A Method for Maintaining New Software

Jennifer Louise Newton

M.Sc. Thesis 1994

Abstract

This thesis describes a novel method for perfective maintenance of software which has been developed from specifications using formal transformations. The list of applied transformations provides a suitable derivation history to use when changes are made to the software. The method uses transformations which have been implemented in a tool called the *Maintainer's Assistant* for the purposes of restructuring code. The method uses these transformations for *refinement.*

Comparisons are made between sequential transformations, refinement calculi and standard proof based refinement techniques for providing a suitable derivation history to use when changes are made in the requirements of a system. Two case studies are presented upon which these comparisons are based and on which the method is tested. Criteria such as scaleability, speed, ease, design improvements and software quality is used to argue that transformations are a more favourable basis of refinement. Metrics are used to evaluate the complexity of the code developed using the method.

Conclusions of how to develop different types of specifications into code and on how best to apply various changes are presented. An approach which is recommended is to use transformations for splitting the specification so that original refinement paths can still be used. Using transformations for refining a specification and recording this path produces software of a better structure and of higher maintainability. Having such a path improves the speed and ease of future alterations to the system. This is more cost effective than redeveloping the software from a new specification.

A Method for Maintaining New Software

Jennifer Louise Newton

M.Sc. Thesis

University of Durham

Computer Science Dept.

1994

Acknowledgements

I wish to thank Prof. Keith Bennett for his supervision and Mum, Dad and David for their endless support and encouragement.

Contents

Chapter 1

Introduction

Over the past few years it has become cost-effective for organisations to develop products which can easily adapt to changes in the requirements. This has caused organisations to re-address the question of software development and, in particular, that of software maintenance. *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" [1]. It can be seen as work carried out upon a developed system to keep up to date with any changes necessary. As the system ages, maintenance develops into a continuing process and becomes a major concern for organisations as more time and effort is needed to maintain the system.

Software maintenance activities are often divided into four categories (see [2] [66] [73]):

1. **corrective maintenance:** performed when the software does not conform to its specification (for example, fixing bugs discovered upon running a program).
2. **adaptive maintenance:** performed on a software system when its environment changes (for example, a new version of the operating system is introduced).
3. **perfective maintenance:** needed when requirements of the software change (for example, tax program changed to reflect new tax laws).

4. **preventive maintenance:** work carried out in anticipation of future malfunctions and to improve maintainability. This differs from the other categories since it is not a direct response to a user's request and some authors do not include it in this list of categories (see [52] [78]).

Lientz and Swanson [51] quantified the amount of effort spent in each area. The distribution is as follows:

Perfective	50%
Adaptive	25%
Corrective	21%
Preventive	4%

The large proportion attributable to perfective maintenance is mainly due to the fact that, to maintain their competitive edge, companies release new products quickly without thinking of the future possible changes in the requirements. Hence software cannot be modified quickly, easily and reliably, resulting in serious delays to changes in the software.

Organisations must now focus on developing products which are easier to maintain, i.e. products which are more *maintainable*. Maintainability is the quality which identifies how maintainable software will be. Longstreet defines it as "the effort required to find and fix or modify an error in operational software... the effects of software failure, and ways to minimize those effects"[55].

Maintainability is a very desirable quality in products where precision and correctness are of prime importance. Safety-critical systems are examples of such products and the need for accuracy in these cases has led to the use of *formal methods*. Formal methods involve the specification of a system represented with strict mathematical notation and the development of lower levels from this, so that code can be formally linked with its specification.

The use of formal notation has provided a new means for producing accurate code and has led to the evolution of a number of formal specification languages and development techniques (e.g. Z, VDM). These methods have often focussed on carefully identifying the requirements of the system and the first implementation of that system, but have not considered the possible implications of changes in the requirements. Having spent so much effort on the production of such accurate systems, it seems a waste to completely re-develop a system according to requirement changes and necessary enhancements. It is this idea which led to the research in this thesis.

The thesis describes a new method which formally specifies and implements code with a view to future adaptations of the system (in other words, aimed at perfective maintenance). By producing software which is easy to change, industry can eliminate many costs needed to reimplement a system and save a lot of time and effort. The method described here uses program *transformations* for forming a *refinement* path which can be re-used when necessary. A *transformation* is defined as the "formal step in which a program is converted to an equivalent with identical semantics" [12]. The transformations used by the method originate from Ward's thesis[82] and many have been implemented in a tool called the *Maintainer's Assistant* [84] for the purposes of restructuring code. The method uses these transformations for *refinement*, "a set of techniques to guide and control the process of producing a piece of software from a description of it; an implementation from a specification" [89]. Chapters 2 and 3 will provide further descriptions of these areas so that the method itself can be understood. This chapter will identify general problems of maintainability and cover basic techniques used in software maintenance which are also connected to this particular research.

1.1 Problems of Maintainability

Problems associated with producing maintainable software have been outlined by Brooks [21] and McDermid [58]. One of these is the complexity inherent within software due to the need to interface complex engineered systems and social or organisational systems; that is, the complexity is due to the complex systems it describes. Also, software can be complex at times when no regular structure exists or a system is so large that no single individual can understand it in its entirety.

Another problem lies in the difficulty of establishing and stabilizing requirements. This is due to users not being able to:

1. know exactly what they want,
2. realize the full limitations and capabilities of computer systems,
3. effectively communicate their needs to the requirements analyst, and
4. provide the complete list of details to produce the system required.

Another cause of this problem is that often requirements need altering due to changes in the environment and to users wanting functional enhancements of the system, even during development.

A third problem lies in the fact that it is important to understand the thought processes which went into writing a program and there are intellectual difficulties in establishing the relationships between different views and perspectives of a program. This can be called a problem of "invisibility" since the programmer's ideas and points of view are not explicitly written down and there is no known way in which to retrieve this information. This problem, like many in the area of software engineering, eventually reduces itself to one of intuition and psychological understanding on behalf of the maintainer.

A further problem is caused by the malleability of software. It is deceptively easy to write and change small programs but when these are part of a large system difficulties arise due to the interaction between different parts of the system. A programmer might wish to make changes to a program without realising the effect this will have on several other programs within the system. This causes an inordinate amount of time and effort to be spent making the necessary adjustments and then testing the software.

A final problem is that systems are often created in different problem domains; every time a new system is developed in a new problem domain a new theory needs to be established. While most engineering disciplines involve the application of existing theory to the development of a new system, software engineering usually involves the development of new theories.

Having analysed the problems that software can undergo, it is evident that a key factor in solving these problems (and ending the software "crisis") would be to find methods and means to improve and *maintain* the maintainability of software when it is first written. Issues concerning a possible solution to this will be discussed in later chapters.

1.2 Techniques

Most maintenance activities occur as a result of requests made by the user for alterations to the developed system. There are a variety of strategies and techniques available which can aid these activities. Areas of research involving these include reverse engineering, restructuring, reengineering, metrics and artificial intelligence.

1.2.1 Reverse Engineering

The word *reverse engineering* originates from the analysis of hardware. In a paper on the reverse engineering of hardware, Rekoff defines it as "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system" [74]. It is this usage of the term which has been directly translated for software (for instance, Chikofsky [25]).

In software development, the term *forward engineering* has come to mean the process of moving from the design of the system to its physical implementation. Hence reverse engineering is defined as the opposite to this, that is as the "process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [25]. Contrary to some people's belief, reverse engineering does not involve changing the system or creating a new system from the reverse engineered system.

There are several subareas of reverse engineering, two of which are commonly referred to as *redocumentation* and *design recovery.* Redocumentation is the simplest and oldest form of reverse engineering and is defined as the "creation or revision of a semantically equivalent representation within the same relative abstraction level" [25]. Tools which can perform this include pretty printers, diagram generators and cross-reference listing generators.

According to Biggerstaff, design recovery "recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains" [13]. He goes further to insist that design recovery must "reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth."

1.2.2 Restructuring

The term *restructuring* comes from code-to-code transform that takes an unstructured program and converts it into a structured form. It is defined as the "transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics)" [25].

Thus it is no longer confined to structuring code as it also covers the reshaping of data models, design plans and requirements structures. A tool which can currently aid the process of restructuring is the "Maintainer's Assistant", a transformation system developed at the University of Durham where transformations are applied to unstructured code to derive its equivalent in a structured form. This system will be described in more depth in the next chapter.

1.2.3 Reengineering

The term *reengineering* is also known as renovation or reclamation and is often confused with the previous terms since it actually involves a form of reverse engineering followed by some form of forward engineering or restructuring. Reengineering is defined by Chikofsky and Cross II as the "examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [25].

Another definition of reengineering can be found in Garnett and Mariani's paper on software reclamation and this states that "reengineering refers to the identification of components within existing systems possessing reuse potential and qualifying them according to some reuse-oriented specification technique" [36]. They go on to refer to reuse reengineering as the "construction of new systems by reusing information from foregoing ones", an approach which they also call "software reclamation".

1.2.4 Metrics

One important area of software maintenance is the use of metrics to provide a comparative measure by which software can be made characteristically maintainable at each iteration of the development process. Metrics integrate maintainability into developing software by identifying high risk areas in the code. This is done by evaluating software according to specific criteria and by producing a quantitative measure on a static scale. In other words, metrics assess the complexity of a procedure by comparing it to other procedures evaluated in the same manner.

Once problem areas are identified, actions are taken to reduce the complexity of the code through further abstraction or reimplementation and to test high risk areas so as to uncover existing errors. Metrics can be described as *predictive*, e.g. when they are used to foretell the future by predicting costs, etc., or *descriptive*, e.g. the use of complexity measures.

According to Conte [27], metrics can be classified as either process or product metrics. Process metrics will "quantify attributes of the development process and of the development environment" [27] while product metrics are "measures of the software product" [27] and are the type of metric usually referred to.

In the method described in this thesis, product metrics are used to assess the maintainability of the code which has been developed. There are three main forms of product metrics:

1. code metrics
2. structure (or coupling) metrics
3. hybrid metrics

Code metrics determine the complexity of a procedure by analysing the amount of information within a procedure or by assessing the logical complexity of the code [50] [43] [46]. *Structure metrics* examine the relationship between a section of code and the rest of the system [43] [50]. These are also known as *coupling metrics* [46] or *design metrics* [10]. *Hybrid metrics* combine the internal view of a procedure with the measure of communication connections between that code and the rest of the system [50].

There are many different metrics but the research described here used only three types of code metrics for the assessment of the maintainability of the code. These can be described as follows:

1. Lines of Code

   This is a measure of how many lines of code exist in a given procedure: the more lines of code, the more complex the procedure [50]. A line of code is generally defined to be "any line of program text that is not a comment or a blank line, regardless of the number of statements or fragments of statements on the line" [27].

2. Halstead's Software Science

   This is also known as Halstead's Effort Metric and was devised in 1977 [50] [43] [57] [10] [78] [27]. This involves the count of operators and operands in a procedure and the following initial values are established:

   (a) n1 = number of unique operators

   (b) n2 = number of unique operands

   (c) N1 = total number of operators

   (d) N2 = total number of operands

   From these another set of values can be calculated:

   (a) Vocabulary Size: $n = n1 + n2$

   (b) Length: $N = N1 + N2$

   (c) Program Volume: $V = N \times \log_2(n)$

   (d) Program Level: $L = (2/n1) \times (n2/N2)$

   (e) Language Level: $\lambda = L^2 \times V$

(f) Effort: $E = V/L$

The Program Volume is a measure of the size based on the length of implementation and the size of the vocabulary and the Program Level is an estimate of the level at which an algorithm is implemented. This level is inversely proportional to the program difficulty: the lower the level, the more difficult it is to implement the algorithm. The Effort is a quantification of the effort required to generate implemented code.

3. McCabe's Cyclomatic Complexity Number

   This was described in McCabe's paper [57] [50] [43] [46] [78] [27]. It is a count of independent logical paths through a procedure and is based on graph theory. That is, the procedure is represented as a strongly connected graph from which measurements are taken. Each node in the graph represents a sequential block of code and each edge a logical branching point through the procedure.

   From the graph, a calculation of the maximum number of linearly independent circuits (i.e. the cyclomatic number) can be made:

$$V(G) = E - N + 2$$

   where G is the graph, V(G) the cyclomatic number, E the number of edges and N the number of nodes.

1.2.5 The Use of A.I. in Software Maintenance

The use of Artificial Intelligence has been increasing in the field of software maintenance over recent years. Techniques within this area promise great improvements in programming productivity and reliability and developments in knowledge representation and automated reasoning have occurred through this. An excellent source for papers on this subject is [75].

The main aim is to develop a form of automatic programming so that the user need only say what he wants and the program would be developed automatically. An ideal solution would be for the user to state requirements which could automatically be transformed into formal specifications and from these specifications into code.

The most active area in developing such techniques is the field of program transformations. Since this is such a large area and most of the work proposed by this report is based upon the subject, it will be described extensively in a later section. For other surveys of this area see [69] [34].

Another active area of artificial intelligence research is in techniques for automatically determining theorem proofs [15] [91]. There are two main forms of this: deductive program synthesis and program verification.

Deductive program synthesis is "based on the observation that constructive proofs are equivalent to programs because each step of a constructive proof can be interpreted as a step of computation" [75]. Constructive proof aims to provide a method for finding an output corresponding to any given input. At present it is only possible to produce small programs from specifications written in logical languages from deductive synthesis since large programs involve large proofs which current theorem provers cannot as yet deal with.

Program verification uses theorem provers to verify that a program satisfies its formal specification. Again this is limited by the fact that theorem provers cannot yet deal with large proofs. Two approaches have been taken to deal with this. The first is that the prover is given some knowledge about programming areas in the form of lemmas and the second is to allow human interaction to guide the theorem prover.

Allowing human interaction has, in fact, assisted most applications of A.I. to software engineering. Since automatic programming is still not possible, artificial intelligence tools aim to provide assistance to the programmer rather than to replace them. Two of these "assistants" are the *Designer/Verifier's Assistant* [60] and *KBEmacs* [87], which uses a knowledge base of standard programming forms in programming construction.

The *Designer/Verifier's Assistant* was developed by Moriconi in 1979. Since it investigates changes made at a design and specification level in addition to program verification, a detailed study of his work was made so that any useful ideas relating this to the work presented by this thesis could be identified. The following section is a critical analysis of the paper he wrote describing the *Designer/Verifier's Assistant.*

No work seems to follow on from this until Moriconi's second paper, published in the *IEEE Transactions on Software Engineering* [61]. Since this also has implications involving the thesis, a critical analysis will also be made of this paper.

A third paper related to the method is a very recent one by Ira Baxter [8]. This also involves changes made at the design level and the approach illustrated is similar to that described by this thesis. It also deals with transformation systems and the tracing of design paths from specification to code. The fact that this approach could be successful and that a thesis was accepted on this topic validates and supports the work presented here.

1.3 Critical Analysis of Moriconi's Papers

1.3.1 A Designer/Verifier's Assistant

This paper describes the first prototype of a system called the *Designer/Verifier's Assistant* which reasons about changes to the design of a system. It represents a theory of Moriconi's concerning the development and maintenance of large formally verified systems. The problem which he was trying to solve in 1979 still faces the computer industry today and is the one which prompted the development of the method proposed by this thesis. The solution which he proposed was novel but unfortunately the theory was impractical and too vague for a full system to be implemented at that time.

Moriconi recognized that specifications, programs and proofs involving a system gradually build up and need frequent revision, so that "developing and maintaining formally verified programs, especially large ones, is an incremental activity... Consequently, one is faced not only with the problem of constructing this data, but also with the complex problem of determining the effects of incremental changes to it."

His proposed solution was to develop a system which parses programs and specifications

1. Suggest fully defining ExchangeSort $\longrightarrow$ Explain
2. $\longleftarrow$ What are the effects of fully defining ExchangeSort?
   ExchangeSort may have additional VCs.
3. $\longleftarrow$ What are the effects of changing the exit assertion of ValueOfMax?
   No effects on ValueofMax. The verification of other programs is not affected.
4. $\longleftarrow$ Why?
   No effects on ValueofMax because it is for specifications only and does not have VCs. No external effects because properties from ValueOfMax have not been used in proving any VCs.
5. $\longleftarrow$ What are the effects of changing the exit of IsPerm?
   No effects on IsPerm. The verification of other programs is not affected if the formula

   $$\textit{Changed exit specification} \Rightarrow \textit{all } Z : \textit{IntArray}, \textit{IsPerm}(Z, Z)$$

   is true. If not, the change invalidates the verification of ExchangeSort.
6. $\longleftarrow$ Why?
   No effects on IsPerm because it is for specifications only and does not have VCs. Invalid verification because a property from IsPerm has been used in proving ExchangeSort#1.
7. $\longleftarrow$ Done
8. Suggest fully defining ExchangeSort $\longrightarrow$ Edit

Figure 1.1: Illustration of dialog with Assistant

and could also generate and prove *verification conditions*, logical formulae used to ascertain that a program is consistent with its specification. The system also needs an "understanding" of the kinds of structures which can be changed or added and the ways in which they interact. It must be able to apply its "knowledge" to integrate new or changed information into the model of the system so that previous work remains valid. To understand more clearly what Moriconi means by "understanding" and "knowledge", a reproduction of his description of an actual scenario from the middle of a session will be presented here.

The example used is a sorting program which is being "incrementally designed and verified". A sequence is illustrated of three events which typically occur for each set of revisions. First, the user converses with the Assistant to gain an understanding of the effects of the changes which he might make. He makes these changes and fits them into the current model while keeping intact previous work that remains valid.

Figure 1.1 shows how the system would appear just after the program **ExchangeSort** has been partially defined and proved. The Assistant suggests completing the definition of ExchangeSort but instead of following this suggestion, the user uses the Assistant to see the effect of intended changes by typing "Explain".

9. Suggest fully defining ExchangeSort ⟶ ReadFileOfChanges.Sort

```
function ExchangeSort(A : IntArray) : IntArray =
begin
    entry N ge 1;
    exit(all I : int,
            I in [1..N] => ExchangeSort(A)[I]
                = ValueOfMax(ExchangeSort(A),1,I))
        and IsPerm(A, ExchangeSort(A));
    var B : IntArray := A;
    var K : int := N;
    keep K in [1..N];
    loop
        assert(all I : int, I in [K + 1..N] => ValueOfMax(B,1,I))
            and K in [1..N] and IsPerm(A, B);
        if K = 1 then leave end;
      [ B := Exchange(B, LocationOfMax(B,1, K), K);
      [ K := K - 1;
    end;
    result := B;
end;

function ValueOfMax(A : IntArray; I, J : int) : int =
begin
  [ exit(all k : int, k in [I..J] and I in [1..N] and J in [1..N]
  [         => A[k] le ValueOfMax(A,I, J)) and ...;
end;

function IsPerm(X, Y : IntArray) : boolean =
begin
    exit(all Z : IntArray, IsPerm(Z, Z))
      [ and(all Z : IntArray,
      [         IsPerm(X, Z) and IsPerm(Z, Y) => IsPerm(X, Y));
end;

[ function LocationOfMax(A : IntArray; I, J : int) : int =
[ begin
[     entry I in [1..N] and J in [1..N] and I le J;
[     exit LocationOfMax(A,I, J) in [I..J]
[         and A[LocationOfMax(A,I, J)] = ValueOfMax(A,I, J);
[     pending
[ end;
[
[ function Exchange(A : IntArray; I, J : int) : IntArray = ...;
[
[ function IsExchanged(A,B : IntArray; I, J : int) : boolean = ...;
```

10. Exec ⟶ Suggest

Figure 1.2: The changed function (changes are indicated by brackets).

11. Suggest generating new VCs for ExchangeSort $\longrightarrow$ $\underline{\$}$
    Tracing new path in loop
    Assume loop assertion

$$(all\, I\#1 : INT, I\#1 in[K+1..N] \\ \Rightarrow B[I\#1] = ValueOfMax(B,1,I\#1))$$

    and K in [1..N]
    and IsPerm(A,B)
    ...
    Generating new verification condition ExchangeSort#4
    ...
    End of path
    Unaffected VCs: ExchangeSort#1, ExchangeSort#2, ExchangeSort#3
12. Suggest proving VC called ExchangeSort#4 $\longrightarrow$

Figure 1.3: Impact of changes on ExchangeSort.

After seeing the potential effects of different kinds of changes, the user types "Done" and the user can invoke a text editor using "Edit". After the editing has finished, the Assistant needs to verify the altered version of ExchangeSort. Figure 1.2 illustrates the new version of the function, with brackets around the parts which have been changed. The user can then see the impact of these changes (see figure 1.3) by accepting the Assistant's suggestions for generating new verification conditions. The user can carry on the development by having these new verification conditions proved.

As there are many directions which the development can follow, the Assistant has a mechanism for providing reasonable suggestions for the next step in design and verification. This suggestion mechanism assigns priorities to tasks and a *scheduling policy* chooses the highest priority task and suggests it to the user.

In addition to generating and proving verification conditions, the Assistant also builds a model of the key parts of a program's design and verification and their relationships. This model is a collection of three models for each task performed by the overall design and verification; parsing and type checking, generating verification conditions and theorem proving. Examples of these models can be seen in the paper. The general model for the scenario displayed in figures 1.1 to 1.3 is also displayed.

Moriconi's paper concludes with experiences in using the Designer/Verifier's Assistant. He maintains that both its utility and the amount of computational efficiency grow proportionately with the size and complexity of the program being developed. However, although the tool reasons at the appropriate level of detail, sometimes it would be better for analysis to take place individually rather than by category (as it does now) and for more structuring in explanations.

A method of change where the effort required to make the change is not proportional to the size of the system would have an advantage over Moriconi's approach. The method described in this thesis has that objective and will be described in more depth later.

1.3.2 Approximate Reasoning About the Semantic Effects of Program Changes

This second paper by Moriconi [61] describes a logic for finding the semantic effects of changes through a direct analysis of the program. This logic is called *approximate* since weak results are sometimes inferred. The approximation is based on the structural interpretation of the information-flow relationships among objects in the program. "Information flow" between objects x and y occurs if a change in the value associated with x changes the value associated with y. Reasoning about the semantic effects of changes is based here on whether any information flows between objects (and not on how much information flows).

The paper briefly describes the characteristics of this logic before comparing the work to other work involving the semantic and structural analysis of programs. In 1972 Floyd [35] described an imagined interaction between a programmer and formal verification system which allowed the computer to maintain the consistency of specifications, programs and lemmas following incremental changes.

Moriconi developed a technique from this in 1979 [60], as discussed earlier in section 1.3.1. This, as are most verification systems, was based on Hoare logic (see [40]). A proof of a program in Hoare logic is a sequence of steps where each step is either an instance of a Hoare axiom, a Hoare sentence derived from a previous step by rule of inference or a theorem in the underlying logic.

The remainder of the paper focuses on information-flow, which is outside the theme of the thesis and so will not be described here. The main interest here is that work from Moriconi's earlier paper is extended and that logic is being applied to software systems. The application of formal mathematics to the development of systems is a major component of the method and will be described later.

1.3.3 Problems

A Designer/Verifier's Assistant

As this work appears not to have been developed further, it is important to assess those problems which prevented this. This will help the development of the method described in this thesis.

First, the Assistant deals with program specifications, a relatively new area at the time. Not many formal methods had been devised and the notation used by this paper resembles short sentences in natural English combined with Pascal. This specification language does not have a rigorous mathematical basis which implies that reasoning is vague and no proofs can be determined. The validity of these specifications is questionable.

Second, the specification language uses constructs found in Pascal which could cause confusion during refinement stages. In fact, these stages cannot be determined exactly since some of the specification can be directly translated from its "Pascal" format while the rest needs to be converted from English to a programming language. This mixture of accuracy and ambiguity does not provide a strong basis for deriving programming languages.

Another problem is that the effects of change are determined by the setting of design and verification flags. This was a popular method used in the past for verification work which has become outdated. Although the flags provide some guide to *impact analysis* (the investigation of how changes to variables affect other variables, functions, predicates and modules), they do not provide any details of the variables, procedures, modules or designs in question.

Finally, the paper needed more examples for illustrating the work. The examples which Moriconi does provide are limited and do not adequately portray the abilities of the Designer/Verifier's Assistant. It is difficult to understand from the paper what sort of examples would work with this tool and how the tool replies to questions in such clear English.

Approximate Reasoning About the Semantic Effects of Program Changes

Moriconi's paper which has just been discussed ends with a description of further work which would be to "evolve a general mathematical framework that explains how to build and extend incremental systems, such as the Assistant". The work in this paper appears to do just this but there are a few problems which can be found here as well.

A major problem is the introduction to the logic and the way in which it fits into prior work in information-flow. Pieces of information concerning the logic appear throughout the opening sections with no apparent order. The inference rules are also presented in a confusing manner since it is only at the end of the paper that the main elements of the logic are described.

However there are few real problems with this work and the examples of the use of the logic at the end give a good grasp of how the logic can work. The main work in deducing the effects of program changes lies in a form of impact analysis. To reason more accurately about the changes it would be necessary to extend this work so other changes can be made (i.e. to functions and predicates as well). Reasoning about larger changes could just imply a recursive extension to the logic or added features. The work in this thesis might be able to involve this in some way.

