototype of the Maintainer's Assistant (described in 5.1.5) is a team effort. The author contributed to the design of the Maintainer's Assistant, the implementation of the Program Structure Database and the General Simplifier. Experiments were carried out with a number of program examples using the Maintainer's Assistant. Up to the summer of 1991 (two years into the project), the following points were noticed by the author:

- Almost all program transformations in the transformation library based on Ward's work were mainly for dealing with functional abstraction (or control abstraction) — most transformations operated on control structures of a program while few transformations on data structures. In another words, the system was only suitable to operate on computation-intensive programs, not data-intensive programs. The program transformer can only deal with the construction of well-structured code.

- To obtain a specification expressed in Z is a long term goal for the REFORM project. Most of the program transformations can only be used for restructuring programs at the code level, i.e., both programs before and after the transformation being applied are in the same abstraction level.

- Most of the program transformations that currently are implemented can only be used for restructuring programs at relatively low levels of abstraction.

- No representations of types, complex data structures and data design yet exist in WSL.

- A new application area of the tool was identified as acquiring data design from data-intensive programs written in e.g. COBOL. After seeing the demonstration of the prototype of the Maintainer's Assistant, many industrialists were disappointed with the tool for being unable to deal with COBOL programs though they confirmed the potential capability of the Maintainer's Assistant.

These facts urged a new research direction to be set up within the REFORM project, i.e., acquiring data designs from data-intensive programs. In particular, the new research direction started with data-intensive programs, employing program transformation technique emphasising data abstraction and to end up with data designs. It was decided by the project leaders to start this research while the original research direction was still going on.

5.3 Recovering Data Designs

The identification of the new research direction raises a number of questions to solve, such as the method to tackle this problem, the theory of a new approach and a new tool to implement the method developed.

5.3.1 Combining Code Analysis with Data Abstraction

The motivation for acquiring a data design from data-intensive code is the same as that of obtaining a Z specification form assembler code — software can be best understood, altered and enhanced at the conceptual level rather than at the code level where the maintainers's view is often obstructed by implementation details. This means that crossing levels of data abstraction is needed to move from code to a data design.

One of the characteristics of data intensive third generation languages is that high level data designs often translate at the implementation level to constructs in both the code and data. For example, a reference in the data design between two data structures is typically implemented in COBOL by a foreign key, i.e., an integer index from one to the other. The relation between the two data structures can only be discovered by examination of the data and the code, not the data alone. Existing reverse engineering techniques have difficulty handling this. It seemed to us that formal transformation offered potential to solve this problem.

Data abstraction is widely used in forward engineering. The use of data abstraction in reverse engineering is in a primary stage. It is proposed in this thesis that the data abstraction process be carried out with the help of code analysis, because code analysis can collect information needed for data abstraction.

5.3.2 Using Program Transformations and WSL

It is considered that the approach using program transformations is also a suitable method for acquiring data designs, because performing data abstraction operations also needs the properties of program transformations, such as the preservation of semantics and suitability for tools, etc.

Although a transformational system for acquiring data designs has to cope with a different kinds of abstraction from both the systems for forward engineering and the transformation system in REFORM by then, all the program transformation systems have one thing in common, i.e., that program transformation changes the syntax but not the semantics of programs both in software development and maintenance. Therefore, a wide spectrum language is also used to build a program transformation system for acquiring data designs just as a wide spectrum language was used by some of the forward engineering transformational system (e.g., CIP project [18]) and by the Maintainer's Assistant.

5.3.3 Analysis of the Problems with Data-Intensive Programs

WSL currently has declarations which introduce the name of an identifier without its type. Therefore, variables are not typed, but all values in WSL have a type which belongs to a distinct set of values. This means that a WSL variable can at different times hold values of different types. Adding type is essential to avoid losing important attributes of the source program, such as logical connections between data. Therefore, data structuring such as records are needed. COBOL is built on a low level model of storage, involving the explicit layout of data in memory, the size of data in characters, etc. A challenging problem for reverse engineering is the use of aliasing to use memory for several purposes. Since COBOL treats all significant data as records, defining "records" in WSL for modelling COBOL records is a clear requirement.

The external calls to the underlining operating system and the embedded database can be modelled as external procedure calls and external functions. WSL already has mechanisms for dealing with external calls. The foreign key problem can be dealt with by program transformations. These transformations analyse the code with foreign keys and relations between modules using foreign keys could be found. An example will be presented in section 7.8.

Entity-Relationship Attribute Diagrams are based on *entity models* [47,53, 55,114]. Entity models provide a system view of the data structures and data relationships within the system. All systems possess an underlying generic entity model which remains fairly static in time. The entity model reflects the logic of the system data, not the physical implementation. Entity models provide an excellent graphical representation of the generic data structures and relationships. Therefore, Entity-Relationship Attribute Diagrams are suitable forms for representing data designs for data-intensive programs and WSL needed to be extended to include Entity-Relationship Attribute Diagrams.

5.3.4 A Design Recovery Method

A method for data design recovery is proposed in this section, which is illustrated in Figure 5.5. The method consists of following major steps:

1. Translating a data-intensive program (in COBOL in this case) into an intermediate language (a wide spectrum language called WSL in this case) (automatically).

2. Applying program transformation(s) to the program in the intermediate language at the code level to obtain entities and relationships also in the same intermediate language but at the conceptual level under control of human.

3. Interpreting the entities and relationships in the intermediate language into the entities and relationships in some language dedicated to a tool for displaying or printing an Entity-Relationship Attribute Diagram.

It is stressed that the main reason for using a wide spectrum language (WSL in this project) is that the language can represent both COBOL programs and Entity-Relationship Attribute Diagrams. Program transformations change programs in WSL to programs in WSL as well and also program transformations themselves are written in the language, and therefore, program transformations need just building once no matter what language in which the source code was written.

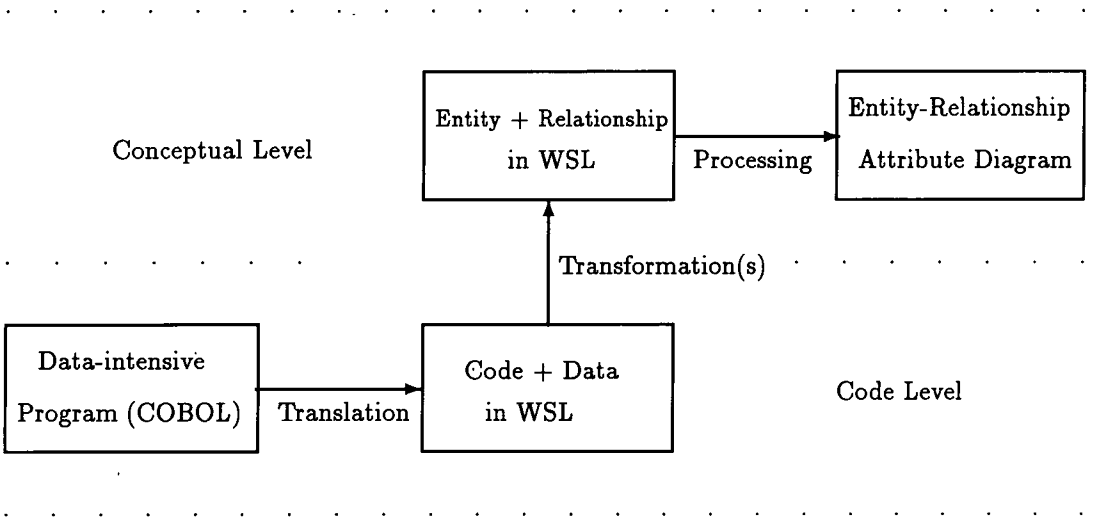

Conceptual Level
Entity + Relationship in WSL
Processing
Entity-Relationship Attribute Diagram
Transformation(s)
Data-intensive Program (COBOL)
Translation
Code + Data in WSL
Code Level

Figure 5.5: A Data Design Recovery Method

5.3.5 Enhancement Design of the Maintainer's Assistant for Acquiring Data Designs from Data-intensive Programs

To implement a tool applying the above method, the problems to be solved can clearly be summarised:

- Design and implementation of the representation for data-intensive programs: after assessing the features of data-intensive programs, the aim is to design and implement the representational form of the programs in an intermediate language on which the programs can be operated by tools. COBOL programs are used and discussed as typical data-intensive programs.

- Design and implementation of the representation for Entity-Relationship Attribute Diagrams: after assessing the features of Entity-Relationship Attribute Diagrams, the aim is to design and implement the representational form of Entity-Relationship Attribute Diagrams in an intermediate language on which they can be operated by tools.

- Development of techniques for crossing levels of data abstraction: the application of data abstraction techniques in reverse engineering is explored even though data abstraction has been widely used in forward engineering; and to develop the techniques of crossing levels of data abstraction which is extremely important to obtaining data designs from the programs.

- Design and implementation of program transformations for data design recovery: the aim is to invent and implement program transformations for manipulating both data-intensive programs (represented in an intermediate language) and Entity-Relationship Attribute Diagrams (represented in the same intermediate language).

After the working environment has been examined, it is found out that the Maintainer's Assistant needs enhancing substantially. The enhancement includes three main parts:

1. **Extension of WSL**

   WSL constructs are needed at both the code level and the conceptual level, e.g., constructs for representing COBOL programs at the code level and Entity-Relationship Attribute Diagrams at the conceptual level.

2. **Extension of Program Transformation Library**

   Program transformations are also needed to manipulate code and data at all levels, particularly for crossing levels of data abstraction.

3. **Extension of Structure Database and Design of Metrics Facility**

   New database queries are required by code analysis and transformation implementation.

   The objectives of using metrics in REFORM are to help the user to select transformations (to help develop heuristics), to measure the progress made in optimising the program code and to measure the resulting quality of the program being transformed.

Existing WSL constructs and program transformations can be directly used in conjunction with newly defined WSL constructs and newly developed transformations.

Chapter 6

Extending and Using WSL

Wide spectrum languages and program transformation techniques in general were discussed in detail in previous chapters. However, when a real program transformation system is built (as will be described in Chapter 8), many practical decisions have to be made. Therefore, the extension and use of WSL, the wide spectrum language used in REFORM, are introduced in this chapter; the definition of program transformations needed for the extension of the Maintainer's Assistant will be introduced in the next chapter.

6.1 Introduction to the WSL Extension

WSL has as its theoretical foundation a kernel language with five statements. Any other WSL constructs are either extensions to the kernel language, or to existing WSL constructs which themselves are initially derived from the kernel. This principle is observed in the research in order that proofs are not invalidated. For example, the first level language was developed observing this principle (Figure 6.1). The first level WSL is used for representing programs (which are not data-intensive) and the first level Meta-WSL is used for implementing program transformations.

The second level language needs designing and this will be discussed in detail later in this chapter. The second level WSL is needed for providing more features, which are mainly needed for representing data-intensive programs. The second

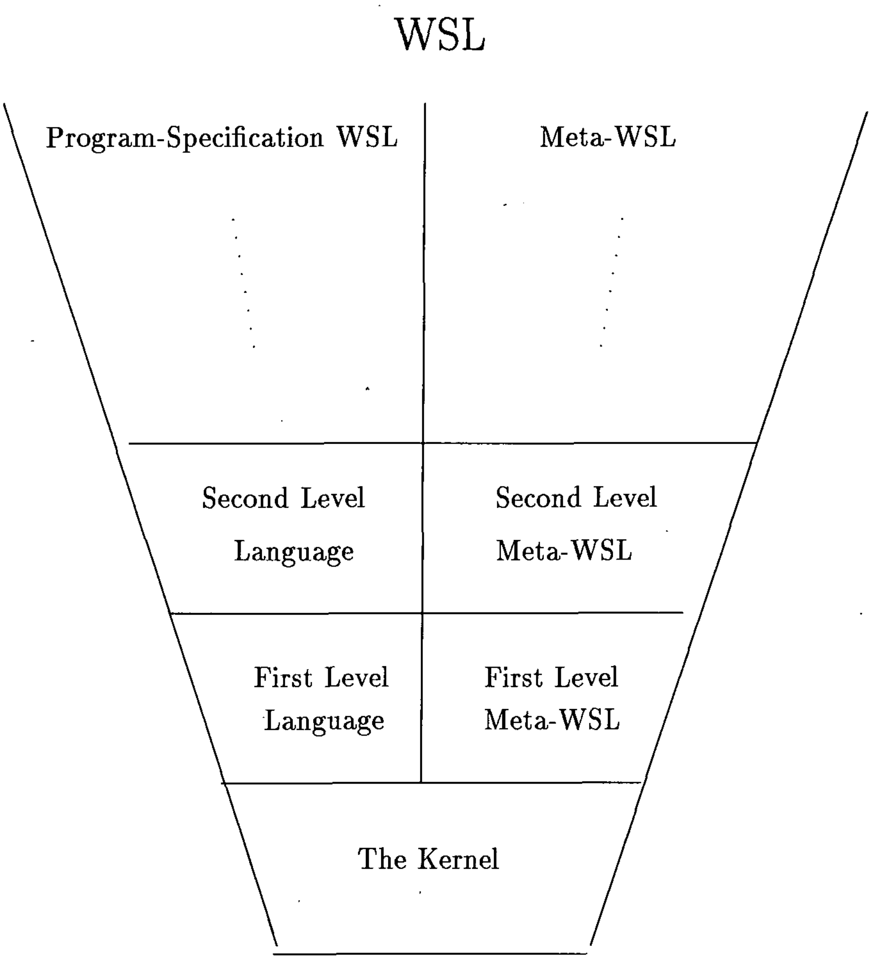


Figure 6.1: WSL Language Levels and Their Usage

level Meta-WSL is needed for implementing supporting tools, such as the Program Structure Database, the General Simplifier and the Metric Facility.

In extending WSL precisely, completely and unambiguously for representing data-intensive programs and data designs, it is essential that the syntax and semantics of such an extension should be well defined.

The specification of the semantics uses the existing WSL kernel and first level language in line with the aforementioned. The specification of the syntax is achieved with a *context-free grammar* (so called because the well-formedness of each phrase is independent of its context), using BNF notation [1].

[1] A formal definition of the syntax of a programming language is usually called a *grammar*. A

Apart from the formal specification of syntax and semantics, an informal semantics specification, possible operations on a newly defined WSL construct and an internal format are also defined, i.e., five jobs need carrying out:

1. Formal specification of syntax — this is written in BNF notation. This is also the format in PASCAL-like form to be displayed in the tool interface.
2. Informal semantics specification — this is written in English and can be used as a comment on the newly defined component by the implementer and the user of the language.
3. Formal semantics specification — this is written in existing WSL whose semantics is denotational semantics.
4. Operations — these are written both in English and WSL to describe available operations on the newly defined construct.
5. Internal format (LISP format) — this is the same LISP form in which WSL is represented and is used for writing program transformations.

6.2 Extension of WSL

As introduced in section 4.3.3, fundamental data types in programming languages include *unstructured data type*, *Cartesian product*, *discriminated union*, *array*, *set*, *sequence*, *sparse data structure* and *pointer*. We shall extend WSL to have all these types to meet the needs of dealing with data-intensive programs. However, it must be ensured that adding a type does not invalidate existing proofs of equivalence and transformations.

grammar consists of a set of definitions (termed *rules* or *productions*) which specify the sequences of characters (or lexical items) that form allowable programs in the language being defined. A *formal grammar* is just a grammar specified using a strictly defined notation and the best-known notation is *BNF* [8,131,159]. In the original version of BNF, nonterminal symbols were written with angle brackets to distinguish them clearly from terminal symbols. In this thesis, a distinctive font, such as **Program**, is used for this purpose. In addition, the *EBNF*, or Extended Backus-Naur Form [159], is also used in this thesis.

WSL uses the concept of most fundamental data types to define its kernel but most of these data types were not supported by WSL language. WSL provides only *unstructured data type* and *array* for representing programs to be manipulated in the first level language. More data types, such as *sequence*, and its operations should be added to the language. Also, though *set* was already defined in the first level language, more operations such as "insert" are needed. Since COBOL does not support sparse data structures, recursive data structures and pointer types (COBOL programs do have these data types but programmers get round them by using foreign keys, etc.), these data types are not considered for acquiring data designs in this thesis.

COBOL records and "redefine" structures can be viewed as Cartesian products and discriminated unions respectively. They will be defined first in this section together with "file" structure. Then WSL components for supporting data types *set* and *sequence* will be defined, together with two other commonly used data types, *stack* and *queue*. A structure for user-defined abstract data types will also be defined. At the end of this section structures for representing Entity-Relationship Attribute Diagrams will be introduced into WSL.

6.2.1 Representing Records and Files

Records

As discussed earlier in this thesis, COBOL (and also Assembler) is built on a low level model of storage, which includes how data is laid out in memory, size of data in characters, etc.

The particular problem of aliasing, at memory level, means that to solve this problem, low level data information needs to be kept in the WSL translation of COBOL (there is a similar problem in FORTRAN with Common and Equivalence statements).

Since COBOL treats all significant data as records, defining "records" in WSL for modelling COBOL records will receive close attention.

Unless the WSL is extended (WSL has only untyped variables), the only way to model COBOL records is to use simple variables in existing WSL, i.e., COBOL records are translated into simple variables. It would be very difficult to derive Entity-Relationship Attribute Diagrams from these simple variables, because useful information contained in the COBOL records would be lost when these records are first translated into WSL simple variables. For example, the definition of a COBOL record includes the type of the record (character, integer, etc.), the length of the record (number of bytes in the memory) and the relationship between the record and its parent record. This information which is not represented by simple WSL variables is vital for deriving Entity-Relationship Attribute Diagrams and should not be thrown away (or at least not at this early stage of reverse engineering).

Therefore, extending the existing WSL to represent COBOL recods is necessary. The new construct in WSL is also called a *record.*

The five-step process for defining "records" in WSL in this section is illustrated in detail. Owing to the length limitation of the thesis, other new WSL components will not be presented individually in the main text. The full specification of the syntax of the WSL extension developed in this research can be referred to in Appendix A and the full semantics specification in Appendix B.

A WSL **record** is defined in:

1. Formal specification of syntax

   Record-Def

   ::= record Rec-Identifier [Integer-Literal of Type-Char-Literal] end;

   | record Rec-Identifier with Records-Def end;

   Records-Def

   ::= Record-Def | Record-Def Records-Def

   Rec-Identifier ::= Identifier

   Type-Char-Literal ::= char | int

   Note: please see Appendix A for the definitions of Identifier and Integer-Literal.

2. Informal specification of semantics

   This is the WSL declaration for a record variable whose components are variables capable of selective updating. A record can be declared either with *type* and *length* in terms of sequence of bytes in memory or with other records ("subrecord") as its components. A record has its own name (*identifier*). The type of such a record is associated with the *Cartesian product* of the domains with which the component records ("subrecords") are associated. For example, expression *rec*1.*rec*2 may then be type checked by requiring that the type of *rec*1 be a record type having *rec*2 as a record name. Then the type of the whole expression is the corresponding type of the lowest level record, i.e., *rec*2 in this case. A record can be recursively defined.

3. Semantics specification

   A record is defined in terms of sequence in the kernel language, e.g.:

```
record x with
      record i [6 of int]
      record j [6 of int]
      record k [6 of char]
  end;
```

$$=_{DF}\ x \in \{\langle i,j,k\rangle | i \in I \wedge j \in J \wedge k \in K\}$$

$$=_{DF}\ x \in I \times J \times K$$

where "6 **of** *int*" means an integer of 6 digit position and "6 **of** *char*" means a string of 6 characters; I, J, K are sets of values, i.e., I, J ∈ { 0 ... 999999 } and K ∈ { ⟨ ’A’, .., ’Z’, ’a’, .., ’z’ ⟩ ⟨ ’A’, .., ’Z’, ’a’, .., ’z’ ⟩ ⟨ ’A’, .., ’Z’, ’a’, .., ’z’ ⟩ ⟨ ’A’, .., ’Z’, ’a’, .., ’z’ ⟩ ⟨ ’A’, .., ’Z’, ’a’, .., ’z’ ⟩ ⟨ ’A’, .., ’Z’, ’a’, .., ’z’ ⟩ }.

If another record is declared as:

```
record y with
      record l [6 of int]
```

record *m* [6 **of** *int*]

record *n* [6 **of** *char*]

end;

the semantics of $x \; := \; y$ is defined as:

$$
\begin{aligned}
& x := y \\
& =_{\text{DF}} \; x.i := y.l \;\wedge\; x.j := y.m \;\wedge\; x.k := y.n \\
& =_{\text{DF}} \; \langle i, j, k \rangle := \langle l, m, n \rangle
\end{aligned}
$$

4. Operations of records

   Two types of operations are available on records. When two records have an identical structure, one record can be assigned to another, e.g., as shown in the above semantics definition. Secondly, an expression can be assigned to a record whose type should be the same as that of the expression.

5. Internal format

   For example, if the display format of a record is:

   record Rec-Identifier [Integer-Literal **of** Type-Char-Literal] **end**;

   the internal format will be:

```
(record rec-identifier length-literal type-char-literal)
```

When a record is defined in WSL it can be used to represent a COBOL record. For instance, the WSL record

record *student-info* **with**

  record *student-id-no* [*8* **of** *int*]

  record *name* **with**

    record *first-name* [*9* **of** *char*]

    record *last-name* [*11* **of** *char*]

```
    end
  record address with
    record street [15 of char]
    record city [10 of char]
    record county [15 of char]
    record postal-code [8 of char]
    end
  record phone [10 of int]
end;
```

represents a segment of COBOL program:

```
01 STUDENT-INFO.
   05 STUDENT-ID-NO            PIC 9(8).
   05 NAME.
      10 FIRST-NAME            PIC X(9).
      10 LAST-NAME             PIC X(11).
   05 ADDRESS.
      10 STREET                PIC X(15).
      10 CITY                  PIC X(10).
      10 COUNTY                PIC X(15).
      10 POSTAL-CODE           PIC X(8).
   05 PHONE                    PIC 9(10).
```

Aliased Records

Another reason that simple WSL variables cannot cope with representing "structures" such as COBOL records and "record" has to be defined in WSL is the aliasing problem. There are two purposes in using aliases. The first purpose is to share storage space. In the early days of computing when memories were very expensive, it was a "skill" that several variables used the same piece of storage at separate times. The second purpose was to control storage scope. In early languages, which did not have block structure, there was no way to use scope control storage allocation so the only “skill” was to use aliasing, where an alias is used to write a value of one type and read out as a value of another type. This “skill” brought enormous difficulties to reverse engineering today.

For example, the following is a COBOL program with aliased records:

```
01 X.
   03 FOO                PIC X(6).
   03 BAR                PIC X(6).

01 Y REDEFINES X.
   03 TOP                PIC X(4).
   03 MIDDLE             PIC X(4).
   03 BOTTOM             PIC X(4).
```

When the value of x.foo is changed, the values of both y.too and y.middle are also changed.

Aliasing is surely something to be removed in reverse engineering. There might be many solutions but one method is proposed in this research. To get rid of aliasing, the first step may still have to include representing the aliased records in WSL. Apart from defining “records” in WSL, a WSL construct called “redefine” is also needed for representing the aliased COBOL records (in the terms of mathematics, records aliasing can be viewed as a discriminated union) and its WSL external format is:

redefine *record-name1* **with** *record-name2*;

Then the above COBOL program with aliased records can be translated into[2]:

[2] Whether variables are aliased in COBOL can be detected by the COBOL keyword “REDEFINES”. However, the detection of aliased variables in other data-intensive programming languages such as pointers in C may need some further study.

```
record x with
    record foo [6 of char]
    record bar [6 of char]
  end;
record y with
    record top    [4 of char]
    record middle [4 of char]
    record bottom [4 of char]
  end;
redefine x with y;
```

In terms of the WSL kernel language, x occupies a sequence of twelve bytes of memory, i.e., $\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}\rangle$, where $x_i \in \{0..255\}$. This means $x.foo \times x.bar = \langle x_1, x_2, x_3, x_4, x_5, x_6\rangle \times \langle x_7, x_8, x_9, x_{10}, x_{11}, x_{12}\rangle$.

Similarly, we have $y = \langle y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}, y_{12}\rangle$, and $y.top \times y.middle \times y.bottom = \langle y_1, y_2, y_3, y_4\rangle \times \langle y_5, y_6, x_7, x_8\rangle \times \langle x_9, x_{10}, x_{11}, x_{12}\rangle$.

However, the "redefine" statement specifies that the record x and the record y use the same segment of the underlying memory, i.e., $\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}\rangle$ and $\langle y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}, y_{12}\rangle$ represent the same memory. If a new sequence m is given to represent this memory, we have:

$x = y = \langle m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{11}, m_{12}\rangle$; $x.foo \times x.bar = \langle m_1, m_2, m_3, m_4, m_5, m_6\rangle \times \langle m_7, m_8, m_9, m_{10}, m_{11}, m_{12}\rangle$; and $y.top \times y.middle \times y.bottom = \langle m_1, m_2, m_3, m_4\rangle \times \langle m_5, m_6, m_7, m_8\rangle \times \langle m_9, m_{10}, m_{11}, m_{12}\rangle$ (Figure 6.2).

There are two issues to address in solving the aliasing problem: it is necessary to determine which records are aliased; and, more challenging, it is necessary to determine a mapping between the different records, based on the memory used by each component of the record. The former can usually be determined by the declarations and the latter can be done by defining a function (or procedure) which maps from a record to a sequence of bytes (the representation of that record in memory), and from a sequence of bytes to a record. These functions (or procedures) need to know the *structure* of the record in terms of the number of bytes occupied by each component. Thus a "write" to aliased memory is described by

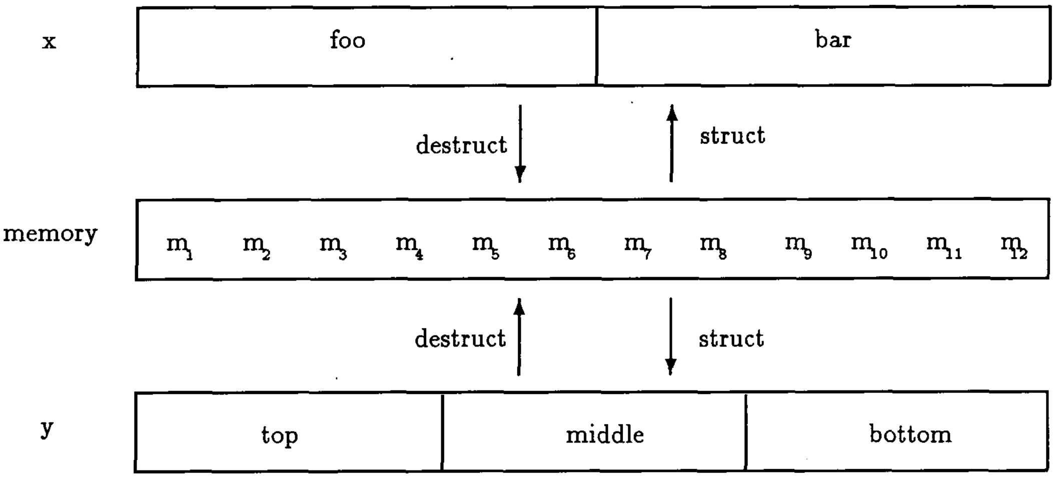

x
foo
bar
destruct
struct
memory
m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8 m_9 m_{10} m_{11} m_{12}
destruct
struct
y
top
middle
bottom

Figure 6.2: Storage Model

a function (or procedure) which maps the COBOL data structure to low level memory; a "read" is represented by a function (or procedure) which describes the mapping in the reverse direction. In our system, these functions (or procedures) are explicitly inserted, in preparation for later simplification using transformations. Because the use of these functions (or procedures) involves accessing the underlying storage (which can be viewed as a hidden state), the concept of an abstract data type is employed here.

If the following statement occurs in the procedure division of the original COBOL program shown previously in this section,

MOVE "PASSED" TO FOO.

we will translate it into five WSL statements (last four statements are added in to address the aliasing problem):

x.foo := "passed";
redefine-write(*x.foo*, "passed");

```
y.top := redefine-read(y.top);
y.middle := redefine-read(y.middle);
y.bottom := redefine-read(y.bottom);
```

where;

```
proc redefine-write(rec, rec-value) ≡
     adt_proc_call redefine-record.destruct (rec, rec-value, state-variable)


funct redefine-read(rec) ≡
     adt_funct_call redefine-record.struct (rec, state-variable).
```

and

```
adt redefine-record (state-variable)(parameters) $=_{DF}$
proc destruct(rec, rec-value, state-variable)
  var (sub-state-variable) :
     sub-state-variable := truncate(rec, state-variable);
     sub-state-variable := if   int-type?(rec) → byte(rec-value)
                              □ char-type?(rec) → ascii * (break(rec-value))
                                         fi end;

funct struct(rec, state-variable) $=_{DF}$
  if   int-type?(rec) → integer(truncate(rec, state-variable))
    □ char-type?(rec) → chr(truncate(rec, state-variable)) fi
    tda.
```

The procedure *redefine-write* and function *redefine-read* will call the corresponding "abstract data type" procedures, *redefine-record.destruct* and function *redefine-record.struct*, and the variable *state-variable* is the hidden state in the abstract data type.

The procedure *destruct* maps a record to a sequence of n bytes (where n is length of the record). The function *struct* takes a record and a sequence of bytes and updates the record so that its destructured representation is the sequence of bytes[3].

In the definition of the abstraction data type, *truncate*(*rec*) returns the segment of underlying state which corresponds to the record *rec* and the length of the segment is the number of bytes of the record *rec*; *int-type?*(*rec*) returns true when *rec* has a type of "integer"; *char-type?*(*rec*) returns true when *rec* has a type of "char"; *byte*(*rec*) returns a sequence of n integers giving the n-byte representation of *rec* (n is the length of *rec*); *break*(s) returns all components of *rec*; *ascii*(s) returns the ASCII value of character of s; *integer*(*rec*) returns an integer assembled from the individual memory bytes; *chr*(*rec*) returns a character string converted from the individual memory bytes[4].

In the above example,

redefine-write (x.foo, "passed");

⇒ <u>**adt_proc_call**</u>*redefine-record.destruct* (*x.foo*, *"passed"*, m)

⇒ *sub-state-variable* := *truncate* (*x.foo*, m);

⇒ *sub-state-variable* := $\langle m_1, m_2, m_3, m_4, m_5, m_6\rangle$;

⇒ $\langle m_1, m_2, m_3, m_4, m_5, m_6\rangle$:= ascii ∗ (break("passed"));

⇒ $\langle m_1, m_2, m_3, m_4, m_5, m_6\rangle$:= ascii ∗(⟨"p", "a", "s", "s", "e", "d"⟩);

⇒ $\langle m_1, m_2, m_3, m_4, m_5, m_6\rangle := \langle 112, 97, 115, 115, 101, 100\rangle$;

y.top := *redefine-read*(y.top);

⇒ y.top := <u>**adt_funct_call**</u> *redefine-record.struct* (*y.top*, m);

⇒ y.top := *chr*(*truncate* (*y.top*, m));

⇒ y.top := *chr*($\langle m_1, m_2, m_3, m_4\rangle$);

⇒ y.top := *chr*($\langle 112, 97, 115, 115\rangle$);

⇒ y.top := "pass";

[3] The procedure *destruct* and the function *struct* here have only dealt with integer and character types; it would be necessary to extend the definitions of the procedure and the function when additional types such as reals (as found in C or FORTRAN) are introduced to WSL.

[4] It should be recognised that aliasing has to take account the underlying machine representation of the data. The definitions of the functions in this paragraph are all based on that the "byte" of the underlying memory is represented in ASCII code. Different function definitions would be needed with other types of representation for underlying memory model.

y.middle can be worked out in a similar way. Since *y.bottom* is not affected by the change of *x.foo*, the statement

y.bottom := *read-rec*(*y.bottom*);

will be made redundant by transformations.

The above example showed the way in which how an alias used for the first purpose (storage sharing) was dealt with. In fact, the variable usage was disjoint, and although very complicated WSL was generated from the alias and much of the WSL can be removed as it was redundant (i.e., aliased variables can be treated separately). It was only needed when aliasing was used for the second purpose (storage allocation control), which is a more difficult use to handle (an example of this case will be addressed in one of the case studies in a later chapter).

Files

A file is defined as a sequence of records, and its external WSL format is:

File-def ::= file File-Identifier with Records end;

Records ::= Record-Def | Record-Def Records

The operations of files will be discussed in a later section.

6.2.2 Representing Basic Data Types and User-Defined Abstract Data Types

Basic Data Types

The main concern in defining the new components described in this section is still to prevent loss of information at an early stage of reverse engineering. For instance, sequence, queue and stack types can be modelled in theory by array, which exists in WSL. But if data in these types in data intensive programs (written in programming languages other than COBOL) are all translated directly to data in array type, the properties of the original data may be lost immediately. The idea used in this thesis is to define new WSL constructs supporting types such as sequence, queue and stack first and then build program transformations to handle data abstraction. More new components defined will be listed in the following subsections, where it is implicit that the reason is to prevent information loss.

New components for supporting basic data types, *set*, *sequence*, *queue* and *stack* are defined for the second level WSL with their usual meanings. Detailed definitions can be seen in Appendix A. In this section, only how new components are supporting a queue data type is defined.

A WSL **queue** and operations on queue are defined as:

- Formal specification of syntax

  Command

  ::= **init-q** Q-Variable;
  | **q-append** Q-Variable Expression;

  Expression

  ::= **q-concat**(Q-Variable1 Q-Variable2)
  | **q-rem-first**(Q-Variable)
  | **q-length**(Q-Variable)

- Informal specification of semantics

  A queue is a sequence in which component selection and deletion are restricted to one end and insertion is restricted to the other end. The concatenation of one queue to another will form a third queue.

- Semantics specification

  Suppose s, s_1 and s_2 are sequences, p, p_1 and p_2 variables, e an expression, and x and y also variables:

$$\underline{\textbf{init\_q}}\ p \ =_{\scriptscriptstyle DF}\ p := s$$

$$\underline{\textbf{q\_append}}\ p\ e \ =_{\scriptscriptstyle DF}\ p := \langle s[1], s[2], ..., s[n], e \rangle$$

$$x := \underline{\textbf{q\_concat}}(p_1\ p_2) \ =_{\scriptscriptstyle DF}\ x := \langle s_1[1], s_1[2], ..., s_1[n], s_2[1], s_2[2], ..., s_2[n] \rangle$$

$$x := \underline{\textbf{q\_rem\_first}}(p) \ = \ _{\text{DF}} \ x := s[1] \wedge p := \langle s[2], s[3], ...s[n] \rangle$$
$$y := \underline{\textbf{q\_length}}(p) \ = \ _{\text{DF}} \ \ell(s)$$

- Internal format

the internal formats of the above five constructs will be:

```
(Init-Q P)
(Q_Append P E)
(Assign (X (Q_Concat P1 P2)))
(Assign (X (Q_Rem_First P)))
(Assign (Y (Q-Length P)))
```

User-Defined Abstract Data Type

As introduced earlier, an abstract data type consists of "objects" and "operations". Objects are usually implemented as variables and operations are implemented as procedures and functions. In reverse engineering, an abstract data type may be formed by looking for a closure of a group of variables and a group of procedures (or functions). No matter whether a closure was originally used for an abstract data type, if an abstract data type is obtained from this closure in the code, it is helpful in viewing the code at a higher abstraction level. To cross levels of data abstraction by looking for user-defined data types is a novel contribution of this thesis.