1.4 Critical Analysis of Baxter's Paper

This analysis is of a paper written by Ira D. Baxter [8] which was published in April 1992 and is based upon a PhD thesis written by Baxter in November 1990. A description of the paper will be given followed by any problems which can be identified. Conclusions as to the applicability of the three analysed papers to the research detailed by the thesis will follow this section.

1.4.1 Design Maintenance Systems: A Summary

This paper suggests that the main objective for the upkeep of systems is *design maintenance* rather than software maintenance, where design maintenance means the updating of design information as changes are made to the system. The article sketches a basic design for a design maintenance system which attempts to do this work.

The approach to a Design Maintenance System (or DMS) outlined here involves several important factors. First, the software system must be formally specified as must be the *maintenance deltas* integrated within it. A maintenance delta is an expression representing desired changes in the program functionality, performance and implementation technology.

Second, the implementation of a DMS must be derived from transformations which, according to Baxter, are the applications of *transforms* at certain places called *locators*. A transform is any function which maps programs into programs while a locator is the place in the program where the locator is applied. The denotation for a transformation with transform t and locator l is t^l.

Other important factors are that a justification exists to prove that the implementation truly solves the problem stated by the specification and that tools exist for modifying the design justification. In summary, this approach needs to ensure that the design is correct and that alterations can be made at this level rather than the implementation level, with a set path of transformations producing code which corresponds to the design.

The application of transformations is controlled via a library of heuristic methods coded in a Transformation Control Language (TCL). Each method formally relates a design plan to a design purpose and a set of such methods can be used to decompose the specification into solvable subproblems. Each subproblem will have its own specification and can be solved by executing the plans from the methods chosen.

Design capture can take place given such a transformation system and a transformational planning language such as TCL. The formal specification will describe what is intended and can be captured easily, while the sequence of transformations will describe how the generated program was constructed and can be captured as a linear *derivation* history. The reason for applying each transformation is captured by storing a trace of the nonprocedural unfolding of goals during the execution of the TCL methods. This trace is known as the *design* history.

In Baxter's own words, the problem of design maintenance is of "updating the specification, the derivation history, and the design history in a way consistent with any new desire, stated as a maintenance delta". The implementation can then be generated by applying the same sequence of transformations in the revised derivation history to the specification.

Maintenance deltas appear in two forms: specification deltas, which affect the problem definition, and support deltas, which affect the implementation of the solution. Most deltas are specification deltas and are denoted either as Δ_f, for changes in the system function, or as Δ_G, for changes in the desired performance. These specifications deltas are applied to the current specification to revise the specification component of design information while support deltas are applied to the transformation system components.

A major problem is the one of integrating the maintenance delta into a design history. The first design history is constructed by either running a transformational implementation on a chosen specification or by reverse engineering such a history from an existing system. It can then be revised according to the delta applied by rearranging and pruning the derivation history and then pruning away the parts of the design history which are no longer useful. The TCL methods can regenerate any incomplete part of the design history.

The derivation history is rearranged through two actions: a *delay* and a *preserve* action. If a particular transformation in the derivation history cannot be preserved then its application is delayed as long as possible. This is done by swapping it with the next transformation, an action which depends upon the commutativity of the two transformations; i.e. whether the effect of performing the first transformation followed by the second is equivalent to that of the second transformation followed by the first. The "offending" transformation continues to be swapped in this manner until it is no longer commutative with the next and has thus been delayed as far as possible.

Thus the derivation history is revised by scanning the original from beginning to end, checking the delta for interference with each transformation. When the transformation interferes with the desired change it is "banished" (i.e. delayed as far as possible); otherwise it is preserved. The scan stops when neither of the two actions can take place and the derivation history is then truncated. The delta can then be applied and the implementation finished using the transformation system itself.

The design history is revised by inspecting its relation to a certain delta and marking those parts which conflict with the delta. Finding these conflicts depends upon the type of delta involved and these parts are then removed from the design history, leaving it incomplete. The pruning of the design history involves removing all portions of the design history which are marked, every agenda item which depends uniquely on some pruned agenda item and agenda items which are generated as descendants of those marked.

The pruned plan can be repaired by carrying out actions for incomplete agenda items which could involve the generation of new agenda items. This involves choosing the earliest incomplete agenda item, as determined by the ordering constraints in the design history, and then executing it according to a specific TCL action taken from some TCL method.

The main use for a DMS is the construction of an incremental maintenance system. Deltas can be applied to only partially completed implementations and new deltas applied as the implementation grows. A DMS can also be used as the foundation for a reusability system. Implemented components can be stored in a library together with their specifications and design histories and a maintainer can choose the component whose specification is near to his desires. This component can be revised by applying the corresponding delta to the stored history.

Work which is related to this includes the transformation systems PDS [24] and the Maintainer's Assistant [84]. PDS is a system which keeps derivation histories and rederives components dependent on changed components. The Maintainer's Assistant maintains existing software by reverse engineering existing concrete programs into abstract ones, applying functional deltas to the abstract programs and reimplementing the abstract programs.

1.4.2 Problems

Using design and derivation histories to maintain software is a novel idea and suggests a useful new approach to solving the problem of software maintenance. Unfortunately the description of this new technique remains a proposal. There is only one example of how this might work, added as an Appendix, but this again provides a theoretical approach to how this method might work rather than how it actually works.

The method can apparently be used with any transformation system but a concrete example of how this can occur does not appear in the paper, suggesting that the method remains theory rather than practice. It would be useful to test this method on an existing transformation system to show its validity. As it stands, the method remains simply a "good idea" and much work needs to be done in order to validate it scientifically or evaluate it from an engineering perspective.

This identifies another major problem with the described method. Transformation systems vary greatly and the affirmation that this method can be used for *any* transformation system needs proof. Work is needed to ascertain which transformation systems can be involved and how the method must be adjusted in each case.

Another problem is that the method relies upon a design history already existing. This is often not the case and the problem of recreating one for a specific system outweighs the problem of changing the system: it is probably more difficult to describe the system using formal specifications and formally developing the design history than it is to apply formal changes to the system. The author should have attempted to tackle this problem first before developing the method for change.

As it stands, the method is a simple one of reusing the design history until it is no longer valid and banishing any transformations that are no longer useful, adding extra ones to complete the derivation history. While this appears a good approach, it is remains very subjective, relying upon the maintainer to truncate the design history when transformations are no longer valid and to reapply new correct transformations to complete the implementation. There is nothing to guide the maintainer as to which transformations should be banished or which should be introduced: it remains entirely based upon his judgement.

Furthermore, the method uses a strictly *top-down* approach. While this allows complete records of both the derivation and design histories and eases alterations made to both, it does not always correspond with the thinking process behind software maintenance. When making changes to a system, a maintainer might use a combination of a top-down and bottom-up approach for altering the system more effectively. Unfortunately, the method allows him only to work in the one direction and might involve more work on his behalf when trying to introduce new transformations or banish unnecessary ones.

However, this allows for a more systematic maintenance of the system and for one which is easier to trace as more changes are included and the system diverges from the original. While a combination of top-down and bottom-up approaches might aid the maintainer initially, eventually the maintainer might not be able to trace which steps allowed the creation of the new system. If he did, this information still remains unique to the maintainer in question and other people might not understand how the new system was eventually created. This would become a major obstacle for others attempting to maintain the system, especially if the original maintainer were no longer accessible.

1.5 Conclusion from Papers

The three papers which were analysed provided ideas as to an original method for producing *maintainable* software, that is software which can be easily changed and updated (this will be discussed in more detail in the next chapter).

Moriconi's first paper illustrated that knowledge based systems could be used to reason about changes made to a design or specification. This supported the use of transformation systems as a basis for a method of change. Also, the models he referred to help to describe the relationship between sections of code and the specification. Ideas presented for the *Designer's* half of the Assistant could be used to aid work in the development of a method.

The second paper provides ideas on the use of a logic for investigating the effects of program changes. It was initially hoped that the method could use a similar logic for determining the semantic effect of changes, but it was found that this approach was better suited for changes of variables within the code rather than changes in the specification. The method currently uses none of the ideas presented in this paper but it is important to keep these in mind for future developments.

Baxter's paper is the most relevant to this project since it proposes a method for the upkeep of systems based upon a design path built from transformations. However, this method does not appear to have been tested on any real transformation system so its claim to hold for *any* transformation system is not proved. In fact, it does *not* hold for the transformation system upon which the method described by this thesis is based, the *Maintainer's Assistant.* Baxter's method of *delaying* and *preserving* transformations proved of no consequence towards the maintenance of the case studies. If a transformation cannot take place at a certain point, then swapping it with the next will not mean that it can hold later on.

1.6 Outline of Thesis

This thesis will describe a method which assists the perfective maintenance of software produced using a formal method and uses ideas from the papers described earlier. As the method uses *program transformations* for *refinement* and for improving maintainability, these will be discussed in **Chapter Two** of the thesis. This chapter looks at the maintainability of software and how activities in the software life cycle might improve this before considering the applications of transformations to maintainability.

Chapter Three describes the method for producing maintainable software using transformations in a tool called the *Maintainer's Assistant.* A description of how these transformations can be used for refinement is provided and a comparison of different refinement techniques given.

Chapter Four describes how the method could apply to a simple case study of finding an integer square root. This problem was originally presented as an example of the use of a refinement calculus in Morgan's book, [59]. The effect of changes to the specification is investigated for this original development and is compared to the method.

Chapter Five describes how the method works with a larger case study which depends more upon the organisation of data rather than the functionality of the code. Different types of transformations are needed to refine this and a comparison is made between this and the original development of the problem. This case study is the description of a library system described using Z in a paper by King and Sørensen, *From specification, through design, to code: a case study in refinement* [45]. How changes affect this development are investigated in the chapter and compared with the development by transformation approach. The latter will prove more favourable a refinement technique when considering perfective maintenance.

Chapter Six revises what was learned from the case studies and provides an assessment of the maintainability of the software in each case. An indication of how the method could work for more complex case studies is provided.

Chapter Seven is the conclusion of the thesis. This investigates problems which were encountered while doing the work and how the method will apply for more general situations. Future work on applying the method to other case studies and the development of a tool is described at the end.

1.7 Summary

This chapter gave a brief introduction to the field of software maintenance and described general techniques used to deal with the problem of maintaining large systems. Critical analyses were made of three papers which used some of these techniques and which provided ideas for the method described in the thesis. The outline of the thesis concluded the chapter.

Chapter 2

Maintainability

The *maintainability* of a software system is a term which describes how easy it will be to maintain that system and can be determined before the system is implemented. According to Longstreet, "maintainability examines the effects of software failure, and ways to minimize those effects" [55]. Identifying all the future problems of a system is difficult and so one can only hypothesize about the qualities that make a system maintainable.

Maintainability can be classed as either *internal* or *external* depending upon whether attributes of the software product or those of the environment are being considered. Examples of attributes of the product which make it more maintainable include modularity, good documentation and structured code. Those of the environment include the skills of the maintainers and the tools which are available.

In the next section, both types of attributes will be discussed as they figure in the various stages of the software life cycle. The actions necessary for the production of maintainable software will be identified as they occur within each stage of the life cycle. Although it remains difficult to decide which factors would enhance maintainability at such an early stage, general guidelines for the way in which each activity should be carried out can be determined.

It is also possible to examine qualities of the software itself and, judging from past maintenance problems, rewrite the code so that it is more maintainable. Issues concerning maintainable software will be discussed once the activities during the software life cycle phases which may aid maintainability are considered.

2.1 Analysis Activities

During the analysis stage, a variety of activities enhancing software maintainability include the development of standards and guidelines, the setting of milestones for supporting documents, the specification of quality assurance procedures, the identification of likely product enhancements, the determination of resources required for maintenance and preliminary budget estimates [32] [27].

The costs of maintenance are difficult to estimate in advance as they vary depending upon the specific application used. However, for large software systems, the actual maintenance cost can be said to be approximately four times development costs [78] [10]. Boehm [16] uses a formula for approximating the cost of software development but this depends upon the existence of previous data.

According to Somerville [78], there are mainly five external factors which affect the cost: application support, staff stability, the program's lifetime, the external environment and hardware stability. Maintenance costs are also governed by such internal factors as module independence, programming language and style, program validation and the quality and quantity of program documentation.

2.2 Design Activities

Design activities can be divided into two parts: architectural and more detailed design. Architectural design is the "process of defining a collection of hardware and software components and their interfaces to establish a framework for the development of a computer system" [1].

The first architectural design activity must be to ensure that the design is clear, modular and easy to modify. By modular we mean that the design should comprise distinct components, enabling a change in one area of the design to not affect any other design areas. To achieve this ideal design, design concepts such as information hiding, data abstraction and top-down hierarchical decomposition must be used.

Information hiding and data abstraction involve the suppression of information in some form or other. Information hiding usually refers to modules in the system hiding the internal details of its processing activities, especially design decisions that are likely to change. Data abstraction is effectively a case of information hiding but involves hiding the data structure, its internal linkage and the implementation details of the procedures that manipulate it.

Another activity could be to try to determine where changes or enhancements in the design might possibly take place and to design the system so as to ensure the ease of these alterations. This could be aided by the further activity of using standardized notations such as data flow diagrams and structure charts to make the design easier to understand and to verify for completeness and consistency.

More detailed design includes "specifying algorithmic details, concrete data representations, and details of the interfaces among routines and data structures" [32]. Again, a useful activity would be to utilize standard notations to specify algorithms, data structures and interfaces. It would also be advantageous for each routine to be documented, specifying possible side effects and exception handling (the dealing of events which suspend normal execution of a program).

Finally, a call graph and cross-reference directory should be included; these can provide the information which determines the routines and data structures affected by modifications to other routines.

2.3 Implementation Activities

One of the main goals of implementation is to write source code and internal documentation so that modification is eased; this can be achieved by making source code as clear and straightforward as possible. Clarity is enhanced by structured coding techniques, good coding style, good comments and general documentation.

More specifically, future maintenance will be easier if single entry, single exit coding constructs are used, standard indentation of constructs observed and a simple, clear coding style employed. It will also be improved by symbolic constructs to parameterize software, by data encapsulation techniques, by margins on resources and by standard documentation prologues for each routine.

These standard prologues should include details such as the author, date of development, maintenance programmer and date and purpose of each modification. In addition to this, one final improvement would be to follow standard internal commenting guidelines when writing the source code. The following section will describe in more detail the various implementation activities which will ensure that software is *maintainable*; i.e. written so that future modification will be easy.

2.3.1 Maintainable Software

There are certain qualities which software should have to ensure its future maintainability. If code can be written so that future changes can be implemented easily without drastic side effects, then it can be termed "maintainable". According to Boehm *et al* [17], maintainable software must have three characteristics: testability, modifiability and understandability. All of these depend upon the system complexity and system modularity.

Complexity can either be computational, when it is difficult to prove the correctness of the code, or psychological, when it is difficult to understand the code. To minimise these forms of complexity, one can use high level languages, good documentation (meaningful comments) and standard coding conventions. By following these guidelines, the code should be easier to understand and hence alter.

Modularity involves the extent to which the system can be decomposed into smaller sections. Software is more maintainable if it remains as independent as possible but yet includes comprehensive links within itself which "glue" it together. Thus one aims to provide minimum external coupling and maximum internal cohesiveness [26].

According to Longstreet [55], there are a number of constructs which must be avoided within a section of code to determine its future maintainability. These are:

- Deeply nested DO loops,
- Excessive IF statements,
- Excessive use of global variables,
- Excessive GOTO statements,
- Embedded parameters, literals, constants,
- Self-modifying code,
- Excessive interaction between modules,
- Multiple entry-exit modules and
- Redundant modules.

Having attempted an assessment of what makes software maintainable by looking at it through the various life cycle activities, it is now appropriate to consider the technique which the method described by this thesis will use to ensure maintainability; *refinement.*

2.4 Refinement

Refinement is a technique for developing stricter definitions of specifications and programs without losing any of the semantics. There are two different refinement processes: operation refinement and data refinement. *Operation refinement* involves the "refinement of operations (or, more generally, of algorithms) to produce executable equivalents" [53] while *data refinement* involves the derivation of a "formal documentation of the relationship between abstract and concrete states" [72]. These two processes usually occur in tandem but depend upon the refinement technique in question.

There are two forms of refinement technique: refinement methods and refinement calculi [53]. *Refinement methods* involve the production of a more concrete version from the more abstract and a demonstration that the concrete version meets the requirements of the specification through a sequence of formal proofs. *Refinement calculi* are based upon the successive application of provably correct transformation rules and do not require proofs at each stage of the refinement.

The main refinement methods are the VDM refinement method [42], the IBM Hursley Park method [45] and the rigorous refinement method for Z [62]. All three follow a similar approach: operation refinement is performed stepwise while data refinement involves showing that the abstract and concrete views of the data are analogous. This is done by defining a *retrieve* relation which describes the relationship between the two views in mathematical terms.

There are several versions of a refinement calculus but the most popular is detailed in Carroll Morgan's *Programming from Specifications* [59]. This is based upon the work of Dijkstra, Hoare and Floyd and relies on a series of development steps which are dependent upon a *refinement law.* All refinement calculi use Dijkstra's guarded command language [30] as the final product of the refinement process and the refinement steps are really formalisations of Dijkstra's ideas on program development [29].

Morgan's book provides a list of the possible refinement laws which can be used for developing one program into another. It also presents some case studies where specifications are refined into algorithms which can be easily translated into code. One of these will be described later for illustrating how the method for implementing change could eventually work.

A comparison of refinement techniques will be made when describing the method in the next chapter. Since the method proposes to use *transformations* for refinement purposes, these will be described next.

2.4.1 Program Transformations and Transformation Systems

Program development by the use of transformations is a "method of software development in which a program is derived from a formal problem specification by manageable, controlled transformation steps which guarantee that the final product meets the initial specification" [7]. In other words, program transformations involve the identification of changes made to a program which leave it logically equivalent to its original.

Program transformations are useful in software maintenance research as they could identify ways in which new software might be written so as to achieve ease in future maintenance of the system. They can also be applied to software that has already been written so that it can be transformed into a program that will be more easily maintainable or so that maintenance problems regarding the program can be identified or solved.

Transformations do not necessarily apply to code alone; they can also be involved with the specifications of the system. Specifications can be "transformed" into sections of code and vice versa. The achievement of a tool which could do this would benefit maintainability research immensely; viewing how changes to a specification affect the code could help to establish a new method for transforming the code into a more maintainable form.

Transformation systems are tools which enable the programmer to transform sections of code or specifications. The main goals of a transformation system include providing general support for program modification (for example, optimization of control structures), generating a program from the formal description of the problem (that is, program synthesis), adapting the program to different environments and verifying the correctness of a program [92].

There are many types of transformation systems; some of these will be discussed in their chronological order.

Burstall and Darlington

Burstall and Darlington were the first to work on program transformations in the mid 1970's [23] [92]. They produced two systems which are mainly automatic; i.e. the system selects appropriate rules through the use of built-in heuristics or other strategic considerations.

The first system was based on a schema-driven method for transforming recursive programs into imperative ones and used built-in rules such as recursion removal, the elimination of redundant computations, unfolding and structure sharing. The main goal here was to improve efficiency.

The second system was designed to manipulate applicative programs by using only six basic rules: definition, instantiation, unfolding, folding, abstraction and data-structure "laws". Other functions can be created by a combination of these rules or by a definition from the user. The user can enter functions if they are written as a set of equations in a restricted form of NPL, an applicative language for first order recursion equations.

Balzer

Balzer's work in the early 1980's [4] resulted in an implementation system for program transformations. This system allowed a formal specification (written in GIST) to be systematically converted into an implementation in three phases: explication, reorganisation and representation selection.

The explication phase is an attempt to understand the algorithmic structure behind the specification by converting implicit structures to explicit ones and dealing with any constraints. The following phase involves the reorganisation of a program so as to mitigate computational expense. The last phase is to choose a representation suitable for this reorganized program.

CIP-S

This system derived from the Munich project CIP (Computer-aided Intuition-guided Programming) which took place between 1976 and 1983 [7] [22] [68]. The main objectives of this project were to:

- produce a method for guiding the process of formal reasoning in program development,
- design a "wide-spectrum language" in which to write specifications and programs at any level as well as to carry out transformations,
- develop an interactive system for supporting the evolution of programs.

Thus a transformation system was developed in accordance with the CIP view of inferential programming (see [7]) and involves the transformational manipulation of program schemes. These schemes are produced by CIP-L (the wide-spectrum language) and are basically algebraic specifications for introducing data types.

DRACO

The DRACO system bases its software construction on the paradigm of "reusable software"; i.e. the reuse of a library program's design but not its code [92]. It is an interactive system allowing the user to refine a problem written in a high-level language into a LISP program and enabling the user to define his/her own level of abstraction.

TAMPR

The TAMPR (Transformation-Assisted Multiple Program Realization) system supports Fortran programming at the Argonne National Laboratory [18]. The system performs transformations within the Fortran language, aids in the translation of Fortran to Pascal and transforms LISP programs into Fortran ones.

ZAP

The ZAP system and language was devised by Feather [33] and is based on the fold/unfold work of Burstall and Darlington mentioned previously. The ZAP language is a language for expressing transformation and developments but cannot express higher level means of structuring developments; these need to be applied informally.

REFINE

REFINE is a programming environment which includes a high level executable specification language, a specification language compiler, an object oriented database, an editor interface and tracing and debugging tools [65]. The tool converts code into design, providing an electronic path between code and its corresponding design language. It also allows the maintainer to edit the structure chart, cut and paste the code and generate high level documentation to describe the code structure.

The work of Burstall and Darlington and the project CIP were the main influences on the transformation system upon which this research is based: the **Maintainer's Assistant** (described in the next section). Ideas which particularly led to the development of the method for producing maintainable software are:

- the use of a system with built-in heuristics for transforming programs (Burstall and Darlington)
- improving efficiency with a schema-driven method for transforming programs (Burstall and Darlington)
- manipulating programs by combining rules or introducing new definitions (Burstall and Darlington)
- developing a method for guiding formal program development (CIP)
- using a "wide-spectrum-language" to represent many levels of specification and program (CIP)
- using the above language for performing transformations (CIP)
- using an interactive system for "evolving" a program (CIP)

2.4.2 The Maintainer's Assistant

The **Maintainer's Assistant** is a system developed by a reverse engineering project called **ReForm** which involved the University of Durham, Durham Software Engineering

Ltd and IBM. The project's aim was to provide a transformation system which could carry out software maintenance work partially under the maintainer's control.

The main goals of the system are to:

1. increase the programmer's understanding of the program,
2. enable the programmer to re-express the program in terms which match the problem being solved, and
3. discover and prove relationships between data structures in a program which would be difficult to discover from the source code alone.

The architecture is best represented using a diagram of the relationships between the components of the system (see figure 2.1). The tool can work with assembler code by having an **assembler** to generate an assembler listing file. This is then translated into equivalent low-level WSL statements with a **lexical analyser**. WSL represents the Wide Spectrum Language used by the Maintainer's Assistant since it allows both high and low level code. The tool's transformations act directly on WSL, whether it be equivalent to specification or code. A description of WSL can be found in appendix A.

As displayed in figure 2.1, the maintainer is faced with an X-window or PC (menu based) front end which is connected to the **Browser Interface**. Menu choices and mouse actions from the front end generate ASCII commands for the browser interface. Commands which concern information on the current program version are executed immediately by the interface and returned as ASCII strings. The front end then updates its display. Any other type of command is passed on to the structure editor or program transformer as appropriate.

The **Structure Editor** executes movement and editing commands and is the only means for manipulating the source code's internal representation. This allows the maintainer to edit the program text and is the only way in which the code can be changed into a version which is not logically equivalent to the original.

The **Program Transformer** executes the transformation commands. It contains a library of proven transformations and a set of correctness conditions for each transformation.