The way of implementing this idea is to provide a structure first in WSL. Five constructs are defined for the definition of a user-defined data type, user-defined data type procedure call and user-defined data type function call. The key words for these constructs are: **user-adt**, **user-adt-funct**, **user-adt-proc**, **user-adt-funct-call** and **user-adt-proc-call** (refer to Appendix A for details). The application of these constructs will be described in the following chapters.

A few previous projects have addressed recognising user-defined abstract data types and Canfora's work [45] is a typical example. The method proposed in [45] is based on user defined data types and exploits the relationships existing between these types and the procedure-like components that use them in their headings (i.e., declared formal parameters and/or return values). [45] also proposed a logic based approach to the definition of both the candidature criterion and the model to apply it. According to such an approach the model to apply the candidature criteria consists of a set of direct relations which summarise and describe the meaningful relationships among the components of a software system. A candidature criterion consists of summary relations obtained by combining direct relations in expressions. The application of a logic-based candidature criterion requires (i) a repository to collect the direct relations produced by the static code analysis; (ii) a query language to express the abstractions to be looked for and (iii) a formalism to link the direct relations to form the summary relations which will enable the above queries to be answered. Canfora's algorithm is much more complex than the one proposed in this thesis and therefore, his algorithm has not been implemented in the thesis (however, this could be easily included if necessary).

6.2.3 Representing Entity-Relationship Attribute Diagrams

Entities and their relationships are objects at high abstraction levels. To cross levels of abstraction and to represent Entity-Relationship Attribute Diagrams, entities and their relationships have to be defined in WSL. Statements representing entities and their relationships are called *specification statements* because they represent program designs, which indicate what programs do without saying how they do it.

Specification statements cannot be executed and therefore, there are usually no operations on them within the language. Their semantics can usually be defined as pairs, triples, quadruples, etc. However, they can be operated on by program transformations. For example, many of the simple statements may be interchanged directly with these specification statements. Also, the specification statement can be mixed freely with other statements because of the wide spectrum nature of WSL. Three specification statements are defined in this research.

A Construct for Relating Two Data Objects

In the process of abstraction, some data objects may be in conceptual form and others may be still in code form. If these two kinds of data objects need to occur in the same program statement, a WSL construct for relating these data objects is needed. This construct is defined as **relate**. The semantics of the construct is a pair. For example, the following WSL statement represents that the first record (or entity) is related to the second record (or entity).

Relate-Def
 ::= **relate** Rec-Identifier1/Ent-Identifier1 **to** Rec-Identifier2/Ent-Identifier2;

In the term of WSL kernel, this means that these two objects form a pair.

Definitions of Entity and Relationship

Entities and entity relationships are defined as:

Entity-def
 ::= **entity** Ent-Identifier **end**;
 | **entity** Ent-Identifier **with** Attributes **end**;

Attributes
 ::= **attr** Attribute-Identifier
 | Attributes **attr** Attribute-Identifier

Relationship-Def
 ::= **relationship** **entity** Ent-Identifier1
 has Relation-Degree1 Relationship-Name
 relation **with** Relation-Degree2 **entity** Ent-Identifier2;

$$=_{DF} \; (\; \langle \text{ Ent-Identifier1, Relation-Degree1 } \rangle, \; \langle \text{ Ent-Identifier2, Relation-Degree2 } \rangle \;) \in \text{Relationship-Name}$$

i.e., a relationship is a mathematical "relation" of two sequence in which the first element of the sequence is the object and the second element is the number of times it appears.

The corresponding diagrams are shown in Figure 6.3 for the following examples of Entity-Relationship Attribute diagrams in WSL display format:

(A)

```
entity E1 with
      attr A1
      attr A2
      attr A3
      attr A4
   end;
```

(B)

```
entity E2 end;
```

(C)

```
paragraph
  entity E3 end;
  entity E4 end;
  relationship entity E3 has one R1 relation with many entity E4;
end;
```

(D)

```
paragraph
  entity E5 end;
  entity E6 end;
  relationship entity E5 has one R2 relation with one entity E6;
end;
```

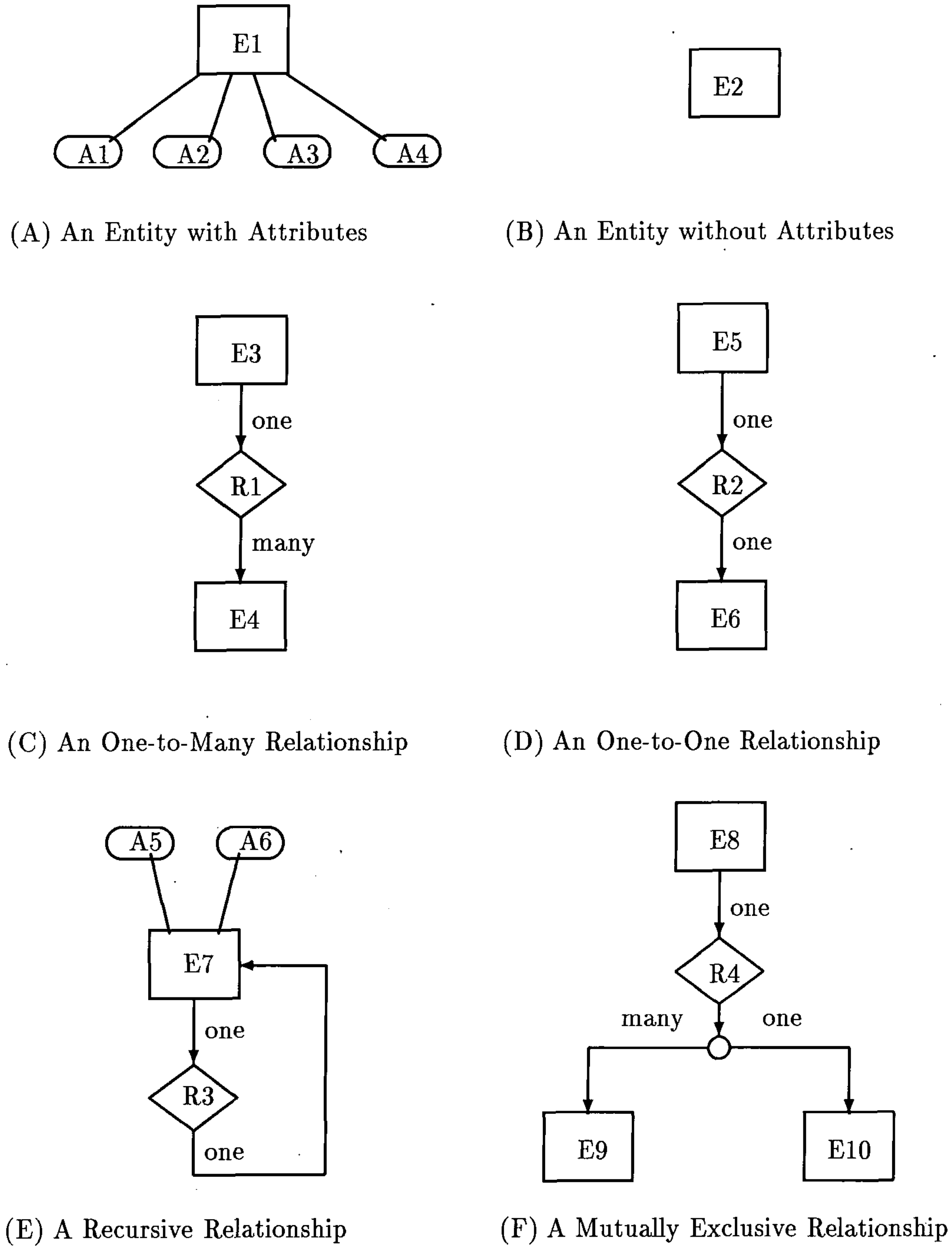
(E)

```
paragraph
```

```
  entity E7 with
          attr A5
          attr A6
   end;
  relationship entity E7 has one R3 relation with one entity E7;
end;
(F)
paragraph
  entity E8 end;
  entity E9 end;
  entity E10 end;
  relationship entity E8 has one R4 relation with
                      {many entity E9} or {many entity E10};
end;
```

6.3 Extension of Meta-WSL

As introduced in the previous chapter, the Maintainer's Assistant consists of several supporting tools. When the new research direction — recovering data designs — was decided, the way of implementing those supporting tools was also reviewed. Because the Program Structure Database, the General Simplifier and the Metric Facility are most relevant to data abstraction, they are discussed in the thesis. It is found necessary that each single service provided by the supporting tools, such as a database query or a metric measure, should be a Meta-WSL procedure or function. This is because each Meta-WSL construct can be defined by existing WSL constructs and this approach will make the prototype and future extension of the prototype theoretically well-founded. Therefore, the services provided by these three supporting tools are defined in terms of Meta-WSL. Detailed Meta-WSL extension for the three supporting tools are systematically defined in Appendix C. According to the definition, the design and the implementation of these three tools will be discussed in detail in Chapter 8.



(A) An Entity with Attributes

(B) An Entity without Attributes

(C) An One-to-Many Relationship

(D) An One-to-One Relationship

(E) A Recursive Relationship

(F) A Mutually Exclusive Relationship

Figure 6.3: Entities and Relationships in WSL

6.4 Embedding WSL in COMMON LISP

One of the advantages of formally defining the semantics of a language (using denotational semantics) is that the effect of a program written in the language can be obtained by analysing the semantics of the program rather than actually executing the program. This suggests that the language used to represent programs to be transformed in a transformation system can have some non-executable statements. The WSL used in the Maintainer's Assistant is such a language, which can have non-executable statements for representing objects at a high abstraction level, such as Entity-Relationship Attribute Diagrams.

In essence, the effect of applying program transformations on the programs represented in WSL is to change the syntax of those programs in WSL. It is crucial to understand how the WSL is represented in the Maintainer's Assistant.

WSL is embedded in COMMON LISP. WSL has two forms, external form and internal form. The external form uses familiar notations which are commonly used in programming language such as ALGOL and PASCAL. The internal form uses syntax trees on which transformations are easily performed. The external form is more user-friendly than the internal form. The interface of the Maintainer's Assistant converts between the external and the internal forms.

In a syntax tree, each node (and its corresponding subtree) represents a single syntactic object. The branches of that node are the components of the object. For instance, a portion of the tree for an assign statement is shown in Figure 6.4. In fact, the syntax tree shown in the diagram is a simplified internal form. An internal form of WSL also includes the database tables, embedded comments, and some information relating to the type of each item.

The figure shows an "Assign" statement represented as a tree. The "Assign" statement has as its component one "Assignment", which has as its component a variable which will be assigned to an expression. This piece of code would itself be part of a large structure. At the top level, a program is a single "Statements" object; that is, it consists of a sequence of statements.

In order to enable the LISP program to work on these trees they are represented internally as nested lists, so that the code above would be stored as

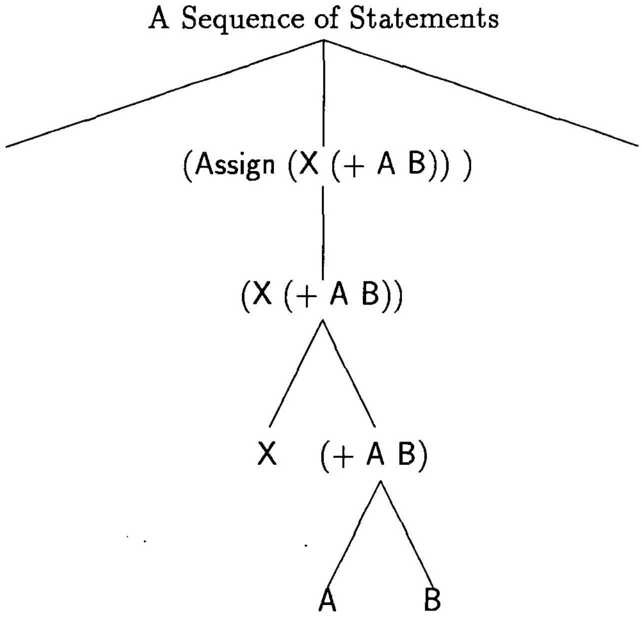


Figure 6.4: Diagrammatic Form of a WSL Syntax Tree

(Assign (X (+ A B))).

The main advantages of adopting the tree-based approach include:

- It is relatively easy to construct an interpreter to execute most WSL code within LISP;
- We can then "move" through the program (left, right, up and down) in order to select the piece of code, that is a leaf or branch of the tree, that we wish to transform;
- The transformations work mostly on syntactic objects, and these can be easily manipulated as branches or leaves within the tree structure.

From the representation of WSL programs, it can been seen that applying a program transformation to a part of a program is equivalent to replacing that part with something which is syntactically different but semantically the same. In particular, the core activity is to search a node in the syntax tree which meets certain conditions, and replace this node with another node according to the rule defined in the transformation.

6.5 Translating Data Intensive Programs to WSL

Although the discussion in this section employs COBOL as the data-intensive programming language, the results of the discussion should be applicable to other data-intensive programming languages.

6.5.1 Consideration and Decision

As a COBOL to WSL translator is not yet available, translating COBOL programs to WSL has been done manually in the research described in this thesis.

A set of general rules of translation was first established based on the features of the two languages, including a mapping table between some COBOL constructs and their equivalent WSL constructs. For instance, translating a COBOL verb **MOVE** to a WSL assignment statement is one such rule.

Translation is carried out entirely according to the semantics of the programs. For example, a COBOL record can be initialised in its declaration by a **VALUE** construct:

```
01 YEAR PIC X(4) VALUE "1994".
```

Since WSL records cannot be initialised in a similar way, the above COBOL statement will be translated into two WSL statements:

```
record year [4 of char] end;
year := "1994";
```

As manual translation is an informal process, it is impossible to prove that the translation is correct. Therefore, the approach used in the research is to make the translation rules as simple as possible, taking pains to capture *all* the effects of each COBOL statement. Once these are captured in WSL (a formal language), program transformation can be used to eliminate any redundancies which this simple approach to translation may have introduced.

Accurate translation of every COBOL construct may not be practical or even desirable: getting the last few percent of the language to translate accurately may increase the amount of effort by an order of magnitude. In this research the occurrences of those constructs which are rarely used are treated as special cases, e.g., the COBOL verb **ALTER**.

A typical COBOL program has four divisions[3,91,92]:

1. Identification Division. This division identifies a program and its origin.
2. Environment Division. This division may contain a Configuration section that defines the computer environment and an Input-Output section that defines the input/output devices.
3. Data Division. This division contains sections that define data and areas that a program references.
4. Procedure Division. This division contains all the executable statements that perform the program's logic and processing.

The Identification Division and the Environment Division are not directly translated into WSL, but the information in these two divisions might be used in translating the other two divisions. The Data division and Procedure Division are directly reflected in the WSL programs.

6.5.2 An Example of Translating A COBOL Program into WSL

The example program used in this section was taken from a COBOL text book [92] and its COBOL source code is as follows:

```
************************************************************
*                                                          *
*                                                          *
*    THIS PROGRAM SEQUENTIALLY ACCESSES TO TWO SEQUENTIAL  *
*    FILES, ONE IN INPUT MODE AND ONE IN OUTPUT MODE.      *
*                                                          *
************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COPY-CUSTOMER-LIST.

 ENVIRONMENT DIVISION.
     INPUT-OUTPUT SECTION.
     FILE-CONTROL.
          SELECT CUSTOMER-LIST ASSIGN TO XYZ
              ORGANISATION SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.
          SELECT CUSTOMER-LIST-BACK ASSIGN TO WXY
              ORGANISATION SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.

 DATA DIVISION.
     FILE SECTION.
     FD CUSTOMER-LIST.
     01 CUSTOMER-RECORD.
        02 NAME                  PIC X(20).
        02 ADDRESS               PIC X(50).
        02 PHONENUM              PIC X(20).

     FD CUSTOMER-LIST-BACKUP.
     01 BACKUP-RECORD.
        02 B-NAME                PIC X(20).
        02 B-ADDRESS             PIC X(50).
        02 B-PHONENUM            PIC X(20).

     WORKING-STORAGE SECTION.
     01 EOF PIC X.

 PROCEDURE DIVISION.
 MAIN.
     OPEN INPUT CUSTOMER-LIST
          OUTPUT CUSTOMER-LIST-BACKUP
     PERFORM, WITH TEST AFTER, UNTIL EOF = "T"
          READ CUSTOMER-LIST NEXT;
          AT END
              MOVE "T" TO EOF
          NOT AT END
              MOVE "F" TO EOF
              MOVE CUSTOMER-RECORD TO BACKUP-RECORD
              WRITE BACKUP-RECORD;
     END-PERFORM
* THE STOP RUN STATEMENT CLOSES THE FILES
     STOP RUN.
```

Organisation		Sequential			
Access Method	Operation Open Mode	I	O	I/O	EX
Sequential	Read	×		×	
	Read Next	×		×	
	Write		×		×
	Rewrite			×	

(× indicates where an operation is available.)

Figure 6.5: COBOL Sequential Files

The identification division is translated into a **comment** statement. Information in the environment division will be used when data division and procedure division are translated, i.e., the files in the code are sequential files.

In the data division, COBOL records and files are translated into WSL records and files. COBOL files sequentially organised and sequentially accessed (see Figure 6.5) are used to illustrate the method developed in this thesis. The same principle can be applied to COBOL indexed and random access files by modelling them using arrays which are available in WSL.

The following file operations are translated into WSL as external procedures (denoted by **!p** which is the WSL function existing in the first level language to call an external procedure for which it is known definitely which variables will be changed) or external functions (denoted by **!f** which the the WSL function existing in the first level language to call a named external function):

COBOL Constructs	WSL Constructs
OPEN	!p open_file
CLOSE	!p close_file
READ	!p read_file
READNEXT	!p readnext
WRITE	!p write_file
REWRITE	!p rewrite
EOF?	!f eof?

In the procedure division, a PERFORM statement is translated into a while statement in WSL, the IF statement into if statement in WSL, an OPEN statement into !p *open-file* and a MOVE into an assignment. Therefore, this program is translated into:

```
segment
  comment: "program-id: copy-customer-list";
  file customer-list with
    record customer-record with
        record name [20 of char]
        record address [50 of char]
        record phonenum [20 of char]
    end;
  end;
  file customer-list-backup with
    record backup-record with
        record b-name [20 of char]
        record b-address [50 of char]
        record b-phonenum [20 of char]
```

```
      end;
   end;
   record eof [1 of char] end;
   !p open_file (i var customer-list);
   !p open_file (o var customer-list-backup);
   while (eof ≠ "T") do
          if non_empty? (!f eof? (customer-list))
            then eof := "T"
             else eof := "F";
                  !p read_file (customer-record var mailing-list);
                  backup-record := customer-record;
                  !p write_file (backup-record var customer-list-backup);
          fi;
            od;
end;
```

It is worth noting that in some languages such as in PASCAL variables can be declared in two ways:

```
(1) type r = record
             x, y: int
             end
         var i, j: r.

(2) var i: record
           x, y: int
           end
    var j: record
           x, y: int
           end.
```

and these two declarations are equivalent. However, WSL does not support the first approach. If this case occurs, the second approach is used to translate the source code into WSL.

6.6 Program Transformation Writing

Transformations are all written in Meta-WSL (which contains WSL). A new transformation is added into the Transformation Library by a function called "ADD-TRANS" (also in Meta-WSL). The "ADD-TRANS" function takes eleven parameters. For example, if a transformation is needed to swap positions between two records, i.e., from

record *name* [20 **of***char*] **end**;
record *address* [50 **of** *char*] **end**;

to

record *address* [50 **of** *char*] **end**;
record *name* [20 **of***char*] **end**;

the transformation is as follows:

```
(Add_trans
  'Record
  'Any
  'Swap-with-next-record
  'Global
  'Always
  '(Rewrite)
  "-------
   To swap a record with next record.
   -------"
  ""
  Nil
  '((Cond ((And ([_S_Type?_] Record)
                ([_>>?_]))
           (@>>)
           (Cond (([_S_Type?_] Record) (@Pass))
                 ((Else) (@Fail))))
          ((Else) (@Fail))))

  '((@Del) (@Undel_after))
  )
```

The eleven parameters include:

1. the general type on which the transformation operates (in this case, "Record");
2. the specific type on which the transformation operates (in this case, "Record");
3. the name of the transformation (in this case, "Swap-with-next-record");
4. the scope of the transformation, either "Global" or "Local" (in this case, "Global");
5. an indicator which determines when the transformation would appear on a menu (in this case, "Always");
6. the list of menus on which the transformation appears (in this case "Rewrite");
7. the documentation for the transformation;
8. the prompt for any information which may need to be entered by the user (in this case, (" ");
9. the type of any information which may need to be entered by the user, (in this case, "Nil");
10. the code in "Meta-WSL" for testing for the transformation's applicability; and
11. the code in "Meta-WSL" to perform the changes required to effect the transformation.

The last two parameters are the most important because they require the writing of some "Meta-WSL" code.

The 10th parameter,

```
'((Cond ((And ([_S_Type?_] Record)
             ([_>>?_]))
        (@>>)
```

```
          (Cond (([_S_Type?_] Record) (@Pass))
                ((Else) (@Fail)))
        ((Else) (@Fail)))),
```

means: if the currently selected item is a "Record" and after this item there is another item (the current item is not the last item of a sequence of items), follow-on tests are carried out; otherwise the transformation is not applicable to the item. In the follow-on testing steps, the next item is selected by moving forward one step (i.e., (@>>)). If the newly selected item is also a "Record", the applicability of the transformation is confirmed by the "(@Pass)" statement. If the newly selected item is not a "Record", the applicability of the transformation is denied by the "(@Fail)" statement.

The 11th parameter,

```
'((@Del) (@Undel_after)),
```

means: delete the current item (actually the next item becomes the current item) and undelete it after the current item (i.e., the deleted item becomes next item), so that two records are swapped.

Chapter 7

Program Transformations and Data Abstraction

In this chapter, the use of program transformations and data abstraction techniques, and the definition of new program transformations are explained.

7.1 Introduction

This chapter starts addressing the addition of program transformations for data abstraction into the prototype system by discussing a number of questions that may concern reverse engineerers. These questions include:

- Which process should be used for crossing level of data abstraction, *top-down* or *bottom-up*?
- What role does human knowledge play?
- What types of abstraction are needed for data abstraction?
- When should *back-tracking* be used?
- How can program transformations be defined formally?
- How to prove program transformations?

Answers to the above questions would certainly be contributions to the research field, but the application of the answers to handling data-intensive programs is more important. Perhaps the main novel contribution of this research is in abstracting from *both* the low level code and data constructs; exploring what information should be thrown away in moving from code and data to higher level abstractions represented especially in Entity-Relationship Attribute Diagrams; and handling commonly encountered problems, such as foreign keys and aliasing. These are implemented by program transformations for manipulating data-intensive programs.

There are classes of low level to high level data abstractions: from a record to an entity; from a segment of unstructured code to a user-defined data type; from code and data to high level data; from aliased memory to separate variables; etc. Each category may consist of a number of transformations. Transformations developed in this research are introduced under these categories.

Transformations from different categories may be used to solve one problem in one data-intensive program. For example, there may a foreign key in one segment of code. The solution is to combine the analysis of code and data. It may need a number of transformations from the class, "from record to entity ", a number of transformations "from code and data to high level data", etc.

The organisation of the program transformations developed in thesis, and the addition of newly defined transformations to the prototype will be introduced in the next chapter.

7.2 Influence of Forward Engineering on Reverse Engineering

7.2.1 Crossing Levels of Data Abstraction

As well as their use in forward engineering, abstraction techniques are of importance in reverse engineering. In reverse engineering, abstraction is the process of identifying the important qualities or properties of the phenomenon being modelled. They abstract from irrelevant details, describing only those details that are relevant to the problem at hand, e.g., understanding the design.

Usually, program code and its design are not at the same level of abstraction — the design is more abstract than the code. It is necessary to reduce the amount of complexity that must be considered at any one time, so that certain number of abstraction levels may exist during the specification extraction process. Each "layer" can be considered as a program in a language provided by a virtual computer, implemented by the layer below. At the lowest layer, we have a real machine.

The process of acquiring a program design or specification from program code has three different ways to cross levels of abstraction. The first way is to refine a high level hypothesis of program design which the program code might have heading to the code (see case (A) in Figure 7.1) (P_{code} stands for program at the code level, $PD_{conceptual}$ for program design at the conceptual level and P_1, P_2, ..., P_n stand for intermediate forms in between P_{code} and $PD_{conceptual}$). The advantage of this is that the technique of refining specifications is relatively well developed, but the disadvantage is that it is very difficult to guide the refinement towards the given program code. When program code is obtained from the program design, it is hard to prove the obtained program is equivalent to the originally given program code and in general it is an undecidable problem.

The second way is to move from the program code towards the specification (see case (B) in Figure 7.1). We do not need to prove the equivalence of the obtained program design and the suspected program design, but attention must be paid to strategic direction because the obtained design may not be the best one. In general, there are an indefinite number of designs which a given piece of code satisfies. Also, because the reverse steps are usually difficult, necessary guidance must be provided by the user to keep the process in the correct direction.

The third way is to move from both ends — to abstract the program and to refine the design — to meet in the middle. This also has the same problems as the first and the second method (refer to case (C) in Figure 7.1).

However, it seems clear that satisfactory abstraction cannot be obtained without a user who is an expert both in software engineering and in the application domain. For example, we may wish to reverse engineer a compiler from its source code. Writing down the program design of the compiler, and working top down, has all the difficulties of verifying an existing program. In contrast, an expert compiler writer, starting from the source code, will look for the lexical and syntax analysis, access to the symbol table etc., and use domain knowledge informally but effectively in guiding the process. Therefore, instead of using the third method (viz. both ends towards the middle) it may be appropriate to use the second method.

$$P_{code} \longleftarrow P_1 \longleftarrow P_2 \longleftarrow \ldots \longleftarrow P_n \longleftarrow PD_{conceptual}$$

(A) Refinement from a Program Design Towards a Program Code

$$P_{code} \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \ldots \longrightarrow P_n \longrightarrow PD_{conceptual}$$

(B) Stepwise Abstraction of Program Towards Program Design

$$P_{code} \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_i \longleftarrow \ldots \longleftarrow P_n \longleftarrow PD_{conceptual}$$

(C) Meeting of Program Abstraction and Program Design Refinement

Figure 7.1: Three Ways of Crossing Levels of Abstraction

The "bottom-up" process is used in this thesis. The aim of the process is the acquisition of the design of the source code. The design derived may not be equivalent to the program design which was originally used even if it existed. Furthermore, the original program design no longer exists after heavy modification. Our research will pay attention to these points. In any case, the derived design will be both helpful to *software maintenance* (maintenance carried out at the conceptual level) and to *software reuse* (components of the software at the conceptual level can be potentially used in another context).

7.2.2 Role of Human Knowledge

To obtain a model of acquiring a program design or specification from program code, program understanding techniques, cognitive models and personal experience are decisive factors.

Approaches to program comprehension are summarised by [136], in which three program comprehension problems are discussed.

- Theories of program comprehension:

    1. examining of the entire program and working out the interactions between various modules,
    2. understanding the program by syntactic and semantic knowledge,
    3. setting a hypothesis of a mapping between the problem domain and the programming domain,
    4. using both top-down and bottom-up strategies at the same time.

- Code reading: The crudest method of understanding program is code reading. Factors affecting code reading are:

    1. the design method employed in the implementation of the program,
    2. the style of writing the program, for example, using meaningful variable names, indentations, comments, etc.

- Program analysis: Static and dynamic analysis — to obtain useful information, such as cross reference listings, call graphs, slicing, and symbolic execution, etc.

Soloway and Eirlich claim [145] that expert programmers have and use two types of programming knowledge: **programming plans**, which are generic program fragments that represent stereotypic action sequences in programming, and **rules of programming discourse**, which capture the conventions in programming and govern the composition of the plans into programs.

The personal experience (of more than ten years) of the author in programming supports the above arguments, i.e., that programs are composed from programming plans that have been modified to fit the needs of the specific problem and that the composition of those plans are governed by rules of programming discourse. Programming knowledge will also play a powerful role in program comprehension [77]. Usually, advanced programmers have strong expectations of what programs should look like and programming knowledge is the base of a programmer's expectation.

Human knowledge plays an important role throughout the whole process of acquiring a data design from code. This fact is both crucial to the researcher (tool builder) and maintainer (tool user). To the reverse engineer, human knowledge assists in the following aspects:

1. Modularisation of source code. The first step in dealing with real software is to modularise the software into manageable sized modules which ought to be functionally independent. This is done by program reading, i.e., the maintainer reads the source code and divides it into smaller modules according to the information found in the source code, e.g., the division identifiers in COBOL.

2. Searching for and naming abstract data types. An abstract data type is an important concept of data abstraction. An important method of crossing levels of data abstraction in this thesis is to gather information in the source code and to form an abstract data type. It is the maintainer who guides the Maintainer's Assistant in searching for an abstract data type and names the obtained abstract data type. The name of an abstract data type ought to be given according to the information gathered from the code. Also the name of an abstract data type affects further abstraction from the abstract

data type. Though tools can help in this case, e.g., work of Sneed [144], the role of human is decisive.

3. Searching and naming entity relationships. In extracting relationships of entities from code, it is again the maintainer who directs the search for relationships between entities. This includes questions of where to look for, and how to name, relationships.

4. Selecting other transformations. Apart from aiding the selection of transformations for abstract data types and entity relationships, the maintainer's knowledge assists selecting all other transformations. Selecting transformations relies on a combination of the following:

    - any potentially useful information visible in the code, e.g., meaningful variable names, comments, indentation, procedure and function names, etc.
    - program syntax components, e.g., controlled variable of a loop, assignment statement, etc.
    - user's hypothesis made according to software engineering knowledge and domain knowledge. The user's hypothesis can be continuously updated all the time as the process of applying transformations is going on.
    - help information from the Maintainer's Assistant. There is a built-in manual facility for all the transformations in the library. Also the help information will be prompted by the program transformer when necessary.

Examples of the above four points will be illustrated later in this thesis.

To the reverse engineering researcher, the problem of how to accommodate the use of human knowledge in the tool (the Maintainer's Assistant in this research) has to be solved. In fact, the use of a program transformer covers the aspect of static program analysis. Other aspects, such as presenting useful information (e.g., comments in a program), providing the user with a facility for naming entity relationships, etc., will be discussed in the section 8.1, "Design of the Prototype".

7.3 Acquiring Data Designs from Program Code Using Program Transformation

7.3.1 Different Types of Transformation and Abstraction Levels

To understand the process of acquiring a data design from code, it is necessary to define clearly three types of program transformation and their relations with program abstraction levels (Figure 7.2).

In simplifying the illustration, it is assumed that only one presentation of the program design or program exists in each abstraction level, i.e., Program I may have more than one semantically equivalent form but only one form is presented in the diagram.

There are three types of transformation: *equivalence transformation*, *refinement transformation* and *abstraction transformation.* When an equivalence transformation is applied to a program (e.g., Program I) the program derived (Program II) has the same semantics as the original. The equivalence transformation is represented by $\Leftrightarrow$. Usually, at the code level, programs are represented by a concrete programming language with defined syntax and the semantics so that these programs can be analysed by the program transformer. Therefore, in the Maintainer's Assistant, the program transformations in the transformation library are all equivalence transformations, and are applicable only to programs at the code level. At the conceptual level, "programs" (in fact, program specifications or program designs) are represented in a conceptual form, such as, Entity-Relationship Attribute Diagrams. At this level, equivalence transformations may also exist and can be used to transform a program design from one form to another equivalent form.

Since the aim of the research is to acquire a data design from program code,

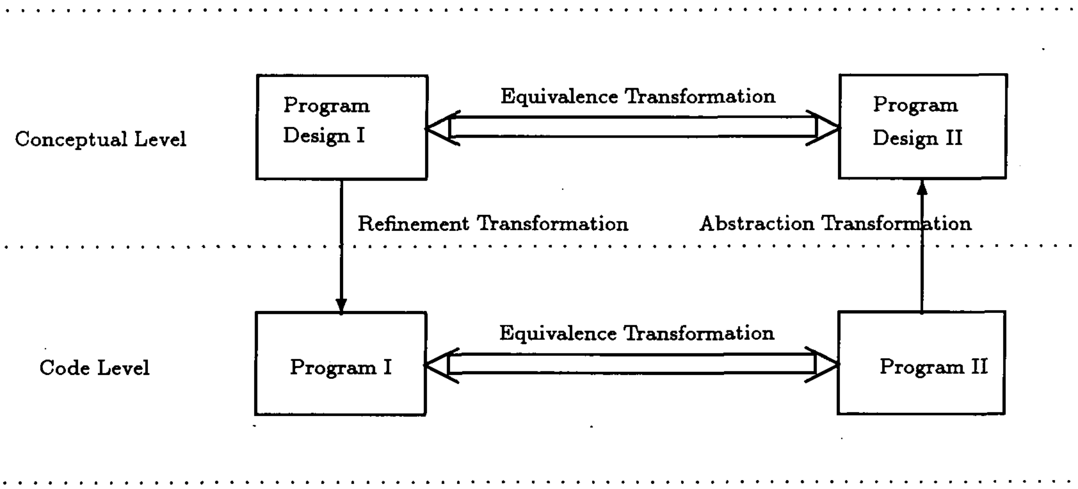

Conceptual Level
Program Design I
Equivalence Transformation
Program Design II
Refinement Transformation
Abstraction Transformation
Code Level
Program I
Equivalence Transformation
Program II

Figure 7.2: Three Types of Program Transformation

the concerned representation of a "program" at the conceptual level is restricted to program designs. A refinement transformation is used to transform a program design (Program-Design I, in this case) into a program (Program I, in this case). This type of transformation is only unidirectional — from an object at the conceptual level to an object at the code level. It means that after a refinement transformation is applied the program derived is the semantic refinement of the program design, but not necessarily vice versa.

An abstraction transformation is used to transform a program (Program II, in this case) back to program design (Program-Design II, in this case). This type of transformation also can only be applied to a program. In another words, though Program-Design II is obtained by applying an abstract transformation to Program II, there does not have to be a transformation available to transform Program-Design II into Program II.