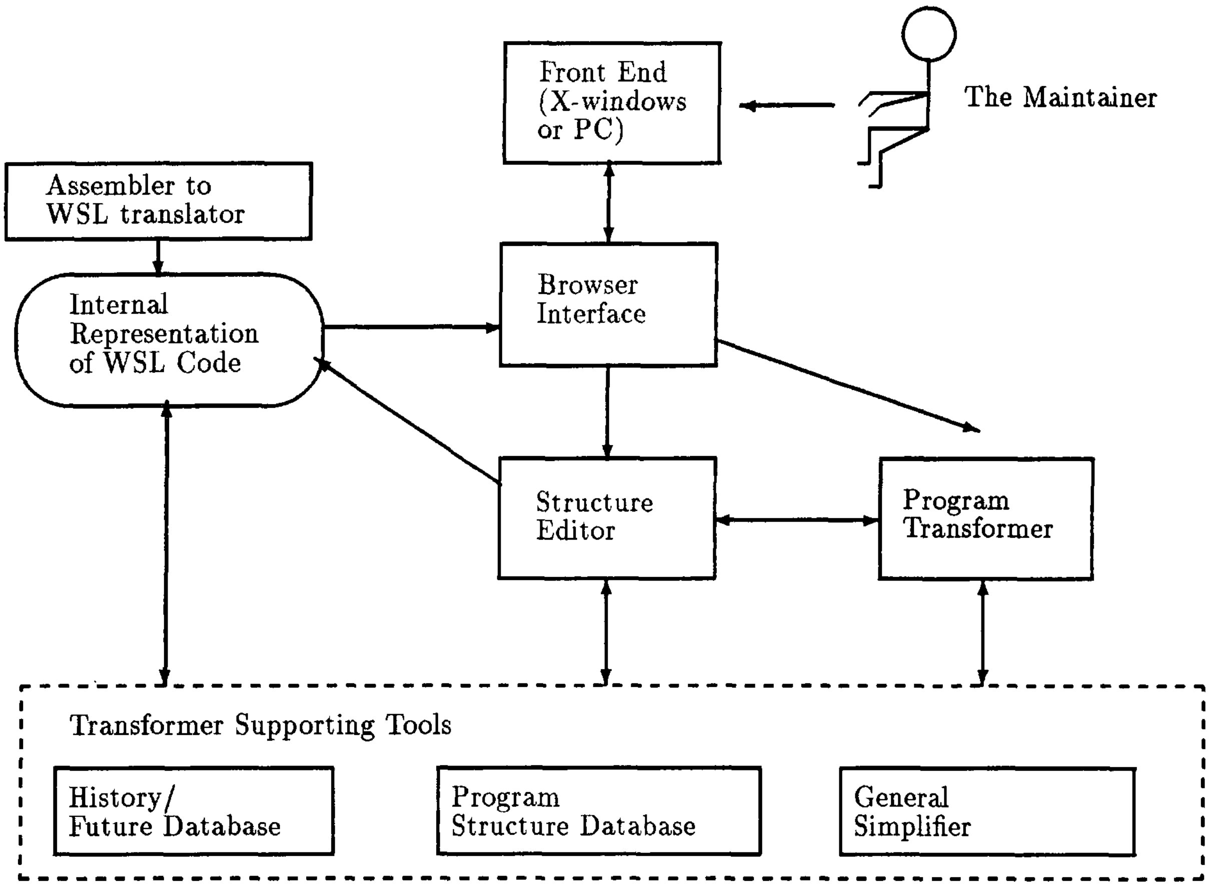

Front End (X-windows or PC)
The Maintainer
Assembler to WSL translator
Internal Representation of WSL Code
Browser Interface
Structure Editor
Program Transformer
Transformer Supporting Tools
History/ Future Database
Program Structure Database
General Simplifier

Figure 2.1: The Architecture for the Maintainer's Assistant

When the maintainer selects a transformation, the appropriate transformation is recalled from the library together with the applicability conditions. If the system is satisfied that a transformation is applicable then a sequence of edit commands are sent to the structure editor so that the transformation is applied at the selected piece of the program text.

There are a number of tools for assisting the Program Transformer. A **History/Future Database** tool allows the maintainer to go back to earlier versions of the transformed program. "Undo" is used for retrieving the previous version of the program and "Redo" for undoing the last "Undo" command.

The **Program Structure Database** is accessed by the Program Transformer via a Database Manager. When code is being transformed, questions about the program are sent to the Manager. The Manager will go through the program structure, calculate the answer and record this in the database before replying to the Transformer. When the question is asked again, the database manager checks the database and returns the result immediately.

The **General Simplifier** carries out symbolic calculations in mathematics and logic (for example, +, -, *, /, Min, =). It also accepts two commands: *Simplify* and *Prove*. The first returns an expression in its simplest form and the second allows very simple algebraic proofs.

The tool at present can apply numerous transformations to a section of code (rewritten in WSL) so that it appears in a logically equivalent form. Appendix B lists the basic transformations in the system and the next chapter illustrates how sequences of these transformations from specification to code provide the basis for a method of producing maintainable software.

2.5 Summary

This chapter focussed on the area of software maintenance which this research aims to improve: software maintainability. Activities within the software life cycle which can produce maintainable software were investigated. The method concentrates on design and implementation activities and so more emphasis was placed on these stages of the life cycle. An introduction to refinement and program transformations was provided along with a survey of different transformation systems. The system with which the method works, the Maintainer's Assistant, was described in the final section.

Chapter 3

A Method for Maintaining New Software

3.1 Ideas for Method

The development of the *Maintainer's Assistant* involved building transformations for restructuring code and for forward engineering. This indicated the possibility of using transformations for refinement purposes. A specification written in a formal language could be developed into code through the sequential application of transformations, thus *refining* the specification into its programmable form.

From this, ideas arose for a formal approach of developing code from a specification so that changes are easier to make in the software. Each transformation could be recorded in the sequence in which it was applied to the specification and the refined versions until code was reached. As with Baxter's method [8] (described in chapter 1), this could be recorded in a *derivation history* as follows:

$$S_0 \xrightarrow{a} D_1 \xrightarrow{b} D_2 \xrightarrow{c} D_3 \xrightarrow{d} C_1$$

where S_0 represents the initial specification, D_1, D_2, D_3 the different versions of the design as the specification is transformed into code and C_1 the first version of the implementation. The arrows represent the application of transformations and a, b, c, d the different transformations which are applied.

If requirements of the system change, these changes could be inserted at the specification level and the history used again to develop a new altered version of the code. The derivation history might then appear as:

$$S_0' \xrightarrow{a} D_1' \xrightarrow{b} D_2' \xrightarrow{c} D_3' \xrightarrow{d} C_1'$$

where S_0' represents the altered version of the specification, D_1', D_2', D_3' the different altered versions of the design as the specification is again transformed into code and C_1' the altered version of the implementation. The transformations a, b, c, d remain the same.

In some cases the derivation history might be extended for certain changes or only part of the old one needs changing for minor changes. For instance, perhaps only part of the design is affected and so the following new history would be:

$$S_0' \xrightarrow{a} D_1' \xrightarrow{b} D_2' \xrightarrow{c} D_3 \xrightarrow{d} C_1$$

where S_0' represents the new specification, D_1', D_2' the different altered versions of the design but D_3 and C_1 remain the same as before. Alternatively, the specification might not be altered but the lower levels changed instead and so:

$$S_0 \xrightarrow{a} D_1 \xrightarrow{b} D_2' \xrightarrow{c} D_3' \xrightarrow{d} C_1'$$

would become the new version of the derivation history.

Obviously, some transformations might no longer be applicable due to the changes made. It was found that Baxter's method of delaying and preserving transformations could not be used for producing maintainable software since the fact that a transformation could not be applied at a certain stage often implied that it could not be applied at all. Also, removing that transformation from the sequence sometimes made the following transformations redundant. More of this will be described when the results from the case studies are presented.

From this, the idea of inserting *extra* transformations from the *Maintainer's Assistant* at a certain stage of the derivation history and recording a new version of the derivation history each time came about. For instance, the derivation history used before might become:

$$S_0' \xrightarrow{a} D_1' \xrightarrow{b} D_2' \xrightarrow{c} D_3' \xrightarrow{e} D_4 \xrightarrow{f} D_5 \xrightarrow{g} C_2 \xrightarrow{h} C_3$$

where e, f, g, h are new transformations needed to develop D_3' into code. The new versions of the design and code corresponding to the changes which were made and the new transformations used are D_4, D_5, C_2, C_3.

Another idea was initially to add transformations to the altered specification (S_0') until there were two parts, one of which was identical to the original specification (S_0). The derivation history could then be applied to this second part, developing altered code (C_1) as in the original, and new transformations could be applied to the first part. Referring back to the original example, the path would appear something like:

$$\begin{tikzcd}
S_0' \arrow[r, "x"] \arrow[dr, "y"'] & S_1 \arrow[r, "n"] & S_2 \arrow[r, "o"] & D_8 \arrow[r, "p"] & C_8 & & \\
 & S_0 \arrow[r, "a"] & D_1 \arrow[r, "b"] & D_2 \arrow[r, "c"] & D_3 \arrow[r, "d"] & C_1 &
\end{tikzcd}$$

The new derivation history would involve the initial transformations which separated the specification (x, y), the old derivation history (with transformations a, b, c, d) and the additional transformations applied to the first part (n, o, p). These ideas will be expanded upon in the section on the method.

It was decided that an incremental case study approach should be adopted where the effect of changes on the original refinement of the problem and upon the refinement by transformation method could be investigated. Two case studies which provided a framework for the method will be described in the following chapters. One was refined by way of Morgan's refinement calculus [59] and the other by the IBM Hursley Park method [45]. Refining each case study with transformations provided a good way of comparing the refinement by transformation with a calculus and a method to test whether it was indeed valid to produce code in such a way. A brief description of this calculus and method is given next.

3.2 Techniques

As mentioned before, there are two types of refinement technique: the use of a refinement *calculus* and a refinement *method.* A *refinement calculus* is based on the successive application of provably correct transformation rules and does not require proofs at each stage of the refinement. A *refinement method* involves the production of a more concrete version from the more abstract and a demonstration that the concrete version meets the requirements of the specification through a sequence of formal proofs. To promote the use of transformations for refinement, descriptions of both must be made before these can be compared to the transformation method. The calculus which will be used is Morgan's refinement calculus while the method is the IBM Hursley Park method. These are used for the case studies upon which the method was tested and will be compared in each to the refinement by transformation method both for producing code and for making changes in the requirements.

3.2.1 Morgan's Refinement Calculus

Morgan's refinement calculus assumes a knowledge of predicate calculus and Dijkstra's *guarded commands* (see [29]). Specifications are written in algebraic notation and are developed into code by using various laws of refinement. Some of the basic laws of refinement will be described here but for a more complete list, refer to Morgan's book [59].

A specification of a program describes its function and involves three main features:

1. a *precondition* which describes its initial states,
2. a *postcondition* which describes its final states, and
3. a *frame* for listing variables whose values might change.

If the initial state satisfies the precondition then only the variables in the frame will change so that the final state satisfies the postcondition. The form used here for illustrating a specification with precondition *pre*, postcondition *post* and frame w is:

$$w : [pre, post].$$

An example of this could be the specification which assigns the square root of x to y, provided x lies between 0 and 9:

$$y : [0 \leq x \leq 9, y^2 = x].$$

A refinement of this could be:

$$y : [0 \leq x \leq 9, y^2 = x \wedge y \geq 0]$$

since the user now knows more about the final state without losing any information previously given. This is known as *strengthening the postcondition* and is one of the main techniques of refinement. It can be expressed in a law as follows:

strengthen postcondition If $post' \Longrightarrow$[1] $post$ then

$$w : [pre, post] \sqsubseteq w : [pre, post'].$$

where $\sqsubseteq$ means 'is refined by'.

Another refinement would be to weaken the precondition so that the old precondition still implies the new. With reference to the earlier specification, an example of this would be:

$$y : [0 \leq x, y^2 = x \wedge y \geq 0]$$

which can be described by a law as follows:

weaken precondition If $pre \Longrightarrow pre'$ then

$$w : [pre, post] \sqsubseteq w : [pre', post].$$

[1]A $\Longrightarrow$ B means that 'for all states, if A is true then so is B'.

A refinement which develops a specification into code is the law of refinement for assignments. An assignment can be written as $w := E$ and this changes the state so that w is mapped to the value E. The law is as follows:

assignment If $pre \Longrightarrow post[w \setminus E]$ then

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

The formula $post[w \setminus E]$ is obtained by replacing in *post* all occurrences of w by E. Using this law, the refinement of the earlier example could appear as $y := \sqrt{x}$.

3.2.2 IBM Hursley Park Method

The IBM Hursley Park Method uses the specification language of Z. This formal language has developed over the last decade and is well described in the books, *The Z Notation* [79] and *An Introduction to Formal Specification and Z* [72]. There are several ways for developing Z specifications into programs but the IBM Hursley Park Method is the most well known.

This method begins with the description of the abstract states of the system using Z, as shown in figure 3.1. Schemas are drawn to represent the abstract elements and their relationships to one another. Preconditions and postconditions are again represented but usually in the form of schemas, with schema names as the frames in these cases.

The first step in the refinement of this abstract representation is to think of a more concrete version of the system and to develop schemas using concrete (instead of abstract) elements. Conditions relating the two versions of the system can be described in relations expressed logically and these are put together in a schema known as *Retrieve* (see fig. 3.1). This Retrieve schema relates all abstract states to concrete ones and can be used to *refine* all the operations based upon abstract elements.

Besides using the *Retrieve* schema, some theorems need to be proved to show that the refinement is correct. The three main theorems are as follows:

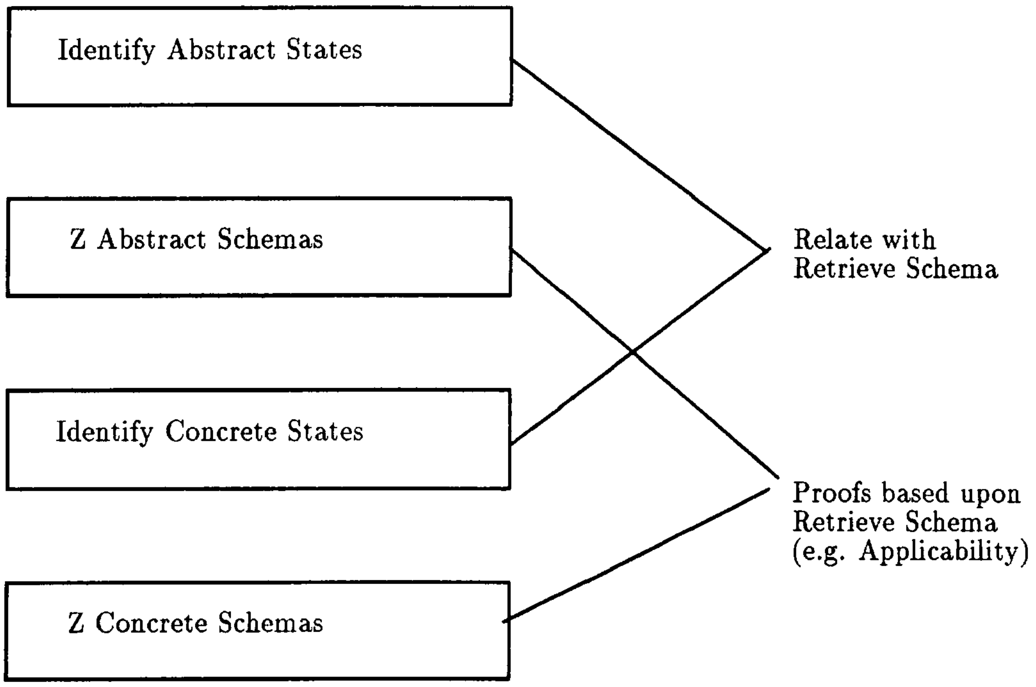


Figure 3.1: The IBM Hursley Park Method

1. Initial States Theorem

   Every initial concrete state corresponds under the retrieve relation to an initial abstract state.

2. Applicability Theorem

   Whenever the abstract operation is applicable in a given abstract state then the concrete operation must be applicable in any concrete state representing the abstract state.

3. Correctness Theorem

   Proves that concrete operations behave correctly. This is done by proving that the image of the concrete operation, as seen through the retrieve function, is consistent with the abstract operation.

Further details on these theorems and examples can be found in [72] and [45]. Once these theorems have been proved, rules of refinement similar to Morgan's laws of the refinement calculus can be used for what is known as *operation decomposition.* These rules include the introduction of local variables, reducing the frame, assignment to simple variables and alternation. A combination of rules such as these are used to develop concrete schemas into forms which can be implemented.

3.3 Refinement via Transformations

The idea of using successive transformations to develop an abstract specification into an efficient program began in the 1970's with the works of Bauer and Griffiths. In 1973 Bauer presented a series of lectures which stressed the need for a unified conceptual basis of programming [6]. Griffiths uses this as the basis for his technique of producing programs with successive transformations [37] [38]. This technique usually involves identifying a recursive version of the program but then eliminating the recursion to obtain an efficient, iterative text. This is one of the refinement strategies which was used with the method for producing maintainable software.

Since this early work, different transformations have been developed for different purposes and implemented in various transformation systems. These were surveyed in chapter 2 so this section will focus on the transformations used for this particular research and on how these were used as a refinement technique.

The transformations used by the method originate from Ward's thesis [81] and many have been implemented in a tool called the *Maintainer's Assistant* [82] for the purposes of restructuring code. This tool was described in the section on transformation systems and one of the aims of this research was to adapt the use of the tool to developing programs from specifications.

The Maintainer's Assistant has a large library of transformations which are basic *rules* for changing programs written in a *wide spectrum language* (WSL) while preserving semantics. That is, it contains a number of ways in which parts of programs can be rewritten without altering their meaning. Programs which can be manipulated by the Maintainer's Assistant must be written in WSL, a language designed to cover both low and high level constructs and upon which Ward's transformations were built. For a full description of WSL see [83] and appendix A of this thesis.

Using WSL and extending the set of transformations provided a way of refinement through the use of program transformations. The application of transformations or sets of transformations can follow determined patterns known as **strategies.** It is particularly difficult to categorise refinement strategies since they vary according to the technique of refinement in use and to the particular aim of the strategy. At times several strategies might be used together to form one large strategy, or only part of a strategy used at a particular moment. It is also possible to say that refinement follows only one strategy: decomposing a problem into subproblems, simplifying them and reassembling the implementable components in an efficient manner.

An attempt will be made to list and categorise the main strategies used when refining specifications with program transformations. This is a general list which has been composed from various texts upon the application of program transformations (see [67] [70] [19]) and the use of the Maintainer's Assistant. Strategies used for refinement vary from the application of particular laws to a defined set of actions which determine a particular strategy. The following list will begin with the simplest strategies (based upon the application of laws of logic and algebra), will be followed by basic transformations which can be used as strategies and will conclude with complex strategies which might be used (based upon compositions of transformations and/or algebraic laws).

3.3.1 Simple Strategies

The following strategies are simple algebraic laws used to refine program specifications:

- **Laws about predicates.** These are laws determined in predicate calculus such as existential and universal quantification. Described in many introductory books to logic and used with both refinement calculi and methods.
- **Axiomatic Laws of Language Definition.** Laws determined by axioms of the particular specification or programming language used.
- **Basic Set Theoretic Laws.** Laws about sets (such as algebraic properties of set operators).

- **Axioms of Underlying Data Types.** These are laws involving the data types used by the specification.

3.3.2 Simple Transformations

These are simple strategies formed by the application of particular transformations to specifications, code or intermediate refinement levels. There are many such transformations (see Appendix B for those in the Maintainer's Assistant), but examples are:

- **Simplification.** Transformations used to write a simpler version of the specification/code or part of specification/code selected. The Maintainer's Assistant has a specific transformation which does this, known as **Simplify**.
- **Re-arrangement.** Transformations used for changing the order in which specification/code is written. An example of this is the Maintainer's Assistant **Swap-With-Next** transformation which swaps the next part with the selected part (whether that "part" be a variable, name, action, definition, condition, expression or statement).
- **Relations between Specification Constructs.** Often parts of specifications can be written in different ways which mean the same thing. An example from the Maintainer's Assistant is when guarded commands are introduced by the transformation **Separate-Cases**.
- **Transformations for Conditional and Guarded Expressions.** There are many transformations for conditional and guarded statements. For example, the Maintainer's Assistant has one for combining cases in a conditional statement, known as **Partially-Join-Cases**. There are also ones for applying cases from conditions, merging similar conditions and removing redundant conditions.

3.3.3 Case Introduction

This is often a basic strategy within several other strategies and usually involves the introduction of new variables and the application of properties of the basic data types.

The aim is to introduce cases so that the problem can be subdivided. In the Maintainer's Assistant, this strategy would combine the use of the editor with the strategies in 3.3.1.

3.3.4 Introduction of Invariants

This strategy involves introducing a predicate which is always true. This is also often used within other strategies and depends upon whether there is something which can be added to a specification which is always true no matter what transformations are later applied. The Maintainer's Assistant actually has a transformation to perform this strategy, known as **Insert-Invariant**.

3.3.5 Embedding

Embedding involves solving a more general problem with the original as a special case. There a three possible classifications of this:

- Embedding of data types.
- Embedding of domain.
- Embedding of range.

All of these rely upon the user's knowledge and ability with the editor (for the Maintainer's Assistant).

3.3.6 Unfold/Fold

This is a larger refinement strategy which can incorporate some of the previous strategies. *Unfold* means to replace a function call by the body of a function and to replace formal parameters with the actual ones. *Fold* is the formation of a (recursive) call from an expression which is the instance of a function body or it can be the introduction of an identifier for a certain expression. Strategies which can be used to do this include:

- Apply axioms and theorems of underlying data types.
- Introduce new function declarations (**define**).
- Evaluate function call for concrete values (**instantiate**).
- Introduce new name for expression (**abstract**).

The Maintainer's Assistant has a special facility for using this *Unfold/Fold* strategy.

3.3.7 Introduce Recursion

To simplify an expression it is often a good strategy to introduce recursion, simplify the problem as it now stands and then to remove the recursion. This is one of the most common refinement strategies and has been described already. The next chapter illustrates the use of transformations from the Maintainer's Assistant for introducing recursion and removing it, although complex transformations do now exist for performing these strategies (**Loop** $\rightarrow$ **Recursion** , **Recursion** $\rightarrow$ **Loop**, **Remove - Recursion**).

3.3.8 Divide-and-conquer

Divide-and-conquer is another general strategy which is very common. It often includes the use of several other strategies which have already been mentioned. This involves determining the type of function (identifying qualities such as monotonicity), deciding where the function might be false and solving the problem for the part which is true. That is, the bounds within which the problem is true are restricted and the problem is solved for a smaller range. A good example of the use of this strategy is the first case study for testing the method, the integer square root, described in the next chapter.

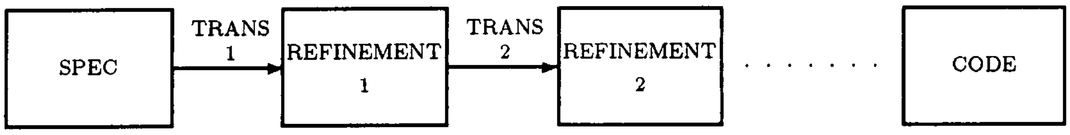


Figure 3.2: Sample derivation history

3.4 Description of the Method

The starting point is an abstract specification of the problem written in a formal language such as Z. Successive transformations can be applied to this until an executable version is achieved (see figure 3.2).

The developer has complete control over which transformation is next applied but can use a variety of tactics. For instance, (s)he might wish to get an inefficient version of the implementation quickly before making this more efficient with the automatable transformations from the *Maintainer's Assistant.*

A wide spectrum language (see Appendix A) is needed to express the specification, implementation and intermediate forms so that these successive transformations can be applied. The algebraic laws described in 3.3.1 will also be needed for stages in the development of the executable. The combination of transformations and axioms, specification, executable and intermediate forms make up the derivation history. This derivation history forms the basis of the method and is continually updated when changes are needed.

Stated formally, the derivation history appears as described at the beginning of this chapter:

$$S_0 \xrightarrow{a} D_1 \xrightarrow{b} D_2 \xrightarrow{c} D_3 \xrightarrow{d} C_1$$

where S_0 represents the initial specification, D_1, D_2, D_3 the different versions of the design as the specification is transformed into code and C_1 the first version of the implementation. The arrows represent the application of transformations and a, b, c, d the different transformations which are applied.

The derivation history records each transformation applied in its sequential order from specification to implementation. When a change in the requirements is requested, this change is first made at the specification level or at some further point in the development, depending upon how specific the change is. If the change alters data from the specification level, then the change must be applied at this level. If it does not affect data in the specification, but only a specific case introduced in an intermediate stage between specification and code, then the change can be applied at this stage instead.

Ideas of how the derivation history appears after different types of changes were presented at the beginning of this chapter. The method itself is based upon the re-use of this derivation history. This is done by applying transformations to an altered specification until the old derivation history can be applied to part of this specification (see section 3.1). It is difficult to describe the method without concrete examples and so the next two chapters illustrate the application of the method to two case studies. These illustrate how the method works with code intensive and then data intensive examples.

3.5 Summary

This chapter described how ideas for the method arose from the use of transformations for restructuring in the *Maintainer's Assistant* and from the papers criticised in chapter 1. The method was applied to two case studies originally developed through other refinement techniques, so these techniques were summarised in this chapter. That is, basic ideas of Morgan's refinement calculus [59] and the IBM Hursley Park method [79] [72] were presented. The method uses Ward's transformations [82] for refinement and so a description was provided of how this might work. The chapter concluded with a general description of the method itself.

Chapter 4

Integer Square Root Problem

The method was first tested on a simple case study: the integer square root problem described in the book [59]. This case study was selected because it was a simple way in which to test the method for a function based specification. That is, it provides indications of how the method should work on a small mathematical problem. The same tactics can be used within larger scale specifications, such as the second case study of a library system. Combining the approaches to these different types of specification provides a more general method for producing maintainable software which can be tested on real specifications.

In the first case study, Morgan uses his refinement calculus to develop a specification of the square root problem into executable code. The refinement steps are based upon concepts of strengthening postconditions, weakening preconditions, compositions and invariants. The main technique is to refine the specification until it consists of composite statements which can each be refined separately. These partial specifications are each developed and the code is found by composing the bottom branches of the 'development tree' (see figure 4.1).