Although three types of transformation are all needed in acquiring program designs (the need for refinement transformations is discussed in the next section "Back Tracking") most attention has been paid to abstraction transformations and equivalent transformations. Refinement transformations have been studied by Ward [155] and therefore, are not a focus in this thesis; for the same reason, equivalent transformations discussed in this thesis are mainly on data abstraction and are supplementary to Ward's work.

7.3.2 "Back Tracking"

The diagram illustrated in Figure 7.2 also supports one argument proposed earlier in this thesis that the program design extracted from a source program can be different from the original program design of the source program. This is because extra information was added by the implementor when the source code was first implemented according to the original program design, and information can be lost when the program design is abstracted out from the source program. The use of a program transformation approach can best preserve information when the source is manipulated, since equivalence transformations will not change the semantics of the source program. For example, Program I and Program II have the same semantics, so this approach keep the information loss to a minimum.

It should be noticed that the scenario described in Figure 7.2 is an ideal case. In practice, there may be several choices at each stage of the transformation process. Some of these choices may not lead in the right direction and the maintainer may need to retreat to some previous stage and start again. Here the "back tracking" technique has to be employed (Figure 7.3).

Figure 7.3 shows that it takes a number of steps to transform Program I into Program Design I. In this figure, circle 1 represents Program I and circle 13 program Design I; other circles represent the intermediate programs; and lines with arrows represent the application of transformations. For some intermediate programs, there is more than one transformation available, e.g., the program represented by circle 2. Assuming the route 2-3-4-5 is selected, the user will soon find out that no suitable transformation is available at 5 to reach 13. The user has to undo a few transformations to go back to 2. The whole diagram indicates all the steps possibly done by a maintainer, i.e., 1-2-3-4-5-4-3-2-6-7-8-7-9-10-11-10-12-13. Though the best known route is 1-2-6-7-9-10-12-13, this route can seldom be found immediately. Program understanding approaches can aid this process.

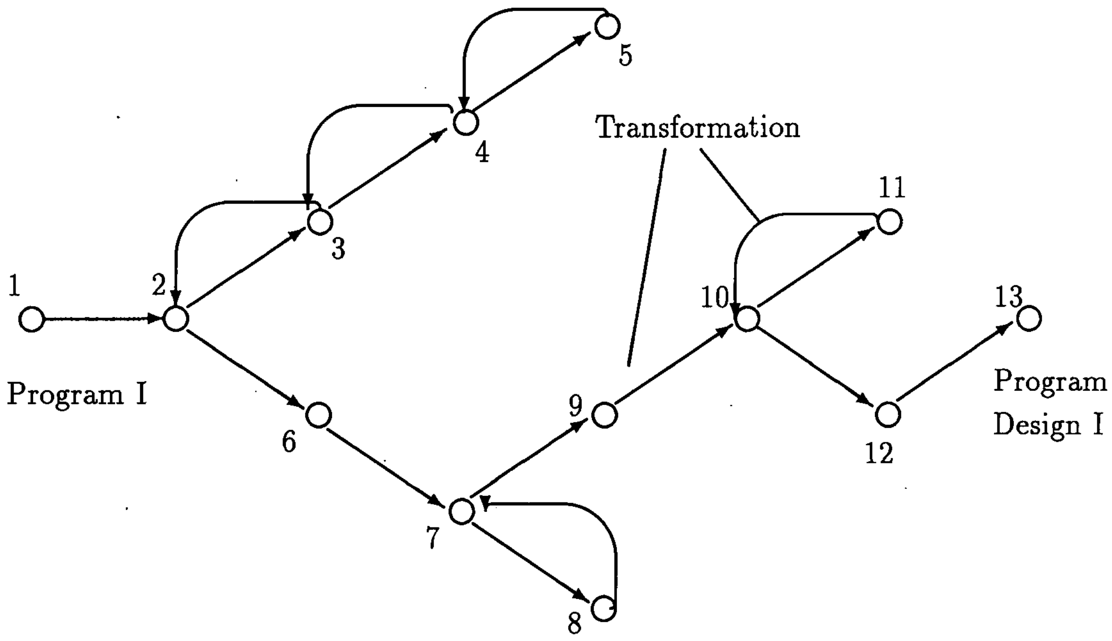


Figure 7.3: "Back Tracking"

7.3.3 Formal Definition of Three Types of Transformation

Three types of program transformation are defined in this subsection[1].

Meanings of Program

A program S is a piece of formal text, i.e., a sequence of formal symbols. There are two ways in which the meaning of these texts can be given [155]:

1. Given a structure [2] M for the logical language $\mathcal{L}$ from which the programs are constructed, and an initial state space (from which a suitable final state

[1] The "refinement" part of the subsection is based on Ward [155] and the "abstraction" part was extended by the author.

[2] A *structure* for a logical language $\mathcal{L}$ consists of a set of values, plus a mapping between constant symbols, function symbols and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values.

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

2. Given any formula **R** (which represents a condition on the final state), a formula WP(**S**,**R**) can be constructed. This formula is usually called the *weakest precondition* of **S** on **R** and is the weakest condition on the initial state such that the program **S** is guaranteed to terminate in a state satisfying **R** only if it started in a state satisfying WP(**S**,**R**).

Because of these two ways of interpreting programs, two corresponding refinement methods were generated: *semantic refinement* and *proof-theoretic refinement.*

Semantic Refinement

A *state* is a collection of variables (the *state space*) with values assigned to them; thus a state is a function which maps from a (finite, non-empty) set V of variables to a set $\mathcal{H}$ of values. There is a special extra state $\perp$ which is used to represent nontermination or error conditions. A state transformation f maps each initial state s in one state space, to the set of possible final states $f(s)$, which may be in a different state space. For convenience, if $\perp$ is in $f(s)$ then so is every other state and $f(\perp)$ is also required to be the set of all state (including $\perp$). However, the set of final states is not required to be empty.

Semantic refinement is defined in terms of these state transformations. A state transformation f is a refinement of a state transformation g if they have the same initial and final state spaces and $f(s) \subseteq g(s)$ for every initial state s. Note that if $\perp \in g(s)$ for some s, then $f(s)$ can be anything at all, i.e., an "undefined" program can be correctly refined to do anything. If f is a refinement of g (i.e., g is refined by f), it is denoted by $g \leq f$. If the interpretation of statement $\mathbf{S}_1$ under the structure M is refined by the interpretation of statement $\mathbf{S}_2$ under the same structure, then this is: $\mathbf{S}_1 \leq_M \mathbf{S}_2$. If this is true for every structure which is a model [3] for a set Δ of sentences of $\mathcal{L}$ then this is written $\Delta \models \mathbf{S}_1 \leq \mathbf{S}_2$.

Proof-Theoretic Refinement

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

A fundamental result, proved in [155], is that these two notions of refinement are equivalent:

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

Semantic Abstraction

If F(s)$\in V_{\mathcal{H}}$ and $G\in V_{\mathcal{H}}$' are state transformations where V$\subseteq$V' and $\forall$ s $\in$V.F(s)=G(s) (i.e., F and G have the same values on variables in V) then it is said that F(s) is more abstract than G(s) (or G(s) is more concrete than F(s)), and it is written $G \sqsupseteq F$(s), i.e., F is an abstraction of G (F abstracts G, or G is abstracted by F). If the interpretation of statement $\mathbf{S}_1$ under the structure M is abstracted by the interpretation of statement $\mathbf{S}_2$ under the same structure, then this is written $\mathbf{S}_1 \sqsupseteq_M \mathbf{S}_2$.

[3] A model for a set of sentences (formulae with no free variables) is a structure for the language such that each of the sentences is interpreted as true.

Refinement, Equivalence and Abstraction Transformations

If $\mathbf{S}_1$ is refined by $\mathbf{S}_2$, there is a program transformation from $\mathbf{S}_1$ to $\mathbf{S}_2$. This is written:

$$\mathbf{S}_1 \leq \mathbf{S}_2$$

and the program transformation is defined as a *refinement transformation.*

If $\mathbf{S}_1$ refines $\mathbf{S}_2$ and also $\mathbf{S}_2$ refines $\mathbf{S}_1$, there is a program transformation from $\mathbf{S}_1$ to $\mathbf{S}_2$ and *vice versa.* This is written:

$$\mathbf{S}_1 \Leftrightarrow \mathbf{S}_2$$

and the program transformation is defined as a *equivalence transformation.*

If $\mathbf{S}_1$ is abstracted by $\mathbf{S}_2$, there is a program transformation from $\mathbf{S}_1$ to $\mathbf{S}_2$. This is written:

$$\mathbf{S}_1 \sqsupseteq \mathbf{S}_2$$

and the program transformation is defined as a *abstraction transformation.* This means that any specification which $\mathbf{S}_2$ satisfies is guaranteed to be satisfied by $\mathbf{S}_1$.

7.4 Issues on Inventing and Proving Program Transformations

In this thesis, program transformations have been invented, based on the research result of data abstraction, to best serve the need of acquiring data designs from programs.

It is crucial to ensure that all program transformations and and abstractions preserve the correctness of program semantics. Since proving program refinements and transformations is not the subject of this thesis (please refer to [155] for more details), one example demonstrates here how a transformation is proved.

The example chosen addresses two programs:

$PUSH\,(S\ x);$

$y\ :=\ Pop\,(S).$

and

$y\ :=\ x.$

Suppose that one program transformation called "Merge-Push-Pop" is valid to transform the first program into the second one. Two methods are presented to prove the validity.

Method 1: To use the weakest preconditions. This is to compare the weakest pre-conditions of the two programs. If their weakest pre-conditions are same, these two programs are equivalent.

WP(PUSH (x,S); y := POP (S), R)

⇔ WP(S := ⟨ x ⟩++ S; y := HD(S); S := TL(S), R)

⇔ R[TL(S)/S] [HD(S)/y] [⟨ x ⟩++ S/S]

⇔ R[TL(⟨ x ⟩++ S/S] [HD(⟨ x ⟩++ S)/y]

⇔ R[S/S][x/y]

⇔ R[x/y]

⇔ WP (y := x, R)

Method 2: To use the existing WSL constructs and their properties.

PUSH (x,S); y := POP (S)

⇔ S[2 ...] := S[1 ...]; S[1] := x; y := S[1]; S[1 ...] := S[2 ...];

⇔ S[2 ...] := S[1 ...]; y := x; S[1 ...] := S[2 ...];

⇔ y := x; S[2 ...] := S[1 ...]; S[1 ...] := S[2 ...];

⇔ y := x; S[1 ...] := S[1 ...];

⇔ y := x;

In essence, the second method is same as the first one, because the exising WSL constructs were derived at earlier stages via definitional transformations which were proven by using weakest preconditions.

7.5 Program Transformations of Data Abstraction

Transformations developed in this thesis for data abstraction are divided into seven categories. Each category will be discussed in one subsection in this section. Because it will not be appropriate to explain every transformation, one or a few examples are usually given in each category.

The approach proposed in the research is shown in Figure 7.4. Lines with arrows in the figure represent where transformations are applied. Data Designs represented in Entity-Relationship Attribute Diagrams are mainly derived from combining entities whose source is data, and relations whose source is code.

It can be seen easily that program transformations fall into the following categories: (1) Data not in the form of records are transformed into records; (2) Records are abstracted to entities or even entities plus relationships; (3) In the code part, statements representing operations on data are abstracted to relations between entities. A typical example is that operations on files are transformed to operations on basic data types first, and then from operations on basic data types to relations between data objects; (4) Control statements and statements representing operations on data together are abstracted to relations or to user-defined abstract data types; (5) User-defined abstract data types are abstracted to both entities and relations; And (6) Entities and their relations are abstracted to form Entity-Relationship Attribute Diagrams.

The seventh category consists of a number of supporting transformations for data abstraction.

7.5.1 Transformations for Deriving Records

Transformations in this category are used to transform data, e.g., in the form of files and variables, which are not in the form of record, into records. For the sake of convenience in the research, the only form of data at the code level which will be abstracted to the data at the conceptual level (i.e., entities) is the record. Section 7.5.3 will address transforming the operations on files to the operations

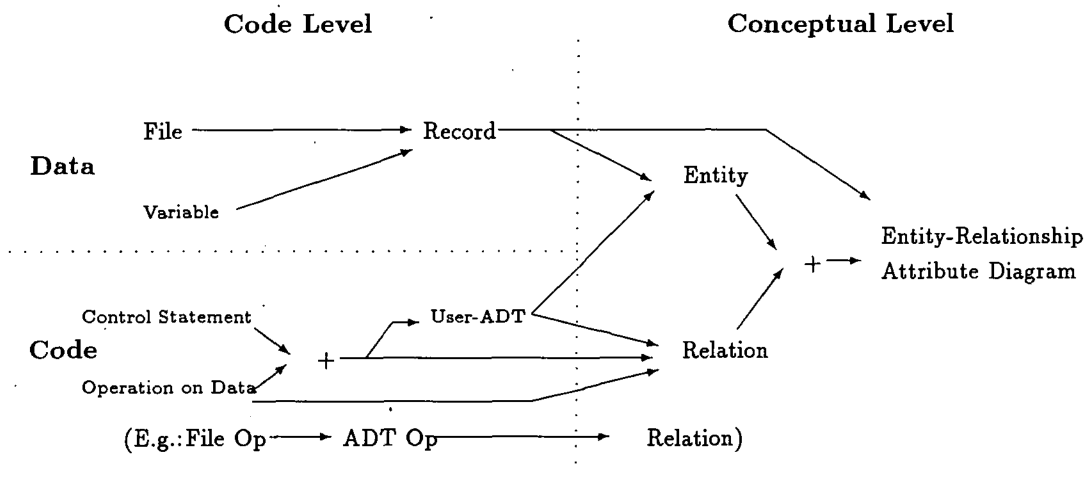


Figure 7.4: Approach of Deriving Entity-relationship Attribute Diagrams

of records.

As discussed earlier in the thesis, simple variables are unable to represent records but fortunately records can represent simple variables properly because a simple variable can be modelled by a record without losing any information. When a simple variable is transformed into a record, the type information of the variable is also recorded in the WSL record. For example, the following is a piece of code in WSL, where local variables *a* and *b* are declared and used.

```
var ⟨ a := 0, b := “ ” ⟩ :
    a := 100;
    b := “ok”;
    end;
```

This program can be transformed into the program shown below. The local variable environment (denoted by var) is changed to local record environment (denoted by segment). The Program Transformer checks the whole segment of the code, and decides the types for each record (in this case, the type for *a* is *int* and *b char*) and gives the length of each record depending on the longest constant assigned to that record (in this case, 3 for *a* and 2 for *b*). Note that, the display form for a variable and a record variable are the same though their internal formats are different.

```
segment
record a [3 of int];
record b [2 of char];
a := 100;
b := "ok";
end;
```

7.5.2 From Records to Data at the Conceptual Level

From Record to Entity

In forward engineering, a "01 level" COBOL record is usually used to implement an entity in an Entity-Relationship Attribute Diagram. Therefore, in reverse engineering, a record without any field can be abstracted to an entity without any attribute; and a record with just fields can be abstracted to an entity with attributes. For example, a record and an entity abstracted from the record are illustrated below. When the record is transformed into the entity, information such as length and type of each field is thrown away, because the information was not usually in the original data design and was added in by the implementor.

```
record E1 with                      entity E1 with
   record A1 [n of char]                   attr A1
   record A2 [n of char]        ⊒          attr A2
   record A3 [n of char]                   attr A3
   record A4 [n of char]                   attr A4
   end;                                end;
```

Acquiring a Relationship from a Record with Subrecords

When a subrecord of a record can be abstracted into an entity and the record can be abstracted into an entity as well, there exists a relationship between the entity derived from the record and the entity derived from the subrecord. For example, in the given record *author*:

```
record author with
   record name [40 of char]
   record address [50 of char]
   record book with
         record title [50 of char]
         record ISBN [20 of char]
   end
end;
```

the subrecord *book* can be abstracted into an entity while the record *author* can be extracted into an entity *author*, according to the knowledge in forward engineering. At the same time, a relationship "write" is put in by user from the logical connection between the record and the subrecord, i.e.:

```
paragraph
entity author with
        attr name
        attr address
  end;
entity book with
        attr title
        attr ISBN
  end;
relationship entity author has one write relation with many entity book;
end;
```

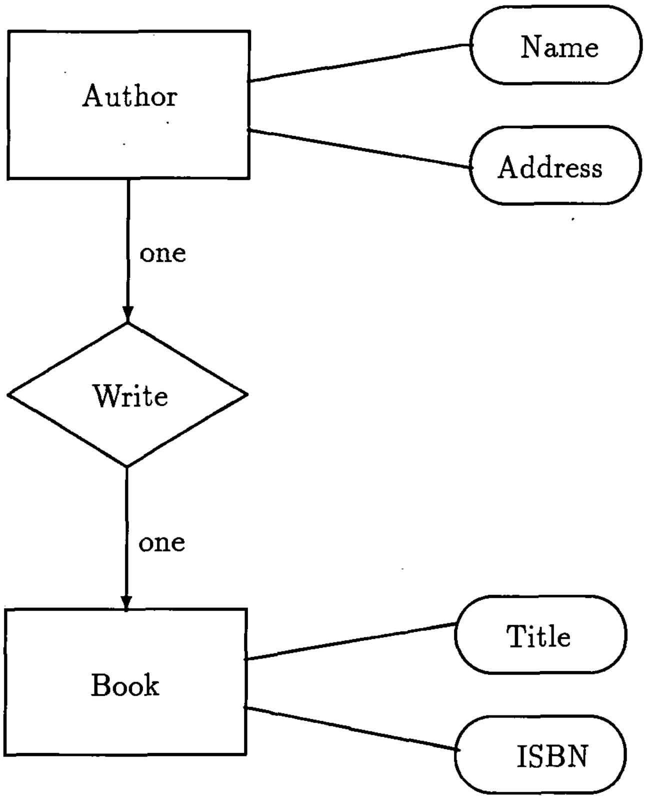


Figure 7.5: Deriving a Relationship from a Record with Subrecord

Note that information on the implementation details is also thrown away in the process of abstraction. An Entity-Relationship Attribute Diagram for this example is shown in Figure 7.5.

7.5.3 From Code Level Data Operations to Data Relations

Transformations for Modelling Sequential Files

File input/output is a central problem in most data-intensive programs. File operations in a programming language usually involve access to external storage. In COBOL, a serial file is a sequence of records, i.e., a record is the unit with which a physical file can be accessed. Though COBOL file operations can be translated into WSL as external procedures and external functions (Section 6.5.2), more suitable forms of data presentations are required to replace these external procedures and functions in order to examine file operations at a high abstraction level.

A queue data type is proposed to model COBOL sequential files and operations on these files, in order that files (external storage objects) can be transformed into queues (internal mathematical objects).

Assuming a file has n records, R_1, R_2, ... , R_i, R_{i+1}, R_{i+2}, ... , R_{n-1}, R_n, sequential file operations can be modelled by operations on two queues (Head-Queue and Tail-Queue) of records, and one record variable V_r:

Open-file $\equiv$ *Head-Queue* := {};
Tail-Queue := $R_1 +\!\!+ R_2 +\!\!+ \ldots +\!\!+ R_n$. ($+\!\!+$ *is concatenation*)

Read-file $\equiv$ V_r := *head* (*Tail-Queue*);
Tail-Queue := *tail* (*Tail-Queue*);
Head-Queue := *Head-Queue* $+\!\!+$ V_r.

Write-file $\equiv$ *Head-Queue* := *Head-Queue* $+\!\!+$ V_r;
Tail-Queue := *tail*(*Tail-Queue*).

Close-file $\equiv$ *Tail-Queue* := *Head-Queue* $+\!\!+$ *Tail-Queue*;
Head-Queue := {}.

eof? $\equiv$ *empty?*(*Tail-Queue*).

Accordingly, five transformations with the above meaning can be invented:

- Model-OpenFile-by-QueueOP

  definition:

Status of a File	Head-Queue	Tail-Queue
(A) Just opened	Empty	R_1 R_2 ... R_{n-1} R_n
(B) After the ith record being accessed	R_1 R_2 ... R_i	R_{i+1} ... R_{n-1} R_n
(C) After the (i+1)th record being accessed	R_1 R_2 ... R_{i+1}	R_{i+2} ... R_{n-1} R_n
(D) After the last record being accessed	R_1 R_2 ... R_{n-1} R_n	Empty
(E) Closed	Empty	R_1 R_2 ... R_{n-1} Rn

Figure 7.6: Modelling a Sequential File by Two Queues

```
!p open_file DATAFILE; ⇔
   init_q DATAFILE-head;
   DATAFILE-tail := q_concat(DATAFILE-head, DATAFILE-tail);
```

- Model-ReadFile-by-QueueOP

  definition:

```
!p read_file (DATARECORD, DATAFILE); ⇔
   DATARECORD := q_rem_first(DATAFILE-tail);
   DATAFILE-head := q_concat(DATAFILE-head, DATARECORD);
```

- Model-WriteFile-by-QueueOP

  definition:

```
!p write_file (DATARECORD, DATAFILE); ⇔
   DATAFILE-head := q_concat(DATAFILE-head, DATARECORD);
   DATARECORD := q_rem_first(DATAFILE-tail);
```

- Model-Eof?-by-QueueOP

  definition:

```
!p eof? (DATAFILE); ⇔
   empty? (DATAFILE-tail);
```

- Model-CloseFile-by-QueueOP

  definition:

!p *close_file DATAFILE*; ⇔
  init_q *DATAFILE-head*;
  DATAFILE-tail := **q_concat**(*DATAFILE-head*, *DATAFILE-tail*);

Transformations on Basic Data Types

Transformations in this class deal with simplifying data objects in basic data types according to the properties of the data type. The data types that the transformations in this class can deal with include stack, set, sequence and queue. For example, the transformation,

push(S, x); **pop**(y, S) ⇔ y := x.

is based on the properties of a stack. Another two examples are:

q_append *p e* $\sqsupseteq$ **relate** *p* **to** *e*;

and

  x := **q_rem_first**(p)
⇔ $x := head(p);\ p := tail(p);$
$\sqsupseteq$ **relate** x **to** p; **relate** p **to** p;

Handling Aliased Records

Suppose we have a COBOL program as below:

```
01 X                  PIC X(8).
01 Y REDEFINES X.     PIC X(8).
01 Z                  PIC X(8).
```

```
... ...
MOVE Z TO X.
```

It will be translated into WSL:

<u>**record**</u> x [*8* <u>**of**</u> *char*];

<u>**record**</u> y [*8* <u>**of**</u> *char*];

<u>**redefine**</u> x <u>**with**</u> y;

<u>**record**</u> z [*8* <u>**of**</u> *char*];

......

$x := z$;

$y := read\text{-}rec(x, y)$;

The second statement in the above program can be simplified by transformations, i.e.:

$$\begin{aligned} & y := read\text{-}rec(x, y); \\ \Leftrightarrow\ & y := z; \end{aligned}$$

7.5.4 Abstraction from Code

A number of examples are given in this subsection to illustrate how code is abstracted towards data design.

Abstracting from Assignment Statement

An assignment statement is a simple but straightforward measure to implement a relation between two data objects, which, at the data design level, may be two entities. Therefore, an assignment can be usually abstracted to a relation, e.g.,

$x := y$; $\sqsupseteq$ <u>**relate**</u> x <u>**to**</u> y;

This simply means that x relates to y. How this relation will be used to obtain Entity-Relationship Attribute Diagrams will be discussed later in this section.

Abstracting from Branching Structure

A branching statement, such as **if .. then .. else .. if**, is used as a control structure in implementing programs. But the structure itself did not appear in the original data design and neither did the condition part of the branching structure. Therefore, these parts will not contribute to the Entity-Relationship Attribute Diagram. Information appearing in both arms of an **if .. then .. else .. fi** statement may exist in the Entity-Relationship Attribute Diagram. Therefore, an **if .. then .. else .. fi** statement can be abstracted to a sequence of two groups of statement (each group comes from each arm of the **if .. then .. else .. fi** statement). For instance,

```
if x > 0
  then  x := y
   else  x := z
fi;
```

is abstracted to

```
relate x to y;
relate x to z;
```

Looping Statement

Looping statements, such as **while** and **for**, are also used as a control structure in implementing programs but do not appear in the original data design. A looping statement can be treated as enumerating the same operation on every instance of entities. The condition part of the loop also does not contribute to the Entity-Relationship Attribute Diagram. So a **while** loop can be removed just leaving the body of the loop. For example,

```
i := 0;
while i < 10 do
```

```
        x[i] := y[i];
        i := i + 1;
od;
```

is equivalent to

```
for i = 0 to 9 step 1
    begin
      x[i] := y[i];
      end;
```

and they both can be abstracted to:

```
relate x to y;
```

The example shows that, at the code level, the elements of the array x are assigned with the elements of the array y and this can be abstracted to that, at the data design level, entity x has a relation with entity y.

Transformations for Forming Abstract Data Type

Transformations in this class address looking for a user-defined abstract data type.

An example is given below:

```
var iset := 0, rest := 0, x := 0, y := 0, m := 0, n := 0,
    int1 := 0, int2 := 0, real1 := 0, real2 := 0 :
    begin
      ......
      iinsert (int1);
      idelete (int2);
      rinsert (real1);
      rdelete (real2);
      ......
    where
      proc iinsert (x) == iset := iset ∨ {x};
proc idelete (y) == iset − {y};
proc rinsert (m) == rset := rset ∨ {m};
proc rdelete (n) == rset − {n}.
```

Here we assume that the variables x and y are not used in (...) parts (this can be easily confirmed in Meta-WSL). If we start with variable x which is used by procedure definition *iinsert*, it is found that variable *iset* is also used by the same procedure definition. There is no other procedure definitions using x, but the procedure definition *idelete* uses variable *iset* and y. Following the same steps, a closure can at last be recognised which contains three variables (x, y , *iset*) and two procedures (*iinsert, idelete*). Hence an abstract data type is formed (and named *intset*). The above program can be transformed into the following:

```
var rest := 0, m := 0, n := 0, int1 := 0, int2 := 0,
    real1 := 0, real2 := 0 :
    begin
      .......
      user-adt-proc-call intset.insert (int1);
      user-adt-proc-call intset.idelete (int2);
      rinsert (real1);
      rdelete (real2);
      ......
    where
      user-adt intset (iset := 0, x := 0, y := 0) (nil)
          user-adt-proc iinsert (x) == iset := iset ∨ {x} :
          user-adt-proc idelete (y) == iset − {y} :
      proc rinsert (m) == rset := rset ∨ {m} :
      proc rdelete (n) == rset − {n}.
```

Note that in the original program any procedure call which is involved in the newly formed abstract data type is transformed into *adt-proc-call.* The search for a closure can be started with a procedure definition (or function definition) as well as a variable, and the result is the same. For example, if we start with procedure definition *iinsert*, we will first get variable x and *iset* involved; then search for procedure definitions using these two variables; and finally get the same closure as we start with the variable x.

Program transformations for searching and forming an abstract data type only involve presentational changes to programs so that their correctness can be easily proven.

It should be pointed out that the above method of identifying a user-defined abstract data type can be implemented satisfactorily in the program transformation approach because the program transformer is a powerful analyser in searching for a closure. Other methods of identifying an abstract data type, such as using *retrieve function*, though they have been considered, still need further study before they can be implemented in the tool.

7.5.5 From User-Defined Data Types to Data Design

Transformations in this category deal with transforming data objects, such as records and abstract data types, into entities.

For example, an abstract data type usually involves a data object and a number of operations on this object. The operations are implemented in terms of procedures and functions, which take parameters. At the data design level, the data object can be viewed as an entity; each parameter can be viewed as a different entity; and the operations can be viewed as relations between the data object and other data objects that are represented by the parameters.

Therefore, a user-defined abstract data type can be abstracted to an entity while all statements accessing this abstract data type have to be changed accordingly and "ADT->Entity" is such a transformation. By applying this transformation, the following program will be abstracted from

```
var ......
    begin
      ......
```

```
        user-adt-proc-call intset.insert (int1);
        user-adt-proc-call intset.idelete (int2);
        ......
    where
        user-adt intset ($iset := 0, x := 0, y := 0$) (nil)
            user-adt-proc iinsert ($x$) == $iset := iset \vee \{x\}$;
            user-adt-proc idelete ($y$) == $iset - \{y\}$;
        ......
    to

paragraph
entity int1
entity int2
entity intset
......
relate int1 to intset; comment : insert int1 to intset;
relate int2 to intset; comment : delete int1 from intset;
......
end
```

In this process, the abstract data type *intset* becomes an entity, and so do the two parameters involved. Two procedure call statements become two relations. In order to abstract the program further, useful information is recorded by comment statements.

7.5.6 Deriving Data Designs from Data and Code

Transformations in this category deal with deriving entity relationships.

Handling Relations

When a **relate** statement relates two entities there must by definition be a relationship between these two entities. So a relationship can be derived from the statement and the name of the relationship is provided by the user according to the information in the program. When a **relate** statement relates one or no entities, i.e., one or two of the components of the **relate** statement are neither entities nor records which may be transformed into entities, it means that this **relate** statement does not represent a relationship of any entity and can be deleted. For example, the following statement can be abstracted to a **skip** because "3" is a.constant and was usually not part of the data design but was just used for the purpose of control in the implementation.

relate x to 3; $\sqsupseteq$ **skip**;

Acquiring Relationship from Abstract Data Type

Let us carry on with the example in Section 7.5.5. A **relate** statement has two components of which one is an entity derived from an abstract data type. Then the other component is a variable or a record without any subrecord (these two are equivalent and there are transformations available in the prototype for transforming one into the other), this **relate** statement can be transformed into a relationship. The name of the relationship has to be provided by the user according to the information existing in the code. The program can be transformed into the following WSL form and its Entity-Relationship Attribute Diagram is shown in Figure 7.7.

paragraph
entity *intset*;
entity *int1*;
entity *int2*;
......
relationship entity *int1* has *one is-member-of* relation with *one* entity *intset*;
relationship entity *int2* has *one is-not-member-of* relation with *one* entity *intset*;
......
end

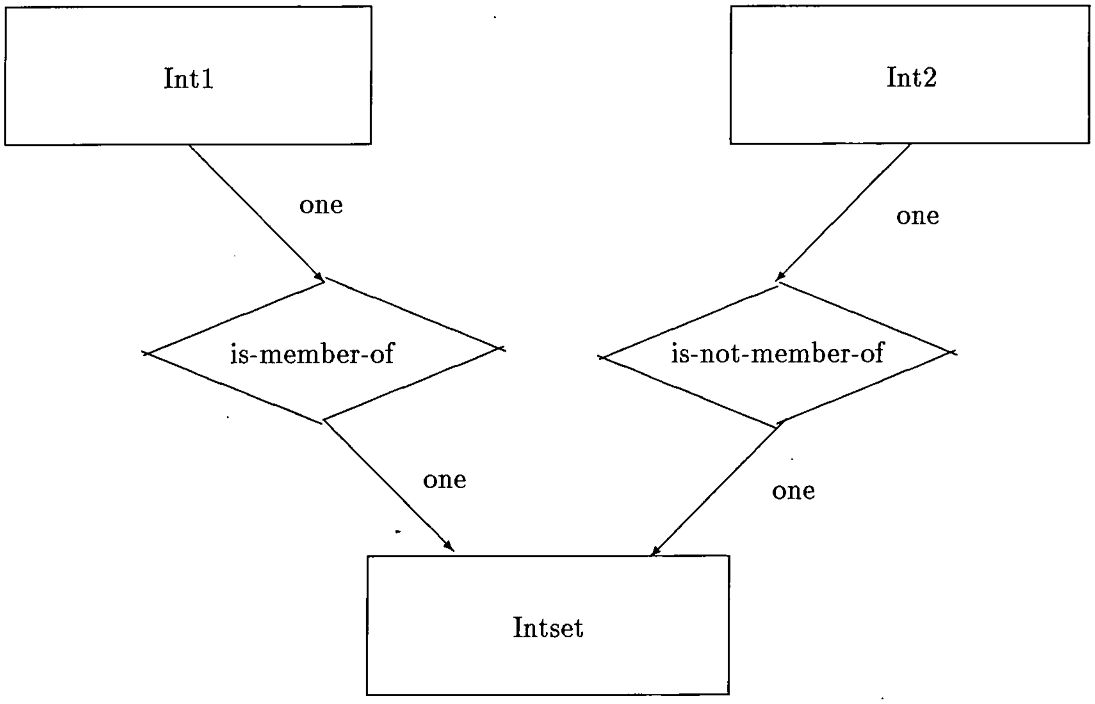


Figure 7.7: Deriving a Relationship from an Abstract Data Type

Acquiring a Relationship from a Foreign Key

One example is given here for *Acquiring Relationship from Foreign Key.* A relationship can exist between two entities that both have the same attribute (known as a *foreign key*). Suppose two entities have been derived already from source code (e.g., record definitions) and two relations between two pairs of entity attributes (e.g., assignment statements):

```
paragraph
entity employee
       attr nhs-number
       attr name
       attr department
       attr vehicle-num-plate
```

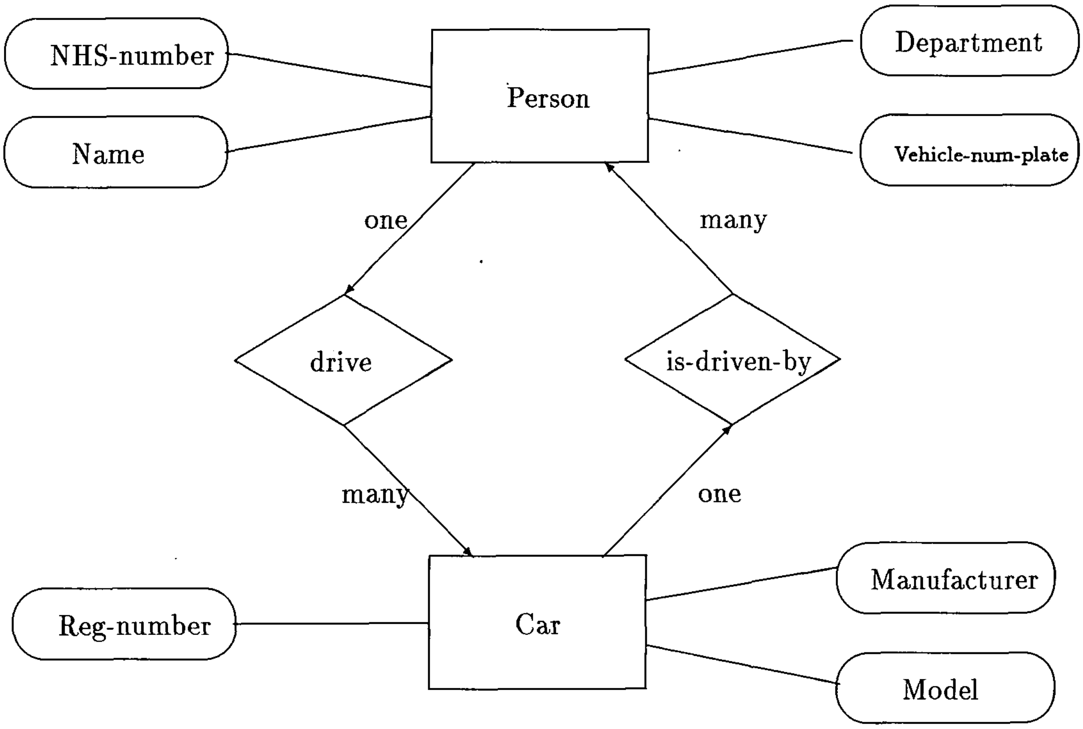


Figure 7.8: Deriving a Relationship from a Foreign Key

```
    end;
entity car
        attr reg-number
        attr manufacturer
        attr model
        attr driver
    end;
relate employee.vehicle-num-plate  to  car.reg-number;
comment :  one employee.vehicle-num-plate relates to many car.reg-number;
relate car.driver  to  employee.nhs-number;
comment :  one car.driver relates to many employee.nhs-number;
end;
```

Program transformations analyse code as well as data. For example, before these entities and relations were obtained, *employee* was a record variable and *employee.vehicle-num-plate* a field variable. The first relate statement records the fact that the variable *employee.vehicle-num-plate* ("*one*" variable) was assigned to by an expression typed *car.reg-number* for more than once ("*many*" times); and this statement is used by a user in asserting a relationship later. By this stage, as shown above, it can be identified that the first entity has a foreign key, *employee.vehicle-num-plate* and the second entity has an entity *car.driver*. If we start with the entity *employee*, a one-to-many relationship is derived, i.e., that a person can drive more than one car. When we start with the entity *car*, a one-to-many relationship is also asserted by the user, i.e., that one car can be driven by more than one person. Therefore, the WSL presentation and the Entity-Relationship Attribute Diagram is shown below and in Figure 7.8 respectively.

```
paragraph
entity employee
        attr nhs-number
        attr name
        attr department
        attr vehicle-num-plate
    end;
entity car
        attr reg-number
        attr manufacturer
        attr model
        attr driver
    end;
 relationship entity employee has one drive relation with
                         many entity car;
 relationship entity car has one driven-by relation with
                         many entity employee end;
 end;
```

7.5.7 Transformations for Manipulating Program Items

Transformations in this category deal with manipulating program items for the preparation of data abstraction. Examples of these transformations are, "Join-Records" (when more than one record can be joined together in order to form an entity), "Split-Record-Into-Subrecords" (when a record needs to be split up in order to form more than one entity), "Swap-With-Next-Record" (when a record can be moved a position where is closer to another record and they may be joined together to form an entity), etc.

Chapter 8

Design and Implementation

This chapter describes the prototype system's design and implementation in terms of those major components of the Maintainer's Assistant built by the author: an extension to the WSL language (Chapter 6), and extension to the Transformation Library (Chapter 7), the Program Structure Database, the General Simplifier and the Metric Facility. Although the prototype is still not an industrial-strength tool, all components are aimed to be fully operational and to be able to demonstrate the feasibility of the method developed in this thesis. Case studies will be described in the next chapter. In this thesis, unless otherwise stated, the interface software is running on a SUN 3/50 workstation and the rest of the prototype is running in COMMON LISP on an IBM RS6000 workstation.

8.1 Design of the Prototype

The components of the prototype needing design and implementation/enhancement includes the Representation of WSL, the Transformation Library, the Program Structure Database, the General Simplifier and the Metric Facility (Figure 8.1) .

8.1.1 Transformations for Data Design Recovery

Transformations for acquiring data design will be the extension of the transformation library and these transformations will be divided into the following categories:

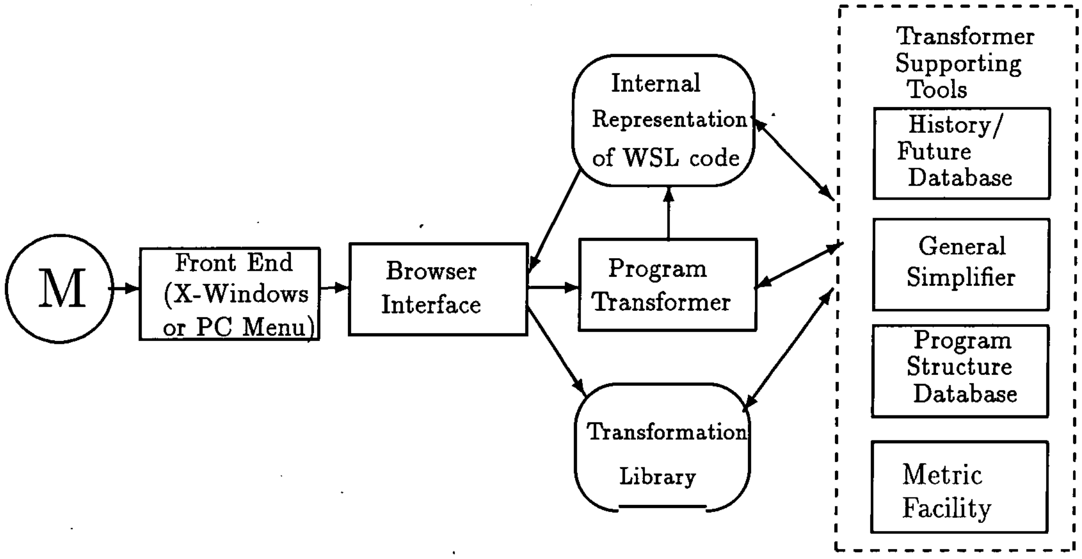

M
Front End (X-Windows or PC Menu)
Browser Interface
Internal Representation of WSL code
Program Transformer
Transformation Library
Transformer Supporting Tools
History/ Future Database
General Simplifier
Program Structure Database
Metric Facility

Figure 8.1: Design of the Prototype

1. deriving records,
2. from records to data in design level,
3. from code level data operation to data relations,
4. abstraction from code,
5. from user-defined data type to data design,
6. deriving data design from data and code, and
7. manipulating program items.

The program transformations to be implemented in the prototype is listed in Appendix D.

8.1.2 Program Structure Database

Examples of the database query definitions (in English) are listed in Figure 8.2, and the specification of all the database queries is in Appendix C which was originally defined in [154,165].

8.1.3 General Simplifier

Basically, the General Simplifier is able to simplify a mathematical or logical expression, or to prove the equivalence of two expressions.

Mathematical and logical operations defined in the system are: +, −, *, /, **, Min, Max, Div, Mod, =, >, <, <>, Not, And, Or, etc. [1]; and three commands defined are:

- [*Simplify*] *Expression*
- [*Equivalent?*] (*Expression*, *Expression*)
- [*Implies?*] (*Expression*, *Expression*)

In the first command, the "Expression" can be any symbolic algebraic expression. When receiving this query from the Program Transformer, the General Simplifier returns the expression in its simplest form. See the following for examples:

- [*Simplify*] (2 + 3) === 5
- [Simplify] ((A + B) / (A + B)) === 1
- [Simplify] (A ** (B ** 0)) === A
- [Simplify] (B * B * B * B) === (B ** 4).

When receiving the second or the third query from the Program Transformer, the General Simplifier returns **T** or **Nil** accordingly. See the following for examples:

- [*Equivalent?*] ((And A B) **or** (And A C), (A **and** (Or B C))) === T

[1] These definitions are in normal mathematical and logical sense and subject to normal limitations, e.g., *zero* cannot be divided by other expressions.

Queries	Description
[Trans?]	Returns “T” or “Nil” depending on whether the named transformation is applicable.
[Variables] [Used] [Assigned]. [Used-only] [Assigned-only]	Returns lists of variables in the currently selected program item according to whether variables are assigned to, referenced, both, or neither.
[Depth]	Returns the depth which is defined as the number of enclosing unbounded (do...od) loops.
[Primitive?]	Returns “T” or “Nil’ depending on whether the statement given is primitive. A primitive statement is defined to be neither an unbounded loop nor conditional statement.
[Terminal-Value]	Returns the terminal value of the given statement. The terminal value is the capacity of a statement for jumping out loops.
[Terminal?]	Returns “T” or “Nil” depending on whether the statement given is terminal, i.e., whether the statement is in a terminal position or causes termination of a loop which is in a terminal position.
[Proper?]	Returns “T” or “Nil” depending on whether the statement is proper. The statement S is a proper sequence if every terminal statement of S has terminal value zero.
[Regular?]	Returns “T” or “Nil” depending on whether the program item is regular, e.g., an item is regular if every execution of that item leads to an action call.
[Reducible?]	Returns “T” or “Nil” depending on whether a statement is reducible. The statement S is reducible if replacing any terminal statement EXIT(k), which has terminal value one, by EXIT(k-1) gives a terminal statement of S.
[Improper?]	Returns “T” if all the terminal statements of S have a terminal value greater than zero.
[Dummy?]	Returns “T” if the statement S is both reducible and all terminal statements of S have a terminal value greater than zero.
[Calls]	Returns all the action call names and how many times they are called in the given program item.
[Statements]	Returns all the statement names in the given program item.

Figure 8.2: Program Structure Database Queries

- [*Implies?*] (X > Y, (Max X Y) = X) === T
- [*Equivalent?*] ((A + B) * (A – B), (A * A) – (B * B)) === T
- [*Equivalent?*] ((A + B) * (A + B), (A * A) – (B * B)) === Nil .

8.1.4 Metric Facility

The objectives of using metrics in REFORM are to help the user to select transformations (to help develop heuristics), to measure the progress made in optimising the program code and to measure the resulting quality of the program being transformed.

The question with which a user is mainly concerned is how to make the program "better" (easier to understand). This is an optimising process which includes removing redundant code, removing unreachable nodes, detecting and removing bugs in the program, restructuring the program and reducing data complexity.

Therefore, the measures needed initially in REFORM are "code" measures. They must reflect the complexity of the code and include control flow complexity (connections between nodes, branches and loops), structural complexity, data flow complexity and the size of the program.

Six metrics were defined for WSL programs in the prototype of the Maintainer's Assistant.

A button named "Metrics" in the X-Windows interface of the Maintainer's Assistant is needed. By clicking this button, a user can calculate any one or all the metrics, applied either to the current program item on which he or she is working or to the whole program.

During the process of transforming a program, the metrics at each stage can be recorded and the results can be plotted as and when required.

The six metrics defined (to be implemented) in the prototype are:

- McCabe Complexity (MCCABE) — the number of linearly independent circuits in a program flowgraph [116]. It is calculated as the number of predicates plus one.

- Structural (STRUCT) — the sum of the weights of every construct in the program. The weight of each WSL construct is defined subjectively according to experience gained by REFORM researchers and users. A loop, for example, is more difficult to understand than an assignment statement, so a loop statement is given a bigger weight than an assignment statement.
- Lines Of Code(I) (LOC) — the number of statements.
- Lines Of Code(II) (LOC2) — the number of nodes in the abstract syntax tree. This reflects the overall size of the program.
- Control-Flow and Data-Flow Complexity (CFDF) — the number of edges in the flowgraph plus the number of times that variables are used (defined and referred). It is a modification of the measure defined by Oviedo [175].
- Branch-Loop Complexity (BL) — the number of non-loop predicates plus the number of loops. It is a modification of the measure defined by Moawad and Hassan [175]. The measure is sensitive both to branches and to loops.

The implementation details of program transformations and components of the prototype designed in this section will be described in the following sections.

8.2 Extension of WSL

The WSL is embedded in LISP which means that executable WSL is represented in a LISP form so that it can be executed by a LISP interpreter simply by providing suitable macro and function definitions and non-executable WSL is also represented in a LISP form so that a LISP interpreter can check the correctness of the syntax. The interpreter checks the syntax of WSL by looking up a syntax table in which every WSL component is defined in terms of its name, type, etc. When new WSL components are needed, their definitions are added to the table. The table has to be loaded before the system can interpret any WSL components.

The implementation of expanding the WSL only involves defining entries of the WSL Syntax Table. The table in Figure 8.3 shows examples of the newly defined entries, for representing data abstraction and Entity-Relationship Attribute

Diagrams. Some of the new WSL components are defined in terms of existing WSL components (please see Appendix A and Appendix B for their syntax and semantics). In the table there are the following entries:

Number This is the number of the type number that is passed to the pretty-printer as a more efficient alternative to passing the actual type of the object. This both reduces the amount of information which needs to be passed, and also speeds up the process of finding the form of the pretty-printed version. The newly defined entries start with 400.

Name This is the name of the item.

Generic Type This is the "parent" type of the given type. For example, "Relate" is a type of statement and "Depth" is a type of expression.

Leading Token This is either "yes" or "no" if and only if the type of the item is the first part of the printed form. For example, an "Relate" statement begins with the word "Relate".

Minimum Size This is the least number of components that the type can have. Example are an "Relate" statement which must have at least two (in fact only two) components and a "Record" which must have at least five components, whereas a list of records can have any number.

Component Types This holds the types of components of the given type (if there are any). For example, the components of an "Insert" are an assigned variable and an expression. If there is an unlimited number of components for a given item, any additional components must have the same type as the last component. For example, a "Segment" must have a section of files, a section of records, and it can have any number of statements in it.

8.3 Transformations

Program transformations defined in the previous chapter using WSL are implemented in Meta-WSL (about 4000 lines of source code). A general process of

Num	Name	Generic Type	Leading Token	Min Size	Component Types
401	A_Def	Thing	Yes	4	Name Number Name Expression
402	E_Def	Thing	No	2	Name A==S
403	F_Def	Thing	No	0	—
404	R_Def	Thing	No	0	—
405	Segment	Statement	Yes	3	Files Records Statement ...
406	Records	A_List	No	0	Record ...
407	Record	R_Def	Yes	5	Name Expression Name Expression Records
408	Rec	Variable	Yes	1	Variable ...
409	Redefine	R_Def	Yes	2	Record Record
410	File	F_Def	Yes	2	Name Records ...
411	Files	A_List	No	0	File ...
412	Paragraph	Statement	Yes	2	E==S Statement ...
413	Ent	Variable	Yes	1	Variable ...
414	E==	E_Def	Yes	2	Name A==S
415	E==S	A_List	No	0	E_Def ...
416	A==	A_Def	Yes	4	Name Number Name Expression
417	A==S	A_List	No	0	A_Def ...
418	Relationship	Statement	Yes	5	Name Expression Name Expression Name
419	And_Relationship	Statement	Yes	7	Name Expression Name Expression Name Expression Name
420	Or_Relationship	Statement	Yes	7	Name Expression Name Expression Name expression Name
421	Adt	Definition	Yes	4	Name Assignments Records Definition ...
422	Adt_Proc_Call	Statement	Yes	4	Name Name Expressions Variables
423	Adt_Funct_Call	Expression	Yes	3	Name Name Expressions
424	Create	Statement	Yes	1	Assd_Var
425	Insert	Statement	Yes	2	Expression Assd_Var
426	Del_Element	Statement	Yes	2	Expression Assd_Var
427	Dis_Union	Expression	Yes	1	Expression
428	Dis_Intersection	Expression	Yes	1	Expression
429	Init_Q	Statement	Yes	1	Assd_Var
430	Q_Append	Statement	Yes	2	Assd_Var Expression
431	Q_Concat	Expression	Yes	2	Expression Expression
432	Q_Rem_First	Expression	Yes	1	Assd_Var
433	Relate	Statement	Yes	2	Expression Expression
434	And_Relate	Statement	Yes	3	Expression Expression Expression
435	Or_Relate	Statement	Yes	3	Expression Expression Expression
436	Seq_Concat	Expression	Yes	2	Expression ...
437	Seq_Remove	Statement	Yes	2	Expression Assd_Var
438	Sub_Seq	Expression	Yes	3	Expression Number Number
439	Sub_Seq?	Condition	Yes	2	Expression Expression
440	Make_Seq	Statement	Yes	1	Assd_Var
441	Seq_Append	Statement	Yes	2	Assd_Var Expression
442	Init_Stack	Statement	Yes	1	Assd_Var
443	Top	Expression	Yes	1	Expression
444	Depth	Expression	Yes	1	Expression

Figure 8.3: WSL Syntax Table

writing a program transformation consists of the following steps:

1. Study the definition of a transformation and design the implementation in pseudo code.
2. Write the transformation according to the design. This includes:
    - checking the applicability of the transformation to the selected program item — pattern matching the selected item with the defined pattern and collecting information from the given item; and
    - editing the given item to what is defined by the transformation definition.
3. Test the implemented transformation. This is done using *path analysis.*

In illustrating these steps, a previously used example is presented again. If the following two statements appear adjacent in the program, where S is a stack:

PUSH (*S x*);
y := *Pop* (*S*).

they can be merged into one statement:

y := *x*.

This transformation is named as "Merge-Push-Pop". It is designed that this transformation can be applied by either selecting the "Push" statement or the "Assignment" statement with a "Pop" function as the expression to assign. Firstly the design is written in pseudo code which is like:

Transformation: Merge-Push-Pop

(a) Applicability

```
if (current-statement = "Push") ∧
  (current-statement ≠ last-statement)
  then current-statement := next-statement fi.
if (current-statement ≠ "Pop")
  then flag = "Fail"
   else buffer1 := stack-name-in-pop;
        current-statement := previous-statement
        if (current-statement = "Push") ∧
          (stack-name-in-push = buffer1)
          then flag = "Pass"
           else flag = "Fail"
        fi
fi.
```

(b) Transforming

```
if (current-statement = "Push")
  then current-statement := next-statement fi.
buffer1 := variable-name-in-pop;
current-statement := previous-statement;
delete next-statement;
buffer2 := variable-name-in-push;
(insert a "Assignment" statement) ∧ (buffer1 := buffer2);
delete next-statement.
```

Secondly, the transformation itself is written in Meta-WSL:

```
(Add_trans
  'Statement
  'Any
  'Merge_Push_Pop
  'Global
  'Always
  '(Rewrite)
  "-------
   Transformation to merge a PUSH statement and a statement using
   POP function.
   -------"
  ""
  Nil
  '((Var ((Table Empty))
          (Cond ((And ([_S_Type?_] Push) ([_>>?_])) (@>>)))
          (Assign (Table ([_Match_] Statement
                                    (Assign ((~>?~ V)
                                             (Pop (~>?~ S))))
                                    Table)))
          (Cond ((Empty? Table) (@Fail))
                ((Else)
                 (Cond ((Not ([_<<?_])) (@Fail))
                       ((Else)

                        (@<<)
                        (Assign (Table ([_Match_] Statement
                                                  (Push (~<?~ S)
                                                        (~>?~ E))
                                                  Table)))
                        (Cond ((Empty? Table) (@Fail))
                              ((Else) (@Pass))))))))
   '((Var ((Table Empty))
          (Cond (([_S_Type?_] Push) (@>>)))
          (Assign (Table ([_Match_] Statement
                                    (Assign ((~>?~ V)
                                             (Pop (~>?~ S))))
                                    Table)))
          (@del_back)
          (Assign (Table ([_Match_] Statement
                                    (Push (~<?~ S) (~>?~ E))
                                    Table)))
          (@Change_To ([_Fill_in_] Statement
                                   (Assign ((~<?~ V) (~<?~ E)))
                                   Table)))))
```

Finally, this transformation is tested with the path analysis method.

This example is still a simple transformation, just for showing how a transformation is implemented. Most transformations written for the research described in this thesis is far more complicated than this transformation (up to several hundred lines of Meta-WSL).

Most of the 60 transformations implemented are "rewrite" transformations because these transformations deal with alterations to the structure of the selected program items. These transformations will therefore appear in the "Rewrite" menu in the interface and the other transformations, such as "Swap-with-next-record", will appear in other menus, such as "(Re)Move" menu.

8.4 Program Structure Database

The Program Structure Database is implemented in COMMON LISP and it consists of a total of around 3000 lines of source code. The features of the implementation are summarised in the following sections.

8.4.1 Use of Recursion Programming Techniques

In the Program Structure Database, a database query is usually implemented by one database query function (LISP function) which may call subsidiary functions. The Database Manager is implemented as a group of LISP functions. To collect adequate information about a given program, a database query function usually needs to examine all the components of the program. For example, to answer the query "which variables are used in the program", the database query function "[Variables]" should check every place in the program where a variable can possibly occur. This can be very complicated and requires some operations to be carried out repeatedly. Recursive functions are therefore useful weapons to deal with such operations. LISP provides a powerful recursion feature and this is fully used by the database query functions.

8.4.2 Dealing with All Kinds of WSL Construct

The result of some database queries entirely depend on the structure of the given program, e.g., whether or not the program consists of a conditional statement, a loop, etc., so that the corresponding database functions must be able to calculate the answers according to different program structures. For instance, whether a given program item is "regular" depends on:

- when the item is a sequence of statements, whether every statement is "regular";
- when the item is a "if...then...else" statement, whether both clauses of the statement are "regular";
- when the item is an "action", whether every execution of the action leads to an action call;
- when the item is an "action system", whether every action in the system is "regular".

Generally, this type of database query functions was organised with a branching section as the following:

```
if       prog-item = "sequence"     then sequence-subfunction
  else_if prog-item = "if-then-else" then if-then-else-subfunction
  else_if prog-item = "do-od"        then do-od-subfunction
  else_if prog-item = "action"       then action-subfunction
  else_if prog-item = "exit"         then exit-subfunction
                                     else other-subfunction fi.
```

8.4.3 Deriving Database Query Functions from Their Specifications

The specifications of database query may directly provide clues for implementing corresponding database query functions. Examples of the clues include:

- an "if-then-else" can be implemented as a "cond" statement in LISP;
- an "$\bigcup_{i=1}^{Size(item)}$" operation can be implemented as a "while" loop with a condition of "$1 \leq i \leq size(item)$";
- a "function" together with a "map" operation (f $*$) can usually be implemented as applying a recursive function to a sequence of components (an example can be seen in the next section);
- logical operations "$\wedge$", "$\vee$" and "not" can be implemented by corresponding LISP functions;
- set operations "$\in$", "$\cup$", "$\cap$" and "$\setminus$" can be directly implemented by corresponding LISP functions; etc.

8.4.4 An Example of Implementing a Database Query Function

The example chosen is a database query "[Statements]" whose definition is

$$\underline{\text{funct}}\ [Statements]\ (item) \equiv$$
$$Specific\text{-}type * \{P \in Posns(item) | [Gen\text{-}type](Get\text{-}n(item, P)) = \texttt{Statement}\}.$$

This definition means that the query function must return all the statement names in the given program item. Firstly, since the function needs to check every component (including leaf node because a statement can be a leaf node in the syntax tree) and a "map" operation "$*$" appears in the definition, a recursive function may be required. Secondly, the result is a set of statement names, a union operation may be needed. Thirdly, the function needs to check the data table to see whether the same query has been made before, and to return the result directly if it was in the datatable or to calculate (and save) the result. So a database query function is developed like this:

```
(Defun [Statements] (Item)
  (If (Leaf_Item? Item)
      (And (Eq (Gen_Type Item) 'Statement)
           (List (Specific_Type Item)))
      (Let ((S (Get_From_Table Item 'Statements)))
        (If S
            (Cdr S)
            (Add_To_Table!
             Item
             'Statements
             (Let ((Args (Args Item))
                   (Result (And (Eq (Gen_Type Item) 'Statement)
                                (List (Specific_Type Item)))))
              (Dolist (X Args Result)
              (Setq Result (Union Result ([Statements] X)))))))))).
```

In this program, "Leaf_Item?" is a function that identifies whether the current item is a leaf node; "Get_From_Table" is a function that retrieves information in the data table indexed by 'Statement; and "Add_To_Table!" is a function that saves the result of this query into the data table.

8.5 General Simplifier (Symbolic Executor)

The prototype of the General Simplifier makes use of two public domain software packages, Maxima and the Boyer-Moore Theorem Prover. The "Simplify" command is mainly based on the Maxima, and the "Prove" command on the Boyer-Moore Theorem Prover. In order to be used by the General Simplifier, the input and output programs of these two packages have been changed.

The Boyer-Moore Theorem Prover is a program developed by Boyer and Moore based on a "computational logic" (the logic) described in their book *A Computational Logic* [30] published in 1979. The logic is both oriented towards discussion of computation and mechanised, so that proofs can be checked by computation; and the logic is quantifier-free logic. Its axioms and rules of inference are obtained from the propositional calculus with equality and function symbols by adding (1) axioms characterising certain basic function symbols, (2) two "extension principles", with which one can add new axioms to the logic to introduce

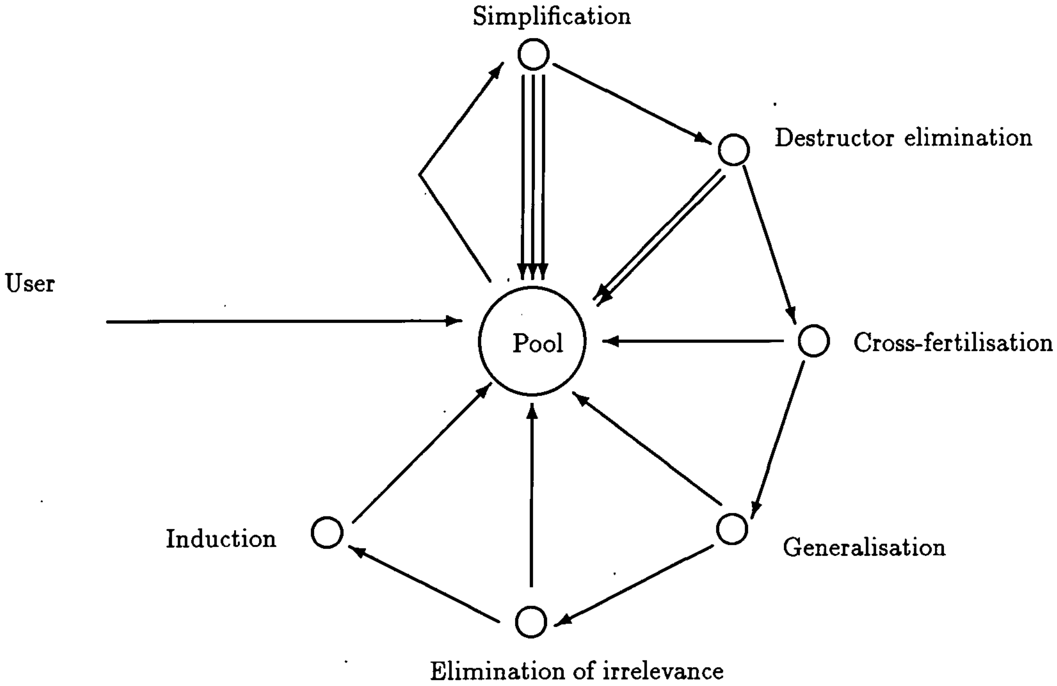


Figure 8.4: Organisation of Boyer-Moore Theorem Prover

"new" inductively defined "data types" and recursive functions, and (3) mathematical induction as a rule of inference.

The Boyer-Moore Theorem Prover is based on the generalised principle of induction and the majority of its heuristics (proof techniques) are oriented towards induction proofs. Any proof using structural induction can be converted into a proof with the given formal system. One advantage of using derived rules of inference is that they permit the formal logic to be relatively primitive while allowing the production of sophisticated proofs. for example, a well-known and very useful rule of inference that the Boyer-Moore Theorem Prover uses is the tautology theorem: a formula of propositional calculus has a proof if and only if it is valid under a truth table. More complicated derived rules are those that justify the use of equality decision procedures and certain arithmetic decision procedure. The Boyer-Moore Theorem Prover uses a variety of such high-level derived rules of inference to discover and describe its proofs. In particular it uses the following six proof techniques:

1. **Simplification** This involves the use of axioms (including definitions and shell axioms) and previously proved lemmas to simplify the conjecture.

2. **Destructor Elimination** This involves the trading of "bad" terms for "good" by choosing an appropriate representation for the objects being manipulated. For example, operations such as / and – might be traded for operations such as $*$ and $+$.

3. **Cross-fertilisation** When the conjecture being proved has an equality as one of its hypotheses, the equality is sometimes used to substitute one of its operands for the other in the remainder of the conjecture and then removed from the conjecture.

4. **Generalisation** This involves the adoption of a more general goal obtained by replacing terms in the given goal by new variables. The generalisation is designed to help prepare a conjecture for induction. Conjectures must frequently be generalised before they can be proved because without generalisation the induction hypothesis may not be sufficiently strong to prove the theorem.

5. **Elimination of Irrelevance** This involves the discarding of apparently unimportant hypothesis.

6. **Induction** Inductions are formulated from information collected when definitions are added to the system and from information available at the time of induction.

As implemented, each of these proof techniques is a computer program that takes a formula as input and yields a set of formulas as output; the input formula is provable if each of the output formulas is. Each of these six programs is called a "process". Not every process is applicable to every formula. For instance, it may not be possible further to simplify a formula. When a process recognises that its input is a theorem, it produces an empty set as the output set.

The Boyer-Moore Theorem Prover is organised around a "pool" of goal formulae [31]. Initially the user places an input conjecture into the pool. Formulas are drawn out of the pool one at each time for consideration. Each formula is passed in turn to the six processes in the order shown in Figure 8.4 until some process is applicable. When an applicable process produces a new set of subgoals, each is added to the pool. The consideration of goals in the pool continues until either the pool is empty and no new subgoals are produced — in which case the system has "won" — or one of several termination conditions is met — in which case the system has "lost". The system may "run forever" until the host system resources are exhausted.

When the system wins — i.e., the pool is finally empty — the trace of the theorem prover is a proof of the input conjecture, provided each of the six processes is considered a derived rule of inference. When system loses, the user should infer nothing about the validity of the original conjecture; the initial problem may or may not be a theorem.

In certain applications the theorem prover resembles a sophisticated proof checker more than an automatic theorem prover. This is because the theorem prover's behaviour on a given problem is determined by a data base of rules. The rules are derived by the system from the axioms, definitions, and theorems summited by the user. Three of those six processes mentioned earlier, namely simplification, destructor elimination and generalisation, can be heavily influenced by these rules. Each time a new theorem is proved it is converted into rule form and stored in the data base. When new theorems are submitted the currently "enable" rules determine how certain parts of the theorem prover behave. In this way the user can lead the machine to the proofs of exceedingly deep theorems by presenting it with an appropriate graduated sequence of lemmas. An experiment shows that the theorem prover cannot prove the equation, $a^2 - b^2 = (a+b)(a-b)$, until the theorem, $ab = ba$, is presented to the theorem prover. In another words, the more rules are in the data base, the more clever the theorem is.

In order to make use of the Boyer-Moore Theorem Prover, the source code of the prover (more than 900,000 bytes) was analysed. Through the analysis, not only the programs to implement those six processes but other details such as input and output functions, global flags, etc. were also found. Therefore, the following changes were made to the Boyer-Moore Theorem Prover to enable it to serve the program transformer as a Symbolic Executor:

- Input — The Boyer-Moore Theorem Prover accepts user commands from the keyboard. In particular, the commands attempting to prove a conjecture is "Prove-Lemma". This command is rewritten together with a new query function, "[Prove]", so the system will accept a query made by another LISP function rather than a command typed in from the keyboard.

- Output — Similarly, the Boyer-Moore Theorem Prover prints its proving results on the screen while our new system requires the output returned back to the query originating function. A LISP function is written to intercept the output originally dedicated to the screen and to return the output to the calling function. Output messages include "Lemma-proved", "Lemma-not-proved" and "Faulty-lemma-format".

- Monitoring messages — while the Boyer-Moore Theorem Prover is attempting to prove a conjecture, monitoring messages are sent to the screen letting the user informed what is going on. These messages range from which process is running to which theorem the theorem prover is employing at the moment to prove the problem. There are several functions in the original Boyer-Moore Theorem Prover to produce these messages, "PPR", "Prind", "PPR1", "PPR2", "Print-States", "Print5*", "PRINC", etc. These functions are all rewritten to become dummy functions and information which used to be generated by those functions is collected in forming the result of the query.

- Forever runs — Forever runs must be prevented from happening when the system serves as a query, because the calling function is not able to stop a forever run (this is different from that the Boyer-Moore Theorem Prover is used since the user can interrupt by pressing keys.). This is done by monitoring check points inserted in the Induction process. When the induction process has been called five times to prove a same lemma, the system assumes that it is not worth while trying any more and sets the global variable, "Do-not-use-induction-flag", to be "True" and no more induction will be invoked.

- Data base of rules — Once a theorem which is found useful for the future use has been proved, the theorem can be added as a new axiom to the data base of rules by the command, "Add-Axiom". For example,

  ```
  (Add-Axiom axiom100 (rewrite)
                  (equal (times (plus a b) (difference a b))
                            (difference (times a a) (times b b))))
  ```

  The data base of rules used by the Symbolic Executor is built with the material in a file included with the standard distribution of the Boyer-Moore Theorem Prover. The file, "basic.events", with about 470 definitions and 970 theorems, contains most of the Appendix A of *A Computational Logic*, which includes the major examples of that book, covering theorems of number theory and theorems of algebra, as well as the Newton's binomial theorem and the Church-Rosser theorem, etc. In this file theorems are in the forms of lemma. These theorems are all treated as axioms by command "Add-Axiom" when being loaded to the data base of rules.

The original plan was to use the simplification part of the Boyer-Moore Theorem Prover as a simplifier. It was soon found out by experiments the simplifying capability of this part is very limited and could not meet the need of serving the program transformer. These experiments triggered an investigation of another public domain software package — Maxima. The system Maxima is a Common LISP implementation due to William F. Schelter, and is based on the original implementation of Macsyma [65,66] at MIT. It was found out through experiments that the simplifying capability of Maxima is much better than that of the Boyer-Moore Theorem Prover.

The process of turning Maxima into a part of the Symbolic Executor to serve the program transformer is very similar to the process of turning the Boyer-

Moore Theorem Prover into a part of the Symbolic Executor, so only main steps are briefly listed here:

- analysing Maxima and obtaining information for changes;
- writing a new input interface for the Maxima;
- writing a new output interface for the Maxima; and
- removing monitoring messages by altering corresponding programs of the Maxima.

The Symbolic Executor is implemented in COMMON LISP and the total of the source code is about 7000 lines including the data base of rules.

8.6 The Metric Facility

The Metric Facility [170] is implemented as a program module with a number of LISP functions and flags. It also uses the Program Structural Database to retain the resultant metrics of code.