A description of the problem and the method by which it was developed using Ward's transformations are first described. So that this work can be understood better, descriptions of WSL and the transformations which can be used are provided in the appendices to this thesis.

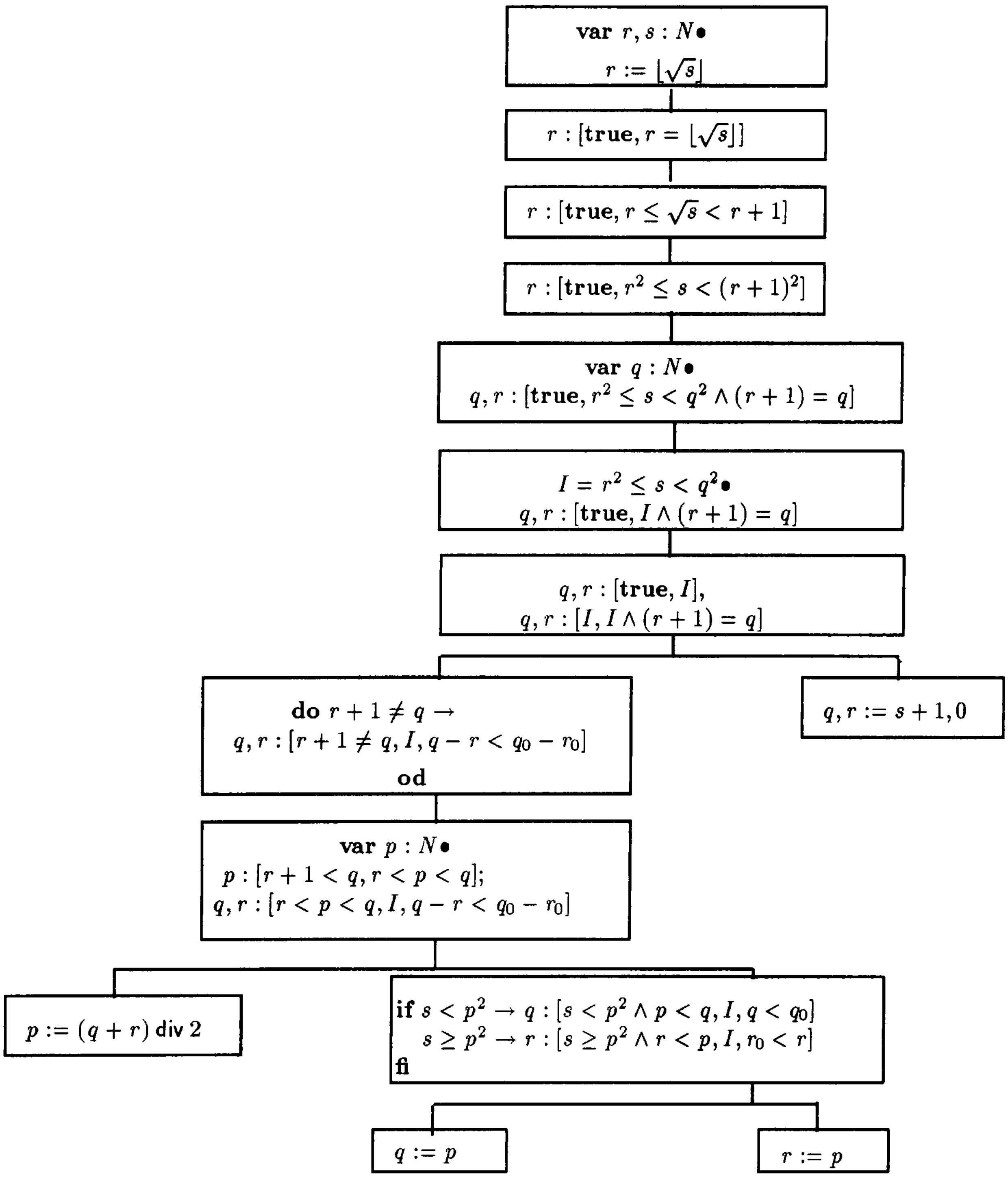

var $r, s : N \bullet$
$r := \lfloor\sqrt{s}\rfloor$
$r : [\mathbf{true}, r = \lfloor\sqrt{s}\rfloor]$
$r : [\mathbf{true}, r \leq \sqrt{s} < r + 1]$
$r : [\mathbf{true}, r^2 \leq s < (r+1)^2]$
var $q : N \bullet$
$q, r : [\mathbf{true}, r^2 \leq s < q^2 \wedge (r+1) = q]$
$I = r^2 \leq s < q^2 \bullet$
$q, r : [\mathbf{true}, I \wedge (r+1) = q]$
$q, r : [\mathbf{true}, I]$,
$q, r : [I, I \wedge (r+1) = q]$
do $r + 1 \neq q \rightarrow$
$q, r : [r + 1 \neq q, I, q - r < q_0 - r_0]$
od
$q, r := s + 1, 0$
var $p : N \bullet$
$p : [r + 1 < q, r < p < q]$;
$q, r : [r < p < q, I, q - r < q_0 - r_0]$
$p := (q + r)$ div 2
if $s < p^2 \rightarrow q : [s < p^2 \wedge p < q, I, q < q_0]$
$s \geq p^2 \rightarrow r : [s \geq p^2 \wedge r < p, I, r_0 < r]$
fi
$q := p$
$r := p$

Figure 4.1: Morgan's refinement of the square root problem

4.1 Description

This is a simple problem for finding the greatest integer not exceeding the square root of a natural number. Given a natural number s, say, one needs to find a natural number r which will not exceed $\sqrt{s}$. The specification of this could be:

$$\textbf{var}\ r, s : \mathbf{N} \bullet r := \lfloor\sqrt{s}\rfloor$$

where we assume that neither $\surd$ (defined as "the square root of") nor $\lfloor\ \rfloor$ (defined as "the largest integer not greater than") is code. So the development of code from this would involve removing these two symbols and producing a version of this specification which might be executable. The application of Ward's transformations will be illustrated in the following section through the use of the wide spectrum language, WSL, described in appendix A.

4.2 Refinement Using Transformations

There are a variety of transformations which can possibly be used for the development of code from the specification of this problem (see Appendix B). The method relies upon selecting the fewest which will develop the specification into an executable and then using additional transformations to make the code more efficient.

The first refinement stage is to use the definitions of the functions $\surd$ and $\lfloor\ \rfloor$ to rewrite the specification as:

$$\{\ s \in \mathbf{N}\ \};\ r := \max\ \{r \in \mathbf{N} \mid r^2 \leq s\ \}$$

This done via the transformation **Substitute-And-Delete** which replaces a function with its definition and removes the 'name' of the function.

The next stage is to set an upper bound so that the *max* function can be implemented.

Since $(s+1)^2 > s$ for all $s \in \mathbf{N}$, the bounds for which r is true can be set (using the **Make-And-Use-Assertions** transformation):

$$\{\ s \in \mathbf{N}\ \};\ r := \max\ \{r \in \mathbf{N} \mid 0 \leq r \leq s \wedge r^2 \leq s\ \}$$

The transformation **Make-loop** can be applied to make the specification executable. This implementation is as follows:

```
{ $s \in \mathbf{N}$ }; $r := 0$;
for $i := 1$ to $s$ do
        if $i^2 \leq s$ then $r := i$ fi od
```

Since this is executable, it is possible to claim that the refinement is complete. However, at this stage such a loop would involve s steps of calculation, which would be very inefficient for large values of s. For more efficiency the number of calculation steps must be reduced and extra transformations need to be applied in order to do this.

The best approach for reducing these steps is to minimise the search space. This is carried out using a "divide and conquer" strategy where special cases are first removed, the search space divided in two and a determination of which half contains the result made.

For the strategy to work, the precondition that the predicate involved is monotonic is needed so that there is some cut-off value x when the predicate becomes false and remains false for all values greater than x. The predicate that we are dealing with is $r^2 \leq s$ which will be labelled as $P(r)$ and its monotonicity implies that $P(l)$ is true and $P(h)$ is false for some values l, h. Since $P(0)$ can be proved true, 0 is represented by l in earlier versions of the specification and a general form of the specification becomes:

$$SPEC(l,h) \underline{\triangle}\ \{\ l,h \in \mathbf{N} \wedge l < h \wedge P(l) \wedge \neg P(h)\ \};\ r := \max\ \{\ r \in \mathbf{N} \mid l \leq r < h \wedge P(r)\ \}$$

where the first part is the new precondition based upon $P(r)$ being monotonic.

The case $h = l + 1$ is taken out since this satisfies the first assertion and makes the assignment of r trivial. That is, if $h = l + 1$ then the assignment to r would become $r :=$ max $\{ r \in \mathbf{N} \mid l \leq r < l + 1 \wedge P(r)$ which would mean that $r = l$ and so the $SPEC(l, h)$ could simplify to $r := l$.

To set up this new case, an **if** statement is introduced out of nothing by using the transformation **Add-Entire-Loop** :

```
$SPEC(l,h) \approx \{ l, h \in \mathbf{N} \wedge l < h \wedge P(l) \wedge \neg P(h) \}$ ;
                 **if** $h = l + 1$ **then** $SPEC(l,h)$
                                  **else** $SPEC(l,h)$ **fi**
```

where $\approx$ means "is transformed to".
If $h = l + 1$ then $SPEC(l, h)$ becomes $r := l$ and so the following is derived from the transformation **Apply-Condition-To-Next**:

```
$SPEC(l,h) \approx \{ l, h \in \mathbf{N} \wedge l < h \wedge P(l) \wedge \neg P(h) \}$ ;
                 **if** $h = l + 1$ **then** $r := l$
                                  **else** $SPEC(l,h)$ **fi**
```

The range is now divided into two by using the editor to introduce new variables m and m' and the transformation **Add-Entire-Loop**:

```
$SPEC(l,h) \approx \{ l, h \in \mathbf{N} \wedge l < h \wedge P(l) \wedge \neg P(h) \}$ ;
                 **if** $h = l + 1$ **then** $r := l$
                                  **else var** $m := m'.(l < m' < h)$ :
                                           **if** $P(m) \rightarrow SPEC(m,h)$
                                           $\Box \neg P(m) \rightarrow SPEC(l,m)$ **fi end fi**
```

To simplify the specification so that it can be implemented a commonly used technique is performed of introducing a recursion, simplifying and then removing the recursion. First, the specification is rewritten as a procedure so that recursion can now be included (using the transformation **Make-Proc**):

```
$SPEC \approx spec(0, s+1)$
where
proc $spec(l,h) \equiv \{\ l,h \in \mathrm{N} \wedge l < h \wedge P(l) \wedge \neg P(h)\ \}$ ;
                    if $h = l+1$ then $r := l$
                                   else var $m := m'.(l < m' < h)$ :
                                             if $P(m) \rightarrow spec(m,h)$
                                             $\square \neg P(m) \rightarrow spec(l,m)$ fi end fi.
```

To reduce $h-l$ as much as possible, the maximum values of $h-m$ and $m-l$ are minimised by refining $m := m'.(l < m' < h)$ to $m := \lfloor (l+h)/2 \rfloor$ and turning m into a global variable:

```
$SPEC \approx$ var $m : spec(0, s+1)$ end
where
proc $spec(l,h) \equiv$ if $h = l+1$ then $r := l$
                                   else $m := \lfloor (l+h)/2 \rfloor$ ;
                                        if $P(m) \rightarrow spec(m,h)$
                                        $\square \neg P(m) \rightarrow spec(l,m)$ fi fi.
```

The parameters are replaced by global variables:

```
$SPEC \approx$ var $m, l := 0, h := s+1 : spec$ end
where
proc $spec \equiv$ if $h = l+1$ then $r := l$
                         else $m := \lfloor (l+h)/2 \rfloor$ ;
                              if $P(m) \rightarrow l := m;\ spec$
                              $\square \neg P(m) \rightarrow h := m;\ spec$ fi fi.
```

This is a tail-recursion which is converted to a **while** loop (using the transformation **Loop → While**):

```
$SPEC \approx$ var $m, l := 0, h := s + 1$ :
        while $h \neq l + 1$ do
            $m := \lfloor (r + h)/2 \rfloor$;
            if $P(m)$ then $l := m$
                    else $h := m$ fi od end
```

Removing l by using r to represent it (transformation **Change-Local-Variable**) , gives:

```
$SPEC \approx$ var $m, h := s + 1$ :
        $r := 0$
        while $h \neq r + 1$ do
            $m := \lfloor (r + h)/2 \rfloor$;
            if $P(m)$ then $r := m$
                    else $h := m$ fi od end
```

and substituting $m^2 \leq s$ for $P(m)$ (transformation **Replace-With-Value**), produces the executable code:

```
$SPEC \approx$ var $m, h := s + 1$ :
        $r := 0$
        while $h \neq r + 1$ do
            $m := \lfloor (r + h)/2 \rfloor$;
            if $m^2 \leq s$ then $r := m$
                    else $h := m$ fi od end
```

This ends the refinement of the square root problem, producing an executable form of the specification which can be calculated using $\log s$ steps instead of the s steps needed by the code produced using the fewest possible transformations.

4.3 Comparisons with Refinement Calculus

The development of the square root specification using Morgan's refinement calculus is illustrated in figure 4.1. The code produced using Ward's transformations is similar to that developed using Morgan's refinement calculus, which, by collecting the bottom leaves of figure 4.1, appears as:

```
|[ var $q : \mathbf{N}$ •
   $q, r := s + 1, 0$;
   do $r + 1 \neq q \rightarrow$
       |[ var $p : \mathbf{N}$ •
           $p := (q + r) \operatorname{div} 2$;
           if $s < p^2 \rightarrow q := p$
              $s \geq p^2 \rightarrow r := p$
           fi
       ]|
   od
]|
```

The refinement steps which produced this are based upon laws of logic applied sequentially to the specification. Unlike our method which develops new versions of the software at each transformation, the refinement calculus works on partial specifications. This means that the specification is refined to the point at which various parts can be separated out and refined individually. As displayed in figure 4.1, the general refinement process eventually resembles a tree, with the development of the separate parts forming branches of that tree. The final version of the code is formed by collecting the last leaves of these branches.

Figure 4.1 illustrates one of the problems with the refinement calculus. Separating parts of the specification makes it eventually difficult to organise a derivation history of the complete specification so that "steps" of the refinement might be listed. An attempt at a derivation history would need to be organised in a tree format but this leads to problems in understanding what the final version of the code actually is. Mistakes in 'collecting' the final branches and assembling them into the code (as displayed above) can also arise. A further problem is that developing branches of the specification restricts which refinement rules can be applied at each point and makes the refinement of each branch independent of other branches.

The method described by this thesis (i.e. using Ward's transformations for refinement) does not involve developing branches in such a way and so is easier to automate, as the derivation history is organised in a sequential order. Automation of the method is further assisted by the fact that many of the transformations are already implemented in the Maintainer's Assistant. There is no complete tool for automating Morgan's refinement calculus.

Another advantage with using transformations is that additional transformations can be added to the refinement path for more efficiency and maintainability. The development of the square root with Morgan's refinement calculus illustrates that developing partial specifications eventually leads to a dead end. There comes a point when a branch has been developed as much as possible and no other refinement rules can be applied. Making the code more efficient or maintainable would involve altering earlier stages of the development. This could affect all the lower parts of the branch and so reusing the refinement path proves difficult.

For instance, it is possible to calculate the value of the executable in less than half the log s steps needed in section 4.2 if we use an estimate of the root to get a better upper bound for the search using Ward's transformation approach. Transformations which involve this can be applied at the end, thus extending the refinement path for efficiency purposes. Morgan's refinement calculus does not allow for a similar extension to its development of a problem.

A better comparison of the two techniques of refinement can be made by making changes to the specification and investigating the effect this has upon the development of the code.

4.4 Application and Implication of Changes

Many changes were made to the specification of the integer square root problem to verify whether the derivation history could be used again or if not, investigate the best method for changing the history. The two which best illustrate the results are:

1. Change from the square root to the cube root of the natural number s.
2. Find the greatest integer not exceeding the square root of a real number instead of a natural number.

Hence the respective new versions of the specification will be:

1. $\mathbf{var}\ r, s : \mathbf{N} \bullet r := \lfloor \sqrt[3]{s} \rfloor$.
2. $\mathbf{var}\ r : \mathbf{Z};\ s : \mathbf{R} \bullet r := \lfloor \sqrt{s} \rfloor$.

The first change examines how internal changes affect the refinement (changing the meaning of terms used within the problem) while the second involves external changes (altering the range over which the problem acts). The following sections will describe the effect that each change has upon the development of the specification using the two methods of refinement.

4.4.1 Change to Cube Root

Ward's Transformations

Changing the specification for cube roots instead of square roots has little effect upon the development of the executable using the method. The specification to be used remains:

$$SPEC(l,h) \underline{\Delta} \{\ l,h \in \mathbf{N} \wedge l < h \wedge P(l) \wedge \neg P(h)\ \};\ r := \max \{\ r \in \mathbf{N} \mid l \leq r < h \wedge P(r)\ \}$$

only $P(r)$ represents $x^3 \leq s$ now. The case $h = l + 1$ can still be taken out using the same transformations as before:

```
$SPEC(l,h) \approx \{\ l,h \in \mathbf{N} \wedge l < h \wedge P(l) \wedge \neg P(h)\ \}$ ;
                  if $h = l + 1$ then $SPEC(l,h)$
                                  else $SPEC(l,h)$ fi
```

... and the same transformations used so that the search space is divided:

```
$SPEC(l,h) \approx \{\ l,h \in \mathbf{N} \wedge l < h \wedge P(l) \wedge \neg P(h)\ \}$ ;
                  if $h = l + 1$ then $r := l$
                                  else var $m := m'.(l < m' < h)$ :
                                          if $P(m) \rightarrow SPEC(m,h)$
                                          $\square \neg P(m) \rightarrow SPEC(l,m)$ fi end fi
```

The technique of introducing a recursion, simplifying and then removing the recursion can again be followed (using the same transformations), so that the specification becomes:

```
$SPEC \approx$ var $m, h := s + 1$ :
          $r := 0$
          while $h \neq l + 1$ do
                $m := \lfloor (r + h)/2 \rfloor$;
                if $P(m)$ then $r := m$
                          else $h := m$ fi od end
```

The only difference is the last step, which involves the substitution of $m^3 \leq s$ for $P(m)$ (instead of $m^2 \leq s$):

```
$SPEC \approx$ var $m, h := s + 1$ :
          $r := 0$
          while $h \neq l + 1$ do
                $m := \lfloor (r + h)/2 \rfloor$;
                if $m^3 \leq s$ then $r := m$
```

else $h := m$ **fi od end**

However, the same transformation is involved with this last step (**Replace-With-Value**) and so the complete derivation history has been used again to transform the altered specification into code. The change from square to cube root has affected the content involved when substituting and removing $P(r)$ but no new transformations are needed.

Morgan's Refinement

Morgan's refinement also involves a change of the definition of the function involving s. It is still possible to introduce q and attempt to bring it close to r. The invariant changes in this case so that now: $I = r^3 \leq s < q^3$. The rest of the refinement carries on as before until $s < p^3$ and $s \geq p^3$ is introduced. This involves rewriting some of the stages to adapt to the change itself. Thus it requires a maintainer's knowledge of the change and his ability to carry out the refinement based upon this knowledge. So for this example Ward's refinement technique is much better for adapting to changes made to the specification since all transformations can be applied in the same sequence (with only the value of $P(r)$ changing). Figure 4.2 illustrates the new version of the refinement path using Morgan's calculus (with arrows pointing to the parts which needed to be changed).

4.4.2 Change to real numbers

Relaxing the restriction on the range from natural numbers to all real numbers affects the way in which the specification of the square root is developed. Changing the specification so that all real numbers can be involved greatly increases the range of values of s and r. The nature of this change makes it difficult to apply Ward's transformations or Morgan's refinement calculus immediately since the meaning of the square root function could be drastically affected. Before analyzing any development of code, it is necessary to examine the effect such a change has upon the specification itself.

Since s is a real number, it could be a positive or negative integer, zero or a rational

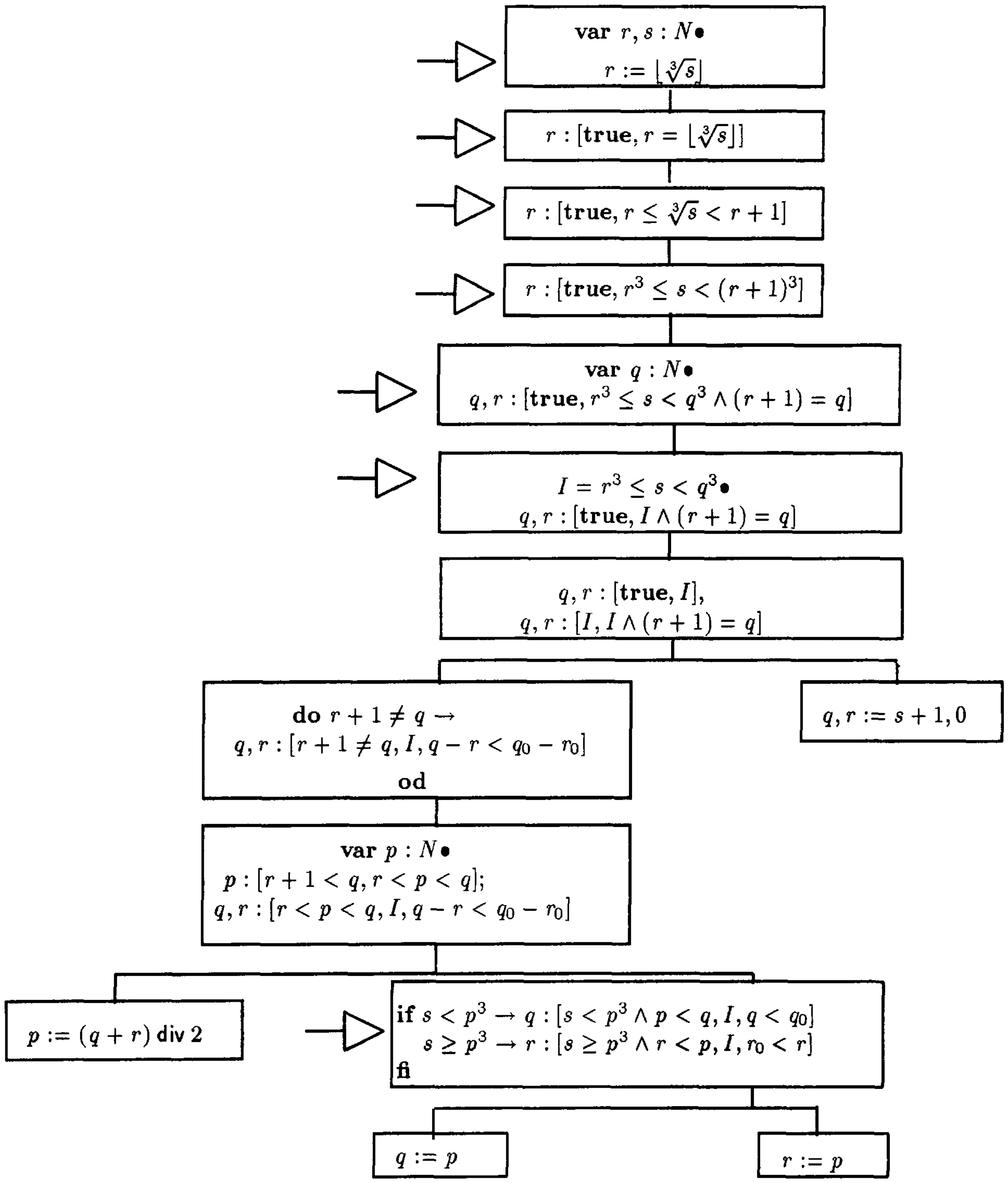

var $r, s : N \bullet$
$r := \lfloor \sqrt[3]{s} \rfloor$
$r : [\mathbf{true}, r = \lfloor \sqrt[3]{s} \rfloor]$
$r : [\mathbf{true}, r \le \sqrt[3]{s} < r + 1]$
$r : [\mathbf{true}, r^3 \le s < (r+1)^3]$
var $q : N \bullet$
$q, r : [\mathbf{true}, r^3 \le s < q^3 \wedge (r+1) = q]$
$I = r^3 \le s < q^3 \bullet$
$q, r : [\mathbf{true}, I \wedge (r+1) = q]$
$q, r : [\mathbf{true}, I],$
$q, r : [I, I \wedge (r+1) = q]$
do $r + 1 \ne q \rightarrow$
$q, r : [r + 1 \ne q, I, q - r < q_0 - r_0]$
od
$q, r := s + 1, 0$
var $p : N \bullet$
$p : [r + 1 < q, r < p < q];$
$q, r : [r < p < q, I, q - r < q_0 - r_0]$
$p := (q + r)$ div 2
if $s < p^3 \rightarrow q : [s < p^3 \wedge p < q, I, q < q_0]$
$s \ge p^3 \rightarrow r : [s \ge p^3 \wedge r < p, I, r_0 < r]$
fi
$q := p$
$r := p$

Figure 4.2: Change to Morgan's development of the square root problem

number. Positive integers are natural numbers anyway, so the specification and its development will remain the same as before for this case. If s was a negative integer, then r would become the largest integer not more than $\sqrt{s}$ by the definition of the floor function and the $\sqrt{s}$ in this case would be a complex number with no real part to it (i.e. $0 + xi$ for some $x \in \mathbf{R}$). Thus r must be the largest integer ≤ 0, i.e. 0 itself.