8.6.1 Collecting Program Property Information

The key program in the Metric Facility is called "Collect-Prog-Info". This program analyses a given program item (in WSL internal format) by going down to the program item recursively and returns eleven properties of the program item. These properties are (refer Section 8.1.4):

1. index number of Mccabe;
2. index number of the Structural Complexity;
3. number of nodes;
4. number of statements;
5. number of edges;

6. number of predicates;

7. number of loops

8. number of data items;

9. number of back-edges;

10. number of procedure or function calls; and

11. number of action calls.

A natural way of constructing this program is to go through the structure of the given WSL program item once and to collect all those eleven properties. These eleven properties are comparatively independent (so defined) and can be directly calculated or collected. For example, the index number of MCCABE is calculated as "the number of predicates plus 1" or "the number of calls in an irregular action system plus 1 (it has been proved that an action call in an irregular action system is equivalent to a predicate in a non-action system)" or "the number of predicates plus number of calls to itself in an recursive definition"; the index number of the Structural Complexity is obtained by checking the Structural Complexity Table in which every WSL construct is given an index number; and the number of statement is assigned to "1" when the current item is a statement, or "0" when the current item is not a statement.

The final calculation result is the sum of the (index) number of the current item and the sum of (index) numbers of its all subitems. The property information collecting program (a LISP function) searches through the given program item recursively, i.e., the function is calling itself when a subitem is entered. At each time the function is called, the function always makes a query to the Program Structure Database. If the result is already in the database, the function can immediately return the information stored in the database as the result. If the result is not in the database, the function does the calculation. While returning the result, the function saves the result in the database if the current item is not a leaf node.

8.6.2 Processing Metrics

Using the Metric Facility, a user may calculate the metrics, applied either to the current program item on which (s)he is working or to the whole program. The format of user command is:

```
[_Metric_] whole-prog|current-item.
```

On receiving this command, the LISP function, "[_Metric_]", parses the parameters; calls the function "Collect-Prog-Info"; calculates metrics according to the obtained information; and finally returns the metrics according to the parameters.

The Metric Facility allows metrics at each stage to be recorded automatically during the process of transforming a program. A user can start the automatic recording mechanism by the command "Start-Auto-Metric"; stop the automatic recording mechanism by the command "Stop-Auto-Metric"; and reset the metric record by the command "Init-Metric-Record". When a transformation is applied to the program, if the auto metric flag is on, the name of the transformation just applied together with the six metrics of the program is recorded. The results can be either saved to a file for future analysis or plotted on the screen.

8.6.3 Plotting Metric Graphs

The Metric Facility provides a command, "[_Plot_Metrics_]", to plot a graph of the metric result, using the record generated by the automatic mechanism. One of the six metrics is plotted in one graph, including the name of the metric, the (index) number of the metric at each line, and the name of the transformation applied. Considering, by the time a graph is going to be plotted, that the number of transformation applied may be more than the maximum number of characters which can be displayed in one line on the screen, we plot the graph vertically, i.e., the graph forwarding downwards. Figure 9.5 and Figure 9.6 are examples of the graphs (refer to the next chapter).

8.7 The Interface

The Interface has been built for the existing Maintainer's Assistant by other members of the research team. It needed expanding when new WSL components for representing data abstraction and Entity-Relationship Attribute Diagrams were introduced. Also a change had to be made to accommodate the Metric Facility.

Similar to expanding the WSL (described in Section 8.2), expanding the interface to display new WSL components involves defining new entries of the WSL Parsing Table in which the display format of all WSL components is defined. The Interface uses this table to convert the internal format of WSL to the external format of WSL. For example, the internal (LISP) WSL format of a record is (entry 407 in the table of Section 8.2):

(Record Name Expression Name Records)

and the external WSL format is:

record *Name* [*Number* **of** *Name*] **with** *Records.*

The internal WSL format of an abstract function call is (entry 420):

(Adt-funct-call Name Name Expression)

and the external WSL format:

user-adt-funct-call *Name.Name* (*Expression*).

When the Interface displays WSL programs, WSL key words (such as **record** and **adt_funct_call**) and other symbols (such as "[", "]", "=", etc.) are added. The complete parsing table is full of tedious symbols and therefore it is omitted here.

The Metric Facility is implemented the Interface as a button with which a pop-up menu can be invoked (Figure 8.5). The Metric menu consists of the following options:

- "Auto Metric" — This option switches "on" or "off" the automatic metrics "history" recording mechanism.

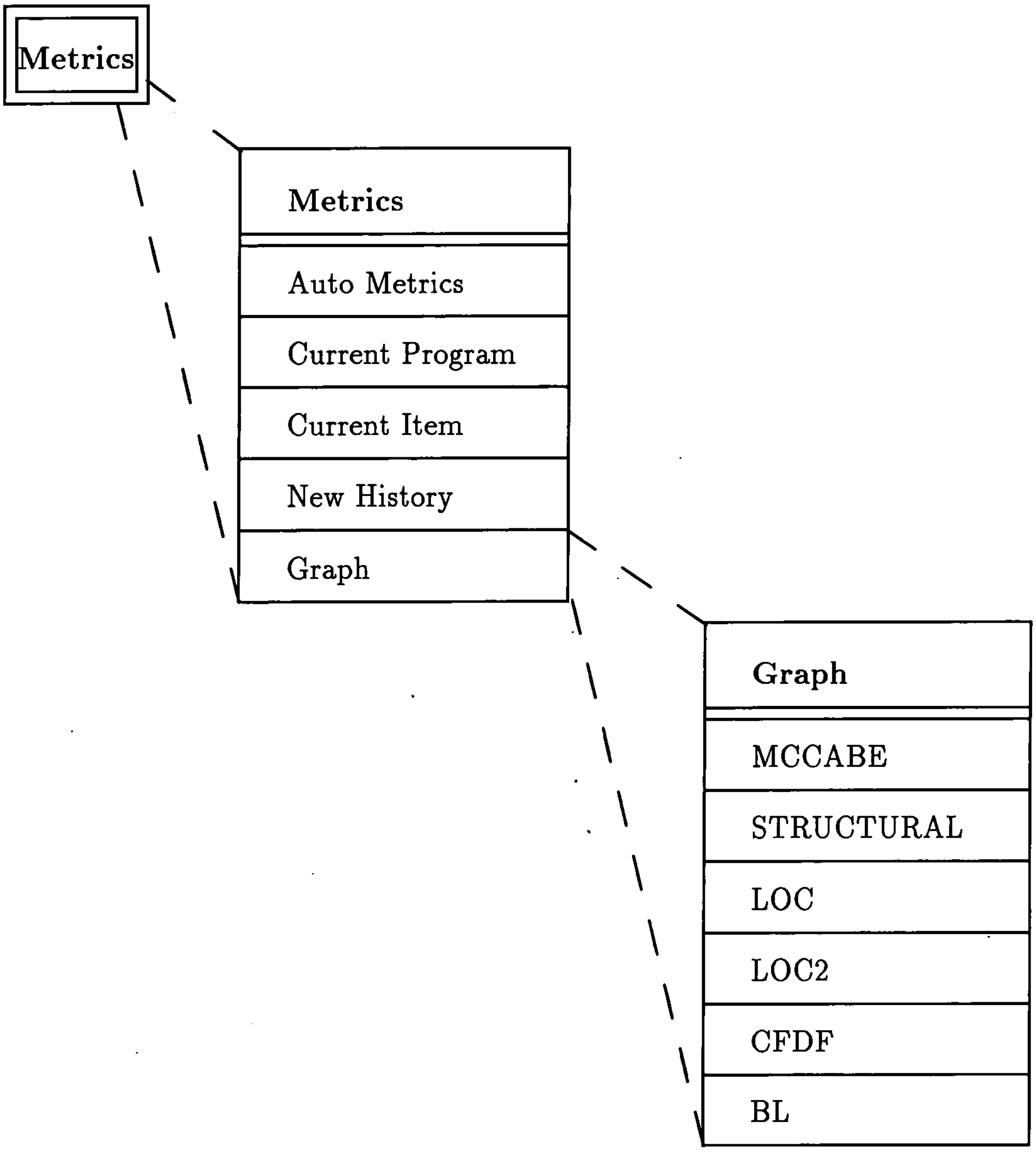


Figure 8.5: The Metrics Menu

- "Current Program" — This option returns the metrics of the current program.
- "Current item" — This option returns the metrics of the current item.
- "New History" — This option resets the metric history record.
- "Graph" — this option invokes a sub-menu, "Graph" menu. When one of the options in the sub-menu is selected, the metric history record for the current program is plotted for that particular metric.

8.8 Integration of the Prototype

To emphasise how implementation of the prototype designed in this thesis extends the Maintainer's Assistant, the following steps of building a new version of the Maintainer's Assistant is shown below:

- to load LISP, and in LISP,
- to load existing components of the Maintainer's Assistant which were not developed in this thesis but includes the WSL part developed in this thesis;
- to load the programs containing transformations developed in this thesis, "dataabstract.clisp";
- to load the Boyer-Moore Theorem Prover, "bm.clisp";
- to load the Maxima, "maxima.clisp";
- to load the Symbolic Executor (which overwrites some parts of the Boyer-Moore Theorem Prover and the Maxima, turning them into the Symbolic Executor), "symbolic.clisp";
- to load the database of rules used by the Symbolic Executor, "basicaxioms.clisp";
- to load the Program Structure Database, "database.clisp";

- to load the Metric Facility, “metric.clisp”; and then
- to save the LISP image as an executable file, which is a new version of the Maintainer’s Assistant.

Chapter 9

The Use of the Prototype System and Results

This chapter describes how the method developed in this thesis and the prototype implemented are used to demonstrate the feasibility of acquiring data designs from data-intensive programs.

9.1 Introduction

Among the research problems set out in Section 4.4, the central problem is the method of extracting Entity-Relationship Attribute Diagrams from data-intensive programs. Although this problem and the other problems have been partly answered in previous chapters, it is still necessary to examine the central problem specifically by looking at the results of applying the method. The following questions are to be considered:

- does the method work?
- how well does it work?
- where does it work well?
- where does it not work well?
- is it better than other approaches, such as those used by Sneed and REDO?

Five case studies are presented in this chapter to demonstrate the use of the prototype and to.examine the method proposed in this thesis. In the first case study, the process of transforming a COBOL program, which copies a file to another, into an Entity-Relationship Attribute Diagram is illustrated in detail. This case study emphases how we obtain Entity-Relationship Attribute Diagrams by analysing COBOL statements in the "Procedure Division". In the second case, a program where an alias was used for the second purpose of aliasing is dealt with. In the third case, a real program (of 3750 lines in COBOL) used in a national telephone company was thoroughly investigated. The fourth case study shows the results of experimenting with a COBOL program which was also used in the same telephone company. The example code used in this case is approximately 7000 lines of COBOL source. In the final case study, a section of COBOL program, which is a Public Library Administration System, is studied in the prototype.

The intention of selecting these five case studies was to cover as much application of the method developed (or in terms of the prototype, as many as the transformations) as possible. Also, the first case shows a detailed process in which entities and their relationships are obtained, so that similar details may be omitted in later cases. The selection of the third and the fourth case was aimed to find out what really happens to a real program at a scale of a few thousand lines. The selection of the fifth case was mainly because it has complicated embedded calls to CICS/TOTAL in code, which is typical in practice.

9.2 Case Study 1 — A File Copying Program

The source code of this COBOL program was presented in Section 6.5.2 and its WSL equivalent form was in Section 6.5. In this case study, this program is manipulated further.

9.2.1 Modelling File Operations by Queue Operations

Following the application of the transformations to represent file operations in terms of queue operations, the program in Section 6.5 can be transformed into:

```
segment
 comment : :"Program-ID: Copy-Customer-List";
 file customer-list with
     record customer-record with
         record name [20 of char]
         record address [50 of char]
         record phonenum [20 of char] end
 end;
 file customer-list-backup with
      record backup-record with
         record b-name [20 of char]
         record b-address [50 of char]
         record b-phonenum [20 of char] end
 end;
 record eof [1 of char] end;
 customer-list-tail := q_concat (customer-list-head, customer-list-tail);
 customer-list-head := empty;
 customer-list-backup-tail := q_concat (customer-list-backup-head,
                                 customer-list-backup-tail);
 customer-list-backup-head := empty;
 while (eof ≠ "T") do
          if empty? (customer-list-tail)
             then  eof := "T"
              else  eof := "F";
                     customer-record := q_rem_first mailing-list-tail;
                     q_append (mailing-list-head, customer-record);
                     backup-record := customer-record;
                     q_append (mailing-list-backup-head, backup-record);
                     backup-record := q_rem_first custom-list-backup-tail;
          fi  od
end;
```

9.2.2 Transforming Records into Entities

A record without any subrecords is transformed into an entity with no attribute. The entity keeps the same name as the record. In this case,

```
record eof [1 of char] end;
```

is transformed into

```
entity eof end;
```

A record with subrecords containing no subrecords is transformed into an entity with attributes. The entity keeps the same name as the record while attributes keep the same names as subrecords. In this case,

```
record customer-record with
   record name [20 of char]
   record address [50 of char]
   record phonenum [20 of char] end;
```

is transformed into

```
entity customer-record with
   attr name
   attr address
   attr phonenum end;
```

By now the program in the previous section is turned into:

```
paragraph
entity customer-record with
   attr name
```

```
    attr address
    attr phonenum end;
entity backup-record with
    attr b-name
    attr b-address
    attr b-phonenum end;
entity eof end;
  customer-list-tail := q_concat (customer-list-head, customer-list-tail);
  customer-list-head := empty;
  customer-list-backup-tail := q_concat (customer-list-backup-head
                                   customer-list-backup-tail);
  customer-list-backup-head := empty;
  while (eof ≠ "T") do
        if empty? (customer-list-tail)
          then  eof := "T"
           else  eof := "F";
                customer-record := q_rem_first mailing-list-tail;
                q_append (mailing-list-head, customer-record);
                backup-record := customer-record;
                q_append (mailing-list-backup-head, backup-record);
                backup-record := q_rem_first custom-list-backup-tail;
        fi  od
end;
```

9.2.3 Turning Assignments into *Relate* Statements

When a record is transformed into an entity, any program statement using this record must be changed accordingly. Because transforming a record into an entity is an abstraction, the statement using the obtained entity must be expressed at a higher abstraction level. The **relate** statement in WSL was just defined for this purpose. The statement takes two parameters which each can be an entity, a record, a variable, a file name, i.e.:

```
relate parameter1 to parameter2;
```

For instance, when the record *customer-record* is transformed into an entity *customer-record*, all the statements using this record are to be changed, i.e., from

```
customer-record := q_rem_first mailing-list-tail;
q_append (mailing-list-head, customer-record);
backup-record := customer-record;
```

to

```
relate customer-record to mailing-list-tail;
relate mailing-list-head to customer-record;
relate backup-record to customer-record;
comment : "copy a record";
```

A newer version of the program in the previous section is obtained:

```
paragraph
entity customer-record with
   attr name
   attr address
   attr phonenum end;
entity backup-record with
   attr b-name
   attr b-address
   attr b-phonenum end;
entity eof end;
relate customer-list-tail to customer-list-head;
relate customer-list-head to empty;
relate customer-list-backup-tail to customer-list-backup-head;
relate customer-list-backup-head to empty;
  while (eof ≠ "T") do
        if empty? (customer-list-tail)
          then eof := "T"
           else eof := "F";
                relate customer-record to mailing-list-tail;
                relate mailing-list-head to customer-record;
                relate backup-record to customer-record;
                comment : "copy a record";
                relate mailing-list-backup-head to backup-record
                relate backup-record to custom-list-backup-tail
        fi od
end;
```

9.2.4 Ignoring Useless *Relate* Statements

When a **relate** statement relates two entities there must by definition be a relationship between these two entities. So a relationship can be derived from the statement and the name of the relationship is provided by the user according to the information in the program. When a **relate** statement relates one or no entities, i.e., one or two of the components of the **relate** statement are neither entities nor records which may be transformed into entities, it means that this relate statement does not represent a relationship of any entity and can be deleted. In the following program:

```
relate customer-record to mailing-list-tail;
relate mailing-list-head to customer-record;
relate backup-record to customer-record;
comment : "copy a record";
```

mailing-list-tail and *mailing-list-head* are not entities or records so that the first two **relate** statements do not represent relationships. So the above can be simplified into:

```
relate backup-record to customer-record;
comment : "copy a record";
```

So far the program in the previous section has become:

```
paragraph
entity customer-record with
   attr name
   attr address
   attr phonenum end;
entity backup-record with
   attr b-name
   attr b-address
   attr b-phonenum end;
entity eof end;
   while (eof ≠ "T") do
          if empty? (customer-list-tail)
            then  eof := "T"
             else  eof := "F";
                 relate backup-record to customer-record;
                 comment : "copy a record";
                  fi
          end;
```

9.2.5 Abstracting Branching Structures and Loop Structures

As discussed in Section 7.5.4, the **if ... fi** statement in the program,

```
if empty? (customer-list-tail)
  then eof := "T"
   else eof := "F"
fi;
```

is abstracted to

```
relate eof to "T";
relate eof to "F";
```

By turning these two statements into **relate** statements and they both do not contribute to a **relationship** statement and hence are ignored eventually.

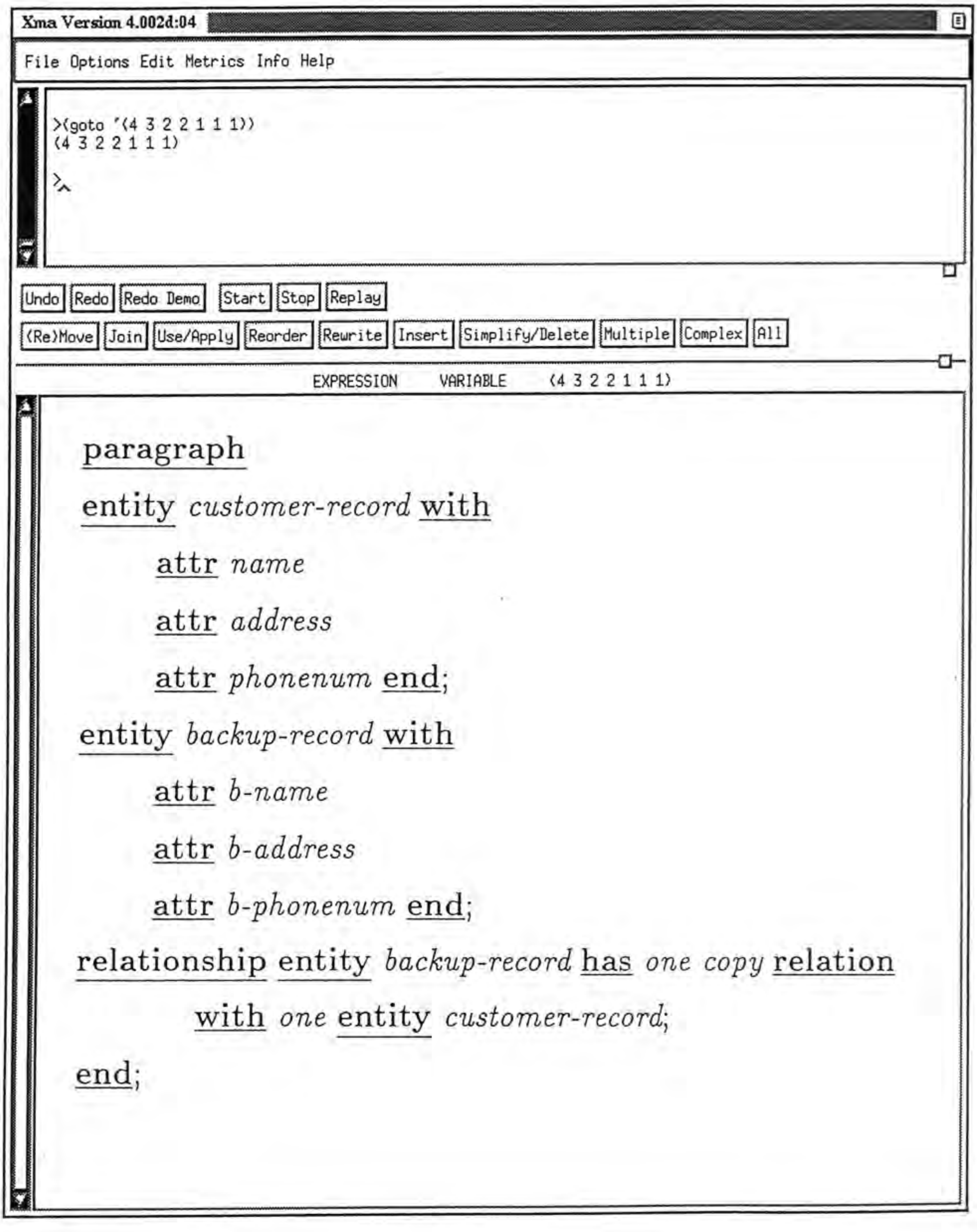
A **while ... do ... od**, is also discussed in Section 7.5.4, i.e., a **while** loop can be removed just leaving the body of the loop. In this example, the **while** statement,

```
while (eof ≠ "T") do
        backup-record := customer-record;
        comment: "copy  a  record";
od;
```

is supposed to implement copying all the records from the original file to the backup file. At the higher abstraction level, it is to say that there is a "copy" relationship between two entities. Each record is one instance of an entity. The above program is transformed into

```
relationship entity backup-record has one copy relation
          with one entity customer-record;
```

9.2.6 The Resultant "Program" in WSL and its Entity-Relationship Attribute Diagram



```
>(goto '(4 3 2 2 1 1 1))
(4 3 2 2 1 1 1)

>
```

```
paragraph
entity customer-record with
        attr name
        attr address
        attr phonenum end;
entity backup-record with
        attr b-name
        attr b-address
        attr b-phonenum end;
relationship entity backup-record has one copy relation
            with one entity customer-record;
end;
```

Figure 9.1: The Result from File Copying Program on X-Windows Front End

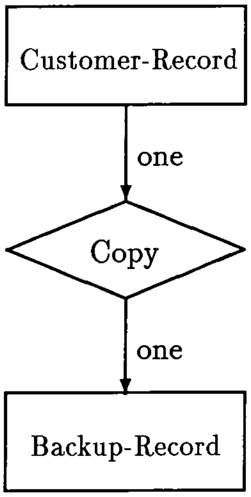


Figure 9.2: The Entity-Relationship Attribute Diagram for "File-Copy" Program

After applying transformations discussed in this section, the final result of the "file-copy" program can be shown by an Entity-Relationship Attribute Diagram. The result displayed on X-Window interface is in Figure 9.1; the following is in WSL representation and the diagram itself is in Figure 9.2.

```
paragraph
entity customer-record with
    attr name
    attr address
    attr phonenum end;
entity backup-record with
    attr b-name
    attr b-address
    attr b-phonenum end;
relationship entity backup-record has one copy relation
        with one entity customer-record;
end;
```

9.3 Case Study 2 — A Program Using Alias

Aliasing problem was already discussed in detail in Chapter 6. An example program in COBOL is given here as an additional illustration to show how an alias used for the second purpose of aliasing. The original program is like this:

```
WORKING-STORAGE SECTION.
01  ALPHANUMERIC-ITEMS.
    05  MONTH-DISP PIC X(09).
01  NUM-MONTH-REC.
    05  NUM-MONTH-IN PIC 9(02).
01  MONTH-TABLE.
    05  FILLER PIC X(09)  VALUE ‘JANUARY’.
    05  FILLER PIC X(09)  VALUE ‘FEBRUARY’.
    05  FILLER PIC X(09)  VALUE ‘MARCH’.
    05  FILLER PIC X(09)  VALUE ‘APRIL’.
    05  FILLER PIC X(09)  VALUE ‘MAY’.
    05  FILLER PIC X(09)  VALUE ‘JUNE’.
    05  FILLER PIC X(09)  VALUE ‘JULY’.
    05  FILLER PIC X(09)  VALUE ‘AUGUST’..
    05  FILLER PIC X(09)  VALUE ‘SEPTEMBER’.
    05  FILLER PIC X(09)  VALUE ‘OCTOBER’.
    05  FILLER PIC X(09)  VALUE ‘NOVEMBER’.
    05  FILLER PIC X(09)  VALUE ‘DECEMBER’.
01  MONTH-TABLE-RED REDEFINES MONTH-TABLE.
    05  MONTH-DESCRN PIC X(09)  OCCURS 12 TIMES.

PROCEDURE DIVISION.
MAIL SECTION.
    ACCEPT NUM-MONTH-REC.
    IF NUM-MONTH-IN < 01 OR > 12
        DISPLAY ‘INVALID MONTH’
```

```
ELSE
    MOVE MONTH-DESCRN (NUM-MONTH-IN) TO MONTH-DISP
    DISPLAY NUMBER-MONTH-IN, ‘ = ’, MONTH-DISP.
STOP RUN.
```

According to the experience obtained from the first case study, the only relevant statement in the procedure section is the MOVE statement. In addition, since the translation process from COBOL to WSL has been demonstrated in previous chapters and therefore, the program shown below is the translated program. Also, initial simplifications were already made to the program by applying program transformations. In particular, record "month-table" was simplified to having one field, "month-name" and record "month-table-red" to having on field, "month-descrn".

```
record alphanumeric-items with
    record month-disp [9 of char] end
end;
record num-month-rec with
    record num-month-in [2 of char] end
end;
record month-table with
    record month-name [9 of char] end
end;
record month-table-red with
    record month-descrn [9 of char] end
end;
redefine month-table with month-table-red;

month-disp := month-descrn;
```

Since record "month-table-read" redefines record "month-table", "month-disp" is also related to "month-name". When records in above program are translated in the entities, the following program is obtained.

```
entity alphanumeric-items with
   attr month-disp end;
entity month-table with
   attr month-name end;
entity month-table-red with
   attr month-descrn end;

relate month-disp  to  month-descrn;
relate month-disp  to  month-name;
```

Further transformations will bring the above program into:

```
entity alphanumeric-items with
   attr month-disp end;
entity month-table with
   attr month-name end;
......
relationship entity alphanumeric-items has one represent relation
         with one entity month-table;
```

9.4 Case Study 3 — A Vetting and Pricing Program Used in a Telephone Company

The program selected for this case study is a real program which could still be in operation in a national telephone company. The emphasis of the case study is on how to tackle a real and heavily maintained program with the method developed in this thesis. Owing to the confidentiality agreement between the telephone company and the REFORM research group, the details of the program will not be presented in this section.

9.4.1 Introduction to the Program

The input of the this program is from specification tape copies and the output consists of two files which priced a valid calls file and a data vet reject file. The program consists of 3750 lines of code written in COBOL. The experiments with the program carried out using the prototype tool are described in detail in the next section.

Experimenting with this program went through several stages.

9.4.2 Translating the Program into WSL and Initial "Tidy-up"

According to the experience obtained in the first case study, some of the COBOL control statements will not contribute to the eventual Entity-Relationship Attribute Diagrams and these statements can be omitted at this stage. These constructs include DISPLAY, REMOVE, INITIALIZE, PERFORM, IF, WHEN, UNTIL, GOTO, SEARCH, ACCEPT, etc.

Some of the "data" statements which are not able contribute to the eventual Entity-Relationship Attribute Diagrams are also omitted. For example, COBOL statements, such as MOVE 0 TO A-VARIABLE, MOVE SPACES TO A-VARIABLE, MOVE "NOTHING" TO A-VARIABLE, ADD 0 TO A-VARIABLE, SUBSTRACT 0 FROM A-VARIABLE, SET A-VARIABLE TO 0, INSPECT statements, COMPUTE statements, etc. were omitted.

After the initial tidy-up, the translated COBOL program in WSL was very much ready for further transformation. It is worth mentioning that information that could be useful in the future abstraction was recorded by WSL **comment** statements.

When the stage finished, there were 1879 lines of WSL code left. Among these WSL statements, 208 were record definitions, and 216 were assignment statements.

9.4.3 Obtaining *relate* Statements

Almost all the assignment statements that were originally translated from the MOVE statements in COBOL were abstracted to **relate** statements. So were those assignment statements originally from ADD and SUBTRACT statements. A **comment** statement was usually added along with each abstraction in order to record information which will be used to decided the degree of the relationship between the two entities which would be obtain from the two records linked by the **relate** statement later on.

9.4.4 Aliased Records

The REDEFINES statements in the original COBOL program were translated into <u>redefine</u> statements in WSL. According to observation, all original records to be redefined were in the same data types as that of the redefining records. The conclusion of applying those functions discussed in Chapter 6 for dealing with alised records was that aliased records would not affect each other in the abstraction process. Therefore, a record and its redefining record was able to be treated as independent records.

27 records were redefined in the original program.

9.4.5 Obtaining Entities

Entities were abstracted from records and this is the starting point of moving from the code level to the conceptual level (refer to Figure 7.4). And this was done when all the restructuring work at the code level had been finished.

9.4.6 Obtaining Relationships

Relationships were mainly derived from the **relate** statements and information recorded by the **comment** statements were used to decide the degrees for the relationships.

Relationships can also be obtained from other sources. For example, relationships were able to be obtained from the declaration of a record in COBOL.

When a subrecord occurs for more than once, the subrecord and the record can both be abstracted to an entity and a relationship between these two entities can also be obtained, i.e.:

```
record X04-TARIFF-SSU-TABLE with
   record X03-TARIFF-SSU-DETAILS with
        record X03-SSU-CHARGE-BAND end
        record X03-SSU-NEW end
   end
end;
comment : "X03-TARIFF-SSU-DETAILS  OCCURS  25";
```

became

```
entity X04-TARIFF-SSU-TABLE end;
entity X03-TARIFF-SSU-DETAILS with
     attr X03-SSU-CHARGE-BAND
     attr X03-SSU-NEW end;
relationship entity X04-TARIFF-SSU-TABLE has one contain
        relation with many entity X04-TARIFF-SSU-DETAILS;
end;
```

The comment statement was translated from the OCCURS construct in the COBOL and was used to decide the degree for the relationship. There were 14 OCCURS constructs in the original COBOL program and therefore, 28 entities and 14 relationships were obtained in this way.

Another example is that a record with multiple levels of subrecords was able to be abstracted to more than one entity, especially when one of the subrecords was assigned to by another record. At the meantime, a relationship between the entities can also be obtained, i.e.,

```
record B43-FACILITY-RATES-RECORD end;
record X03-TARIFF-FAC-TABLE with
   record X03-TARIFF-FAC-DETAILS with
```

```
        record X03-FACILITY-TYPE
        record X03-FAC-LOCAL-NEW
        record X03-FAC-LOCAL-OLD
        record X03-FAC-TRUNK-NEW
        record X03-FAC-TRUNK-OLD
    end
end;
relate B43-FACILITY-RATES-RECORD to X03-TARIFF-FAC-TABLE;
```

was abstracted to:

```
entity B43-FACILITY-RATES-RECORD end;
entity X03-TARIFF-FAC-TABLE end;
entity X03-TARIFF-FAC-DETAILS with
    record X03-FACILITY-TYPE
    record X03-FAC-LOCAL-NEW
    record X03-FAC-LOCAL-OLD
    record X03-FAC-TRUNK-NEW
    record X03-FAC-TRUNK-OLD end;

relationship entity X03-TARIFF-FAC-TABLE has one copy
      relation with one entity X03-TARIFF-FAC-DETAILS;
relationship entity B43-FACILITY-RATES-RECORD has one copy
      relation with one entity X03-TARIFF-FAC-DETAILS;
```

There were 10 cases like the example in this program and therefore, these were dealt with in a similar way.

The "back tracking" technique introduced in Chapter 7 was applied when the above example was experimented. At first the record with multiple levels of sub-records were dealt with independently to obtain entities and relationships. Other routes were also experimented before the solution introduced here was reached.

By the time the stage finished, there were 1486 lines of WSL left.

9.4.7 Final Tidy-up and Result

Finally, duplicate entity relationships which might be obtained from different places of the program, and these were checked and removed. All the comment statements were removed. The resultant WSL program for representing obtained Entity-Relationship Attribute Diagrams for the original COBOL program was 999 in length, containing 267 entities (excluding 482 lines representing attributes to some of the entities) and 250 entity-relationships.

The case study turned out exactly the way as shown in Figure 7.4, in which records, entities and relationships were obtained, in particular. This case study demonstrates true the hypothesis made in Figure 7.4. Nevertheless, foreign keys were not found used in the example.

9.5 Case Study 4 — A Customer Account Ledger Program Used in A Telephone Company

An experiment was also carried out with another real program.

9.5.1 About the Program

The program presented in the section is named "Customer Account Ledger Extract", being composed of 7070 lines of COBOL source code. The program is physically written as "one program", i.e., having one data division, one procedure division, etc., and being stored in one file.

The only information source is the source code itself. According to the comments in the code, there were 23 amendments during the period from August 1985 when the code was first written and November 1991 when the most recent compilation of the program took place before it was sent to the REFORM research group to be studied.

9.5.2 Experiments to the Program

Experiments carried out can be summarised as follows:

Modularising the program by inspection

The first step to analyse the program was to inspect the code and divide it into smaller manageable blocks.

The identification division and the environment division of the COBOL program consists of about 200 lines, mainly including comments serving as an "information area". The data division consists of about 2400 lines, declaring six files and 230 records. The procedure division consists of the remaining 4500 lines of source code, in which there are four modules, namely main control section, initialisation section, main processing section and final section. The main control section calls the other three sections sequentially.

File and record declarations

Translating file and record declarations into WSL is straightforward and when they become WSL records, they would be considered as candidates of entities. Not all WSL records are loaded into the prototype at the same time. Any record is only loaded in conjunction with a block of code in which the record will be used.

The initialisation section

This section consists of 400 lines of code including 100 lines of comments. About 100 lines of code which open those six files declared earlier and initialise data parameters are transformed into WSL. The rest of the code in this section are used to initialise input and output devices and have therefore not been dealt with. A number of entities and relationships are obtained using the prototype from this section.

The main processing section

The main processing section is composed of 32 functionally independent subprograms. One of these subprograms acts as the "main" subprogram which coordinates the remaining subprograms to process all customer account records. Each of the remaining 31 subprograms performs one function, such as "Read-customer-account-balance", "Check-unbilled-amount", "Output-ledger-records", etc. Some of these subprograms were translated into WSL and each time one subprogram was loaded into the prototype together with the records (having been translated earlier) used by the subprogram. A number of Entity-Relationship Attribute Diagrams are obtained by manipulating these subprograms in the prototype tool.

The final section

The final section consists of about 200 lines COBOL source code excluding comment lines. This section mainly writes a final report of customer account ledger. The section of code was translated into WSL. The obtained Entity-Relationship Attribute Diagram (Figure 9.3) clearly shows the data structure of a final customer account ledger report.

9.5.3 Comments

So far, the file record declaration section and a number of code segments in other sections in the program have been studied in depth, and the following results have been obtained:

- One line of COBOL code is typically translated into one line of WSL.

- A number of entities and relationships were successfully derived from the code (approximately 200 entities were extracted from the original 01 level records, and other entities were derived from file operations and from abstract data types; about 50 relationships were derived from those records which had subrecords, abstract data types and foreign keys).