If s was a rational number then it can be described by a fraction. The square root of a positive rational number is also a positive rational number, so r would still be the largest integer not more than this rational number. The square root of a negative rational number will again be a complex number with no real part, so again r must be 0. The same thing would apply to other positive and negative real numbers.

Hence a new version of the specification would be as follows:

$$\mathbf{var}\ r : \mathbf{N};\ s : \mathbf{R} \bullet r := \lfloor\sqrt{s}\rfloor$$

since we have deduced that r must be a natural number for all cases of s.

Ward's Transformations

Using what is known about the values of s and r, the specification is split into two cases:

$$\mathbf{var}\ r : \mathbf{N};\ s : \mathbf{R} \bullet (s \geq 0 \Rightarrow r := \lfloor\sqrt{s}\rfloor;\ s < 0 \Rightarrow r := 0)$$

Using Ward's syntax, the specification will appear as:

$$\mathit{SPEC} = \{\ s \in \mathbf{R} \wedge s \geq 0\ \};\ r := \lfloor\sqrt{s}\rfloor;\ \{\ s \in \mathbf{R} \wedge s < 0\ \};\ r := 0$$

The same transformations used in 4.2.1 can be applied to the first half (i.e. $\{\ s \in \mathbf{R} \wedge s \geq 0\ \};\ r := \lfloor\sqrt{s}\rfloor$) so that the following results:

$\{\ s \in \mathbf{N} \wedge s \geq 0\ \};$

```
var $m, h := s + 1$ :
    $r := 0$
    while $h \neq l + 1$ do
          $m := \lfloor (r + h)/2 \rfloor$;
          if $m^2 \leq s$ then $r := m$
                    else $h := m$ fi od end ;
$\{ s \in \mathbf{N} \wedge s < 0 \}$; $r := 0$
```

Now the two assertions can be transformed so that there are two loops which are merged together (using the **Make-loop** and **Merge** $\rightarrow\rightarrow$ transformations) to give:

```
var $m, h := s + 1$ :
    if $s \geq 0$ then
              $r := 0$
              while $h \neq l + 1$ do
                    $m := \lfloor (r + h)/2 \rfloor$;
                    if $m^2 \leq s$ then $r := m$
                              else $h := m$ fi od
    else $r := 0$ fi end
```

Finally a transformation which takes out the common assignment (the $\leftarrow\leftarrow$ **Take-Out** transformation applied to the complete if-then statement, followed by a **Simplify** transformation for removing the redundant loop) results in the following executable code:

```
var $m, h := s + 1$ :
    $r := 0$
    while $h \neq l + 1 \wedge s \geq 0$ do
          $m := \lfloor (r + h)/2 \rfloor$;
          if $m^2 \leq s$ then $r := m$
                    else $h := m$ fi od end
```

Thus the transformations used before are still applied by splitting the specification into two so that the first part was similar to the original specification and the rest required simple transformations to integrate it with the original code. The method of re-using the original derivation history is still applicable, only it applies to only the first part of the specification. This application of the method is the idea referred to in section 3.1, where the possibility of transforming the altered specification so that one part was identical to the original specification was introduced. The original derivation history could be applied to this part, new transformations to the other and a final set of transformations could merge the two parts. This possibility was verified with this example and will be discussed in the Results chapter (section 6.1).

Morgan's Refinement

Morgan's refinement of the problem is also dealt with in the same way. The specification in this case could be written as:

$$
\begin{array}{l}
\textbf{var } r : \mathbf{N};\ s : \mathbf{R}\ \bullet \\
\qquad r : [s \geq 0, r = \lfloor \sqrt{s} \rfloor]; \\
\qquad r : [s < 0, r = 0];
\end{array}
$$

Refinements can be carried out as before on the first two lines of the specification with the only difference being the value of the pre-condition (instead of **true** it is now $s > 0$). This does not affect the refinements since they rely on *some* pre-condition existing but do not depend upon its value at all. Its effect is only seen when the statement $q, r : [s > 0, I];$ needs to be refined. Originally this was $q, r : [\mathbf{true}, I];$ and was rewritten as $q, r := s + 1, 0$ due to the value of I. In the new version, rewriting I in the statement gives us $q, r : [s > 0, r^2 \leq s < q^2]$. This can be refined to:

$$
\begin{array}{l}
q, r : [s \geq 0, 0 \leq s < q^2]; \\
\quad r := 0
\end{array}
$$

which can then be refined to:

$$q, r : [s \geq 0, s < q^2];$$
$$r := 0$$

Now since $s \geq 0$ then $(s+1)^2 > s$ so q can be assigned to $s+1$ and the following is derived:

$$\{\ s \geq 0\ \};$$
$$q := s + 1;$$
$$r := 0$$

However there is now a problem with the values which q can have. Initially, q was assigned a value of $r+1$ and so was obviously a natural number since r was a natural number. By assigning q to $s+1$, its range has been extended to cover real numbers since s is real. This means that one must return to the earlier steps of the refinement and redefine q as a real number.

This shows one of the advantages that the method has over the Morgan technique. Ward's transformations do not depend so much upon the data type or upon the creation of local variables for developing code. When using Morgan's refinement of the square root problem again for changes to this problem, we need to review the definitions of any of the variables defined during the development of the problem. This means that this technique would be more difficult to apply to a changed problem than would be the case with Ward's transformations.

In fact, at this point in the development of the code, we will need to return to a much earlier stage of the development and redefine q as a real number. We will also need to treat r as a real number due to the definition of q. This will create a number of other problems in the development of the code, since r appears in every stage of the refinement. The variable p will also need to be redefined as a real number.

Another advantage of the method highlighted by this example is that it is more difficult to determine the impact of change with Morgan's refinement calculus. This is due to the development of the specification following a tree format. When a change is made one must be very careful to trace any branches which have been affected and collect all the refined parts in a way which does not adversely affect the other ares of the code.

This is not a problem for the method of using transformations. If the specification is small enough then transformations are applied to the *complete* specification and its intermediate stages so the final product is a single block of code which directly implements the specification. For larger specifications, transformations are applied so that the specification can be separated into smaller independent parts. Each part is transformed into code but then transformations can unite the different parts together again and the whole specification can be implemented.

4.5 Summary

This chapter described how the method of using transformations for refinement of a simple problem worked and the advantages of this for perfective maintenance. The simple problem was a case study of finding an integer square root originally developed through a refinement calculus in the book [59]. A comparison of refinement via transformations and refinement via a calculus was made. Criteria for this comparison were the general advantages of the transformation approach when changes needed to be implemented.

Chapter 5

Library Case Study

The library system is formally represented by a complex abstract state and a set of transactions involving this state. The specification which describes this system originates from a technical paper by King and Sørensen, *From specification, through design, to code: a case study in refinement* [45] and reference should be made to this for a complete description of the library system and its transactions.

A brief description of the system and the transformations which were applied will be provided before the application and implication of changes to this system are considered.

5.1 Description

The library case study involves a large abstract state which consists of people, books and a database together with a number of transactions which are possible in the system. Basic requirements are:

1. There are two types of user: members of staff and borrowers.

2. All copies in the library are available for check out or checked out.

3. No copy may be both available and checked out at the same time.
4. A user may not have more than a predefined number of books checked out at one time.
5. Books can only be checked out to members of staff and borrowers.

Once these requirements have been included in a complete specification of the abstract state of the library, the transactions which can take place are also specified. The Z specification of this [45] includes nine transactions which can be described using schemas based upon the abstract state of the library specification. These transactions can be classified according to which type of person performs the operation and what parts of the database are affected by the operation. A total of 18 schemas are used for describing the abstract state and these possible transactions. Since most of these schemas are several lines long and include other schemas by text inclusion, the complexity of the full problem is high.

Due to this complexity, only one transaction was refined so that the method could be tested. Consequently, our version of the refinement and application of the method are also based upon this transaction. The operation in question is the *Check out* transaction. This is performed only by members of staff when a user wishes to borrow a copy of a book from the library. A requirement is that the copy must be available for loan; all other requirements are inherent from the abstract state of the library. The result of this operation is that the copy is removed from the list of books available for loan, the record of checked-out books is updated and the new borrower is recorded as the last person to check out the book.

This operation provides a good test for the method since it involves most aspects of the library system which can possibly be altered and forms the basic characteristic of a library: the main function of a library is to loan books out to users. Any change in the *Check out* transaction affects the library as a whole and so a method which can apply changes to this operation while minimising the side effects must be valid for perfective maintenance.

Abstract states represented by:

$$STATE =_{DF} \{\langle type1, type2 ... \rangle \mid \\ predicates\ involving\ types\ \}$$

Transactions represented by:

$$TRANS =_{DF} [pre];\ STATE'/STATE.post$$

Error conditions represented by:

$$ERROR =_{DF} [pre];\ [post]$$

Complete transactions represented by:

$$\begin{aligned} T\_TRANS =_{DF} & [pre]; \\ & TRANS;\ SUCCESS \sqcap \\ & ERROR_1 \sqcap \\ & ERROR_2 \sqcap ... \end{aligned}$$

Figure 5.1: WSL syntax for library specification

5.2 Refinement Using Transformations

Specifications using WSL are based upon atomic specifications and guard statements (see [83]). The constructs used for this particular problem are illustrated in figure 5.1.

The WSL specification of the library case study is written according to these constructs and is then refined using transformations. The first transformation which can be applied is **Multi-Move** $\rightarrow\rightarrow$, which changes the order of the nondeterministic choice according to the conditions to be met. The less complex conditions are placed first, ending with the condition for success. Several transformations will be needed in order for the specification to be properly rearranged, and even the first transformation relies upon some user input (deciding the order for the conditions). As there is a wide variety of smaller transformations which are applied to this case study, only the main objectives will be described here within each stage of the refinement. Appendix B lists the main transformations which can actually be used and classifies these under the stages described here. The first stage can be called **rearrange specification**.

The next stage is **rewrite preconditions**, which removes the set operators such as $\in$ and uses the definitions of the states to rewrite these to access parts of the sequences involved. This is followed by rewriting the structure so that guarded commands replace nondeterministic choice; this again is based on definitions involving structure. The remaining stages necessary in the development in the implementation can be summarised as:

- Introduction of Guarded Commands
- Introduction of Global Variables
- Introduction of Procedures
- Replacement of Atomic Specification

The individual transformations used are summarised in Appendix B (according to each stage) and more on the development of this case study is detailed in the report [63]. A simplification of the derivation history produced can be illustrated in figure 5.2. The implementor can choose additional transformations to make this executable more efficient but the transformations of interest for perfective maintenance are the few recorded in the derivation history which will be affected when changes in the requirements arise.

5.3 Application and Implication of Changes

Three modifications are made to the specification of the check out transaction so that the method can be tested. These are that:

1. Books can only be checked out to borrowers (**not** members of staff).
2. Any number of books can be checked out.
3. Only some books can be checked out (the others are **reference only**).

The above modifications affect the basic requirements of the library as listed earlier. The first change restricts requirement 5, the second negates requirement 4 and the last adds new cases to requirements 2 and 3. Since requirement 1 affects USERS only and *Check out*

$T\_D\_Check\_out =_{DF}$
$[id? \in PERSON \wedge bor? \in PERSON \wedge copy? \in COPY \wedge r! \in REPORT]$;
$(...check\ out\ trans; [r! = OK])\sqcap$
$([id? \notin staff\_list]; [r! = unknown\ librarian])\sqcap$
$([bor? \notin \mathrm{dom}\, P\_file]; [r! = unknown\ borrower])\sqcap$
$([copy? \notin \mathrm{dom}\, C\_index]; [r! = book\ not\ in\ stock])\sqcap$
$([status(C\_file(C\_index(copy?))) = out]; [r! = book\ not\ available])\sqcap$
$([length(borrowed(P\_file(bor?))) \geq maxbooks]; [r! = too\ many\ books])$

Rearrange specification

$T\_D\_Check\_out =_{DF}$
$[id? \in PERSON \wedge bor? \in PERSON \wedge copy? \in COPY \wedge r! \in REPORT]$;
$([id? \notin staff\_list]; [r! = unknown\ librarian])\sqcap$
$([bor? \notin \mathrm{dom}\, P\_file]; [r! = unknown\ borrower])\sqcap$
$([copy? \notin \mathrm{dom}\, C\_index]; [r! = book\ not\ in\ stock])\sqcap$
$([status(C\_file(C\_index(copy?))) = out]; [r! = book\ not\ available])\sqcap$
$([length(borrowed(P\_file(bor?))) \geq maxbooks]; [r! = too\ many\ books])\sqcap$
$(...check\ out\ trans; [r! = OK])$

Rewrite preconditions

(... successive transformations and intermediate stages until executable ...)

```
T_D_Check_out ⊑
begin var id, bor : PERSON, cpy : COPY, r : REPORT;
do MEMBER(id); BORROWED(bor); INDEX(cpy);
    STATUS(cpy); LENGTH(bor); MAIN(bor, cpy)
od;
print r end
where proc MEMBER(x var x) ==
            if member?(x, staff_list) then r := unknown librarian; exit fi.
      proc BORROWED(x var x) ==
            if P_file(x) = ⟨⟩ then r := unknown borrower; exit fi.
      proc INDEX(x var x) ==
            if C_index(x) = ⟨⟩ then r := book not in stock; exit fi.
      proc STATUS(x var x) ==
            if status(C_file(C_index(x))) = out then r := book not available; exit fi.
      proc LENGTH(x var x) ==
            if length(borrowed(P_file(x))) ≥ maxbooks then r := too many books; exit fi.
      proc MAIN(x, y var x, y) ==
            ...; r := OK; exit.
end
```

Figure 5.2: Derivation of Check_out operation in WSL

does not alter the USERS field, alterations to this are not tested. The modifications listed above were selected for the variety of ways in which they affect requirements connected to the *Check out* operation and appear in ascending order of complexity.

5.3.1 First Change

Restricting borrowed books to non-members of staff (alteration 1) can be dealt with by altering some basic definitions involving USERS at the specification level. The set PERSON is split into two new sets: STAFF and BORROWER, *P_file* is restricted to members of BORROWER only and *staff_list* to members of STAFF only. The choice of carrying out this modification in this way removes the need to add predicates to the specification. An advantage to changing only the declarations part of the specification is that transformations can be carried out automatically as recorded in the original derivation history.

The new version of the derivation history corresponding to this change is:

$T\_D\_Check\_out =_{DF}$

$[id? \in \mathbf{STAFF} \wedge borrower? \in \mathbf{BORROWER} \wedge copy? \in COPY \wedge r! \in REPORT];$

$(...check\ out\ trans; [r! = OK]) \sqcap$

$([id? \notin staff\_list]; [r! = unknown\ librarian]) \sqcap$

$\ldots$

Rearrange specification

$T\_D\_Check\_out =_{DF}$

$[id? \in \mathbf{STAFF} \wedge borrower? \in \mathbf{BORROWER} \wedge copy? \in COPY \wedge r! \in REPORT];$

$([id? \notin staff\_list]; [r! = unknown\ librarian]) \sqcap$

$\ldots \sqcap$

$(...check\ out\ trans; [r! = OK])$

| **Rewrite preconditions**

(... successive transformations and intermediate stages until executable ...)

```
T_D_Check_out ⊑
    begin var id : STAFF, bor : BORROWER, cpy : COPY, r : REPORT;
    do MEMBER(id); BORROWED(bor); INDEX(cpy); STATUS(cpy);
        LENGTH(bor); MAIN(bor, cpy) od;
    ... as original code
```

5.3.2 Second Change

The second change causes more implications but is also easy to trace. Negating requirement 4 by removing a limit on the number of books which can be borrowed merely implies that there is no longer a need for that restriction. The declaration of the value **maxbooks** can be removed from the specification, as can any predicate which refers to **maxbooks**. The error condition corresponding to too many books being checked out can be removed altogether from the specification.

All transformations can be applied automatically from the derivation history, producing a new version of the code as they did with the first change. The only difference is the removal of the introduction of the *LENGTH* procedure for the **introduce procedure** transformation, but this will not be carried out automatically since the predicate leading to this was removed from the specification.

5.3.3 Third Change

The final change is the most complex. Making certain library books reference only means that library requirement 2 is changed to "all copies in the library are available for check out, are checked out or are reference books only". Requirement 3 is changed to "no **non-reference** copy can be both available and checked out at the same time". The result is that most predicates concerning the database are altered in the specification.

A new section of the COPY record is created to identify the sort of book involved. A copy being checked out must now have sort "available" and must not be registered as checked out. The error condition corresponding to a book of sort "reference" being checked out is added to the specification. The new version of the specification will therefore have an extra precondition to the check out operation, declaring that the book must have sort "available", and an error condition when this is false among the other error conditions.

The derivation history can now be used for developing a corresponding altered version of the code. Transformations are applied in the same way as before: rearranging the choice (*D_Book_is_ref* appears after *D_Book_not_available*), rewriting preconditions (checking the sort becomes an access to part of the sequence by definition, as in the other error cases), changing to guarded commands, adding initial declarations, introducing procedures and replacing the atomic specification.

The only difference lies in the introduction of procedures. The user needs to add the procedure for checking the sort of book being checked out. This will test whether the book is a reference copy and output the corresponding error report if it is. If not, the next error case can be tested for and the *MAIN* procedure remains exactly the same. Hence the transformations can be automated as far as introducing procedures, where the user adds the extra case, and then automated to the end. The new version of the code will correspond to the change which has taken place.

5.4 Comparisons with Z Refinement Method

The effect of changes on the refinement by transformation method of this problem gives an indication as to the benefit of this method for developing maintainable code. A complete examination must include some measure for comparison. The same changes were applied to the Z specification and its refinement and the effect of these examined as well. The results of this can be compared with the conclusions from the previous section to test whether this method is better when changes in requirements arise. The criteria used for this comparison includes scaleability, speed of refinement, ease of refinement, design improvements and software quality. For quick reference, a paper was published on this in the proceedings from the IEEE Conference on Software Maintenance 1993 [64].

First, a general description of the method of Z refinement used in the library example of the paper is given. The Z specification is based upon a number of schemas which are related to one another by reference to states or some "inheritance". This means that when changes are made, all initial schemas need to be considered and the effects traced down to the *Check_out* transaction stage.

The refinement of the specification involves an immediate conversion into a form which includes variables, return codes, procedures and conditionals which do not appear at the specification level. The user must *prove* that this form is equivalent to the earlier schemas before (s)he can be satisfied that this is a correct refinement. Similarly, other refinements involve the direct conversion from schemas to a "code-like" form with procedures, Boolean return codes and conditionals. Assignments are brought in, with no link provided with earlier stages. The user must *prove* that each case is equivalent by regarding the semantics of each stage.

The code produced from the Z refinement of the *Check_out* transaction is displayed in figure 5.3. The lack of connections between this code and the Z specification is reflected by problems when changes are needed. The user must start from the first schemas presented, making changes where necessary, but following through each schema as the effects of the changes are felt. Even when the complete *Check_out* specification is altered, each refinement stage must be modified where the user feels it is necessary and a proof of the semantic equivalence of each stage to the original schemas must be made.

This highlights a basic difference with the refinement by transformation method. The Z refinement is proof-based and a record of the transition from specification to code is difficult to establish. Refinement by transformation, however, relies upon mathematically proven steps which gradually develop the specification into an executable form. These steps and intermediate stages of the specification are all recorded in a derivation history. Changes in requirements can be made at the specification level and the transformations recorded in the history used again for producing a new version of the code. These new versions of the specification, code and intermediate stages are recorded in the updated derivation history. Thus a complete record of the derivation of the code and its alterations

```
$T\_D\_Check\_out \sqsubseteq$
$r1, r2, r3 : Boolean;$
$c\_tkn : TKN;$
$member(staff\_list, id?, r1);$
$domlookup(P\_file, borrower?, r2);$
$dirlookup(C\_index, copy?, c\_tkn, r3);$
$if \neg r1$        $\rightarrow r! := unknown\ librarian$
$\Box \neg r2$       $\rightarrow r! := unknown\ borrower$
$\Box \neg r3$       $\rightarrow r! := book\ not\ in\ stock$
$\Box r1 \wedge r2 \wedge r3 \rightarrow$
    $PR : P\_RECORD;$
    $CR : C\_RECORD;$
    $r4, r5 : Boolean;$
    $filelookup(C\_file, c\_tkn, CR, r4);$
    $pflookup(P\_file, borrower?, PR, r5);$
    $lengthdll(PR.borrowed, l);$
    $if\ CR.status \neq in \rightarrow r! := book\ not\ available$
    $\Box\ l \geq maxbooks$   $\rightarrow r! := too\ many\ books$
    $\Box\ CR.status = in \rightarrow$ ...
      $\wedge\ l < maxbooks$
    $fi$
$fi$
```

Figure 5.3: Code developed from Z library specification

is always present and there is no need to prove semantic equivalence for each stage due to the mathematical correctness of the transformations in use.

5.4.1 Scaleability

Looking in more depth at the differences between the two approaches, the first quality to be tested is scaleability, or the size of effort needed by each method for refinement and change as the specification grows. There are fewer stages of refinement with the Z method, but each stage involves a proof that the new version is equivalent to the previous one. This proof could be lengthy and adds extra work to the refinement which cannot be qualified nor automated. Modifications must be applied both at specification and implementation levels, with new proofs needed each time. As the specification grows, so will the size of the proof.

Our method involves more stages of refinement but each stage is produced by a mathematically proven transformation, implying that the new version is semantically equivalent to the previous one. Once these stages and the transformations used have been recorded in a derivation history, modifications to the specification are quicker to implement and there will be no need for further proofs. Changes can be made to the specification and the history used and updated to produce corresponding versions of the code.

A conclusion from this is that the transformation method is an improvement on the Z method as regards scaleability of the refinement and of the application of change.

5.4.2 Speed

Due to the lack of proofs, the speed of the refinement is improved with our method since the transformations are the only element involved which have already been proved. The speed in which changes can be made is even better since at this stage, the history of transformations can be applied to the altered specification to produce new code. The influence of tools which automate this procedure will also make the use of a derivation history quicker for refinement and change.

5.4.3 Ease of Change

The method of refinement used in the library example paper [45] depends upon the user's intuition and there are no mathematical steps taken. This makes the problem harder to refine and change. With a set of possible transformations available for specifications, the user has some choice as to what refinement steps to take at which point. Although the nature of the refinement is still reliant on the user's intuition, (s)he has some guidelines to follow.

Once the user has made a decision on a refinement path this is recorded and can be used again when changes are needed. This list of possible transformations eases the application of change. The user can just alter the specification (or simpler still, a definition) and then run through the list of transformations until code is reached or an extra transformation might be needed. Keeping track like this of the refinement makes any change infinitely easier to implement than with the Z refinement procedure, where reason and proof is required at every stage of refinement.

5.4.4 Design Improvements

Unfortunately, the simple notation available with WSL at specification level means that the appearance of the specification is more complex and harder to understand than Z schemas. A Z schema shows clearly what declarations and predicates are involved and text inclusion simplifies lengthy schemas. WSL specifications are very basic in form and so more difficult to understand.

However, the simplicity of the language is a benefit when refining via transformations since one can immediately access the parts of the specification which are being refined. This makes a step-wise, automatable production of code easier to achieve rather than relying upon the user to define new stages of the refinement which depend upon knowledge of the system and understanding of the Z specification.

It is also an advantage when changes are needed. Since the states involved appear directly in the specification, one can alter the necessary items in the specification and then refer to their definitions to see where these are different. With Z schemas, one needs to trace back through all the included schemas to find which states are involved and where. In fact, it is easier to write out the complete schema (expanding any other included schemas) to see where changes can be applied. This full schema looks much more complicated than the WSL version!

5.4.5 Software Quality

The quality of the software produced using either method is good since refinement from specification makes the code itself much more compact and well-structured. It is difficult to assess which one has the best quality since there are many factors determining software quality and their importance depends upon the needs of the customer.

Factors which determine software quality include correctness, reliability, efficiency, flexibility and reusability. Maintainability can also determine software quality, but since this is our objective, it will be discussed separately in the next section. The above factors can be measured with a number of software metrics but the true quality of the software in question is invariably subjective.

If a simple metric like LOC (Lines Of Code) [27] is applied to the derived code (*not* to the Z or WSL), one could say that refinement via transformation is better since there are fewer lines of code. Using Halstead's Software Science metric [39], the refinement via transformation code is also superior. Values of 224 for the program length, 12320 for the program volume and .041 for the program level are recorded for the code produced from Z refinement. The respective values for the code produced from our method are 194, 970 and .022. On the other hand, McCabe's cyclomatic complexity [57] is 6 for the code produced from Z and 8 for that produced from WSL, indicating that the first is less complex.