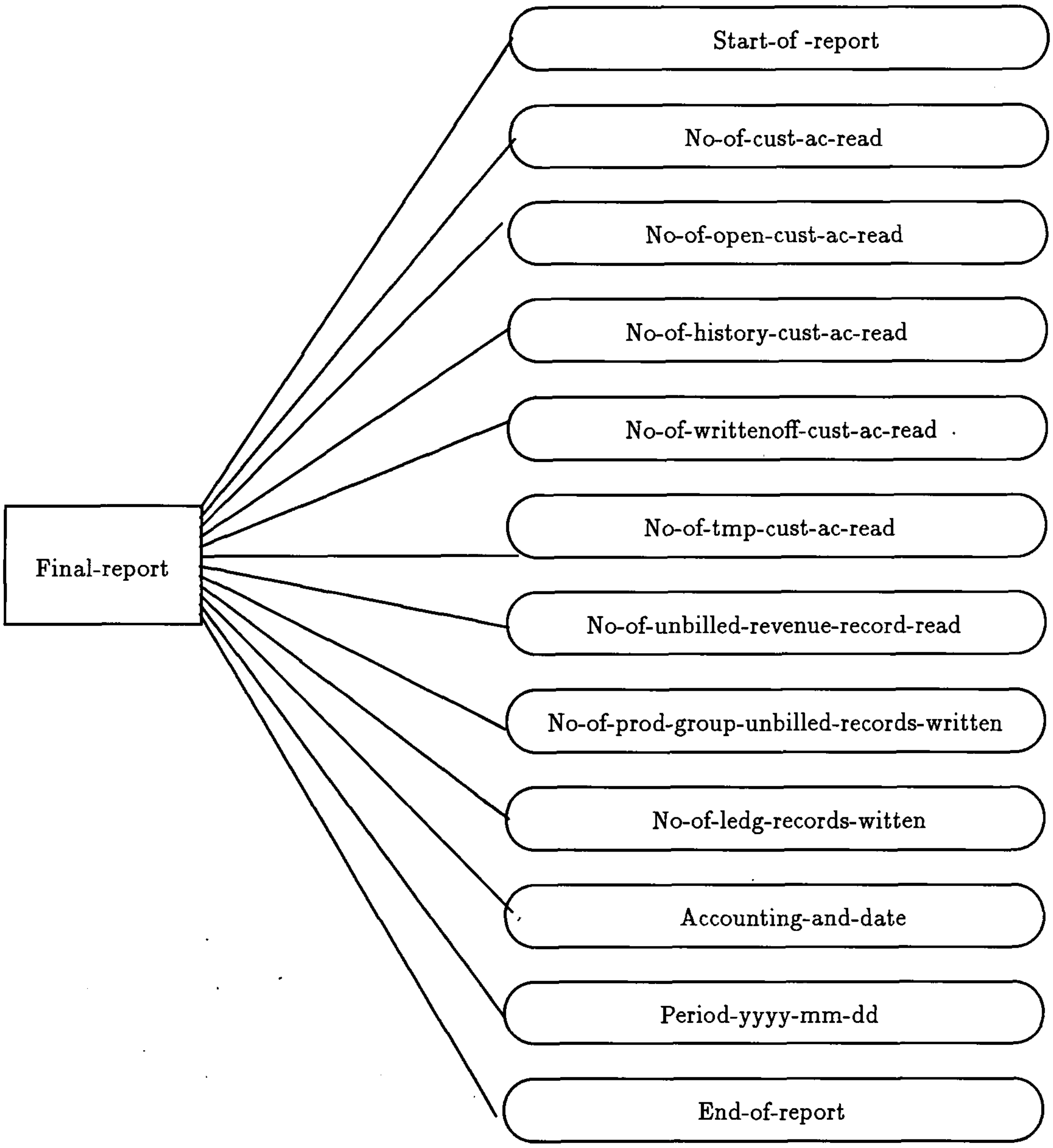

Final-report
Start-of -report
No-of-cust-ac-read
No-of-open-cust-ac-read
No-of-history-cust-ac-read
No-of-writtenoff-cust-ac-read
No-of-tmp-cust-ac-read
No-of-unbilled-revenue-record-read
No-of-prod-group-unbilled-records-written
No-of-ledg-records-witten
Accounting-and-date
Period-yyyy-mm-dd
End-of-report

Figure 9.3: Final Report on Customer Account Records

- Data designs obtained from programs made these programs much more comprehensible.
- Deriving data designs by using the tool is a faster and more reliable process than doing it manually (even when that is possible).
- Metric graphs monitoring the process of transforming programs show that programs have been considerably simplified, i.e., by becoming more abstract.
- The Entity-Relations Attribute model is useful when a static, but not dynamic, view of a software system is sought, because the model only describes the relationships between data objects (or entities) and it does not describe how data flows in a system.
- Attention should be paid and more analysis should be done when using the information obtained during the reverse engineering process. for example, relationships obtain such as "file-backup" in Case Study 1 emerged from reverse engineering but were not really a part of the application domain.

In a word, with the help of the method and the prototype tool built, reading a real program such as a program of 7000 lines is no longer a frightening task.

9.6 Case Study 5 — A Public Library Administration System

In addition to the experiments presented in previous sections, an experiment was carried out with a public library administration system which was implemented in COBOL.

9.6.1 Background of the Program

This program [42] was built to manage a public library in order to get a better control over all the operations related with the books in stock during all their life cycle.

With a source code of over 4000 lines, the program was developed for execution in a typical mainframe environment: DOS/VSE, the most common mainframe operating system; CICS: the most common teleprocessing monitor; and TOTAL, very old but very common DBMS. CICS was used for the communication between this program and the end user; TOTAL was used to store details of books.

9.6.2 Prepreparing the Program for the Prototype

Before it was loaded into the prototype the library management program was pre-processsed as follows:

The source code of this program was first manually divided into eight COBOL modules (temporarily named Lib1, Lib2, ..., Lib8) according to the layout of the source code. It was soon found out that one of the modules (Lib8) was a batch process program which is used for updating a back-up file and printing out an inventory of all the books in stock. Because this program (Lib8) was implemented mainly with CICS calls and TOTAL calls [1], an analysis to this module seemed not directly relevant to the research and therefore not carried out. It was also found that another module, i.e., Lib7, was an "include" file which defined common data structures for the system. Lib1 to Lib6 are main components of the library management system and Lib7 was included by these six components.

Lib1 to Lib7 were then manually translated into WSL separately. CICS calls and TOTAL calls were translated into external procedure calls and external function calls. Because the frequent occurrences of CICS and TOTAL calls in between COBOL source lines literally made a detailed analysis to the control part of the system more difficult, experiments had to be carried out with the following guide-line in mind: to obtain data design as much as possible from the data part of the system. Every effort was made in deriving Entity-Relationship Attribute Diagrams from data divisions in Lib1 to Lib6.

[1] It was observed that an analysis of the parameter list of a CICS or TOTAL call would show the data declaration of that CICS or TOTAL call and this information could be therefore used by the program transformer.

9.6.3 Dealing with CICS Calls

Fortunately, most CICS/TOTAL calls in this COBOL source code are used for handling simple "interfacing" operations so that they can be translated into external procedure (function) calls in WSL. take the following segment of code as an example:

```
... ...

IF OPTION01I = 'C'
   EXEC CICS START TRANSID ('CREATE')
                   TERMID ('EIBTRMID')     END-EXEC
   EXEC CICS RETURN                        END-EXEC
ELSE
   IF OPTION01I = 'D'
   EXEC CICS START TRANSID ('DROP')
                   TERMID ('EIBTRMID')     END-EXEC
   EXEC CICS RETURN                        END-EXEC
... ...
```

This segment was taken from the part which examines user options. When 'C' is input, CICS is called to invoke the execution of 'CREATE' program (to create a book in the library). Similarly user option 'D' invokes 'DROP' program (to drop a book from the library). In both 'IF' cases, after the 'CREATE' or the 'DROP' program is executed, CICS is called afterwards to return the control to the caller. This CICS call will not literally appear in the WSL program, because when the first CICS call was translated into a WSL external procedure call, the return of the control was semantically covered by using a procedure. Therefore, the above program may be translated into:

```
 ......
if $option01i = 1$
```

```
then !p cics (start var red1, eibtrmid)
 else
     if option01i = 2
       then !p cics (start var red2, eibtrmid)
            ......
```

9.6.4 Resultant Entity-Relationship Attribute Diagrams

After being loaded into the prototype, program modules, Lib1 to Lib7, were each manipulated. Typically several entities and their relationships were extracted from each module. For example, among entities and relationships obtained from Lib7, five entities and four relationships can be used to draw an Entity-Relationship Attribute Diagram shown in Figure 9.4.

9.6.5 Understanding the Program Through A Data Design

An analysis to the Entity-Relationship Attribute Diagram in Figure 9.4 has provided a better understanding to the original program.

A library has many books in it. One *book* has one identifier and may have more than one copy. A book *identifier* is composed of a *title*, a *publishing house*, one or many *authors*, *ISBN* and *publishing year*. There may be more than one copy for the same book and each *copy* has a reference number, a piece of information on its physical *location* and a *status* flag on whether or not the copy is in the library.

To identify a book is to confirm the identifier of the book because a book has only one identifier (see the Entity-Relationship Attribute Diagram in Figure 9.4). Information may have to be provided by a user up to all components of the identifier (in the Entity-Relationship Attribute Diagram, the entity "Identifier" has four attributes and another entity "Author" attached), i.e., title, publisher, ISBN, publishing year and author(s). It can be deduced that, in the actual program, the process of consulting about the existence in the library of a certain book should

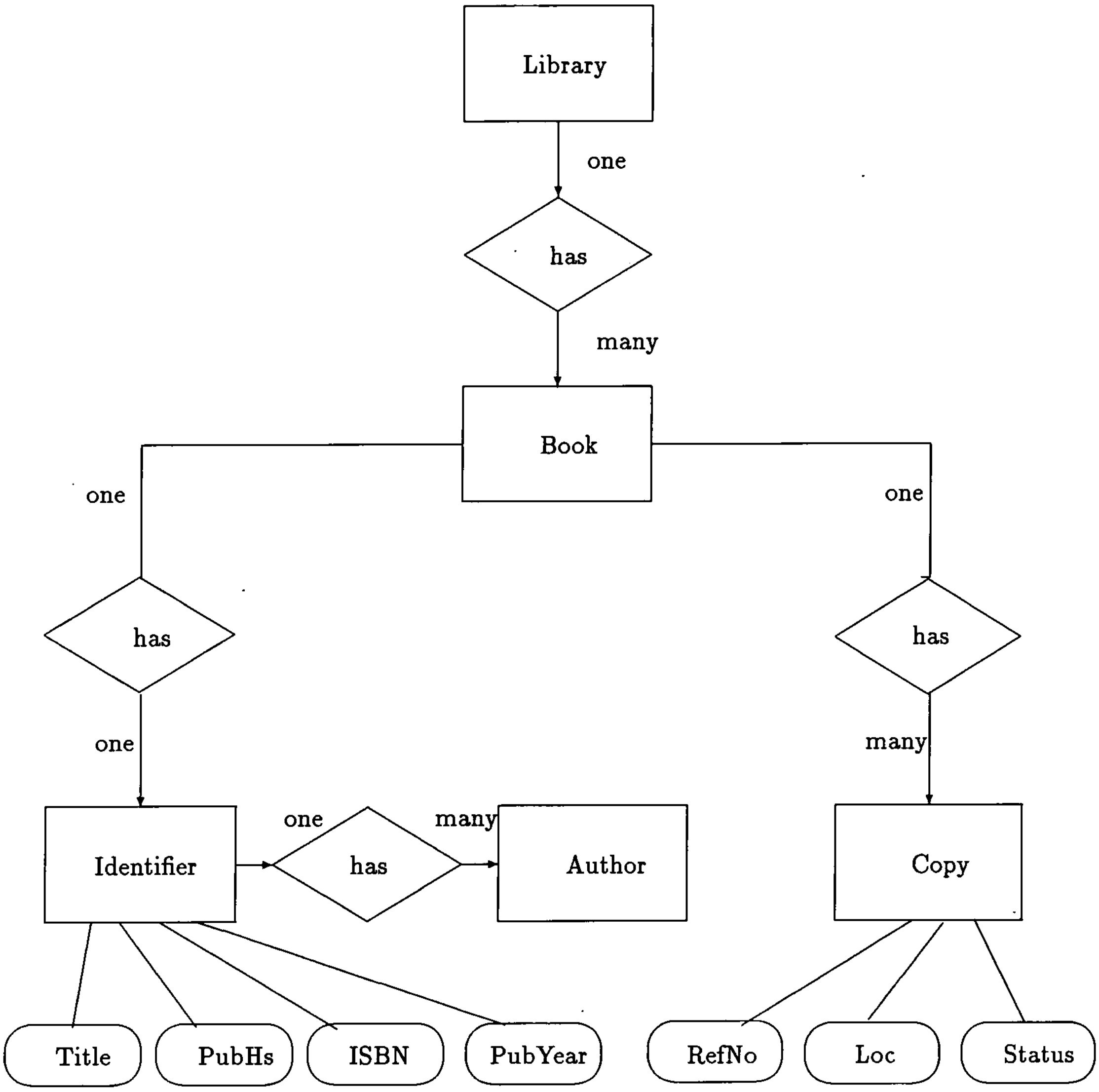


Figure 9.4: Books in a Library

be a process of identifying if there is such a book in the library according to the information supplied by a user.

Similarly, when a new book is created in the library, while all information constructing the identifier of the book would be recorded in the library system, a new copy (of the book) should also be created by allocating a reference number and a location in the library, and setting up the status of the book as “in library”.

9.7 An Example of Using the Metric Facility

An experiment has been conducted in obtaining measuring results on the following program (by monitoring the metrics as transformation progressed):

```
var ⟨m := 0, p := 0, last := “ ”⟩ :
    actions PROG :
    PROG ≡
      ⟨line := “ ”, m := 0, i := 1⟩;
      call INHERE.
    L ≡

      i := (i + 1);
      if (i = (n + 1)) then call ALLDONE fi;
      m := 1;
      if (item[i] ≠ last)
         then write(line var std_out);
              line := “ ”;
              m := 0;
              call INHERE fi;
      call MORE.
    INHERE ≡

      p := number[i];
      line := item[i];
      line := ((line ++ “ ”) ++ p);
      call MORE.
    MORE ≡

      if (m = 1) then p := number[i];
                      line := ((line ++ “, ”) ++ p) fi;
    last := item[i];
    call L.
  ALLDONE ≡

    write(line var std_out);
    call Z. endactions end
```

After transformations have been applied to this program, it becomes:

```
⟨line := " ", i := 1⟩;
while $(i \neq (1 + n))$ do
        line := ((item[i] ++ " ") ++ number[i]);
        i := (1 + i);
        while $((i \neq (1 + n)) \wedge (item[i] = item[(i - 1)]))$
                do line := ((line ++ ", ") ++ number[i]);
                i := (1 + i) od;
        write(linevar std_out) od
```

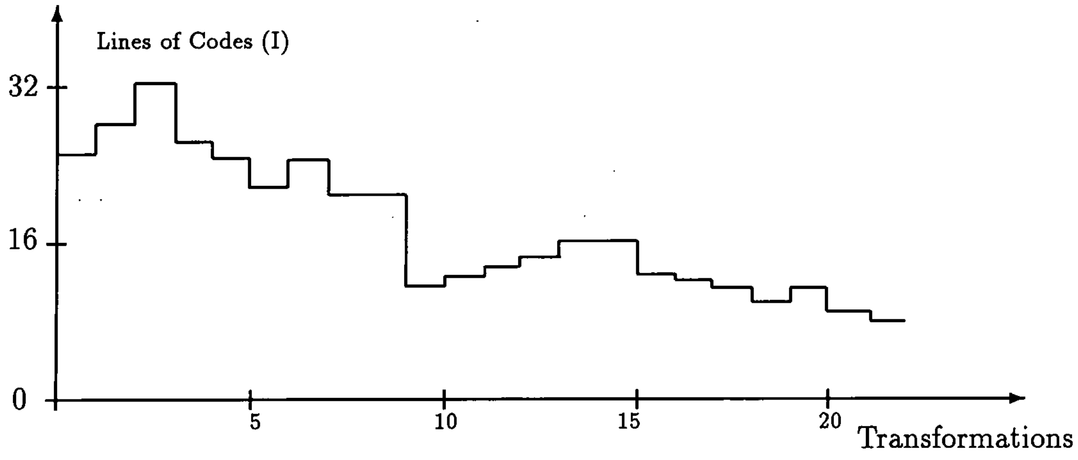
The measuring results for LOC and CFDF can be seen in Figure 9.5 and Figure 9.6 respectively. Results for other metrics are similar to the results shown in those two figures and , therefore, are not included. Figure 9.5 and Figure 9.6 are actual screen dumps. Their equivalent forms are shown in Figure 9.7.

```
                                   Lines of Code
    0                               32
   +----------+---------+---------+---------+->
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$               'EXPAND_ACTION_CALL
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$           'EXPAND_ACTION_CALL
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$                 'COLLAPSE_ACTION_SYSTEM
   !$$$$$$$$$$$$$$$$$$$$$$$$$$                  '<<--MERGE
   !$$$$$$$$$$$$$$$$$$$$$$$                     'APPLY-->>
   !$$$$$$$$$$$$$$$$$$$$$$$$$                   '<<--MERGE
   !$$$$$$$$$$$$$$$$$$$$$$                      'APPLY-->>
   !$$$$$$$$$$$$$$$$$$$$$$                      'APPLY-->>
   !$$$$$$$$$$$$                                'REMOVE_REDUNDANT_VARS
   !$$$$$$$$$$$$$                               'PUT_IN_NEW_OUTER_LOOP
   !$$$$$$$$$$$$$$                              'ADD_ASSERT
   !$$$$$$$$$$$$$$$                             'INSERT_ASSERT
   !$$$$$$$$$$$$$$$$                            'MERGE-->>
   !$$$$$$$$$$$$$$$$$                           'TAKE_OUTSIDE_LOOP
   !$$$$$$$$$$$$$$                              'DELETE_ALL_ASSERTIONS
   !$$$$$$$$$$$$$                               '<<--MERGE
   !$$$$$$$$$$$$                                'JOIN_ALL_CASES
   !$$$$$$$$$$$                                 'LOOP->WHILE
   !$$$$$$$$$$$$                                NIL
   !$$$$$$$$$$                                  'LOOP->WHILE
   !$$$$$$$$$                                   'DELETE
   !$$$$$$$$$                                   NIL
  Y

Transformations                       (Name of The Transformation Used)
```

Figure 9.5: The Measure Result of Lines of Code

```
              Control-Flow and Data-Flow Complexity
    0                                    84
   +--------+--------+--------+--------+--------->
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$           'EXPAND_ACTION_CALL
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  'EXPAND_ACTION_CALL
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  'COLLAPSE_ACTION_SYSTEM
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$    '<<--MERGE
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$           'APPLY-->>
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$      '<<--MERGE
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  b           'APPLY-->>
   !$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$             'APPLY-->>
   !$$$$$$$$$$$$$$$                            'REMOVE_REDUNDANT_VARS
   !$$$$$$$$$$$$                               'PUT_IN_NEW_OUTER_LOOP
   !$$$$$$$$$$$$$$                             'ADD_ASSERT
   !$$$$$$$$$$$$$$$$                           'INSERT_ASSERT
   !$$$$$$$$$$$$$$$$$$                         'MERGE-->>
   !$$$$$$$$$$$$$$$$$$$                        'TAKE_OUTSIDE_LOOP
   !$$$$$$$$$$$$$                              'DELETE_ALL_ASSERTIONS
   !$$$$$$$$$$$$                               '<<--MERGE
   !$$$$$$$$$$$                                'JOIN_ALL_CASES
   !$$$$$$$$$$$$                               'LOOP->WHILE
   !$$$$$$$$$$$$$                              NIL
   !$$$$$$$$$$$$$                              'LOOP->WHILE
   !$$$$$$$$$$$                                'DELETE
   !$$$$$$$$$$$                                NIL
   Y

Transformations                       (Name of The Transformation Used)
```

Figure 9.6: The Measure Result of Control-flow and Data-flow Complexity



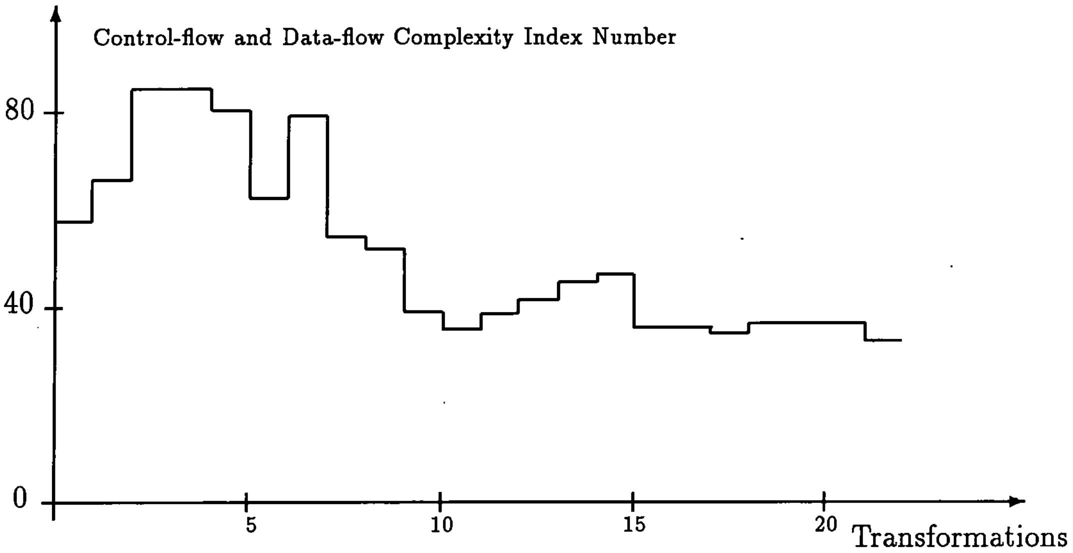


Figure 9.7: Measure Results of Using Metric Facility

Chapter 10

Conclusions

The conclusions of this thesis can be divided into three parts. The first part, *summary of the thesis*, summarises what has been done in this thesis; the second part, *assessment*, demonstrates how the work done in the thesis has achieved the predefined goals and points out what else ought to be carried out in making full use of the results achieved in this thesis; and the third part, *future directions*, suggests some research topics which have been stimulated by this research.

10.1 Summary of Thesis

This thesis has undertaken an original investigation and feasibility study of the problems concerning obtaining program designs from data intensive programs using a program transformation approach. A systematic method has been developed and a prototype has been implemented based on the method developed [169].

The investigation was started by addressing the overall process of software engineering. Software maintenance is nowadays the most expensive stage in the software life cycle and enormous maintenance problems desperately await solutions. Furthermore, reverse engineering is a crucial part of software maintenance.

The work related to reverse engineering was examined in detail. Formal specification and program transformation techniques which were mainly used in forward engineering before are proposed to have their new applications in reverse engineering. Existing program verification tools are reviewed to show not only that one of these tools is borrowed to build a component of the prototype (the Symbolic Executor) but also that transformational programming approach has more strengths in constructing a reverse engineering tool than program proving approach. The main existing reverse engineering approaches have been analysed and criticised so that an original method of reverse engineering can be proposed.

The method proposed in this thesis is to derive a program data design from a data intensive program through program transformations. Program design is represented in Entity-Relationship Attribute Diagrams and the source data intensive programs in the study are in COBOL (mainly), C, etc. In order to apply program transformations, a suitable language which can represent both source programs and Entity-Relationship Attribute Diagrams is needed. We used a wide spectrum language (WSL) for this purpose.

The source program has to be translated into its semantically equivalent form in WSL. Since a program in this form is basically at the "physical" abstraction level, abstraction levels have to be crossed to get a program design which is at the "conceptual" level. Problems of crossing levels of data abstraction are discussed in this thesis.

Another goal of the research is to build a tool based on the method developed. Because the method itself is a substantial extension of the REFORM approach, the prototype is designed as an extension of the existing tool — the "Maintainer's Assistant". The extension includes extending the existing WSL to support representing COBOL programs and Entity-Relationship Attribute Diagrams, implementing program transformations for crossing levels of data abstraction and deriving Entity-Relationship Attribute Diagrams, and implementing supporting components of the tool.

The designed prototype has been fully implemented. Experiments have been carried out and results of those experiments are presented.

10.2 Evaluation

The project is evaluated in this section by examining it against the main criteria predefined in Chapter 4 (Section 4.4) for success.

Crossing levels of abstraction is accomplished by applying those program transformations which transform a program to another at a higher abstraction level. The transformations were invented based on the research result of the project on crossing levels of abstraction.

A prototype has been implemented by extending the existing Maintainer's Assistant. While using the Maintainer's Assistant as a platform, the prototype has made a substantial extension to the existing Maintainer's Assistant, including the extension to the WSL, the extension to the Transformation Library, and extensions to supporting tools.

One way to measure the success of acquiring data designs from the code is to use metrics. Six metric measures (code measures) have been defined to reflect the features of reverse engineering and a metric facility has been implemented as a supporting component of the prototype.

It has been demonstrated viable to extract data design from the code if using the method developed in this thesis. The method works and it works well:

- Entity-Relationship Attribute Diagrams have been produced from a large number of COBOL programs.
- The abstracted Entity-Relationship-Attribute Diagrams are able to represent the designs of the original programs. The correctness of the obtained ERA diagrams is at present checked manually based on human knowledge and expertise. During the duration of this project, experts on COBOL programming were consulted to identify the features of data-intensive programs and for evaluating the resultant Entity-Relationship Attribute Diagrams obtained from the example programs using the prototype tool. It was agreed by the experts that the obtained Entity-Relationship Attribute Diagrams represented viable designs for these example programs and can be used as possible data designs for subsequent forward engineering of the programs.

In some cases it can be done in an inverse way, i.e., checking whether the original program is an implementation of the obtained Entity-Relationship Attribute Diagram. In some cases, such as in Case Study 1, it can be done in an inverse way, i.e., checking whether the original program is an implementation of the obtained Entity-Relationship Attribute Diagram. Of course, the eventual aim is to assure correctness by proving that the transformations are semantic preserving. This has been done for some transformations.

- The method combines analysis of data and code, and therefore, it can address two major problems common to data-intensive programs, i.e., aliases and relations (like foreign keys) that affect both data and code. This distinguishes our approach from others, e.g., Bachman's work.

- The method is sufficiently general to apply to data intensive programs written in other source languages though COBOL programs have been mainly used in experiments to date. For example, a program written in the C programming language can be reverse engineered using the approach as long as the C program is translated into WSL. In fact, a few examples in the experiments were derived from C programs, for instance, the example (in WSL) used in Section 7.5.4 was translated from a C program.

- This method only requires source code as its input and it can be applied to heavily modified code typical in systems which have been maintained over many years.

- There are few restrictions on the approach developed in this research. Perhaps the only restriction is that the user needs to supply the source code to be reverse engineered. The approach is, however, fundamentally interactive, and the tool supports the expert maintainer in restructuring code and then extracting designs. It is not automatic. The user-driven mode is common with the approach used in almost all other reverse engineering approaches. The user's role is well taken into consideration when the approach was studied and a user-friendly interface was designed and implemented when the prototype tool was built.

- The development of the method and the implementation of the prototype show that this approach has covered a scope from theory to practice [170]. It has not been seen so far in the literature that any other system can derive data designs represented in Entity-Relationship Attribute Diagrams. Therefore, it is believed that the approach described in this thesis cater what others do not — no other projects have gone as far as this project has. This can be ensured by re-examining other reverse engineering approaches introduced in Section 3.5. For example, in the REDO project [100,101], it was just shown that there was a method by which COBOL code can be abstracted to produce explicit mathematical descriptions of functionality. However, more work still has to be devoted to make the REDO approach more practical.

It appears to be at first that the method developed could be applied manually (i.e., manipulating COBOL code by hand and extracting entities and their relationships from the code directly) and therefore, it might not necessary to build a tool to implement the method. This argument might be true when a program is fairly small (like the program in the first case), but when the size of a program to be maintained increases building a tool for doing this will apparently show many advantages.

Three kinds of transformations defined earlier have been used alternatively during the process of manipulating the example cases. For example, in the first case, equivalence transformations were used first to handle file operations, and then abstraction transformations are applied to derive entities and entity relationships. Refinement transformations were used when "back tracking" is needed.

The role of human knowledge was to guide the whole application process of the method. When COBOL programs are translated into WSL, because a COBOL-to-WSL translator has not been built (or even when building the translator), human knowledge has to be used to ensure that WSL programs are semantically equivalent to the original COBOL programs. It can be seen that human knowledge plays a decisive role in choosing transformations, in particular, abstraction transformations. Finally it is also human knowledge that determines whether the obtained Entity-Relationship Attribute Diagrams represents a reasonable data design of the original COBOL program.

Where the method developed in this thesis needs improving was observed as well. The method developed in this thesis can powerfully deal with most example cases studied in the project. In particular, COBOL records and files are able to be represented in WSL and this is crucial to the implementation of the prototype as well as the successful application of the method. Therefore, it is comparatively easier to extract Entity-Relationship Attribute Diagrams from a relatively independent (self-contained) segment of COBOL code with record (file) definitions, but it is more difficult for a COBOL segment with many calls to other segments (i.e., with many PERFORM statements) because the structural complexity is increasing. The problem has to be helped by building more powerful "restructuring" transformations, which is not a main thrust of the thesis. Another reason that this method can cater for most example cases is that the translation from COBOL to WSL is currently done manually. Building a translator needs to consider all possible syntax combinations of COBOL language while hand translation can tailor any program as long as the program is semantically translated into an equivalent form in WSL. Once a program is represented in WSL, according the author's experience, there are always some transformations available to simplify and eventually to abstract Entity-Relationship Attribute Diagrams from it, and some entities and entity relationships were almost always derived (representing part of the data design, not the full data design).

10.3 Assessment

The research described in the thesis brings us valuable detailed understanding in this field.

The method developed in the thesis is a successful outcome of the project. This method combines transformational programming and data abstraction and thus give us the confidence when we manipulate the programs.

Another feature of this approach is to employ Entity-Relationship Attribute

Diagrams to represent the final products (data designs). The existing WSL has been extended to represent the original programs and Entity-Relationship Attribute Diagrams. Therefore, the original objects and the target objects can both be represented and manipulated in the same language.

In order to capture any useful information for data abstraction, all materials associated with data are to be made full use, i.e., records, assignment statements, etc.

Common data structures used in programming, data-intensive programming in particular, have been investigated, for example, foreign keys, aliasing problem, file operations, ADTs, etc.

One of the useful features of the prototype is that the system can be easily extended. When more program transformations are available, they can be integrated into the system with ease.

Since the project made a novel attempt to solve the problem of acquiring data designs from data intensive programs, it may have some weaknesses as well as strengths. The research described in this thesis includes several subjects within the scope of software engineering. Although the author has managed tackling the main issues of the project, further research into some side issues may make the project even more promising. During the course of this project, a number of drawbacks have been exposed both in the method and its implementation.

The method relies on a certain amount of user expertise. In particular a user must understand the principle of program transformation and be capable of choosing program transformations in order to reverse engineer the code more efficiently. Also, since a user is heavily involved in naming relationships in Entity-Relationship Attribute Diagrams, the user's misunderstanding to the original program might lead to an inconsistent or even a wrong Entity-Relationship Attribute Diagram.

More supporting components should be built. In this thesis, the translation from program in COBOL or in other data-intensive programming languages to its equivalent program in WSL was done by hand. This is because the techniques of translating programs between programming languages are comparatively mature and therefore, no more effort should be made in the research. Another supporting component is a tool dealing with Entity-Relationship Attribute Diagrams, i.e., printing Entity-Relationship Attribute Diagrams on the screen or to a printer. At present, the end results of Entity-Relationship Attribute Diagram are represented in WSL. When the resource of enhancing the prototype to a fully tool-supported system is available, the COBOL-to-WSL translator can be implemented by existing compiler-writing techniques and the Entity-Relationship Attribute Diagram processor can be built referring existing tools, such as ERDRAW [112,113,150].

More experiments should also be carried out with data intensive programs in other programming languages. This includes not only building translators for translating programs in other languages but also studying the features of programs in those languages. At present, the prototype is mainly COBOL-specific and it can only deal with those data structures that exist in COBOL. However, the method developed in this thesis is general enough to cope with data-intensive programs in other languages. For example, programming languages such as C provide data structures such as pointers that are not supported by COBOL and therefore transformations for dealing with pointers are needed.

When the source code scales up, the method developed in the thesis would still work well provided program segments in a manageable size could still be found. The scale of the source code would affect the method in a situation where smaller program segments are all not comparatively self-contained, i.e., self-contained segments are already not in a manageable size. One possible solution could be to build an Information Database. When the Program Transformer is working on a program segment which has many calls to other segments, it is the Information Database that collects all the necessary information from the called segments for the Transformer so that the Transformer does not need to load in those segments.

One of the leftovers of the project is that some of the transformations used need proving. As argued in early chapters, proving program transformations is not a main thrust of this thesis and it is only worth while proving all the transformations when they are identified suitable for future use. Also, existing transformations may still need optimising and more transformations should be involved particularly when source data intensive programs not written in COBOL are to be experimented with the prototype tool.

However, the original goal of this thesis was **to demonstrate the feasibility** of acquiring data designs from data-intensive programs and it is a huge task. The author of the thesis hopes to be able to argue fairly that this thesis has achieved this original goal extremely well despite the existence of the above mentioned problems.

10.4 Future Directions

The future research directions can be suggested directly from the assessments to this project.

An area for future research is to find out whether the approach developed in this thesis can be used to acquire specifications (e.g., specification written in Z) from programs, which is the original aim of the REFORM project.

Metric measures need to be defined for measuring both codes and specifications — to meet the needs of reverse engineering. It is perhaps important that a metric measure can reflect the process of crossing levels of abstraction.

This research has so far indicated that the approach of program transformation can be used to acquire data designs from data intensive programs. However, the real application of this approach will not be seen until an industrial-strength tool has beed built. Therefore, more research should be conducted to improve the prototype developed in this thesis into a practical tool.

Finally, the work described in this thesis provides a summary of the state of the art in the field of reverse engineering using formal methods [166]. Using formal methods can be a good solution to the software evolution problem, because software maintenance [168], software reuse [171] and software testing [172] are main parts in software evolution. Therefore, the outcomes of this thesis may be a starting point for the research into software evolution using formal program transformations.