The code produced using the Z refinement is a single block with two conditional statements (one inside the other). The code produced through transformations has a set of variable declarations, a loop which makes calls to each procedure until one is satisfied and a print-out of the report at the end. The latter is much easier to understand and to modify but the first is simpler. Since modification is the priority in this example, code produced with transformations can be judged as having better quality.

Issues regarding the *maintainability* of the code will be discussed in the Results chapter of this thesis.

5.5 Summary

This chapter investigated the use of the method for a larger case study based upon the organisation of data rather than the functionality of the code, as described in the first case study of the integer square root. This case study was the specification and implementation of a library system which was originally developed through the use of the IBM Hursley Park method in the paper [45]. A comparison of the transformation approach to this method could be made, based upon the success of perfective maintenance in each case. Criteria for evaluating this success was scaleability, speed, ease of change, design improvements and software quality. Metrics were used as a measurement for this last criterion.

Chapter 6

Results

This chapter draws conclusions on how the method should work for specifications based on computation and for those based on the arrangement of data. This follows from the results found by applying the method to the case studies as described in chapters 4 and 5 and the Appendices.

6.1 The Method

Applying the method to case studies has indicated that there is a different approach needed when dealing with specifications involving functions than with those reliant upon the organisation of data. Most real cases will involve a combination of these two and so the method must finally combine the two approaches which were adopted.

The following sections describe how the method should work in general for computation-based specifications and specifications based on data organisation. The differences in the approaches lie in how the refinement path is originally built and how changes in the requirements are dealt with.

6.1.1 Computation Intensive

There are two stages to the method: developing a derivation history from the refinement of the specification into an implementation and dealing with changes in the requirements of the system. The first part involves the identification of *strategies* to follow according to the specification involved.

Strategies which were found useful for producing maintainable software from specifications based on functions are as follows:

1. Expand functions and definitions.
2. Introduce bounds.
3. Introduce cases (if possible).
4. Divide problem so that part of it is trivial.
5. Repeat steps 2,3,4 as necessary.
6. Introduce Recursion, Simplify, Remove Recursion.

Most of these steps could be carried out with a single transformation or with a collection of transformations from the Maintainer's Assistant and so the derivation history could be produced automatically if the user knew which transformations corresponded to each strategy. Due to the differences between each specification of this type, the user must have some idea of what he is aiming for. For instance, step 2 above depends upon the values that the user chooses to enter for the bounds which depends upon his understanding of the type of function which he is dealing with.

Referring to chapter 4, the set of strategies described above were useful for such a small code intensive specification as the integer square root. The transformations in the order in which they were applied can be listed as:

1. Expand the definition of $\sqrt{}$.
2. Expand the definition of $\lfloor\rfloor$.

3. Use $(s+1)^2 > s$ to set upper bound.
4. Introduce assertion based upon the monotonicity of $P(r)$.
5. Take out case $h = l + 1$ since this is trivial. This can be done by introducing an **if** statement and simplifying.
6. Divide the range into two parts.
7. Introduce recursion.
8. Minimise search space through simplification.
9. Replace parameters by Global Variables.
10. Convert to **while** loop.
11. Use **r** to represent **l**.
12. Substitute $m^2 \leq s$ for $P(m)$.

Once a refinement path had been built from the transformations which were used, it was necessary to investigate if this could be used again for dealing with changes in the requirements. The results from these experiments were described in Chapter 4 and from these the best approach could be identified for problems of a computational nature. The following sequence of actions is suggested:

1. Make change at the highest possible abstraction level (usually the specification itself).
2. Separate this level so that part of it is identical to the earlier version (using strategies described above).
3. Use path recorded earlier for developing the part of the specification which is the same.
4. Use refinement strategies for developing the different part of the specification.
5. Use restructuring transformations for merging the two parts which have been developed.

6. The new refinement path is recorded.

The transformations currently present in the Maintainer's Assistant are mainly for restructuring purposes and these can be used upon the code which is developed for making it simpler or more efficient. These can always be included at the end of the recorded refinement path in the sequence in which they are applied.

The new path referred to in step 6 will consist of those transformations needed to separate the altered specification (in the sequence in which they were applied), followed by the original refinement path, followed by the transformations used in step 4, followed by the transformations used in step 5. If this new path were applied automatically to the altered specification then a correct altered specification would be produced. Future changes to the specification will involve a referral to this new refinement path.

While the refinement path looks long when constructed in such a way, in reality it is not, since many of the above steps will involve only 2 or 3 transformations which will need to be recorded. Also, it is not always necessary to separate the specification as described in step 2 since the type of change might still allow an implementation to be developed using the old refinement path. There are no strict rules which can be followed here but the best suggestion is to try using the old path first. If this fails to produce an implementation, then steps 2 and 3 should be followed. Step 4 might not be needed as it might be a case of restructuring what is now available (step 5), but again this depends on the problem and change in question.

6.1.2 Data Intensive

When dealing with specifications which rely upon the organisation of data, different refinement strategies and changes are involved. Chapter 5 described such a specification, the library system. Work on developing a refinement path and making changes to the requirements was described and from this general conclusions on working with data intensive specifications could be drawn. The following approach was useful for developing a refinement path for a specification involving many parts of a system:

1. Replace abstract states with concrete ones (look at the system from a different perspective).
2. Write specification for *complete* operation in a concrete style.
3. Rearrange so that general precondition is followed by error conditions and finally the successful case.
4. Rewrite conditions to involve sequences.
5. Introduce guarded commands.
6. Create variable declarations (to replace inputs and outputs).
7. Make loop.
8. Introduce procedures (exit out of loop if a procedure is successful).
9. Rewrite conditions in the successful case.

Most of these steps involve single transformations which can be ordered in the sequence in which they are applied. This will form the refinement path, or derivation history. Referring back to the transformations which were used to refine the library specification (as described in Chapter 5), these could form the following specific refinement path:

1. Select operation to be refined.
2. Represent complete operation in a concrete specification.
3. Rearrange parts of this specification using transformations such as **Move**.
4. Use definitions to expand specification.
5. Represent conditions as sequences.
6. Replace nondeterministic choices by guarded commands.
7. Introduce global variables.
8. Represent input with variables.
9. Represent output as assignments to a variable and the printing of this variable.

10. Make loop around body of operation.

11. Represent each conditional branch as a procedure.

12. Simplify so that a satisfactory condition causes an exit from the loop.

13. Use the **Separate** transformation so report only printed at end.

14. Rewrite atomic specification with assignments to the newly represented parts.

15. Introduce separate database which is updated each time a transaction takes place.

Having constructed a refinement path, the effects of changes in the requirements of the library system could be investigated. These changes were selected according to the effects they had on the very basic requirements of the system and on the *check out* specification in particular. The results are described in Chapter 5 and from these a general procedure for dealing with changes to similar types of specifications can be deduced.

It is difficult to draw up a step-by-step approach to dealing with changes to a specification of a data intensive nature since the effect of changes varies very much from case to case. However, the nearest approximation of such an approach for the library system is as follows:

1. Identify the basic data types altered by the change.

2. If other parts of the specification are unaffected, introduce or remove a data type to accommodate this change. That is, the data type can be used to represent the change if this is independent at the highest level of abstraction.

3. If step 2 is possible, follow the method suggested for the computation intensive situation.

4. If step 2 is not possible, investigate whether it is possible to change one of the conditions independently of the rest.

5. If the above is possible, make the change and remove all transformations pertaining to this condition. Use the transformations in the order in which they are now found and add extra transformations to the end of the path.

6. If step 4 is not possible, link all conditions affected by the change and rewrite these according to the new requirements. Remove all the transformations linked to these particular conditions and add extra transformations to the end of the refinement path to develop new code corresponding to these conditions.

The above is very general and only covers simple alterations to the data types or conditions. It is impossible to investigate any further since the problem itself is only an example and not a true indication of the more particular problems found in real cases. For this reason, the problem of the storage control interface was chosen to test the method for both data and computation intensive specifications. However, the range of changes which can be carried out in the real case is wide and has quite far-reaching consequences. This means that parts of the method need extending and a full set of conclusions from this work cannot be drawn up at present since several months will be needed for working on this problem. Initial results indicate that the basis of the method is good, however, and that it is usually possible to use the refinement path again for small changes.

6.2 Maintainability of Case Studies

Having experimented with the use of a refinement path when changes are made to a specification, the last results involved assessing the maintainability of the software for the integer square root and library examples. Ideally the success of the method could be ascertained by evaluating the maintainability of the WSL specification, the intermediate refinement stages and the code which was produced. This "maintainability factor" could be used to compare this approach with standard refinement techniques such as Morgan's refinement calculus and the IBM Hursley Park Method.

Unfortunately there are no standard metrics for assessing maintainability at higher levels which makes the above mentioned evaluation impossible. As with software quality, it is difficult to quantify maintainability but an examination of the code can provide insights into which maintainability factors exist. Factors which have been recognized to contribute towards the maintainability of software include modularity, complexity, consistency and expandability. Each of these can be examined for the code generated when using the method with the integer square root and library case studies.

Modularity refers to the extent to which a system can be decomposed into smaller sections. To be maintainable, software must remain as independent as possible but include enough comprehensive links to bind the code together. Comparing the code produced for the integer square root, the WSL code has the same modularity to the code produced from Morgan's refinement calculus. However, the development of the code using transformations is more modular since it is easy to add or remove transformations in the refinement path to develop code with different efficiency. Morgan's refinement calculus involves a dependency on earlier stages of the development and so one cannot remove and extend sections of the path without drastically altering the code which is produced.

Looking at the library system, the code produced from transformations is more modular than the code developed from Z since it has several distinct procedures which are independent from one another yet linked in a sequence in the loop. The benefit of this is that procedures can be altered, removed or added without affecting the rest of the code drastically. This is an improvement on the Z refined code where half the code is reliant on the results of the earlier section. This code does not consist of several independent, cohesive modules and any change to a part of this software could cause a side effect.

Maintainable software must be as simple as possible; the complexity must be minimised. Evaluating the complexity of the integer square root problem through the use of metrics did not provide much of a fulcrum for comparing the transformational approach with the use of the calculus. For instance, the LOC metric [27] gives values of 6 and 11 respectively for the code produced from transformations and the calculus. This would indicate that the method *does* produce code of a minimum complexity. However, this is very much a matter of how the code is actually written. If the code produced from the calculus was written in WSL syntax, then its complexity would be 7 and only slightly more complex than the code produced using the method.

The metrics described in chapter 5 provide a better indication of the differences between the two pieces of code as far as complexity is concerned since the library problem is larger and less dependent on variations in syntax. However, it was still found that neither is more complex than the other since the values of complexity differ according to the measure being used. On the other hand, the code produced using transformations is simpler for the purposes of modification.

Consistency involves the use of some standards and conventions so that the format of modules is similar. This makes the understanding of the program much easier. Since the refinement by transformation method involves writing all constructs from specification to code in WSL, there is a great deal of consistency. Users need only acquaint themselves with the standard document on the syntax and semantics of WSL [83] (see Appendix A) and from this understand any of the code, specifications and intermediate stages generated from the method. Again, this is an advantage over the refinement calculus or Z refinement method since although there are standard formats for the specification level, the code can be represented in any language.

The last measure for maintainability to be discussed here is expandability, the ease with which the code can be added to. This is reflected by some of the changes discussed earlier, and it was found that the refinement by transformation method produced code which was easier to adapt and expand on. The work carried out on the integer square root problem illustrated this point. It was easy to refine the problem using four simple transformations and produce some basic code. However, to make the code more efficient, extra transformations could be added for more specific code. At earlier stages of this development it was easy to separate parts of the specification and use transformations to expand different sections in different ways. The refinement calculus relied upon a more standard approach which was less flexible and involved dependencies on earlier stages of the development of the code. This made it difficult to expand on the software.

For the library system example, each procedure can be added to without affecting the main body of the software and extra procedures can be added at any stage prior to the *MAIN* procedure. Code which does not involve testing for conditions can be added after the **do** loop. Again, transformations can be used in different orders and different emphases making the software easier to expand upon than with the Z refinement method.

A conclusion from this is that the method produces software which is modular, consistent, of low complexity and easily expandable. This combination of qualities makes for software which is maintainable. A more quantitative approach is still not possible for the specification levels but the possibility of applying function points to test the maintainability of different levels of the software is currently being investigated.

6.3 Summary

This chapter summarised results from the application of the method to case studies reliant upon computation and the organisation of data and described conclusions on the general approach to similar problems. The maintainability of the case studies was assessed.

Chapter 7

Conclusions

This chapter will summarise the problems which were encountered and solved while researching the new method. The way in which the method works for general problems of both a mathematical and organisational nature will be discussed with reference to a third case study which is currently being worked on. Conclusions to the thesis include future developments of this research; the application of the method to further case studies and the development of a tool based upon the *Maintainer's Assistant* for automating the method.

7.1 Problems Encountered

The main aim of this research was to produce a method for maintaining new software. The first problem, therefore, was deciding which areas of software engineering to address. The fact that the *Maintainer's Assistant* was being developed by the University of Durham and hence readily available made transformational programming and refinement focus areas. While the tool was aimed at restructuring, the transformation set could be extended to specification levels and so the possibility presented itself of using the transformations for refinement.

Once these initial decisions had been made there were a number of problems which were encountered while working on this project. Some of these were:

1. how to use WSL to represent formal specifications.
2. which refinement strategies to use when building a refinement path.
3. which extra transformations to implement.
4. identifying the types of changes which could be made to a system.
5. identifying the qualities of "maintainable software".
6. how to produce an automated method involving the "Maintainer's Assistant".

Such problems could not be solved without using concrete examples. It was necessary to select some examples of specifications, refine these into code and see the effect that changes in the specification had upon the development of the code. The only way in which the effectiveness of a method could be judged was by selecting problems which had already been refined using different methods. These could supply suitable comparisons for the development of refinement paths and for the maintainability of the software which was produced.

7.2 Problems Solved

The first step towards tackling the problems listed above involved selecting suitable case studies. The first decision was to choose examples which had utilised different methods of refinement so that the idea for using Ward's transformations (and extensions of these) for refinement could be justified. The two best representatives of the opposing refinement calculi and refinement methods were deemed to be Morgan's refinement calculus and the IBM Hursley Park Method of refinement. So two case studies involving these needed to be selected.

Morgan's refinement calculus is well defined in his book, *Programming from Specifications* [59] and there are a number of case studies described where he applies the calculus to a formal specification and develops an implementable algorithm from this. A very simple one, the integer square root, was selected as a good representative of how to refine a mathematical function. This was a good starting point since it could help to form the basics of the method when functionality alone was important.

The first problem identified earlier, using WSL to represent formal specifications, was easily solved since this formal specification was based on algebraic notation which was already defined in the WSL language. However, deciding upon how to refine this particular case study (item 2) was problematic.

Initial transformations are obviously based upon expanding the functions involved according to their syntactic definitions and introducing bounds to the problem. At this stage a simple transformation which introduces a loop can make the problem implementable. However, the code is not very efficient and changes in its requirements would involve many extra steps of calculation. So it is necessary to extend the refinement path and extra refinement strategies were identified to do this. These included the "divide and conquer" strategy and a strategy of introducing recursion, simplifying and removing recursion. These are recognized strategies when transformations are used for refinement but had not been practised in Morgan's treatment of the integer square root problem.

This work gave some indication of which extra transformations needed to be implemented but identified a further problem. The development of a program from a specification sometimes uses algebraic axioms and definitions. As these are quite extensive and not implemented in the Maintainer's Assistant, the method would need to rely upon an additional tool for pure logical reasoning (such as the Boyer-Moore tool). This was the first indication of hindrances in solving problem 6.

Returning to problem 4 however, identifying the types of changes was very dependent on the specification itself. After investigating the effect which a number of changes in the specification had upon the refinement into code, only two categories were identified for this particular example. The first involved the aspects of the function itself and the second the bounds of the actual problem. All other changes were variations and combinations of these and so it was only necessary to standardise the approach to changes representative of these two categories.

Viewing the effect that changes had on the development of the code indicated whether the refinement by transformation approach was suitable for producing maintainable software. From this, qualities determining maintainability could be identified. Software is easier to change when it is modular, consistent, expandable and of minimum complexity. Having a formal development path ensured that this was the case and allowed changes to be made at higher levels. This eliminated the need to investigate the interactions between sections of code and maintainability became more of a specification issue.

Having identified a method for producing maintainable software based upon a functional specification, it was then necessary to investigate possibilities for larger specifications which relied upon the organisation of data within the system. It was also time to compare the transformation approach with the IBM Hursley Park Method. A case study of a library system which had been developed from its Z specification using this method was thus selected.

Returning to the list of problems to be solved, the first proved quite difficult to solve for this case study. WSL has a very limited set of constructs for higher level specifications. As everything relies upon atomic specifications and guard statements (see [83] and Appendix A), attempting to present an equivalent to Z schemas was a major problem. Constructs for representing abstract data types and relationships between parts of a complex specification are not present in WSL.

This problem was solved by defining a subset of WSL for representing the library specification based upon how the Z had been originally used. Syntax for abstract and concrete states and operations was developed for testing out the method on this particular case study. The result was quite complex and indicated a further need to define new constructs in WSL. The problem of how to produce a general WSL specification language into which *all* major specification languages can be translated is still being addressed.

The types of constructs used greatly affected which transformations could be used for the development of the code. As comparing the transformation approach to the Z Method was essential, it was necessary to begin from the same levels of abstraction so that later changes could be made at these levels. Many of the early transformations are ones also used for restructuring code and are already automated in the *Maintainer's Assistant.* These were of the form of rearranging sections of the specification and joining parts together. Transformations not present in the tool involved those which interpreted the meaning of the newly defined high level constructs and rewrote these using different syntax and introducing more specific cases.

Again, this work indicated some extra transformations to be implemented but identified further problems. The syntax and semantics of the specification greatly affects the transformations which can be applied. Does this imply that new transformations need to be built into the *Maintainer's Assistant* each time a new system is being maintained?

The types of changes which might be made to the library system could again be categorised but differed greatly from those of the integer square root. Changes within the library came under three categories:

1. a change in the refined operation which affects only this operation.
2. a change in the refined operation affecting other operations.
3. a change in a basic data element of the system.

As expected, these changes were in ascending order of complexity and altered the development of the code in a variety of ways. They were selected for the effect that they each had on the requirements of the system and further changes proved again to be special combinations of these. These changes proved that the development of the system had been carried out with a view to maintainability and in some cases caused a revision of this development. In fact, the refinement path described for developing the check out operation of the library system is an edited version of the original (such as reordering the conditions in the specification was added as an early transformation to make it easier to reuse the path).

While qualities of modularity, complexity, consistency and expandability are still important for maintenance, this case study illustrated that the order in which parts of the system were described and the aspects which determined the definition of these parts both affected the maintainability of the software. However the order of the transformations as applied to the specification was not always that important unless cases, bounds and procedures were being introduced. That is, many steps in the refinement path are purely for rearrangement so that a transformation which imports new detail and knowledge can be carried out.

The idea that there is no ideal refinement strategy or path makes automation of the method even more difficult. Having decided that the Maintainer's Assistant is not sufficient due to laws of logical reasoning and definitions which need to be used in a refinement path, producing a tool to automate the method is further complicated by the fact that new transformations can be defined for each different system and refinement paths can be constantly altered according to the needs of the user. A tool must enable the user to decide what refinement strategy to use, to define new transformations if needed, to rearrange the order of transformations in a refinement path and to change transformations within this path. Such a tool is not impossible but could be inadvisable! Further ideas of how the method could be automated are described in the section of future developments, after an attempt to generalise the method is made.

7.3 Application Of Method

The work on the case studies has provided indications of a general method for producing maintainable software for a formally specified system which combines mathematical functions with the organisation of data. This method is currently being tested on a Z specification of a real system. This case study involves a storage control interface which was developed by IBM but followed no real method for its refinement. The formal Z specification was provided together with the code which was produced but linking the two has been extremely difficult. This has provided the scope for developing a refinement path from transformations without following any current trends.

With such a wide horizon of possible refinement paths, work is still being carried out on this case study and no ideal refinement path has been identified. In fact, the way in which this problem was specified is still being questioned and further syntax for the WSL is being defined. However, it has been possible to see whether a combination of the approach taken to the two case studies can be used for systems which combine mathematical functions with data arrangement. The size of the case study makes its presentation difficult and so only the conclusions of what the general method actually is and the results from applying it will be described.

The method basically combines the approach used for the first case study with that for the second. The refinement strategy used for the library system is initially followed and it is only when the specification takes the form of procedural code that the need for refining mathematical functions arises. So a very crude description of the refinement strategy needed for producing code from a large WSL specification of the Z model is as follows:

1. Identify operation to be refined into procedural code.
2. Write specification for *complete* operation based on concrete states.
3. Rearrange so that like postconditions can be assembled together.
4. Expand on any definitions to produce extra postconditions.
5. Substitute concrete for abstract (if possible).
6. Combine and simplify conditions where possible.
7. Rearrange conditions so that general precondition is followed by error conditions and the successful case is last.
8. Rewrite conditions to involve sequences.
9. Introduce guarded commands.
10. Create variable declarations (to replace inputs and outputs).
11. Make loop.
12. Replace sections with more specific definitions.
13. Introduce procedures (exit out of loop if a procedure is successful).
14. Add assignments and delete assertions where possible.
15. Expand functions where necessary.
16. Introduce bounds.

17. Introduce cases.

18. Divide-and-conquer (for more efficiency).

19. Introduce Recursion, Simplify and Remove Recursion (if necessary).

20. Use additional refinement-by-transformation strategies (where necessary).

This a very generalised form of any refinement strategy which might be used but illustrates how it is first essential to deal with the way in which data is organised and re-interpret this *before* working upon the actual functions. So the general approach is to use the method for refining the library followed by that for refining the square root.

The way in which changes should be implemented is not as straightforward. The types of changes described in the square root case study obviously take effect in the lower levels of the development of the storage control interface. Changes involving mathematical functions rely solely upon pinpointing where the function occurred and dealing with the change from that point onward in a similar manner as suggested for the problem of the square root.

Changes which affect the data or operation involved have a much larger effect on the refinement path since the higher levels are affected here. Again, this can be dealt with as with the library case study but causes a lot more work in redeveloping and changing the refinement path. Changes in the data parts themselves usually require a new set of transformations and reusing the path becomes much more difficult. Problems of locating where a change is best made and incorporating new transformations in the middle of a refinement path are still being tackled. There is no ideal solution but whether there is an optimal one is still difficult to foresee with such a real situation.

7.4 Future Developments

Future developments include applying the method to further case studies to test it on problems not originally specified using Z. Once the method has been successful with a single specification it can be extended to systems where many interconnected specifications exist. That is, there are components within these systems which can each be specified but may affect each other in some way. For instance, the library system discussed previously includes specifications of checking books out, returning books, adding and removing books from the shelves, producing lists of borrowed books and finding the last borrower. If a restriction is placed so that members of staff cannot borrow books, then not only is the specification for checking books out affected but so is that of the last borrower and borrowed books.

Some database needs to be developed so that any similarities between the specifications are connected in some way; alterations in one specification can influence the other. This requires the use of tree structures so that links between specifications can be established. The original method will be applied to the first altered specification and then this tree structure should indicate where the method can be applied to other specifications.

Thus the method is dependent upon three things: transformations developed already for the Maintainer's Assistant, a history of the transformations and changes which have taken place and a tree structure for linking specifications and their changes. With so many details involved, an automation of the method should be attempted. Ideas for a tool are described in the next section. The tool will be directly linked to the Maintainer's Assistant and a tool for algebraic reasoning.

7.4.1 The Tool

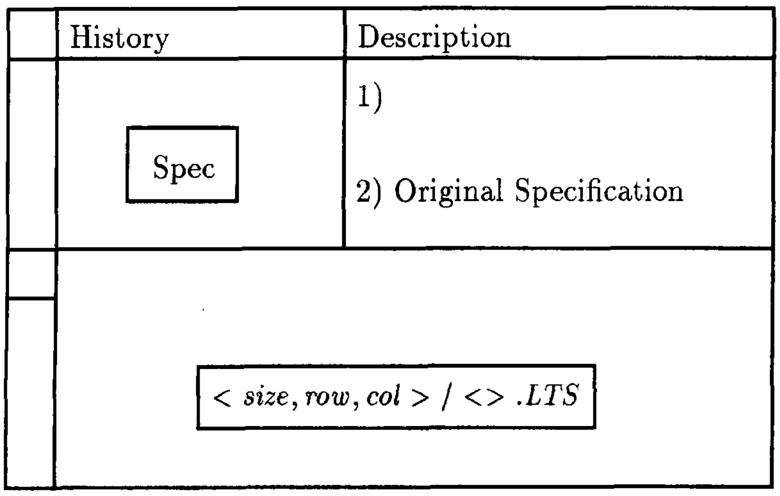
There will be two separate versions of the tool. The first will consist of a single independent specification and changes made to it, while the second will extend this idea to systems where many specifications exist and are interrelated. Both will involve the use of a transformation editor and change editor and will consist of three windows at the basic level: a refinement window, a descriptive window and a history window.