Appendix A

Syntax of WSL Extension

A.1 Introduction

This appendix just specifies the abstract syntax of the WSL extension for data abstraction and basic WSL definitions which are used for the extension. The abstract syntax of existing WSL has not been included. The semantics specification of the WSL extension can be seen in Appendix B.

A.2 Programs

Program ::= Command

A.3 Commands

Command

::= – empty

| Command

| Command Command

| **segment** Files Records Command **end**;

| **paragraph** Entities Specification-Statements **end**;

| **create-set** Set-Variable;

| **set-insert** Expression Set-Variable;

| **del-element** Expression Set-Variable;

| **init-q** Q-Variable;

| **q-append** Q-Variable Expression;

| **make-seq** Seq-Variable;

| **seq-remove** Seq-Variable;

| **seq-append** Seq-Variable Expression;

| **init-stack** Stack-Variable;

| **push** Expression Stack-Variable;

| **redefine** Rec-Identifier1 **with** Rec_Identifier2;

| **user-adt-proc-call**(Actual-Parameter-Sequence);

Files ::= File-def | File-def File-def

Records ::= Record-Def | Record-Def Records

Entities ::= Entity-Def | Entities Entity-Def

Specification-Statements

::= Specification-Statement

| Specification-Statements Specification-Statement

A.4 Names

V-name :: = Identifier | V-name.Identifier

A.5 Expressions

Expression

::= Integer-Literal

| Character-Literal

| V-name

| Operator Expression

| Expression Operator Expression

| Record-Aggregate

| Ent-Identifier

| **dis-union**(Expression)

| **dis-intersection**(Expression)

| **q-concat**(Q-Variable Q-Variable)

| **q-rem-first**(Q-Variable)

| **q-length**(Q-Variable)

| **seq-concat**(Expression Expression)

| **sub-seq**(Expression Index1 Index2)

| **sub-seq?**(Expression Expression)

| **pop**(Expression)

| **top**(Expression)

| **depth**(Expression)

| **user-adt-funct-call**(Actual-Parameter-Sequence)

Rec-Aggregate

::= Rec-Identifier

| Rec-Identifier.Rec-Identifier

Set-Variable ::= V-Name

Q-Variable ::= V-Name

Seq-Variable ::= V-Name

Stack-Variable ::= V-Name

A.6 Specification Statements

Specification-Statement

::= – Empty

| Specification-Statement

| Specification-Statement Specification-Statement

| Entity-Def

| Relate-Def

| Relationship-Def

Entity-def

::= **entity** Ent-Identifier **end**;

| **entity** Ent-Identifier **with** Attributes **end**;

Attributes

::= **attr** Attribute-Identifier

| Attributes **attr** Attribute-Identifier

Relate-Def

::= **relate** Rec-Identifier1/Ent-Identifier1 **to** Rec-Identifier2/Ent-Identifier2;;

| **relate** Rec-Identifier1/Ent-Identifier1 **to**

{ Rec-Identifier2/Ent-Identifier2 **and** Rec-Identifier3/Ent-Identifier3};

| **relate** Rec-Identifier1/Ent-Identifier1 **to**

{ Rec-Identifier2/Ent-Identifier2 **or** Rec-Identifier3/Ent-Identifier3};

Relationship-Def

::= **relationship** **entity** Ent-Identifier1 **has** Relation-Degree1 Relationship-Name

relation **with** Relation-Degree2 **entity** Ent-Identifier2;

| **relationship** **entity** Ent-Identifier1 **has** Relation-Degree1 Relationship-Name

relation **with** { Relation-Degree2 **entity** Ent-Identifier2 } **and**

{Relation-Degree3 **entity** Ent-Identifier3 };

| **relationship** **entity** Ent-Identifier1 **has** Relation-Degree1 Relationship-Name

relation **with** { Relation-Degree2 **entity** Ent-Identifier2 } **or**

{ Relation-Degree3 **entity** Ent-Identifier3 };

Ent-Name-Def ::= Ent-Identifier

Ent-identifier ::= Identifier

Entity-Identifier ::= Identifier

Relation-Degree ::= one | many

Relationship-Name ::= Identifier

A.7 Declarations

Declaration

::= Declaration Declaration

| File-Def

| Record-Def

| User-Adt-Def

| User-Adt-Funct-Def

| User-Adt-Proc-Def

File-def ::= **file** File-Identifier **with** Records-Def **end**;

Records-Def

::= Record-Def | Record-Def Records-Def

Record-Def

::= **record** Rec-Identifier [Integer-Literal **of** Type-Char-Literal] **end**;

| **record** Rec-Identifier **with** Records-Def **end**;

Rec-Identifier ::= Identifier

Type-Char-Literal ::= char | int

User-Adt-Def ::= **user-adt** Identifier (Formal-Parameter-Sequence)

User-Adt-Defs **end**;

User-Adt-Defs ::= User-Adt-Def User-Adt-Defs

User-Adt-Def ::= User-Adt-Funct-Def | User-Adt-Proc-Def

User-Adt-Funct-Def

::= **user-adt-funct** Identifier (Formal-Parameter-Sequence)

Expression **end**;

User-Adt-Proc-Def

::= **user-adt-proc** Identifier (Formal-Parameter-Sequence)

Command **end**;

A.8 Parameters

Formal-Parameter-Sequence

::= – empty

| Formal-Parameter

| Formal-Parameter, Formal-Parameter-Sequence

Formal-Parameter ::= Identifier

Actual-Parameter-Sequence

::= – empty

| Actual-Parameter

| Actual-Parameter Actual-Parameter-Sequence

Actual-Parameter ::= Expression

A.9 Lexicon

Program ::= (Token | Comment | Blank)*

Token ::= Integer-Literal | Character-Literal | Identifier
| Operator | . | : | ; | , | := | ~ | (|) | [|] | { | }
| **attr** | **create-set** | **depth** | **del-element**
| **dis-intersection** | **dis-union** | **ent** | **entity** | **file** | **in** | **init-q**
| **init-stack** | **set-insert** | **make-seq** | **paragraph**
| **q-append** | **q-concat** | **q-rem-first** | **rec**
| **relate** | **relationship** | **segment** | **seq-append**
| **seq-concat** | **seq-remove** | **sub-seq** | **sub-seq?** | **top**
| **user-adt** | **user-adt-funct** | **adt-funct-call** | **user-adt-proc**
| **user-adt-proc-call**

Integer-Literal ::= Digit Digit*

Character-Literal ::= 'Graphic'

Identifier ::= Letter (Letter | Digit)*

Operator ::= Op-character Op-character*

Comment ::= **comment** Graphic* end-of-line

Blank ::= space | tab | end-of-line

Graphic
::= Letter | Digit | Op-character | space | tab
| . | : | ; | ,| ~ | (|) | [|]
| { | } | _ | | | ! | ' | ` | " | # | $

Letter
::= a | b | c | d | e | f | g | h | i | j | k | l | m

| n | o | p | q | r | s | t | u | v | w | x | y | z
| A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Op-character ::= + | - | ∗ | / | = | | | \ | & | @ | % | ^ | ?

Note: EBNF is used in this section.

Appendix B

Semantics of WSL Extension

B.1 Introduction

This appendix specifies the semantics of the WSL extension for data abstraction. It is based on the syntax of the WSL extension in Appendix A. It is assumed that the semantics of the existing WSL has already been specified.

B.2 Commands

$\underline{\mathbf{segment}}\ \underline{\mathbf{file}}\ f\ =\ p;\ \underline{\mathbf{record}}\ r\ =\ q;\ S\ \underline{\mathbf{end}}\ =_{\mathrm{DF}}$

$\quad(\underline{\mathbf{file}}\ f,\ \underline{\mathbf{record}}\ r)/\langle\rangle.(\underline{\mathbf{file}}\ f\ =\ p;\ \underline{\mathbf{record}}\ r\ =\ q;\ S);\ \langle\rangle/(\underline{\mathbf{file}}\ f,\ \underline{\mathbf{record}}\ r).\underline{\mathbf{true}}$

$\underline{\mathbf{paragraph}}\ \underline{\mathbf{entity}}\ e;\ S\ \underline{\mathbf{end}}\ =_{\mathrm{DF}}$

$\quad\underline{\mathbf{entity}}\ e/\langle\rangle.(\underline{\mathbf{entity}}\ e;\ S);\ \langle\rangle/\underline{\mathbf{entity}}\ e.\underline{\mathbf{true}}$

$\underline{\mathbf{create\_set}}\ p\ =_{\mathrm{DF}}\ p := \{\ \}$

$\underline{\mathbf{set\_insert}}\ e\ p\ =_{\mathrm{DF}}\ p \cup e$

$\underline{\mathbf{delete\_element}}\ e\ p\ =_{\mathrm{DF}}\ p := p - e$

$\underline{\mathbf{init\_q}}\ p\ =_{\mathrm{DF}}\ p := \langle\ \rangle$

$\underline{\mathbf{q\_append}}\ p\ e\ =_{\mathrm{DF}}\ p := \langle s[1], s[2], ..., s[n], e\rangle$

$\underline{\mathbf{make\_seq}}\ p\ =_{\mathrm{DF}}\ p := \langle\ \rangle$

$\underline{\mathbf{seq\_remove}}\ s\ =_{\mathrm{DF}}\ p := \langle s[2], ..., s[n]\rangle$

$\underline{\mathbf{seq\_append}}\ s\ e\ =_{\mathrm{DF}}\ p := \langle s[1], s[2], ..., s[n], e\rangle$

$\underline{\mathbf{init\_stack}}\ t\ =_{\mathrm{DF}}\ t := \langle\ \rangle$

$\underline{\mathbf{push}}\ e\ t\ =_{\mathrm{DF}}\ \langle e, t[1], ..., t[n]\rangle$

$\underline{\textbf{user-adt-proc-call}}\ \mathcal{X} \equiv\ S.\ =_{\text{DF}}\ (\mu\mathcal{X}.S)$

B.3 Expressions

$x := \underline{\textbf{dis\_union}}(p)\ =_{\text{DF}}\ x := \cup/p$

$x := \underline{\textbf{dis\_intersection}}(p)\ =_{\text{DF}}\ x := \cap/p$

$x := \underline{\textbf{q\_concat}}(p_1\ p_2)\ =_{\text{DF}}\ x := \langle s_1[1], s_1[2], ..., s_1[n], s_2[1], s_2[2], ..., s_2[n]\rangle$

$x := \underline{\textbf{q\_rem\_first}}(p)\ =_{\text{DF}}\ x := s[1]\ \wedge\ p := \langle s[2], s[3], ...s[n]\rangle$

$y := \underline{\textbf{q\_length}}(p)\ =_{\text{DF}}\ \ell(s)$

$x := \underline{\textbf{seq\_concat}}(s_1\ s_2)\ =_{\text{DF}}\ x := \langle s_1[1], s_1[2], ..., s_1[n], s_2[1], s_2[2], ..., s_2[n]\rangle$

$x := \underline{\textbf{sub\_seq}}(pij)\ =_{\text{DF}}\ x := \langle\ s[i], ..., s[n]\rangle$

$y := \underline{\textbf{sub\_seq}}?(p_1 p_2)\ =_{\text{DF}}\ \underline{\textbf{if}}\ \langle s_1[1], ..., s_1[n]\rangle = \langle\ s_2[i], ..., s_2[j]\rangle\ \wedge\ (n-1 = j-i)$

$\qquad\qquad\qquad\qquad \underline{\textbf{then}}\ y := \textbf{true}\ \underline{\textbf{else}}\ y := \textbf{false}\ \underline{\textbf{fi}}$

$x := \underline{\textbf{pop}}(t)\ =_{\text{DF}}\ x := t[1] \wedge t := \langle t[2], ..., t[n]\rangle$

$x := \underline{\textbf{top}}(t)\ =_{\text{DF}}\ x := s[1]$

$y := \underline{\textbf{depth}}(p)\ =_{\text{DF}}\ y := \ell(t)$

$y := \underline{\textbf{user-adt-funct-call}}\ \mathcal{X} \equiv\ S.\ =_{\text{DF}}\ y\ :=\ (\mu\mathcal{X}.F)$

B.4 Specification Statements

$\underline{\textbf{entity}}\ x\ =\ e\ \underline{\textbf{end}}\ =_{\text{DF}}$

$\qquad \underline{\textbf{entity}}\ x/\langle\rangle.\underline{\textbf{entity}}\ x\ =\ e;\ \ \langle\rangle/\underline{\textbf{entity}}\ e.\underline{\textbf{true}}$

$\underline{\textbf{relate}}\ x\ \underline{\textbf{to}}\ y\ =_{\text{DF}}\ \langle x,\ y\rangle \in \mathcal{R}\quad * \ \mathcal{R}\ \textit{is a relation.}$

$\underline{\textbf{relationship}}\ \textit{ENTITY}\ x\ \underline{\textbf{has}}\ \text{d}1\ r\ \underline{\textbf{relation}}\ \underline{\textbf{with}}\ d2\ \underline{\textbf{entity}}y$

$=_{\text{DF}}\ \langle x\ d1\ r\ d2\ y\ \rangle \in \mathcal{P}\quad * \ \mathcal{P}\ \textit{is a five-tuple.}$

B.5 Declarations

$$\underline{\textbf{user-adt}}\ adt(X)\ \underline{\textbf{user-adt-funct}}\ adtfunct\ \mathcal{Y}\ \equiv\ S.\ \underline{\textbf{end}};$$

$$=_{\text{DF}}\ adt(X)[\underline{\textbf{funct}}\ \mathcal{Y} \equiv S./\mathcal{Y}]$$

$$\underline{\textbf{user-adt}}\ adt(X)\ \underline{\textbf{user-adt-proc}}\ adtproc\ \mathcal{Z}\ \equiv\ S.\ \underline{\textbf{end}}$$

$$=_{\text{DF}}\ adt(X)[\underline{\textbf{proc}}\ \mathcal{Z} \equiv S./\mathcal{Z}]$$

Appendix C

Syntax and Semantics of Meta-WSL Extension

C.1 Notations

Basic definitions and functions are presented here in order to read the specification of Meta-WSL extension. These notations are in WSL [165].

1. Programs

Definitions:

$$\begin{aligned}
\mathit{Program} \;&\equiv \mathit{Items}^*\\
&\quad(\mathit{set\ of\ finite\ sequences\ of\ elements\ of\ Items} = \bigcup_{n<\omega} \mathit{Items}^n\\
&\quad\mathit{where\ Items}^1 = \mathit{Items}, \mathit{Items}^{n+1} = \mathit{Items} \times \mathit{Items}^n)\\
&\quad\mathit{Items} \;\equiv \{\langle \mathit{table}, \mathit{comment}, \langle t_1, t_2, c\rangle\rangle \mathbin{++} \mathit{components} \mid (\mathit{table} \in \mathit{Tables}) \wedge\\
&\qquad\qquad (\mathit{comment} \in \mathit{Comments}) \wedge (t_1 \in \mathit{Specific\text{-}types}) \wedge\\
&\qquad\qquad (t_2 \in \mathit{Gen\text{-}types}) \wedge (\mathit{components} \in \mathit{Items}^*)\}
\end{aligned}$$

Items is a set of quadruples. Nth component is a sequence of items.

$\mathit{Names} \;\equiv \mathit{name\ of\ item}$

$\mathit{Numbers} \;\equiv \mathbb{Z}$

$Strings \;\equiv\; set\ of\ ASCII\ string$

$Comments \;\equiv\; any\ combination\ of\ ASCII\ strings$

$Content \;\equiv\; Names \cup Numbers \cup Strings \cup \{\mathbf{true}, \mathbf{false}\}$

$Gen\text{-}types \;\equiv\; \{\texttt{Thing}, \texttt{Statement}, \texttt{Expression}, \texttt{Variable}, \ldots\}$

$Specific\text{-}type \;\equiv\; \{\texttt{Name}, \texttt{Var}, \texttt{Skip}, \texttt{Call}, \ldots\}$

Functions:

$Gen\text{-}type : Items \longrightarrow Gen\text{-}types$

$Specific\text{-}type : Items \longrightarrow Specific\text{-}types$

$Size : Items \longrightarrow \mathbb{N}$

$Get\text{-}n : Items \times \mathbb{N} \nrightarrow Items$

$Contents : Items \nrightarrow Content$

$Val\text{-}N(item, n) \;=_{\text{DF}}\; Contents(Get\text{-}n(item, n))$

2. Database Table

Definition:

$Tables \;\equiv \mathbb{P}((Names \cup Strings) \times Contents)$

An element of Tables is a set of pairs — a function.

Functions:

$\textit{Get-table} : \textit{Items} \rightarrow \textit{Tables}$

$\textit{Set-table} : \textit{Items} \times \textit{Tables} \rightarrow \textit{Items}$

$$\textit{Add-to-table} : \textit{Items} \times \textit{Key} \times \textit{Data}$$
$$\rightarrow \textit{Items} \approx \textit{Set-table}(\textit{item}, (\textit{get-table}(\textit{item}) \times \textit{Key}) \oplus \textit{Data})$$

$\textit{Get-from-table}(\textit{item}, \textit{key}) \equiv \textit{get-table}(\textit{item}) \times \textit{key}$

The Meta-WSL extension consists of three parts of construct for implementing the Program Structure Database, the General Simplifier and the Metric Facility.

Please note that all Meta-WSL construct defined in the appendix are in square brackets, e.g., [*Variables*]. Other entries are all assistant definitions.

C.2 Specification of Program Structure Database Queries

The following database queries are specified in this section: [Variables], [Used], [Assigned], [Used-Only], [Assd-Only], [Assn-to-self], [Regular?], [Regular-system?], [Depth], [Primitive?], [Terminal-Value], [Terminal?], [Proper?], [Reducible?], [Improper?], [Dummy?], [Calls], [Calls-terminal?], [Calls-term-sys?], [Statements] and [Size]. More information can be found in "Catalogue of Program Transformations" [154].

C.2.1 Query: [Variables]

Definition:

```
funct [Variables](item) ≡
    if Size(item) = 0
      then if Gen-type(item) ∈ {Variable, Assd-var }
             then {Contents(item)}
              else ∅ fi
      else
```

```
            $\bigcup_{i=1}^{Size(item)} [Variables](Get\text{-}n(item, i))$
    fi
```

Comment: The query function returns all variables occurring ([Variables] : Items → ℙ(Names)).

C.2.2 Query: [Assigned]

Definition:

```
funct $[Assigned](item)$ $\equiv$
    if $Size(item) = 0$
      then if $(Gen\text{-}type(item) =$ Assd-var )
              $\wedge$ $(Specific\text{-}type(item) =$ Variable)
              then $\{Contents(item)\}$
               else $\emptyset$ fi
      else
            $\bigcup_{i=1}^{Size(item)} [Assigned](Get\text{-}n(item, i))$
    fi
```

Comment: The query function returns those variables assigned in the assignment statements.

C.2.3 Query: [Used]

Definition:

```
funct $[Used](item)$ $\equiv$
    if $Size(item) = 0$
      then if $Specific\text{-}type(item) =$ Variable
              then $Contents(item)$
               else $\emptyset$ fi
      else
            $\bigcup_{i=1}^{Size(item)} [Used](Get\text{-}n(item, i))$
    fi
```

Comment: The query function returns those variables used in the assignment statements and variables used in non-assignment statements.

C.2.4 Query: [Used-Only]

Definition:

funct $[\mathit{Used\text{-}Only}](\mathit{item}) \equiv [\mathit{Used}](\mathit{item}) \backslash [\mathit{Assigned}](\mathit{item})$

Comment: The query function returns those variables used (refer to [Used]) but not assigned (refer to [Assigned]).

C.2.5 Query: [Assd-Only]

Definition:

funct $[\mathit{Assd\text{-}Only}](\mathit{item}) \equiv [\mathit{Assigned}](\mathit{item}) \backslash [\mathit{Used}](\mathit{item})$

Comment: The query function returns those variables assigned but not used.

C.2.6 Query: [Assn-to-self]

Definition:

```
funct $\mathit{Purely\text{-}used}(\mathit{item}) \equiv$
    if $(\mathit{Size}(\mathit{item}) = 0) \vee (\mathit{Gen\text{-}type}(\mathit{item}) = \mathit{Assignment})$
      then if $\mathit{Gen\text{-}type}(\mathit{item}) = \mathit{Assignment}$
             then $[\mathit{Used\text{-}Only}](\mathit{item})$
             else $\emptyset$ fi
      else
          $\bigcup_{i=1}^{\mathit{Size}(\mathit{item})} \mathit{Purely\text{-}used}(\mathit{Get\text{-}n}(\mathit{item}, i))$
    fi
```

Comment: Purely-used returns those variables used but not assigned to themselves.

Definition:

```
funct $[\mathit{Assn\text{-}to\text{-}self}](\mathit{item}) \equiv [\mathit{Assigned}](\mathit{item}) \backslash \mathit{Purely\text{-}used}(\mathit{item})$
```

Comment: The query function returns those variables only used in assignments to itself or only assigned to.

C.2.7 Query: [Depth]

Definition:

```
funct $\mathit{If\text{-}floop}(\mathit{item}) \equiv$ if $\mathit{Specific\text{-}type}(\mathit{item}) = \texttt{Floop}$
                                then 1
                                 else 0 fi
```

Comment: It returns 1 if the item is a Floop, otherwise 0.

Definition:

```
funct $[\mathit{Depth}]\ (\mathit{item}, \mathit{posn\text{-}offset}) \equiv +/\ \mathit{If\text{-}floop} * \mathit{Get\text{-}n}\ \#_{\mathit{item}}\ \mathit{posn\text{-}offset}$
```

Comment: The query function "Depth" returns the depth of the given program compared to the position represented by posn-offset. Depth is defined as the number of enclosing unbounded (do...od) loops (Floop).

C.2.8 Query: [Primitive?]

Definition:

```
funct $[\mathit{Primitive?}]\ (\mathit{item}) \equiv$
    if $\mathit{Specific\text{-}type}(\mathit{item}) \notin \{\texttt{Floop}, \texttt{Cond}, \texttt{D-if}\}$
      then true
       else false fi
```

Comment: The query function "Primitive?" returns "true" or "false" depending on whether or not the statement given is primitive. A primitive statement to be either an exit statement or an assignment or an assertion.

C.2.9 Query: [Terminal-value]

Definition:

```
funct [Terminal-value] (item, posn-offset) ≡
    ⌈floop-layers := +/If-floop * Get-n $\#_{item}$ posn-offset;
     exit-value :=
       if Specific-type(Get-from-posn(item, posn-offset)) = Exit
         then Contents(Get-from-posn(item, posn-offset), 1)
              if Specific-type(Get-from-posn(item, posn-offset)) = Call
                then Contents(Get-from-posn(item, posn-offset), 2)
                 else 0 fi fi;
     terminal-value := exit-value − floop-layers;
     terminal-value⌋
```

Comment: The query function "Terminal-Value" returns the terminal value of the given program compared to the comparative position represented by posn-offset. The terminal value is the difference of the capacity for jumping out loops (exiting value, e.g., k in "Exit k") and the number of loops.

C.2.10 Query: [Regular?]

Definition: $$a \uplus b =_{\mathrm{DF}} \begin{cases} (a - \{0\}) \cup b & \text{if } 0 \in a \\ a & \text{otherwise} \end{cases}$$

Comment: It is a conditional union operation.

Definition:

```
funct Tv-list (item) ≡
    if Specific-type(item) = Exit
      then {Val-N(item, 1)}
    elsif Specific-type(item) = Call
        then {Val-N(item, 2)}
    elsif Specific-type(item) ∈ {Cond, D-if}
        then
              $\biguplus_{i=1}^{Size(item)}$ Tv-list(Get-n(item, i))
    elsif $(Specific\text{-}type(item) \notin \{\texttt{Statements}, \texttt{Floop}, \texttt{Guarded}\}) \vee$
          $(Size(item) = 0)$
          then 0
          else
               $\biguplus_{i=1}^{Size(item)} \; Tv\text{-}list(Get\text{-}n(item, i))$
    fi
```

Comment: It returns a set of terminal values of the given item.

Definition:

```
funct $Tv\text{-}list\text{-}for\text{-}calls\ (item) \equiv$
    if $Specific\text{-}type(item) = \texttt{Exit}$
      then $\{Val\text{-}N(item, 1)\}$
    elsif $Specific\text{-}type(item) = \texttt{Call}$
          then $\{*big\text{-}num*\}$
    elsif $Specific\text{-}type(item) \in \{\texttt{Cond}, \texttt{D-if}\}$
          then
               $\bigcup_{i=1}^{Size(item)} \; Tv\text{-}list(Get\text{-}n(item, i))$
    elsif $(Specific\text{-}type(item) \notin \{\texttt{Statements}, \texttt{Floop}, \texttt{Guarded}\}) \vee$
          $(Size(item) = 0)$
          then $\{0\}$
          else
               $\biguplus_{i=1}^{Size(item)} \; Tv\text{-}list(Get\text{-}n(item, i))$
    fi
```

Comment: The function is the same as Tv-list, apart from that the terminal value caused by Call statement is marked $*big\text{—}num*$.

Definition:

```
funct $[Regular?]\ (item) \equiv Tv\text{-}list(item) = \{*big\text{-}num*\}$
```

Comment: This query function returns "true" or "false" depending on whether or not the program item (or called "action") is regular or not. For example, an action is regular if every execution of the action leads to an action call.

C.2.11 Query: [Regular-system?]

Definition:

```
funct [Regular-system?] (item) ≡
    if Specific-type(item) = Action
      then $\forall i, 2 \leq i \leq Size(item) \bullet [Regular?](Get\text{-}n(item, i))$
       else false
    fi
```

Comment: This query function returns "true" or "false" depending on whether or not the given action system is regular or not. An action system is regular if every action in the system is regular.

C.2.12 Query: [Terminal?]

Definition:

$$Get\text{-}from\text{-}posn(item, posn) =_{\text{DF}} \begin{cases} item & \text{if } posn = \langle\rangle \\ Get\text{-}from\text{-}posn(Get\text{-}n(item, posn[1]), posn[2, \ldots]) & \text{otherwise} \end{cases}$$

Comment: It returns the sub-item in the position offset of the given item.

Definition:

```
funct Sub-items (item, posn-offset) ≡

    $\langle 0 \leq i \leq l(posn\text{-}offset) | Get\text{-}from\text{-}posn(item, posn\text{-}offset[i])\rangle$
```

Comment: The function returns all sub-items from the item going down to the sub-item at position offset, e.g., sub-item (item, '$\langle 1, 2, 3, 4\rangle$) = { item, Get-from-posn(item,

'⟨1⟩), Get-from-posn(item, '⟨1,2⟩), Get-from-posn(item, '⟨1,2,3⟩), Get-from-posn(item, '⟨1,2,3,4⟩) }.

Definition:

```
funct terminal-component? (item, n) ≡
    if (Specific-type(Get-n(item, n)) ∈ {Cond, D-if}) ∨
      ((Specific-type(Get-n(item, n)) ≠ Floop) ∧ (Size(item) = n)) ∨ (n = 0)
      then true
       else false fi
```

Comment: It returns "true" if the nth element of the item is a terminal component, otherwise "false".

Definition:

```
funct Terminal-posn? (item, posn-offset) ≡
    ∧ / (terminal-component? * zip(sub-items(item, posn-offset), {0}∪
                                            posn-offset))
```

Comment: It returns "true" if the sub-item at position offset of the item is in a terminal position, otherwise "false".

Definition:

```
funct Posn-prefix (item, key, ⟨p1, p2, ..., pn⟩) ≡
    ⟨p1, p2, ..., pi⟩ | μ i • Specific-type(get-from-posn(item, posni)) = key
```

Comment: It returns a sub-position-offset of the position offset. This "i" is the smallest.

Definition:

funct *Sub-posn* $(item, j, \langle p_1, p_2, \ldots, p_n\rangle) \equiv \exists j : j < n | \{p_1, p_2, \ldots, p_j\}$

Comment: It returns a sub-position-offset of the position offset.

Definition:

```
funct Outmost-floop-terminal? (item, posn-offset) ≡
      if $Floop \notin Specific\text{-}type * (Sub\text{-}items(item, posn\text{-}offset)$
        then false
         else Terminal-posn?(item, Posn-prefix(item, Floop, posn-offset))
      fi
```

Comment: It returns "true" if the outmost floops is terminal, other "false".

Definition:

```
funct [Terminal?] (item, posn-offset) ≡
      if $i \in \langle 1, \ldots, length(posn\text{-}offset)\rangle \bullet (j \in \langle 1, \ldots, p_i - 1\rangle \bullet$
        $(\{0\} \in / Tv\text{-}list * (Get\text{-}from\text{-}posn(item, sub\text{-}posn(posn\text{-}offset, i-1) \cup$
                  $j)))) = \mathbf{true}$
        then
              if Terminal-posn? (item, posn-offset)
                then true
              elsif $(Specific\text{-}type(Get\text{-}from\text{-}posn(item, posn\text{-}offset)) =$ Exit$) \wedge$
                    $(Val\text{-}N(Get\text{-}from\text{-}posn(item, posn\text{-}offset), 1) >$
                    $[Depth](item, posn\text{-}offset)) \vee$
                    $(Specific\text{-}type(Get\text{-}from\text{-}posn(item, posn\text{-}offset)) =$ Call$) \wedge$
                    $(Val\text{-}N(Get\text{-}from\text{-}posn(item, posn\text{-}offset), 2) >$
                    $[Depth](item, posn\text{-}offset))$
                    then true
              elsif $((Specific\text{-}type(Get\text{-}from\text{-}posn(item, posn\text{-}offset)) =$ Exit$) \wedge$
                    $(Val\text{-}N(Get\text{-}from\text{-}posn(item, posn\text{-}offset), 1) =$
                    $[Depth](item, posn\text{-}offset)) \vee$
                    $(Specific\text{-}type(Get\text{-}from\text{-}posn(item, posn\text{-}offset)) =$ Call$) \wedge$
                    $(Val\text{-}N(Get\text{-}from\text{-}posn(item, posn\text{-}offset), 2) =$
                    $[Depth](item, posn\text{-}offset))) \wedge$
                    $(Outmost\text{-}floop\text{-}terminal?\ (item, posn\text{-}offset))$
                    then true
                     else false
```

<u>**fi**</u>

Comment: The query function "Terminal?" returns "true" or "false" depending on whether or not the statement given is terminal. I.e., for any statements T, S (where T is primitive) and integers n, d, the predicate ts(n, T, S, d) is interpreted "the nth occurrence of T in S is a terminal statement of S which leave d enclosing loops".

C.2.13 Query: [Proper?]

Definition:

$Posns_0 = \{\langle\rangle\}$
$Posns_{n+1} = \{P \mathbin{++} \langle N\rangle | P \in Posns_n \wedge 1 \leq N \leq Size(Get\text{-}from\text{-}posn(item, P))\}$
$\underline{\textbf{funct}}\ Posns(item) = \bigcup_{n<\omega} Posns_n$

Comment: It returns all position offsets in the item.

Definition:

$Posns\text{-}of\text{-}primitive\text{-}statements(item)$
$\equiv \{P \in Posns(item) | [Primitive?](Get\text{-}from\text{-}posn(item, P))\}$

Comment: It returns all positions of the primitive statements in the item.

Definition:

$\underline{\textbf{funct}}\ Terminal\text{-}Posns(item) \equiv$
$\{P \in Posns(item) | Terminal\text{-}posn?(Get\text{-}from\text{-}posn(item, P))\}$

Comment: It returns all terminal positions in the item.

Definition:

$\underline{\textbf{funct}}\ Terminal\text{-}statements(item) \equiv$
$\{P \in Posns(item) | [Terminal?](Get\text{-}from\text{-}posn(item, P))\}$

Comment: It returns all positions of the terminal statements in the item.

Definition:

```
funct [Proper?] (item) ≡
        $\wedge$ / $(= 0) * ([\mathit{Terminal\text{-}value}](\mathit{item}, \bullet) * \langle \mathit{Terminal\text{-}statements}(\mathit{item})\rangle)$
```

Comment: The query function "Proper?" returns "true" or "false" depending on whether or not the statement at the current point in item is proper. The statement S is a proper sequence if every terminal statement of S has terminal value zero.

C.2.14 Query: [Reducible?]

Definition:

```
funct If-reducible(item, posn-offset) ≡
    if (Terminal-posn?(item, posn-offset)) $\vee$
      (([Depth](item, posn-offset) $> 0$) $\wedge$
       (Outmost-floop-terminal?(item, posn-offset)))
      then true
       else false
    fi
```

Comment: It returns "true" if the sub-item in the position offset of the item is reducible, otherwise "false".

Definition:

```
funct [Reducible?] (item) ≡
        $\wedge$ / If-reducible?(item, $\bullet$)$*$
        $\{P \in$ Posns(item)$|$([Terminal?](Get-from-posn(item, P))) $\wedge$
        ([Terminal-value](Get-from-posn(item, P)) $= 1$)$\}$
```

Comment: The query function "Reducible?" returns "true" or "false" depending on whether or not a statement is reducible. I.e., The statement S is reducible if replacing any terminal statement EXIT(k), which has terminal value one, by EXIT(k-1) gives a terminal statement of S.

C.2.15 Query: [Improper?]

Definition:

<u>**funct**</u> $[\mathit{Improper?}]\ (\mathit{item})\ \equiv$

$$\wedge\ /\ (>0) * ([\mathit{Terminal\text{-}value}](\mathit{item}, \bullet) * \langle \mathit{Terminal\text{-}statements}(\mathit{item})\rangle)$$

Comment: The query function "Improper" returns "true" if all terminal statements of S have terminal value greater than zero, otherwise "false".

C.2.16 Query: [Dummy?]

Definition:

<u>**funct**</u> $[\mathit{Dummy?}]\ (\mathit{item})\ \equiv\ [\mathit{Reducible?}]\ (\mathit{item})\ \wedge\ [\mathit{Improper?}](\mathit{item})$

Comment: The query function "Dummy?" returns "true" if both a statement S is reducible and all terminal statements of S have terminal value greater than zero, otherwise "false".

C.2.17 Query: [Calls]

Definition:

<u>**funct**</u> $\mathit{Make\text{-}pair}(\mathit{thing})\ \equiv\ (\mathit{thing}, 1)$

Comment:

Definition:

$S := \{(x_1, l_1), (x_2, l_2), \ldots, (x_k, l_k)\}$

<u>**funct**</u> $\mathit{Summarize}(S)\ \equiv\ \{(x, n| \exists\, m \bullet (x, m) \in S\ \wedge\ n = +/\{m|(x, m) \in S\})\}$

Comment: For example, Summarize { (a 3), (a 4), (b 3) } = { (a 7), (b 3) }

Definition:

```
funct $[Calls]\ (item) \equiv$
    $\ulcorner C_1 := Get\text{-}n(item, \bullet)*$
            $\{P \in Posns(item) | [Specific\text{-}type](Get\text{-}n(item, P)) = \texttt{Call}\};$
      $C_2 := Val\text{-}N(\bullet, 1) * C_1;$
      $C_3 := Make\text{-}pair * C_2;$
      $C_4 := Summarize(C_3);$
      $C_4 \lrcorner$
```

Comment: The query function returns all the action call names and how many times they are to be called in the given program item.