The refinement window will illustrate the specification throughout its refinement stages and should highlight each change to the original specification. The descriptive window will have two sections: one describing which transformations have taken place during the last refinement and the other describing the current stage of refinement with some English text. The history window draws a new box for each refinement and highlights the current situation compared to the history of events. The function of these windows will be easier to understand in the ensuing descriptions of the tool.

A brief description of how the tool might appear in its initial stages of development follows.

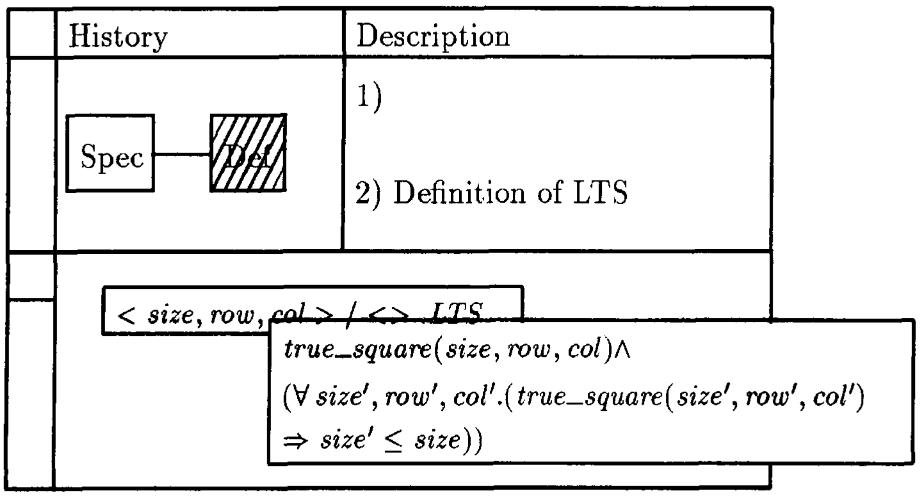
Initial Stages

The first screen will consist of three different sections: the refinement window, descriptive window and history window, as mentioned previously. The refinement window will consist of the specification itself while the descriptive window will remain blank as far as listing transformations are concerned but contain the words: "Original Specification" under the English text section. The history window will include a single highlighted box to illustrate that no refinement has as yet taken place. An illustration of how this screen will appear for a computation-based specification is as follows:



There are three options possible at this point. The user can select LTS and request its definition, select the whole specification and request a first refinement of it, or select the whole specification and request a change. Requesting the change requires the change editor and is a fairly complex process which will be developed and explained later. First, examples of defining and refining this specification will be illustrated.

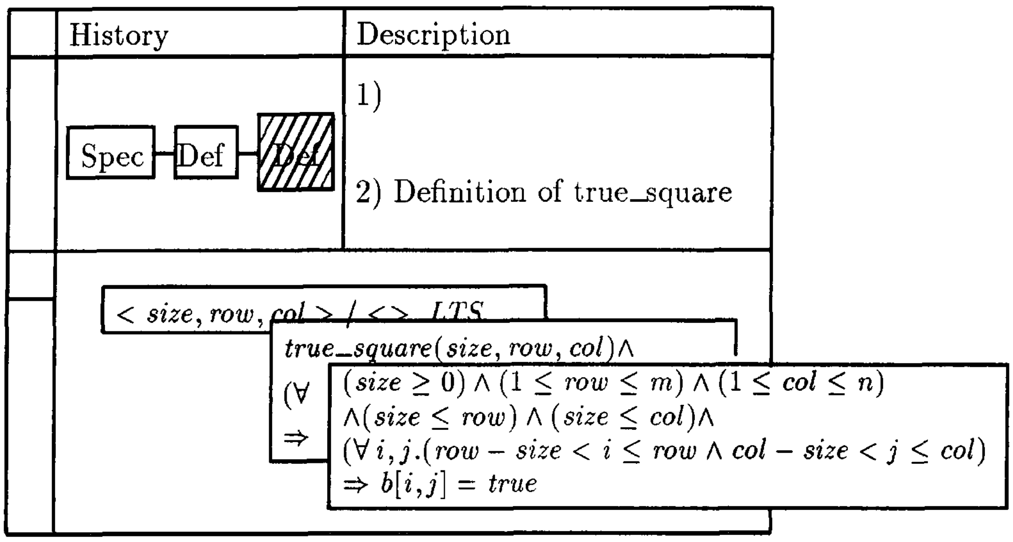
If the user selects LTS using the mouse and clicks on the menu option **Define**, then the following screen will result:



$< size, row, col > / <> . LTS$

$true\_square(size, row, col) \wedge$
$(\forall\, size', row', col'.(true\_square(size', row', col')$
$\Rightarrow size' \leq size))$

Selecting the *true_square* predicate and requesting its definition will further result in a screen as follows:

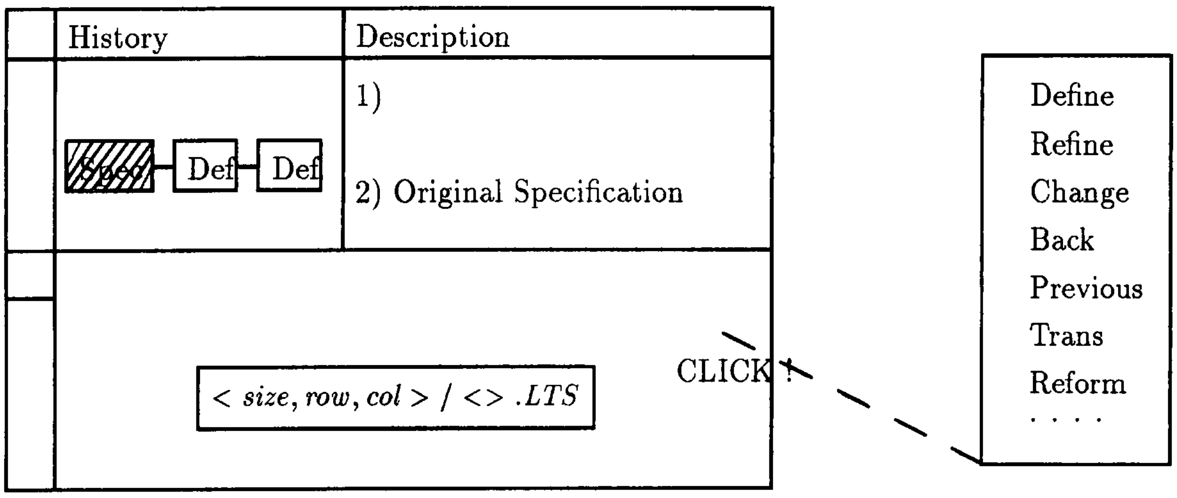


$< size, row, col > / <> . LTS$

$true\_square(size, row, col) \wedge$
$(\forall$
$\Rightarrow$

$(size \geq 0) \wedge (1 \leq row \leq m) \wedge (1 \leq col \leq n)$
$\wedge (size \leq row) \wedge (size \leq col) \wedge$
$(\forall\, i, j.(row - size < i \leq row \wedge col - size < j \leq col)$
$\Rightarrow b[i, j] = true$

This is the furthest that we can get as far as definitions are concerned (and clicking on the **Define** option on the menu will result in nothing occurring) and we must return to the original specification so that refinement or change can be illustrated. This is done by selecting the option **Back** on the menu. The screen which results is as follows:



To the right of the screen is a picture of how the menu might appear. The various options listed can be described as follows:

Define provides definitions of abbreviated terms if these exist.

Refine suggests a refinement strategy using transformations from the Maintainer's Assistant.

Change invokes the change editor so that changes to the specification can be made.

Back displays the original specification.

Previous displays the previous screen.

Trans displays a list of the transformations used for refinement of the current specification.

Reform requests the use of the Maintainer's Assistant.

There will be other possible options but until the tool is implemented it is difficult to decide how these will appear. The ideas described in this section have given an outline to how the tool could appear. Decisions of how it will actually automate the method can be made once work on the storage control interface case study (and possibly others) has been completed and the application of the method to *real* problems has been standardised.

7.5 Summary

This chapter provided the conclusion of the thesis. Problems which were encountered during the research are described, followed by those which were actually solved. Conclusions on how the method applies for real examples which combine computation and data were described. Initial results from applying the method to such a real case were provided. Future developments for this work involve the application of the method to further case studies and the development of a tool based upon the *Maintainer's Assistant* for automating the method. Ideas of how such a tool should be built are presented at the end of the chapter.

Appendix A

WSL Syntax and Semantics

This Appendix briefly defines the syntax and semantics of the Wide Spectrum Language, WSL, as detailed in the document [83].

A.1 Background

A *wide spectrum language* is one which includes both low-level programming constructs and high-level abstract specifications within a single language. Such a language forms an ideal tool for developing methods for formal program development.

The language is defined in a series of stages or levels, with the lowest level being an extremely simple and tractable "kernel" language whose syntax is based on infinitary logic, and whose semantics is defined denotationally. In contrast to other work, a purely applicative kernel is not used; instead, the concept of state is included, using a *specification statement* which also allows specifications expressed in first order logic as part of the language, thus providing a genuine wide spectrum language.

Fundamental to the approach is the use of infinitary first order logic both to express the weakest preconditions of programs [29] and to define assertions and guards in the kernel language.

A program S is a piece of formal text, i.e. a sequence of formal symbols. There are two ways in which to interpret this:

1. Given a structure[1] M for the logical language $\mathcal{L}$ from which the programs are constructed, and an initial state space (from which a suitable final state space is constructed), a 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. A program can therefore be interpreted by itself as a function from structures to state transformations.

2. Given any formula **R** (which represents a condition on the final state), the formula $\mathbf{WP}(\mathbf{S}, \mathbf{R})$ (the *weakest precondition* of S on **R**) can be constructed. This is the weakest condition on the initial state such that the program S is guaranteed to terminate in a state satisfying **R** if it is started in a state satisfying $\mathbf{WP}(\mathbf{S}, \mathbf{R})$.

A.2 The Kernel Language

A.2.1 Syntax of Expressions

Expressions include variable names, numbers, strings of the form "*text* ...", the constants **N, R, Q, Z**, and the following operators and functions: (in the following e_1, e_2 etc. represent any valid 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 ** e_2, $e_1 \bmod e_2$, $e_1 \operatorname{div} e_2$, $\operatorname{frac}(e_1)$, $\operatorname{abs}(e_1)$, $\operatorname{sgn}(e_1)$, $\max(e_1, e_2, \ldots)$, $\min(e_1, e_2, \ldots)$, with the usual meanings.

Bit operators: (these are used in ReForm in translating Assembler code): $e_1 \overset{bit}{\wedge} e_2$, $e_1 \overset{bit}{\vee} e_2$, $e_1 \overset{bit}{\oplus} e_2$, $\overset{bit}{\neg}\ e_1$. The following implement floating point, decimal and

[1]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.

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

Map: The map operator $*$ returns the sequence obtained by applying a given function to each element of a given sequence: $(f * \mathrm{seq}\, a_1, a_2, \ldots, a_n) = \langle f(a_1), f(a_2), \ldots, f(a_n)\rangle$. A map to a set can also be applied in the obvious way.

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

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

A.2.2 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: $e_1 = e_2$, $e_1 \neq e_2$, $e_1 < e_2$, $e_1 \leq e_2$, $e_1 > e_2$, $e_1 \geq e_2$, equal(e_1, e_2), eq(e_1, e_2), even?(e_1), odd?(e_1);

Logical operators: $\neg\mathbf{Q}$, not$(\mathbf{Q})$, $\mathbf{Q}_1 \vee \mathbf{Q}_2$, $\mathbf{Q}_1 \wedge \mathbf{Q}_2$;

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

A.2.3 Syntax of Statements

The kernel language consists of four primitive statements, two of which contain formulae of infinitary first order logic, and two compound statements. Let $\mathbf{P}$ and $\mathbf{Q}$ be any formulae, and $\mathbf{x}$ and $\mathbf{y}$ be any non-empty sequences of variables. The following are primitive statements:

1. **Assertion:** {¶} is an assertion statement which acts as a partial **skip** statement. If the formula **Q** is true then the statement terminates immediately without changing any variables, otherwise it aborts (abnormal termination and non-termination are treated as equivalent, so a program which aborts is equivalent to one which never terminates);

2. **Guard:** [**Q**] is a guard statement. It always terminates, and enforces **P** to be true at this point in the program without changing the values of any variables. It has the effect of restricting previous nondeterminism to those cases which will cause **P** to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition (including **P**);

3. **Add variables:** add(**x**) adds the variables in **x** to the state space (if they are not already present) and assigns arbitrary values to them;

4. **Remove variables:** remove(**y**) removes the variables in **y** from the state space (if they are present).

There is a rather pleasing duality between the assertion and guard statements, and the **add** and **remove** statements.

The compound statements are as follows, for any kernel language statements $\mathbf{S_1}$ and $\mathbf{S_2}$, the following are also kernel language statements:

1. **Sequence:** ($\mathbf{S_1}$; $\mathbf{S_2}$) executes $\mathbf{S_1}$ followed by $\mathbf{S_2}$;

2. **Nondeterministic choice:** ($\mathbf{S_1} \sqcap \mathbf{S_2}$) choses one of $\mathbf{S_1}$ or $\mathbf{S_2}$ for execution, the choice being made nondeterministically;

3. **Recursion:** ($\mu X.\mathbf{S_1}$) where X is a *statement variable* (taken from a suitable set of symbols). The statement $\mathbf{S_1}$ may contain occurrences of X as one or more of its component statements. These represent recursive calls to the procedure whose body is $\mathbf{S_1}$.

This very simple kernel language is all that is needed to construct the wide spectrum language, for example an assignment such as $x := 1$ is constructed by adding x and restricting its value: add($\langle x \rangle$); $[x = 1]$. For an assignment such as $x := x + 1$ the new value of x must be recorded in a new variable, x' say, before copying it into x. So $x := x+1$ can be constructed as follows: add($\langle x' \rangle$); $[x' = x + 1]$; add($\langle x \rangle$); $[x = x']$; remove(x')

Three fundamental statements can be defined immediately:

$$\textbf{abort} =_{DF} \{\textbf{false}\} \qquad \textbf{null} =_{DF} [\textbf{false}] \qquad \textbf{skip} =_{DF} \{\textbf{true}\}$$

where **true** and **false** are universally true and universally false formulae, defined as: $\textbf{true} =_{DF} \forall x.(x = x)$ and $\textbf{false} =_{DF} \neg\forall x.(x = x)$.

A statement whose set of final states may be empty is called a "null statement". An example is the guard, [**false**], which is a "correct refinement" of *any* specification whatsoever. While any null statement, and guard statements in general, cannot be directly implemented, they are nonetheless a useful theoretical tool. Only null-free statements can be implemented, so it is important to be able to distinguish easily which statements are null-free. This is the motivation for the definition of the specification statement.

The notation $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ is defined where $\mathbf{x}$ is a sequence of variables and $\mathbf{x}'$ the corresponding sequence of "primed variables", and $\mathbf{Q}$ is any formula. This assigns new values to the variables in $\mathbf{x}$ so that the formula $\mathbf{Q}$ is true where (within $\mathbf{Q}$) $\mathbf{x}$ represents the old values and $\mathbf{x}'$ represents the new values. If there are no new values for $\mathbf{x}$ which satisfy $\mathbf{Q}$ then the statement aborts. The formal definition is:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q} =_{DF} (\{\exists \mathbf{x}'.\mathbf{Q}\}; (add(\mathbf{x}'); ([\mathbf{Q}]; (add(\mathbf{x}); ([\mathbf{x} = \mathbf{x}']; remove(\mathbf{x}'))))))$$

An important property of this specification statement is that it is guaranteed null-free.

A.3 WSL Syntax and Semantics

The kernel language is 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 "zero level" language which forms the foundation for all the others.

A.3.1 The First Level Language

The first level language consists of the following constructs:

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

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

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

$$\underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{S_1}\ \underline{\mathbf{else}}\ \mathbf{S_2}\ \underline{\mathbf{fi}} =_{DF} (([\mathbf{B}]; \mathbf{S_1}) \sqcap ([\neg\mathbf{B}]; \mathbf{S_2}))$$

3. Specification statement: This was discussed in Section A.2.3:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q} =_{DF} (\{\exists\, \mathbf{x}'.\mathbf{Q}\}; (add(\mathbf{x}'); ([\mathbf{Q}]; (add(\mathbf{x}); ([\mathbf{x} = \mathbf{x}']; remove(\mathbf{x}'))))))$$

4. Simple Assignment: If $\mathbf{Q}$ is of the form $\mathbf{x}' = \mathbf{t}$ where $\mathbf{t}$ is a list of terms which do not contain $\mathbf{x}'$ then the assignment is abbreviated as follows:

$$\mathbf{x} := \mathbf{t} =_{DF} \mathbf{x} := \mathbf{x}'.(\mathbf{x}' = \mathbf{t})$$

If $\mathbf{x}$ contains a single variable, $x := t$ is written for $\langle x \rangle := \langle t \rangle$.

5. Nondeterministic Choice: The "guarded command" of Dijkstra [29]:

$$\begin{array}{lll} \underline{\mathbf{if}}\ \mathbf{B_1} \rightarrow \mathbf{S_1} & =_{DF} & (\{\mathbf{B_1} \vee \mathbf{B_2} \vee \ldots \vee \mathbf{B_n}\}; \\ \Box\ \mathbf{B_2} \rightarrow \mathbf{S_2} & & \quad (\ldots((([\mathbf{B_1}]; \mathbf{S_1}) \sqcap \\ \quad\ldots & & \quad\quad ([\mathbf{B_2}]; \mathbf{S_2})) \sqcap \\ \Box\ \mathbf{B_n} \rightarrow \mathbf{S_n}\ \underline{\mathbf{fi}} & & \quad\quad \ldots)) \end{array}$$

6. Deterministic Iteration: A **while** loop is defined using a new recursive procedure X which does not occur free in S:

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

7. Nondeterministic Iteration:

$$\begin{array}{ll}
\underline{\mathrm{do}}\ \mathrm{B}_1 \rightarrow \mathrm{S}_1 & =_{DF} \underline{\mathrm{while}}\ (\mathrm{B}_1 \vee \mathrm{B}_2 \vee \ldots \vee \mathrm{B}_n)\ \underline{\mathrm{do}} \\
\square\ \mathrm{B}_2 \rightarrow \mathrm{S}_2 & \quad \underline{\mathrm{if}}\ \mathrm{B}_1 \rightarrow \mathrm{S}_1 \\
\ldots & \quad\ \square\ \mathrm{B}_2 \rightarrow \mathrm{S}_2 \\
\square\ \mathrm{B}_n \rightarrow \mathrm{S}_n\ \underline{\mathrm{od}} & \quad \ldots \\
 & \quad \square\ \mathrm{B}_n \rightarrow \mathrm{S}_n\ \underline{\mathrm{fi}}\ \underline{\mathrm{od}}
\end{array}$$

8. Local Variables: These should be properly initialised, for example by using the scheme in [29]:

$$\underline{\textbf{begin}}\ \mathrm{x} : \mathrm{S}\ \underline{\mathrm{end}} =_{DF} (\mathit{add}(\mathrm{x}); (\mathrm{S}; \mathit{remove}(\mathrm{x})))$$

9. Initialised Variables:

$$\underline{\textbf{begin}}\ \mathrm{x} := \mathrm{t} : \mathrm{S}\ \underline{\mathrm{end}} =_{DF} (\mathit{add}(\mathrm{x}); ([\mathrm{x} = \mathrm{t}]; (\mathrm{S}; \mathit{remove}(\mathrm{x}))))$$

10. Counted Iteration. Here, the loop body S must not change i, b, f or s:

$$\begin{array}{l}
\underline{\textbf{for}}\ i := b\ \underline{\mathrm{to}}\ f\ \underline{\textbf{step}}\ s\ \underline{\mathrm{do}}\ \mathrm{D}\ \underline{\mathrm{od}} =_{DF} \underline{\textbf{begin}}\ i := b : \underline{\mathrm{while}}\ i \leq f\ \underline{\mathrm{do}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{S};\ i := i + s\ \underline{\mathrm{od}}\ \underline{\mathrm{end}}
\end{array}$$

A.3.2 Exit Statements

The programming language includes statements of the form $\underline{\mathrm{exit}}(n)$, where n is an integer, (*not* a variable) which occur within loops of the form $\underline{\mathrm{do}}$ S $\underline{\mathrm{od}}$ where S is a statement. They are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form $\underline{\mathrm{exit}}(n)$ which causes the program to exit the n enclosing loops.

To simplify the language **exit**s which leave a block or a loop other than an unbounded loop are not allowed. More formally, the notion of a *simple statement* is defined:

Definition A.3.1 *A simple statement is any first level language statement apart from: (1) a sequence, (2) a deterministic or nondeterministic choice, and (3) a* **do** *...* **od** *loop.*

A simple statement in the body of an action may not have an action call as a component or subcomponent. A simple statement may not have an **exit**(k) statement as a component unless it occurs within k or more nested **do** ... **od** loops. Note that **do** ... **od** loops and action systems are allowed as components of a simple statement, but no simple statement apart from **call** can affect the value of **action** and no simple statement apart from **exit** can affect the value of **depth**.

The interpretation of these statements in terms of the first level language is as follows (but see the next section for the full interpretation when action systems are also involved):

There is an integer variable **depth** which records the current depth of nesting of loops. At the beginning of the program the assignment **depth** := 0 is inserted and each **exit** statement **exit**(k) is translated: **depth** := **depth**$-k$ since it changes the depth of "current execution" by moving out of k enclosing loops. To prevent any more statements at the current depth being executed after an **exit** statement has been executed, all statements are surrounded by "guards" which are **if** statements which will test **depth** and only allow the statement to be executed if **depth** has the correct value. Each unbounded loop **do** S **od** is translated:

depth := n; **while** **depth** = n **do** $\mathbf{guard_n}$(S) **od**

where n is an integer constant representing the depth of the loop (1 for an outermost loop, 2 for double nested loops etc.) and $\mathbf{guard_n}$(S) is the statement S with each component statement guarded so that if the depth is changed by an **exit** statement then no more statements in the loop will be executed and the loop will terminate. The important property of a guarded statement is that it will only be executed if **depth** has the correct value. Thus: $\{\mathbf{depth} \neq n\}$; $\mathbf{guard_n}$(S) $\approx$ $\{\mathbf{depth} \neq n\}$; **skip**. So for example, the program:

```
do do $last := item[i]$;
        $i := i + 1$;
        if $i = n + 1$ then $write(\textbf{count})$; exit(2) fi;
        if $item[i] \neq last$ then $write(\textbf{count})$; exit(1)
                    else $\textbf{count} := \textbf{count} + \textbf{number}[i]$ fi od;
   $\textbf{count} := \textbf{number}[i]$ od
```

translates to the following:

```
depth := 1;
while depth = 1 do
        depth := 2;
        while depth = 2 do
            last := item[i];
            $i := i + 1$;
            if $i = n + 1$ then $write(\textbf{count})$; depth := depth − 2 fi;
            if depth = 2
                then if item[i] ≠ last then $write(\textbf{count})$; depth := depth − 1
                                    else count := count + number[i] fi fi od;
        if depth = 1 then count := number[i] fi od
```

A.3.3 Action Systems

An *Action System* is a set of parameterless mutually recursive procedures together with the name of the first action to be called. A program written using labels and jumps translates directly into an action system. Note however that if the end of the body of an action is reached, then control is passed to the action which called it (or to the statement following the action system) rather than "falling through" to the next label. The exception to this is a special action called the terminating action, denoted Z, which when called results in the immediate termination of the whole action system.

Definition A.3.2 *An action is a parameterless procedure acting on global variables. It is written in the form $A \equiv \mathbf{S}$ where A is a statement variable (the name of the action) and* S *is a statement (the action body). A set of (mutually recursive) actions is called an action system. The difference between actions and ordinary procedures is that an action system may include calls to the special action name Z which has no definition. A call to Z causes termination of the whole action system even if there are unfinished recursive calls. An occurrence of a statement* __call__ *X within the action body refers to a call of another action. It should be noted that an action system is a statement (in fact a simple statement as defined above) which can form a component of compound statements—including other action systems. A simple statement (apart from an action system) may not contain action calls.*

An example of an action system is:

> __actions__ $\mathbf{A_1}$:
>
> $\mathbf{A_1} \equiv \mathbf{S_1}$.
>
> $\mathbf{A_2} \equiv \mathbf{S_2}$.
>
> ...
>
> $\mathbf{A_n} \equiv \mathbf{S_n}$. __endactions__

(where statements $\mathbf{S_1}, \ldots, \mathbf{S_n}$ must have no __exit__(k) statements within less than k nested loops). More details on how this is defined using the kernel language can be found in [83].

A.3.4 Procedures with Parameters and Local Variables

Recursive procedures and action systems are similar in several ways, the differences are:

- There is nothing in a **where** statement which corresponds to the Z action: all procedures must terminate normally (and thus a "regular" set of recursive procedures could never terminate);
- Procedure calls can occur anywhere in a program, for example in the body of a **while** loop: action calls cannot occur as components of simple statements.

An action system which does not contain calls to Z can be translated to a **where** clause (the converse is only true provided no procedure call is a component of a simple statement):

```
actions A_1 :
A_1 ≡ S_1.
A_2 ≡ S_2.
...
A_n ≡ S_n. endactions
```

is equivalent to:

```
begin
    A_1
where
    procA_1 ≡ S_1[A_i/callA_i].,
    procA_2 ≡ S_2[A_i/callA_i].,
    ...
    procA_n ≡ S_n[A_i/callA_i].
end
```

A.3.5 Functions and Boolean Function

A **where** clause may also include functions and boolean functions with parameters. These are defined in terms of their "procedural equivalent" (a procedure which stores the result of the function in a given variable) and they are allowed to use local variables and have side effects (using the $\lceil S;\, e \rfloor$ notation for expressions with side effects). For the purposes of this transformation system we have decided to assume that all expressions, functions and Boolean functions are side-effect free and non-recursive: in other words, all side effects must be made explicit as statements. This simplifies the design of many transformations without adding significantly to the complexity of translators into WSL. Note that the implementation of the transformation system (which will be largely written in WSL) may use functions with side effects (particularly in the data table system), however, the specification of these functions must be as pure functions. In other words, the side effects are purely for implementation efficiency and any user of the functions can assume they are pure functions.

Appendix B

Transformations

This appendix will describe some of the very basic transformations used by the Maintainer's assistant and will list the full set of those used by the case studies described in this thesis.

B.1 Basic Groups

The transformations implemented in the Maintainer's Assistant can be classified into nine groups which represent the type of action which the transformation performs. The groups and some of the main transformations which compose them can be described as follows:

1. **MOVE.**

   Transformations in this group involve the movement of selected parts within a specification, code or intermediate level. Examples of these include $\leftarrow\leftarrow$ **Multi – Move, Separate** $\rightarrow\rightarrow$, **Swap-With-Next**, **Swap-With-Previous**. As their names suggest, they all involve swapping sections of code or specification if the semantics remain the same. The arrows indicate which direction this move will take place. The following illustrates an example of this:

var a:=0 ;

actions: MAIN:

MAIN == **if (a<5) then a:=4; a:=(a+1)**

else a:=(a+1) fi.

end-actions end

TRANSFORMATION = **Separate** $\rightarrow\rightarrow$

(The code in bold face is the part which has been selected for transformation. This transformation should pull the assignment a:=(a+1) out of the loop, leaving a redundant else...)

var a:=0 ;

actions: MAIN:

MAIN == **if (a<5) then a:=4 else skip fi**; a:=(a+1).

end-actions end

2. **JOIN.**

These transformations involve combining sections of code or specification. Examples of these include $\leftarrow\leftarrow$ **Absorb, Merge** $\rightarrow\rightarrow$, **Join-Adjacent-Cases, Forward-Expand-Cond, Put-Into-Next-Loop**. Again, the names are representative of the function of the transformation and the following illustrates a simple example of one of these:

if (a>5) then a:=6 fi; **if (a<=5) then a:=4 fi**

TRANSFORMATION = $\leftarrow\leftarrow$ **Merge**

(This will include the highlighted statement in the previous statement so in this case there is only one main conditional statement which can easily be simplified.)

if (a>5) then a:=6 ; if (a<=5) then a:=4 fi elseif (a<=5) then a:=4 fi

3. **USE/APPLY.**

These transformations involve using values or conditions within the selected part to simplify other parts of the specification/code. Examples include **Apply $\rightarrow\rightarrow$, $\leftarrow\leftarrow$ Use, Apply-Assignment-Forwards**. An illustration of the use of the last transformation is:

```
a:=8;
b:=5;
if (a>5) then b:=(a-5); a:=(a-1); b:=(b+2);
while (a>5) do b:=(a-5); a:=(a-1); b:=(b+2) od fi
```

TRANSFORMATION = **Apply-Assignment-Forwards**

(Rewrites the if statement, using the assigned value of a.)

```
a:=8;
b:=5;
b:=(a-5);
a:=(a-1);
b:=(b+2);
while (a>5) do b:=(a-5); a:=(a-1); b:=(b+2) od
```

4. **REORDER.**

The REORDER group of transformations rearranges the order of specification or code (as long as the semantics are not affected). Transformations in this group include **Take – Out $\rightarrow\rightarrow$, Reorder-Condition, Invert-Loop, Fully-Factor-While.** An example of one of these is:

```
var a:=0 ;
actions: MAIN:
MAIN == do if (a>=5) then a:=8; exit(1) else a:=(a+2) fi od.
end-actions end
```

TRANSFORMATION = **Reorder-Condition**

(Changes the order of the conditional statement so that $a < 5$ is first tested.)

var a:=0 ;
actions: MAIN:
MAIN == do **if (a<5) then a:=(a+2) else a:=8; exit(1) fi** od.
end-actions end

5. **REWRITE.**

   These transformations replace the selected part of the specification with something of the same semantics but different syntax. Examples include **Change-Action-Name, Else → Elsif, Cond-Abort → Assert, While → Proc, Expand-Proc-Call.** An illustration of this last transformation is:

   var a:=0;
   begin **main(var)**;a:=8 where proc main(var)==
   while (a<5) do a:=(a+2) od. end end

   TRANSFORMATION = **Expand-Proc-Call**

   (This replaces the procedure call with the procedure itself.)

   var a:=0;
   begin **while (a<5) do a:=(a+2) od; a:=8**
   where proc main(var)==while (a<5) do a:=(a+2) od. end end

6. **INSERT.**

   The INSERT transformations will introduce new constructs such as loops, invariants, assertions and comments. The transformations in this group include **Add-Assert** , **Add-Comment** , **Add-New-Definition, Add-Invariant-After** , **Include-Dummy-Else** , **Add-Entire-Loop**. The example of code transformed in the REWRITE description can be further transformed with an INSERT transformation:

   var a:=0;
   begin **while (a<5) do a:=(a+2) od; a:=8**
   where proc main(var)==while (a<5) do a:=(a+2) od. end end

TRANSFORMATION = **Add-Entire-loop**

(This makes a loop around the expanded procedure call so that is self-contained.)

var a:=0;
begin **do while (a<5) do a:=(a+2) od; a:=8 od**
where proc main(var)==while (a<5) do a:=(a+2) od. end end

7. **SIMPLIFY/DELETE.**

These transformations remove redundant statements or replace expressions with simpler forms. They include **Collapse-While, Delete, Prune-Loop, Remove-Assignment, Remove-Redundant-Proc, Simplify, Use-Condition-In-If.** The example of code transformed with an INSERT transformation (above) can be further transformed by:

var a:=0;
begin do while (a<5) do a:=(a+2) od; a:=8 od
where proc main(var)==while (a<5) do a:=(a+2) od. end end

TRANSFORMATION = **Remove-Redundant-Proc**

(The procedure is removed since it is no longer called.)

var a:=0;
do while (a<5) do a:=(a+2) od; a:=8 od end

8. **MULTIPLE.**

This is a group of transformations which carries out many of the above transformations in one step. Examples of these are **Delete-All-Assertions, Expand-All-Calls, Merge-All-Assignments, Replace-All-Values, Simplify-All-Expressions.** MULTIPLE Transformations involve selecting *all* the specification or code and performing the same transformation many times.

9. **COMPLEX.**

This last group of transformations involves combining several types of transformation for a particular purpose. These transformations are really *strategies* (see chapter 3) for refining or simplifying specifications or code, but can be selected as options if the user has a particular objective in mind. Examples of COMPLEX transformations are **Collapse-Action-System** , **Expand-And-Factor** , **Fully-Parameterise** , **Loop → Recursion**, **Proc-Body → Action-System**, **Remove-Recursion** , **Substitute-And-Delete**, **Remove-Tail-Recursion-1**.

An illustration of this last transformation is:

var a:=0;
begin main(var) where **proc main(var)==if (a<5) then a:=(a+2);**
main(var) fi. end;
a:=8 end

TRANSFORMATION = **Remove-Tail-Recursion-1**

(This is so as to convert the basic recursive statement into an iterative one; i.e. if-then statement becomes a while statement)

var a:=0;
begin main(var) where **proc main(var)==**
while (a<5) do a:=(a+2) od. end; a:=8 end

B.2 Transformations Used in Case Studies

A list of transformations is given for each case study (followed by the group each transformation is in, as described in section B.1). This list is the set of transformations which were first applied to the specification to produce the code and hence form part of the *derivation history* for each case study.

B.2.1 Integer Square Root

1. **Remove-Redundant-Function** (SIMPLIFY/DELETE) This is used to replace $\sqrt{}$ and $\lfloor\,\rfloor$ with the definitions of these functions.
2. **Substitute-And-Delete** (COMPLEX)
3. **Make-And-Use-Assertions** (COMPLEX)
4. **Add-Entire-Loop** (INSERT)
5. **Apply-Condition-To-Next** (USE/APPLY)
6. **Add-Entire-Loop** (INSERT)
7. **Make-Proc** (REWRITE)
8. **Change-Local-Variable** (REWRITE)
9. **Replace-With-Value** (REWRITE)
10. **Remove-Parameter** (REWRITE)
11. **Loop $\rightarrow$ While** (REWRITE)
12. **Simplify** (SIMPLIFY/DELETE)
13. **Change-Local-Variable** (REWRITE)
14. **Replace-With-Value** (REWRITE)

B.2.2 Library Case Study

1. **Multi-Move $\rightarrow\rightarrow$ (MOVE)**
2. **Expand-Funct-Call (REWRITE)**
3. **Simplify (SIMPLIFY/DELETE)**
4. **Add-New-Guarded-Clause (INSERT)**
5. **Expand-General-Case (REWRITE)**
6. **Make-Assign-Before (INSERT)**
7. **Make-Into-Parameter (REWRITE)**
8. **Add-New-Definition (INSERT)**
9. **Simplify (SIMPLIFY/DELETE)**
10. (Transformations 6 to 9 repeated for each variable)
11. **Insert/Assert (INSERT)**
12. **Add-Loop-Body (INSERT)**
13. **Prune-Loop (SIMPLIFY/DELETE)**
14. **Elsif_True $\rightarrow$ Else (REWRITE)**
15. **Make-Proc (REWRITE)**
16. **Make-Proc-Call (REWRITE)**
17. (Transformations 14, 15 and 16 repeated to introduce each of the six procedures within a loop)
18. **Separate $\rightarrow$ (MOVE)**
19. **Prune-Cond (USE/APPLY)**
20. **Use-Assertion (USE/APPLY)**
21. **Brackets $\rightarrow\rightarrow$ (REORDER)**

22. **Add-New-Definition (INSERT)**

23. **Add-Assign (INSERT)**

24. **Replace-And-Simplify (SIMPLIFY)**

25. **Add-New-Definition (INSERT)**

26. **Substitute-And-Delete (SIMPLIFY)**

27. (... and a sequence of transformations 22 through to 26 for each type)

28. **Make-Funct-Call (REWRITE)**

29. **Simplify (SIMPLIFY)**

Chapter 5 describes the refinement of the library case study. To simplify this description, only stages of the derivation history were listed, rather than the individual transformations. The above list provides more detail on these transformations and the correspondence between this list and the stages is as follows:

- Rearrange specification = Transformation 1 (performed several times)
- Rewrite preconditions = Transformations 2,3 (performed several times)
- Introduction of Guarded Commands = Transformations 4,5
- Introduction of Global Variables = Transformations 6 - 13
- Introduction of Procedures = Transformations 14 - 20
- Replacement of Atomic Specification = Transformations 21 - 29

Bibliography

[1] ANSI/IEEE., *IEEE Standard Glossary of Software Engineering Terminology.* ANSI/IEEE Std 729, 1983.

[2] Arnold, R.S. and Parker, D.A., The dimensions of healthy maintenance. *Proceedings of the 6th International Conference on Software Engineering*, 1982, pp 10-27.

[3] Babich, W.A., *Software Configuration Management: Coordination for Team Productivity.* Addison-Wesley Publishing Company, 1986.

[4] Balzer, R., Transformational Implementation: An Example. *IEEE Trans. Software Eng.*, vol.SE-7, no.1.

[5] Back, R. J. R., *Correctness Preserving Program Refinements.* Mathematisch Centrum, Mathematical Centre Tracts 131.

[6] Bauer, F.L., A Philosophy of Programming. *Lecture Notes in Computer Science.*, vol.46. Berlin-Heidelberg- New York: Springer 1976, pp 194-241.

[7] Bauer, F.L., Moller, B., Partsch, H. and Pepper, P., Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming. *IEEE Trans. Software Eng.*, vol.15, no.2, pp 165-180.

[8] Baxter, I.D., Design Maintenance Systems. *Communications of the ACM*, April 1992, vol.35, no.4, pp 73-89.

[9] Bersoff, E.H., Henderson, V.D. and Siegel, S.G., *Software Configuration Management: An Investment in Product Integrity.* Prentice-Hall, Inc., N.J., 1980.

[10] Bennett, K.H., *NATS Procurement Strategies for Highly Maintainable Software.* Centre for Software Maintenance Ltd., 1991.

[11] Bennett, K.H., The Software Maintenance of Large Software Systems: Management, Methods and Tools. *Reliability Engineering and System Safety,* vol.32, 1991, pp 135-154.

[12] Bennett, K.H., Cornelius, B., Munro, M. and Robson, D., Software Maintenance, In: McDermid, J. (ed.), *Software Engineer's Reference Book.* Butterworth-Heinemann Ltd, 1991, pp 20/1-18.

[13] Biggerstaff, T.J., Design Recovery for Maintenance and Reuse. *Computer,* July 1989, pp 36-49.

[14] Bergeretti, J.F. and Carré, B.A., Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.,* vol.7, no.1, pp 37-61.

[15] Bledsoe, W. and Loveland, D. (ed.), *Automated Theorem Proving: After 25 Years.* American Mathematical Society Contemporary Mathematical Series, 1984.

[16] Boehm, B., *Software Engineering Economics.* Prentice-Hall, NJ, 1981.

[17] Boehm, B., Brown, J., Kaspar, H., Lipow, M., MacLeod, G., Merritt, M., *Characteristics of Software Quality.* North-Holland Publishing Company, Amsterdam, 1987.

[18] Boyle, J.M., Lisp to Fortran - Program Transformation Applied, In: Pepper, P. (ed.) *Program Transformation and Programming Environments. Report on a Workshop directed by F.L.Bauer and H.Remus.* NATO ASI Series F: Computer and System Sciences, vol.8, Springer-Verlag, 1984.

[19] Boyle, J.M. and Muralidharan, M.N., Program Reusability through Program Transformation. *IEEE Transactions on Software Engineering,* vol.SE-10, no.5, 1984, pp 574-588.

[20] Brooks, F.P., Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies,* vol.18, 1983, pp 543-554.

[21] Brooks, F.P., No silver bullet: essence and accidents of software engineering. *Computer,* vol.20, no.4, 1987, pp 10-19.

[22] Broy, M., Algebraic Methods for Program Construction: The Project CIP, In: Pepper, P. (ed.) *Program Transformation and Programming Environments. Report on a*

Workshop directed by F.L.Bauer and H.Remus. NATO ASI Series F: Computer and System Sciences, vol.8, Springer-Verlag, 1984.

[23] Burstall, R.M. and Darlington, J.A., A Transformation System for Developing Recursive Programs. *Journal of the ACM*, vol.24, 1977, pp 44-67.

[24] Cheatham, T.E., Jnr., Holloway, G.H. and Townley, J.A., Program Refinement by Transformation. *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, Calif., March 1981, pp 430-437.

[25] Chikofsky, E.J. and Cross II, J.H., Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, vol.7, no.1, 1990, pp 13-17.

[26] Constantaine, L.L., Stevens, W.P. and Myers, G.J., Structured Design. *IBM Systems Journal*, vol. 2, pp 115-139, 1974.

[27] Conte, S.D., Dunsmore, H.E. and Shen, V.Y., *Software Engineering Metrics And Models.* The Benjamin/Cummings Publishing Company, Inc., 1986.

[28] Denning, D.E., A lattice model of secure information flow. *Commun. ACM*, vol.19, no.5, pp 236-242.

[29] Dijkstra, E.W., *A Discipline of Programming.* Prentice Hall, 1976.

[30] Dijkstra, E.W., Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, vol.18, no.4, pp 453-457.

[31] DTI and NCC, *The STARTS Guide.* NCC, 1987.

[32] Fairley, R.E., *Software Engineering Concepts.* McGraw-Hill Book Company, 1985.

[33] Feather, M.S., A System for Assisting Program Transformation. *ACM Transactions on Programming Languages and Systems 4, (1).* Jan. 1982, pp 1-20.

[34] Feather, M.S., A Survey and Classification of Some Program Transformation Approaches and Techniques. *IFIP Working Conf. on Program Specification and Transformation*, Bad Tolv Germany, April 1986.

[35] Floyd, R.W., Toward interactive design of correct programs. *Proc. IFIP Congress '71*, 1972, pp 7-10.

[36] Garnett, E.S. and Mariani, J.A., Software reclamation. *Software Engineering Journal*, May 1990, pp 185-191.

[37] Griffiths, M., Program Production by Successive Transformation. *Lecture Notes in Computer Science.*, vol.46. Berlin-Heidelberg- New York: Springer 1976, pp 125-152.

[38] Griffiths, M., Development of the Schorr-Waite Algorithm. *Lecture Notes in Computer Science.*, vol.69. Berlin-Heidelberg- New York: Springer 1979, pp 464-471.

[39] Halstead, M.H., *Elements of Software Science.* New York, Elsevier North-Holland, 1977.

[40] Hoare, C.A.R., An Axiomatic Basis for Computer Programming. *Communications of the ACM*, vol.12, 1969.

[41] Horwitz, S., Reps., T. and Binkley, D., Interprocedural slicing using dependence graphs. *Proc. ACM SIGPLAN 88 Conf. Programming Language Design and Implementation*, Atlanta, GA, June 1988, pp 35-46.

[42] Jones, C.B., *Systematic Software Development using VDM.* Second Edition. Prentice Hall, 1990.

[43] Kafura, D. and Reddy, G.R., The Use of Software Complexity Metrics in Software Maintenance. *IEEE Trans. Software Eng.*, vol.SE-13, pp 335-343.

[44] Kenning, R., Configuration Management : State of the Art. *Technical Report.*

[45] King, S. and Sørensen, I., *From specification, through design, to code: a case study in refinement.* IBM United Kingdom Laboratories Ltd, 1988.

[46] Leach, R.J., Software Metrics and Software Maintenance. *Software Maintenance: Research and Practise*, vol.2, pp 133-142.

[47] Lehman, M.M., Programs, life cycles, and laws of software evolution. *Proceedings of IEEE*, vol.19, pp 1060-1076.

[48] Lehman, M.M. and Belady, L.A., A model of large program development. *IBM Systems Journal*, vol.15, pp 225-252.

[49] Letovsky, S., Cognitive Processes in Program Comprehension, In: Soloway, E. and Iyengar, S.(eds.) *Empirical Studies of Programmers.* Ablex, Norwood, 1986.

[50] Lewis, J.A. and Henry, S.M., On the Benefits and Difficulties of a Maintainability via Metrics Methodology. *Software Maintenance: Research and Practise*, vol.2, pp 113-131.

[51] Lientz, B. and Swanson, E.B., *Software Maintenance Management.* Addison-Wesley, 1980.

[52] Lientz, B., Swanson, E.B. and Tompkins, G.E., Characteristics of application software maintenance. *Communications of the ACM*, 21, 1978, pp 466-471.

[53] Litteck, H.J. and Wallis, P.J.L., Refinement methods and refinement calculi. *Software Engineering Journal*, May 1992, vol.7, no.3, pp 219-229.

[54] Littman, D.C., Pinto, J., Levovsky, S. and Soloway, E., Mental Models and Software Maintenance, In: Soloway, E. and Iyengar, S.(eds.) *Empirical Studies of Programmers.* Ablex, Norwood, 1986, pp 80-96.

[55] Longstreet, D., *Software Maintenance And Computers.* IEEE Computer Society Press, 1990.

[56] Mair, P.A., *IPSE: State of the Art Report.* NCC Ltd., 1986.

[57] McCabe, T.J., A Complexity Measure. *IEEE. Trans. Software Eng.*, vol.SE-2, no.4, pp 308-320, 1976.

[58] McDermid, J., Introduction and Overview to Part II, In: McDermid, J. (ed.), *Software Engineer's Reference Book.* Butterworth-Heinemann Ltd, 1991, pp 1-15.

[59] Morgan, C., *Programming from Specifications.* Programming Research Group, University of Oxford, 1990.

[60] Moriconi, M., A designer/verifier's assistant. *IEEE Trans. Software Eng.*, vol.SE-5, pp 387-401.

[61] Moriconi, M., Approximate Reasoning About the Semantic Effects of Program Changes. *IEEE Trans. Software Eng.*, vol.16, no.9, Sept.1990.

[62] Neilson, D., Hierarchical refinement of a Z specification. In Nori, K.V. (ed.): Foundations of Software Technology and Theoretical Computer Science '87. *Lect. Notes Comput. Sci. 287*, 1987, pp 376-399.

[63] Newton, J., *Library Specification, Refinement and Change in Z and WSL.* University of Durham, 1993.

[64] Newton, J. and Bennett, K., Designing Systems for Future Maintainability: A Case Study. *Proc. IEEE Conference on Software Maintenance 1993*, Sept. 1993, pp 272-280.

[65] Oman, P., Maintenance Tools. *IEEE Software*, May 1990, pp 63.

[66] Parikh, G., *Techniques of Program and System Maintenance.* Winthrop Publishers, 1982.

[67] Partsch, H., *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[68] Partsch, H., The CIP Transformation System, In: Pepper, P. (ed.) *Program Transformation and Programming Environments. Report on a Workshop directed by F.L.Bauer and H.Remus.* NATO ASI Series F: Computer and System Sciences, vol.8, Springer-Verlag, 1984.

[69] Partsch, H. and Steinbruggen, R., Program Transformation Systems. *ACM Computing Surveys V15 No.3*, 1983, pp 199-236.

[70] Partsch, H., Transformational Program Development in a Particular Problem Domain. *Science of Computer Programming*, Vol. 7, No.2, 1986, pp 99-248.

[71] Perry, D.E., Software interconnection models. *Proc. 9th Int. Conf. Software Engineering*, Monterey, CA, Mar.1987, pp 61-69.

[72] Potter, B., Sinclair, J. and Till, D., *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[73] Pressman, R.S., *Software Engineering: A Beginner's Guide.* McGraw-Hill Book Company, 1988.

[74] Rekoff Jr., M.G., On Reverse Engineering. *IEEE Trans. Systems, Man, and Cybernetics*, March 1985, pp 244-252.

[75] Rich, C. and Waters, R. (ed.), *Readings in Artificial Intelligence and Software Engineering.* Morgan Kaufmann Publishers, Inc., 1986.

[76] Shneiderman, B. and Mayer, R., Syntactic Semantic Interactions in Programming Behaviour: A Model. *International Journal of Computer and Information Science*, vol.8, 1979, pp 219-238.

[77] Simon, A., *Requirements for a Software Maintenance Support Environment.* MSc Thesis, Centre for Software Maintenance, University of Durham, October 1991.

[78] Sommerville, I., *Software Engineering.* Addison-Wesley Publishing Company, 1985.

[79] Spivey, J.M., *The Z Notation - A Reference Manual.* Programming Research Group, University of Oxford, December 1989.

[80] Terry, B. and Lodge, D., Terminology for Software Engineering Environment and Computer-Aided Software Engineering. *Software Engineering Notes*, Vol.15, No.2, pp 83-94.

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

[82] Ward, M.,*The Architecture of the "Maintainer's Assistant".* Centre for Software Maintenance, University of Durham, 1989.

[83] Ward, M., *WSL Syntax and Semantics.* Centre for Software Maintenance Ltd, 1993.

[84] Ward, M., Calliss, F.W. and Munro, M., The Maintainer's Assistant. *Proceedings of Conference on Software Maintenance*, Miami, Florida, October 1989, pp 22-29.

[85] Ward, M., Calliss, F.W. and Munro, M., *The Use of Transformations in "The Maintainer's Assistant".* Centre for Software Maintenance, University of Durham.

[86] Ward, M.,*The Largest* **true** *Square Problem.* Centre for Software Maintenance, University of Durham, April 1990.

[87] Waters, R., The Programmer's Apprentice: A Session with KBEmacs. *IEEE Trans. Software Eng.*, Vol.11, No.11, pp 1296-1320, November 1981.

[88] Weiser, M., Program Slicing. *IEEE Trans. Software Eng.*, vol.SE-10, pp 352-357, July 1984.

[89] Whysall, P., Refinement, In: McDermid, J. (ed.), *Software Engineer's Reference Book.* Butterworth-Heinemann Ltd, pp 1-15, 1991.

[90] Wordsworth, J.B., *A Z Development Method.* IBM United Kingdom Laboratories Ltd, July 1987.

[91] Wos, L., *et al*, *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.

[92] Yang, H., *How Does The "Maintainer's Assistant" Start?* Centre for Software Maintenance, University of Durham.