C.2.18 Query: [Statements]

Definition:

```
funct $[Statements]\ (item) \equiv$
    $Specific\text{-}type * \{P \in Posns(item) | [Gen\text{-}type](Get\text{-}n(item, P)) = \texttt{Statement}\}$
```

Comment: The query function returns all the statement names in the given program item.

C.2.19 Query: [Calls-terminal?]

Definition:

```
funct $[Calls\text{-}terminal?]\ (item) \equiv$
    if $\{P \in Posns(item) | Specific\text{-}type(Get\text{-}n(item, P))$
      $= \texttt{Call}\} = Terminal\text{-}statements\ (item)$
      then true
       else false
    fi
```

Comment: It returns “true” if all Call statements in the item are terminal, otherwise “false”.

C.2.20 Query: [Calls-term-sys?]

Definition:

```
funct [Calls-term-sys?] (item) ≡
    if Specific-type(item) = Actions
      then $\wedge$ / [Calls-terminal?] $* \{i \in 2, \ldots, Size(item) | Get\text{-}n(item, i)\}$
       else false
    fi
```

Comment: It returns "true" if all Call statements in an action system in the given item are terminal, otherwise "false".

C.2.21 Query: [Size]

Definition:

```
funct [Size] (item) ≡
    $length\langle P \in Posns(item) | Size(Get\text{-}n(item, P)) = 0\rangle$
```

Comment: It returns the number of leaf nodes in the given item.

C.3 Specification of General Simplifier

The General Simplifier simplifies a mathematical or logical expression, or to "prove" the equivalence or implication of two expressions. Mathematical and logical operations defined in the system are: +, −, *, /, **, Min, Max, Div, Mod, =, >, <, <>, Not, And, Or, etc. and these definitions are in normal mathematical and logical sense.

Definition:

<u>funct</u> [*Simplify*] (*expression*) ≡ *simplified-expression*

Comment: The "Expression" can be any symbolic algebraic expression. The function returns the expression in its simplest form.

<u>funct</u> [*Equivalent?*] (*expression1, expression2*) ≡ T $\vee$ Nil

Comment: The function returns **T** or **Nil** according to whether the two expression are equivalent.

<u>funct</u> [*Implies?*] (*expression1*, *expression2*) $\equiv$ **T** $\vee$ **Nil**

Comment: The function returns **T** or **Nil** according to whether the first expression implies the second one.

C.4 Specification of Metric Facility

The six metrics have been defined.

C.4.1 Preliminary Definitions For Metrics Functions:

```
funct If-predicate(item) ≡ if (Specific-type(item) = Cond) ∨
                              (Specific-type(item) = D-if) ∨
                              (Specific-type(item) = D-do) ∨
                              (Specific-type(item) = While) ∨
                              (Specific-type(item) = For) ∨
                              (Specific-type(item) = If)
                              then 1
                               else 0 fi
```

Comment: It returns 1 if the item is a predicate, otherwise 0.

```
funct Struct-index(item) ≡ case Specific-type (item) of
                                Abort :  10
                                Assert : −5
                                Assign :  1
                                ......
                                Where :  1
                                While :  15
                           end
```

Comment: It returns the structural complexity index of the item.

```
funct If-node(item) ≡ if Gen-type(item) = Expression ∨ Statement ∨ ...
                                          Thing ∨ Variable
                      then 1
                       else 0 fi
```

Comment: It returns 1 if the item is a node in the abstract syntax tree of the program, otherwise 0.

```
funct If-statement(item) ≡ if Gen-type(item) = Statement
                            then 1
                             else 0 fi
```

Comment: It returns 1 if the item is a statement, otherwise 0.

```
funct If-edge(item) ≡ if Gen-type(item) = Actions
                       then −1
                        else If-statement(item) fi
```

Comment: It returns 1 if the item is a statement, otherwise 0.

```
funct If-loop(item) ≡ if Specific-type(item) = D-do ∨ Floop ∨ ...
                                               For ∨ While
                       then 1
                        else 0 fi
```

Comment: It returns 1 if the item is the key word of a loop, otherwise 0.

```
funct If-data(item) ≡ if (Gen-type(item) = Assd-var ) ∨
                         (Gen-type(item) = Variable)
                         then 1
                          else 0 fi
```

Comment: It returns 1 if the item is a variable, otherwise 0.

C.4.2 Metric: [MCCABE]

Definition:

<u>funct</u> $[\mathit{MCCABE}]\ (\mathit{item}) \equiv$

$$1 + (+/\mathit{If\text{-}predicate} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\})$$

Comment: McCabe Complexity (MCCABE) — the number of linearly independent circuits in a program flowgraph [116]. It is calculated as the number of predicates plus one.

C.4.3 Metric: [Structural]

Definition:

<u>funct</u> $[\mathit{STRUCT}]\ (\mathit{item}) \equiv$

$$+/\mathit{Struct\text{-}index} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\}$$

Comment: Structural (STRUCT) — the sum of the weights of every construct in the program. The weight of each WSL construct is defined subjectively according to experience gained by REFORM researchers and users. A loop, for example, is more difficult to understand than an assignment statement, so a loop statement is given a bigger weight than an assignment statement.

C.4.4 Metric: [LOC]

Definition:

<u>funct</u> $[\mathit{LOC}]\ (\mathit{item}) \equiv$

$$+/\mathit{If\text{-}statement} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\}$$

Comment: Lines Of Code(I) (LOC) — the number of statements.

C.4.5 Metric: [LOC2]

Definition:

<u>funct</u> $[\mathit{LOC2}]\ (\mathit{item}) \equiv$

$$+/\mathit{If\text{-}node} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\}$$

Comment: Lines Of Code(I) (LOC) — the number of statements. Lines Of Code(II) (LOC2) — the number of nodes in the abstract syntax tree. This reflects the overall size of the program.

C.4.6 Metric: [CFDF]

Definition:

<u>**funct**</u> [*CFDF*] (*item*) $\equiv$

$$(+/\mathit{If\text{-}edge} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\})+$$
$$(+/\mathit{If\text{-}data} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\})$$

Comment: Control-Flow and Data-Flow Complexity (CFDF) — the number of edges in the flowgraph plus the number of times that variables are used (defined and referred). It is a modification of the measure defined by Oviedo [175].

C.4.7 Metric: [BL]

Definition:

<u>**funct**</u> [*BL*] (*item*) $\equiv$

$$(+/\mathit{If\text{-}predicate} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\})+$$
$$(+/\mathit{If\text{-}floop} * \{P \in \mathit{Posns}(\mathit{item}) | \mathit{Get\text{-}n}(\mathit{item}, P)\})$$

Comment: Branch-Loop Complexity (BL) — the number of non-loop predicates plus the number of loops. It is a modification of the measure defined by Moawad and Hassan [175]. The measure is sensitive both to branches and to loops.

Appendix D

Program Transformations for Data Abstraction

D.1 Deriving Records

- Split-Record-Into-Subrecords
- Join-Records
- Absorb-Single-Record-In-Record
- Variable->Record
- One-level-Record->Variable

D.2 From Records to Data in Design Level

- Record-Of-File->Entity
- Record-Of-Top-Level->Entity
- Record-Without-Subrecord->Entity
- ArrayOfRecord->Entities-And-Relationship

D.3 From Code Level Data Operation to Data Relations

- Model-OpenFile-By-QueueOP
- Model-CloseFile-By-QueueOP
- Model-EOF?-By-QueueOP
- Model-ReadFile-By-QueueOP
- Model-WriteFile-By-QueueOP
- Simplify-Queue-Initialise
- Simplify-Queue-Append
- Simplify-Queue-Remove-First
- Simply-Set-Initialise
- Simplify-Set-Insert
- Simplify-Set-Delete
- Assert-EmptySet-From-SetInitialise
- Merge-Push-Pop
- Assert-Hd-From-Push
- Assert-EmptyStack-From-Push
- Assert-EmptyStack-From-StackInitialise
- Assert-StackDepth-From-StackInitialise
- Simplify-StackInitialise
- Simplify-Top
- Simplify-Depth
- Simplify-Push
- Simplify-Pop

D.4 Abstraction from Code

- Find-Data-Object
- Recognise-ADT-Starting-With-Proc
- Recognise-ADT-Starting-With-Funct
- Recognise-ADT-Starting-With-Variable
- Intance->Entity
- Assign->Relate
- Remove-Useless-Relate
- Abstract-If-Then-Else-Statement
- Abstract-If-Then-Statement
- Eliminate-Irrelevant-Statement

D.5 From User-Defined Data Type to Data Design

- ADT->Entity

D.6 Deriving Data Design from Data and Code

- Derive-Relationship-From-Foreign-Key
- Derive-Entity-and-Relationship-From-Code1
- Derive-Entity-and-Relationship-From-Code2
- Derive-Entity-and-Relationship-From-Code3

D.7 Manipulating Program Items

- Swap-With-Next-Record
- Create-Segment-From-Var
- Create-Paragraph-From-Var
- Create-Segment-And-Paragraph-From-Var
- Create-Var-From-Segment
- Create-Paragraph-From-Segment
- Remove-Dummy-VarStruct
- Remove-Dummy-Segment
- Remove-Dummy-Paragraph
- Swap-With-Next-Entity
- Swap-With-Next-Relationship
- Rename-Relationship
- Adjust-Relationship-Degree
- Normalise-Entity-Relationship-Diagram

References

[1] ANSI, "Report on Study Group on Database Management Systems", Interim Report, FDT, 1975.

[2] ANSI, *Standard 729*, IEEE Standard Glossary of Software Engineering Terminology, 1983.

[3] Abel, P., *COBOL Programming Language — A Structured Approach*, Prentice-Hall International, Inc., London, 1989.

[4] Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs*, The MIT Press, McGraw-Hill Book Company, 1985.

[5] Abrial, J. R., Gardiner, P. H., Morgan, C. C. and Spivey, J. M., "A Formal Approach to Large Software Construction", Technical Report, Programming Research Group, Oxford, 1988.

[6] Agrawal, H. and Horgan, J. R., "Dynamic Program Slicing", ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, New York , June 1990.

[7] Antonini, P., Benedusi, P., Cantone, G. and Cimitile, A., "Maintenance and Reverse Engineering: Low-level Design Documents Production and Improvement", IEEE Conference on Software Maintenance-1987, Austin, Texas, 1987.

[8] Appleby, D., *Programming Languages: Paradigm and Practice*, McGraw-Hill Book Company, New York, 1991.

[9] Arango, G., Baxter, I., Freeman, P. and Pidgeon, C., "TMM: Software Maintenance by Transformation", IEEE Software, May, 1986.

[10] Arsac, J., *In Foundations of Programming*, Academic Press, Inc., London, 1985.

[11] Ashworth, C. and Goodland, M., *SSADM: A Practical Approach*, McGraw-Hill Book Company, London, 1990.

[12] Bachman, R., *A CASE for Reverse Engineering*, Cahners Publishing Company, July, 1988, reprinted from DATAMATION.

[13] Back, R. J. R., "Correctness Preserving Program Refinements", Mathematical Centre Tracts No. 131, Mathematisch Centrum, 1980.

[14] Balzer, R., "Transformational Implementation: An Example", *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 1, pp. 3–14 (January 1981).

[15] Balzer, R., "A 15 Year Prospective on Automatic Programming", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, pp. 1257–1267 (November 1985).

[16] Balzer, R., Goldman, N. and Wile, D., "On the Transformational Implementation Approach to Programming", The 2nd International Conference on Software Engineering, San Francisco, California, 1976.

[17] Balzer, R. and Sartout, W., "On the Inevitable Intertwining of Specification and Implementation", in *Software Specification Techniques*, Addison-Wesley Publishing Company, 1986.

[18] Bauer, F. L., Berghammer, R. and et. al., "The Munich Project CIP - The Wide Spectrum Language CIP-L ", in *Lecture Notes in Computer Science 292*, Springer-Verlag, New York, 1985.

[19] Bauer, F. L., Moller, B. B., Partsch, H. and Pepper, P., "Formal Program Construction by Transformation — Computer-Aided, Intuition-Guided Programming", *IEEE Transactions on Software Engineering*, Vol. 15, No. 2, pp. 165–180 (February 1989).

[20] Bennett, K. H., "The Software Maintenance of Large Software Systems: Management Method and Tools", Technical Report, Durham University, 1989.

[21] Bennett, K. H., "An Overview of Maintenance and Reverse Engineering", in *The REDO Compendium*, John Wiley & Sons, Inc., Chichester, 1993.

[22] Bennett, K. H., Cornelius, B. J., Munro, M. and Robson, D. J., "Software Maintenance", in *Software Engineer's Reference Book*, Butterworth Heinemann, 1991, pp. 20/1-20/18 .

[23] Bennett, K. H., Denier, J. and Estublier, J., "Environments for Software Maintenance", Technical Report, Durham University, 1989.

[24] Berg, H. K., Boebert, W. E., Franta, W. R. and Moher, T. G., *Formal Methods of Program Verification and Specification*, Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1982.

[25] Biggerstaff, T. J., "Design Recovery for Maintenance and Reuse", *IEEE Computer*, Vol. 22, No. 7, pp. 36–49 (July 1989).

[26] Bird, R., "A Calculus of Functions for Program Derivation", Technical Monograph PRG-64, Oxford University, 1987.

[27] Bird, R., "Lectures on Constructive Functional Programming", Technical Monograph PRG-69, Oxford University , September, 1988.

[28] Bishop, J., *Data Abstraction in Programming Languages*, Addison-Wesley Publishing Company, Wokingham, 1986.

[29] Boehm, B. W., *Software Engineering Economics*, Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1981.

[30] Boyer, R. S. and Moore, J. S., *A Computational Logic*, Academic Press, Inc., New York, 1979.

[31] Boyer, R. S. and Moore, J. S., *A Computational Logic Handbook*, Academic Press, Inc., New York, 1988.

[32] Boyle, J. M., "LISP To FORTRAN - Program Transformation Applied", in *NATO ASI Series F: Computer and Systems Science*, Vol. 8 (P.Peper, Ed.), Springer-Verlag, 1984.

[33] Breuer, P., "Tackling Reverse Engineering", Technical Report (ESPRIT Project: 2487-TN-PRG-1037), Programming Research Group, Oxford University, 1990.

[34] Breuer, P., "Inverse Engineering: The First Step Backwards", Technical Report (ESPRIT Project: 2487-TN-PRG-1031), Programming Research Group, Oxford University, 1990.

[35] Breuer, P., Lano, K. and Bowen, J., "Understanding Programs through Formal Methods", Technical Report, Programming Research Group, Oxford University, 1991.

[36] Brown, A., "Specifications and Reverse Engineering", *Software Maintenance: Research and Practice*, Vol. 5 , No. 3 (1993).

[37] Bull, T., "An Introduction to the WSL Program Transformer", IEEE Conference on Software Maintenance-1990, San Diego, California, 1990.

[38] Bull, T., "Software Maintenance by Program Transformations in A Wide Spectrum Language", Ph.D. Thesis, Durham University, 1994.

[39] Burstall, R. M. and Darlington, J. A., "A Transformation System for Developing Recursive Programs", *Journal of the ACM*, Vol. 24, pp. 44–67 (1977).

[40] Burstall, R. M. and Goguen, J. A., "An Informal Introduction to Specifications Using Clear", in *Software Specification Techniques* (N. Gehani and A. D. McGettrick, Eds.), Addison-Wesley Publishing Company, 1986.

[41] Bush, E., "Reverse Engineering: What and Why", Software Maintenance Workshop, Durham, 1990.

[42] CENTRISA, "Case Study No. 1: Implementation", Technical Report (ESPRIT Project: 2487-MI-CE-1020) , CENTRISA, October, 1989.

[43] Calliss, F. W., "Inter-Module Code Analysis Techniques for Software Maintenance", Ph.D. Thesis, Durham University, 1989.

[44] Calliss, F. W., Khalil, M., Munro, M. and Ward, M., "A Knowledge-Based System for Software Maintenance", IEEE Conference on Software Maintenance-1988, Phoenix, Arizona, 1988.

[45] Canfora, G., Cimitile, A. and Munro, M., "A Reverse Engineering Method for Identifying Reusable Abstract Data Type", Durham University Technical Report, Durham, 1992.

[46] Carey, J. M., "Prototyping: Alternative Systems Development Methodology", *Information and Software Technology*, Vol. 32, No. 2 (1990).

[47] Chen, P. P., "The Entity-Relationship Model — Toward a Unified View of Data", *ACM Transaction on Database Systems*, Vol. 1, No. 1 (March 1976).

[48] Chikofsky, E. J. and Cross, J. H., "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software* , Vol. 7 , No. 1 (1990).

[49] Claybrook, B. G., "A Specification Method for Specifying Data and Procedural Abstraction", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 5 (September 1982).

[50] Cleaveland, J. C., *An Introduction to Data Types*, Addison-Wesley Publishing Company, 1991.

[51] Colbrook, A. and Smythe, C., "The Retrospective Introduction of Abstraction into Software", IEEE Conference on Software Maintenance-1989, Miami, Florida, 1989.

[52] Cross, J. H., Chikofsky, E. J. and Jr., C. H. M., "Reverse Engineering", *Advances in Computers* , Vol. 35 (1992).

[53] Cutts, G., *Structured Systems Analysis and Design Methodology*, Paradigm Publishing Company, London, 1987.

[54] DTI, The Proceedings on the Fifth Refinement Workshop, London, January 1992, sponsored by Lloyd's Register, DTI and Program Validation Limited.

[55] Date, C. J., *An Introduction to Database Systems*, Vol. I, Addison-Wesley Publishing Company, Manchester, 1986.

[56] Dershowitz, N., "Program Abstraction and Instantiation", *ACM Transactions on Programmings and Systems*, Vol. 7, No. 3 (July 1985).

[57] Desclaux, C., "Capturing Design and Maintenance Decisions with MACS", Software Maintenance Workshop, Durham, 1990.

[58] Desclaux, C. and Ribault, M., "MACS: Maintenance Assistance Capability for Software — A K.A.D.M.E. ", IEEE Conference on Software Maintenance-1991, Sorrento, Italy , 1991.

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

[60] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall International, Inc., 1972.

[61] Downs, E., Clare, P. and Coe, I., *Structured Systems Analysis and Design Method — Application and Context*, Prentice-Hall International, Inc., London, 1988.

[62] Ejiogu, L. O., *Software Engineering with Formal Metrics*, McGraw-Hill Book Company, London, 1991.

[63] Engberts, A., Kozaczynski, W. and Ning, J., "Concept Recognition-Based Program Transformation", IEEE Conference on Software Maintenance-1991, Sorrento, Italy, 1991.

[64] Engeler, E., *Formal Language: Automata and Structures*, Markham, Chicago, 1968.

[65] Fateman, R. J., "Macsyma's General Simplifier: Philosophy and Operation ", 1979 MACSYMA User's Conference, Washington D.C., June 1979.

[66] Fateman, R. J., "A Review of Macsyma ", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 1, pp. 133–145 (March 1989).

[67] Feather, M. S., "A Survey and Classification of Some Program Transformation Techniques", in *Program Specification and Transformation*, Elsevier Science Publishers, The Netherlands, 1987, pp. 165–195.

[68] Feather, M. S., "A System for Assisting Program Transformation", ACM Transactions on Programming Language Systems , January, 1982.

[69] Federal Information Processing Standards, "Guidelines on Software Maintenance", U.S. Department Commerce/National Bureau of Standards, Standard FIPS PUB 106, June, 1984.

[70] Fickas, S. F., "Automating the Transformational Development of Software", *IEEE Transactions on Software Engineering*, Vol. SE-11 , No. 11 (November 1985).

[71] Fisher, A. S., *CASE — Using Software Development Tools*, John Wiley & Sons, Inc., 1988.

[72] Fraser, M. D., Kumer, K. and Vaishnavi, V. K., "Informal and Formal Requirements Specification Languages: Bridging the Gap", *IEEE Transactions on Software Engineering*, Vol. SE-17, No. 5 (May 1991).

[73] Georges, M., "MACS: Maintenance Assistance Capability for Software", Software Maintenance Workshop, Durham University, 1990.

[74] Ghezzi, C., "Modern Non-Conventional Programming Language Concepts", in *Software Engineer's Reference Book*, Butterworth Heinemann, 1991, pp. 44/1-44/16 .

[75] Ghezzi, C. and Jazayeri, M., *Programming Language Concepts (2nd Edn.)*, John Wiley & Sons, Inc., 1987 .

[76] Gilb, T., "A Comment on the Definition of Reliability", *ACM Software Engineering Notes*, Vol. 4, No. 3 (July 1979).

[77] Gilmore, D. J., "Models of Debugging", Fifth European Conference on Cognitive Ergonomics, Urbino, Italy, September, 1990.

[78] Goguen, J. A. and Tardo, J. J., "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications", in *Software Specification Techniques*, Addison-Wesley Publishing Company, 1986.

[79] Good, D. I., London, R. L. and Bledsoe, W. W., "An Interactive Program Verification System", *IEEE Transactions on Software Engineering*, Vol. SE-1 (March 1975).

[80] Goos, G. and Hartmanis, J., *Program Construction — Lecture Notes in Computer Science*, Vol. 69, Soringer-Verlag, 1979.

[81] Gray, P. M. D., *Logic, Algebra and Databases*, Ellis Horwood Limited, Chichester, 1984.

[82] Guttag, J. V., "The Specification and Application to Programming of Abstract Data Type", Ph.D Thesis, Department of Computer Science, University of Toronto, 1975.

[83] Guttag, J. V. and Horning, J. J., "Preliminary Report on the Larch Shared Language", Technical Report (CSL-83-6), Xerox PARC, 1983.

[84] Hall, A., "Seven Myths of Formal Methods", IEEE Software, September, 1990.

[85] Hawksley, C., "Coercion in Class-Based Software Environments", Ph.D. Thesis, Keele University, 1987.

[86] Hayes, I., *Specification Case Studies*, Prentice-Hall International, Inc., 1987.

[87] Hayes, I. and Jones, C. B., "Specifications Are Not (Necessarily) Executable", Technical Report (UMCS-89-12-1), University of Manchester, 1989.

[88] Hoare, C. A. R., "Proof of A Structured Program: The Sieve of Eratosthenes", *Computer* , Vol. 14, No. 4 (1972).

[89] Hoare, C. A. R., "Notes on Data Structuring", in *Structured Programming*, Academic Press, Inc., London, 1972.

[90] Horebeek, I. V. and Lewi, J., *Algebraic Specifications in Software Engineering*, Springer-Verlag, Berlin, 1989.

[91] Hutty, R., *COBOL 85 Programming*, Macmillan Education Ltd., London, 1990.

[92] Inglis, J., *COBOL 85 for Programmers*, John Wiley & Sons, Inc., Chichester, 1989.

[93] Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall International, Inc., London, 1986.

[94] Kant, E., "Efficient Synthesis of Efficient Programs", in *Artificial Intelligence and Software Engineering*, 1986, pp. 157–188.

[95] Kemmerer, R. A., "Integrating Formal Methods into the Development Process", IEEE Software, September, 1990 .

[96] Kljaich, J., Smith, B. T. and Wojcik, A. S., "Formal Verification of Fault Tolerance Using Theorem Proving Techniques", *IEEE Transaction on Computers*, Vol. 38 , No. 3 (March 1989).

[97] Knuth, D. E., *The TEXbook*, Addison Wesley Publishing Company, Reading, Massachusetts, June, 1986.

[98] Kopetz, H., *Software Reliability*, Springer-Verlag, 1979.

[99] Lamport, L., LATEX*: A Document Preparation System*, Addison Wesley Publishing Company, Reading, Massachusetts, 1986.

[100] Lano, K. and Breuer, P., "Reverse-Engineering and Validating COBOL", Technical Report (ESPRIT Project: 2487-TN-PRG-1049), Programming Research Group, Oxford University, 1991.

[101] Lano, K. and Breuer, P. T., "From Programs to Z Specifications", Technical Report , Oxford University, 1990.

[102] Lano, K., Breuer, P. T., Haughton, H. and Estdale, J., "Reverse-Engineering COBOL Via Formal Methods", in *The REDO Handbook*, August, 1991.

[103] Lano, K. and Haughton, H., "Applying Formal Methods to Maintenance", Technical Report (ESPRIT Project: 2487-TN-PRG-1042), Programming Research Group , Oxford University, 1990.

[104] Layzell, P. J., "The Identification and Management of Latent Software Assets", *International Journal of Information Management*, Vol. 14, No. 6, pp. 427–442 (1994).

[105] Lehman, M. M., "Programs, Life Cycles, and Laws of Software Evolution", *Proc. IEEE*, Vol. 68, No. 9 (1980).

[106] Lientz, B. P. and Swanson, E. B., *Software Maintenance Management*, Addison-Wesley Publishing Company, 1980.

[107] Lindsay, P. A., Moore, R. C. and Ritchie, B., "Review of Existing Theorem Provers", Technical Report (UMCS-87-8-2), Department of Computer Science, University of Manchester, 1989.

[108] Liskov, B. and Berzins, V., "An Appraisal of Program Specifications", in *Software Specification Techniques* (N. Gehani and A. McGettrick, Eds.), Addison-Wesley Publishing Company, 1979.

[109] Liskov, B. and Guttag, J., *Abstraction and Specification in Program Development*, The MIT Press, McGraw-Hill Book Company, 1986.

[110] Liskov, B. and Zillis, S. N., "Specification Techniques for Data Abstractions", IEEE Transaction on Software Engineering, March, 1975.

[111] Manna, Z. and Waldinger, -., "A Deductive Approach to Program Synthesis", ACM Transactions on Programming Language Systems, February, 1980.

[112] Markowitz, V. and Makowsky, J. A., "Identifying Extended Entity-Relationship Object Structures in Relational Schemas", *ACM Transactions on Software Engineering*, Vol. 16, No. 8, pp. 777–790 (August 1990).

[113] Markowitz, V. and Shoshani, A., "Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach", *ACM Transactions on Database Systems*, Vol. 17, No. 3, pp. 423–464 (September 1992).

[114] Martin, J. and McClure, C., *Structured Techniques for Computing*, Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1985.

[115] Masso, S., "The Power of Algebraic Proofs", M.Sc. Thesis, Computing Laboratory, Oxford University, 1988.

[116] McCabe, T. J., "A Complexity Measure", *IEEE Transaction on Software Engineering*, Vol. SE-2, No. 4, pp. 308–320 (December 1976).

[117] McCall, J., Richards, P. and Walters, G., "Factors in Software Quality", NTIS, November, 1977.

[118] McGettrick, A. D., *Program Verification Using Ada*, Cambridge University Press, London, 1982.

[119] McMorran, M. A. and Nicholls, J. E., "Z User Manual", Technical Report, IBM (UK) Laboratories, Winchester, England, July, 1989.

[120] Meek, B. L., "Software Maintenance", in *Software Engineer's Reference Book*, Butterworth Heinemann, 1991, pp. 43/1-43/17.

[121] Morgan, C., *Programming from Specification*, Prentice-Hall International, Inc., 1990.

[122] Morgan, C., "The Specification Statement", *ACM Transaction on Programming Languages and Systems*, Vol. 10, No. 3, pp. 403–419 (July 1988).

[123] Moriconi, M. S., "A Designer/Verifier's Assistant", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4, pp. 387–401 (July 1979).

[124] Morris, J., "Type Are Not Sets", Proceedings of First ACM Symposium on Principles of Programming Languages, New York, 1973.

[125] Naur, P. and Randell, B., *Software Engineering: A Report on A Conference Sponsored by the NATO Science Committee*, NATO, 1969.

[126] Nielsen, M., Havelund, K. and Wagner, K., "The RAISE Language, Method and Tools", Formal Aspects of Computing, 1989.

[127] Noonan, R. E., "Structured Programming and Formal Specification", *IEEE Transactions on Software Engineering* , Vol. SE-1, pp. 421–425 (December 1975).

[128] Oman, P., "Maintenance Tools", IEEE Software, May, 1990.

[129] Partsch, H., *Specification and Transformation of Programs*, Springer-Verlag, London, 1990.

[130] Partsch, H. and Steinbruggen, R., "Program Transformation Systems", *Computing Surveys*, Vol. 15, No. 3, pp. 198–236 (September 1983).

[131] Pratt, T. W., *Programming Languages: Design and Implementation (2Edn.)*, Prentice-Hall International, Inc., London, 1984.

[132] Pressman, R. S., *Software Engineering — A Practitioner's Approach*, McGraw-Hall Book Company, New York, 1987.

[133] Ramamoorthy, C. V., Prakash, A., Tsai, W. and Usuda, Y., "Software Engineering: Problems and Perspectives ", IEEE Computer, October, 1984.

[134] Ratcliff, B., *Software Engineering Principles and Methods*, Blackwell Scientific Publications, Oxford, 1987.

[135] Reddy, U. S., "Transformational Derivation of Programs Using the FOCUS System", ACM Symposium on Software Development Environments, December, 1988.

[136] Robson, D. J., Bennett, K. H., Cornelius, B. J. and Munro, M., "Approaches to Program Comprehension", Journal of Systems Software, 1991.

[137] Sannella, D., "A Survey of Formal Software Development Methods", Expository Report, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.

[138] Sannella, D., "Toward Formal Development of ML Programs: Foundations and Methodology", Expository Report, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[139] Sannella, D. and Tarlecki, A., "Extended ML: An Institution-independent Framework for Formal Program Development", Expository Report, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[140] Sannella, D. and Tarlecki, A., "Toward Formal Development of Programs from Algebraic Specification: Implementation Revisited", ACTA Informatica, 1988.

[141] Shaw, M., "Abstraction Techniques in Modern Programming Languages Empirical Studies of Programming Knowledge", *IEEE Transaction on Software Engineering*, Vol. SE-10, No. 5, pp. 595–609 (September 1984).

[142] Smith, D. R. and Pressburger, T. T., *Knowledge-Based Software Development Tools*, Kestrel Institute, California, September, 1986.

[143] Sneed, H. M., *Software Engineering Management*, Ellis Horwood Limited, Chichester, 1989.

[144] Sneed, H. M. and Jandrasics, G., "Inverse Transformation of Software from Code to Specification", IEEE Conference on Software Maintenance-1988, Phoenix, Arizona, 1988.

[145] Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge", *IEEE Transaction on Software Engineering*, Vol. SE-10, No. 5, pp. 595–609 (September 1984).

[146] Sommerville, I., *Software Engineering (3rd Edn.)*, Addison-Wesley Publishing Company, Wokingham, 1989.

[147] Spivey, J. M., *Understanding Z*, Cambridge University Press, 1988.

[148] Spivey, J. M., *The Z Notation*, Prentice-Hall International, Inc., London, 1989.

[149] Swanson, E. B., "The Dimension of Maintenance", Second International Conference on Software Engineering, Los Alamitos, California, 1976.

[150] Szeto, E. and Markowitz, V., *ERDRAW: A Graphical Schema Specification Tool Reference Manual*, Lawrence Berkeley Laboratory, Berkeley, California, May, 1991.

[151] Wand, I. C., "Features of Modern Imperative Programming Languages", in *Software Engineer's Reference Book*, Butterworth Heinemann, 1991.

[152] Ward, M., "Transforming A Program into A Specification", Technical Report, Durham University, 1988.

[153] Ward, M., "Constructive Specifications and Program Transformations", Technical Report, Durham University, 1988.

[154] Ward, M., "A Catalogue of Program Transformations", Technical Report, Durham University, 1988.

[155] Ward, M., "Proving Program Refinements and Transformations", Ph.D. Thesis, Oxford University, 1989.

[156] Ward, M., Munro, M. and Calliss, F. W., "The Maintainer's Assistant", IEEE Conference on Software Maintenance-1989, Miami, Florida, 1989.

[157] Wasserman, A. I., "Software Engineering Environments", *Advances in Computers*, Vol. 22 (1983).

[158] Waters, R. C., "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 1 (January 1982).

[159] Watt, D. A., *Programming Language Syntax and Semantics*, Prentice-Hall International, Inc., 1991.

[160] Weiser, M., "Program Slicing ", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 352–257 (July 1984).

[161] Whysall, P., "Refinement", in *Software Engineer's Reference Book*, Butterworth Heinemann, 1991.

[162] Wilde, N. and Thebaut, S. M., "The Maintenance Assistant: Work in Progress", *The Journal of Systems and Software*, Vol. 9 , No. 1 (January 1989).

[163] Wirth, N., "Program Development by Stepwise Refinement", *CACM*, Vol. 14, No. 4 (1971).

[164] Yang, H., "How Does the Maintainer's Assistant Start? ", Technical Report, Durham University, 1989.

[165] Yang, H., "The Specification of Program Structure Database Queries", REFORM Research Group, Durham University, 1991.

[166] Yang, H., "Software Maintenance in Europe and the Maintainer's Assistant", Invited Paper, Conference on Software Engineering Technology, Hong Kong, June, 1994.

[167] Yang, H., "The Supporting Environment for A Reverse Engineering System — The Maintainer's Assistant", IEEE Conference on Software Maintenance-1991, Sorrento, Italy, October, 1991.

[168] Yang, H., "Formal Methods and Software Maintenance — Some Experience With the REFORM Project", Position Paper, Workshop on Formal Methods, Montery, USA, September, 1994.

[169] Yang, H. and Bennett, K. H., "Extension of A Transformation System for Maintenance — Dealing With Data-Intensive Programs", IEEE International Conference on Software Maintenance (ICSM '94), Victoria, Canada, September, 1994.

[170] Yang, H., Bull, T. and Bennett, K. H., "A Transformation System for Maintenance — Turning Theory into Practice", IEEE Conference on Software Maintenance-1992, Orlando, Florida, November, 1992.

[171] Yang, H. and Chu, W. C., "Component Reuse Through Reverse Engineering and Semantic Interface Analysis", Accepted by The 19th IEEE Annual Computer Software Application Conference (CompSac '95), Dallas, Texas, August, 1995.

[172] Yang, H., Luqi and Zhang, X., "Constructing An Automated Testing Oracle: An Effort to Produce Reliable Software", The 18th IEEE Annual Computer Software Application Conference (CompSac '94), Taipei, Taiwan, November, 1994.

[173] Yau, S. S. and Collofello, J. S., "Some Stability Measures For Software Maintenance", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 6 (November 1980).

[174] Zimmer, J. A., "Restructuring for Style", *Software — Practice and Experience*, Vol. 20, No. 4, pp. 365–389 (April 1990).

[175] Zuse, H., *Software Complexity — Measures and Methods*, Walter de Gruyter, New York, 1991.