PROVING PROGRAM REFINEMENTS AND TRANSFORMATIONS

Martin Ward

DPhil Thesis 1989

Abstract

In this thesis we develop a theory of program refinement and equivalence which can be used to develop practical tools for program development, analysis and modification. The theory is based on the use of general specifications and an imperative kernel language. We use weakest preconditions, expressed as formulae in infinitary logic to prove refinement and equivalence between programs.

The kernel language is extended by means of “definitional transformations” which define new concepts in terms of those already present. The extensions include practical programming constructs, including recursive procedures, local variables, functions and expressions with side-effects. This expands the language into a “Wide Spectrum Language” which covers the whole range of operations from general specifications to assignments, jumps and labels. We develop theorems for proving the termination of recursive and iterative programs, transforming specifications into recursive programs and transforming recursive procedures into iterative equivalents. We develop a rigorous framework for reasoning about programs with **exit** statements that terminate nested loops from within; and this forms the basis for many efficiency-improving and restructuring transformations. These are used as a tool for program analysis and to derive algorithms by transforming their specifications. We show that the methods of top-down design and program verification using assertions can be viewed as the application of a small subset of transformations.

Introduction

In this thesis we develop the basis of a theory for program refinement and equivalence which can be used to develop practical tools for program development, analysis and modification. In contrast to many workers in the field of program transformation [Bauer 79], [Bauer & Wossner 82], [Burstall & Darlington 77], [de Bakker & van Vilet 81], [Griffiths 76], [Guttag 77], [Hoare et al 87], [Partsch & Pepper 76], [Pepper 84], [Townley 82], [Wosser et al 79], we will be using constructive (ie model-based rather than algebraic) specifications and an imperative kernel language which is extended to deal with applicative constructs, rather than an applicative kernel which is extended to imperative constructs. [Majester 77] has pointed out some limitations of algebraic specifications, and in this author's opinion a constructive specification based on logic and set theory is easier to understand and work with than a set of algebraic equations. The use of an imperative kernel language reduces the number of "definitional transformations" required to deal with side-effecting functions and other "difficult" programs.

A major aim of our work is to develop a system which can be applied to any program written using any methods. This is so that the theory can be applied to the development of practical systems for software maintenance as well as for the development of programs from specifications. Most of the research into formal methods of programming has concentrated on program development rather than maintenance: however, between 60-80% [Leintz & Swanson 80] of the total software cost is spent on the maintenance of existing programs. This means that even if methods can be developed which dramatically reduce program development cost, these methods will have little impact on total software costs, since any gain from increased development will be swallowed up by the increased maintenance costs. In fact if the resulting programs are harder to maintain there will be a net loss in cost effectiveness if the new methods are used! Current research work at Durham indicates that the theory developed here can form the foundation of software maintenance tools which transform existing programs into versions which are easier to understand and modify, without the transformation process introducing new errors.

Engeler [Engeler 68] used infinitary logic to define a formula corresponding to each deterministic program which will be true exactly on those initial states for which the program terminates. Back [Back 80] extended this idea to nondeterministic programs by showing that the weakest precondition of a program could be defined as a formula of infinitary first order logic. The refinement relation between two programs could then be proved by proving an implication between the two weakest preconditions. Engeler's atomic statements were simple assignments, Back showed that general specifications expressed using first order logic could form the atomic statements. We further develop this method to deal with recursive procedures (which prove to be very important) and use the technique to develop a catalogue of practical transformation rules and techniques which can be used to derive programs from specifications and to analyse existing programs (including unstructured programs). Our main contributions are in the following areas:

* Theorems on proving the termination of recursive and iterative programs (Chapter one).

* The Recursive Implementation of Specifications (Chapter one), this enables the transformation of general specifications (for example in **Z** notation [Hayes 87] or **VDM** [Jones 86]) into programs.

* A rigorous framework for reasoning about programs with **exit** statements that terminate nested loops from within. (Arsac [Arsac 79] [Arsac 82] discusses some transformations which we are able to generalise and prove using our framework), (See chapters two and three). These are important for recursion removal and restructuring).

* Selective unrolling of loops and entire loop unrolling (Chapter four), these are particularly useful for efficiency improving and general restructuring.

* Selective unfolding of procedures. (Chapter five). Our selective unrolling/unfolding theorems provide the basis for a rigorous treatment of "Action Systems" which are systems of parameterless recursive procedures. These are used as the equivalent of **goto** statements.

* Theorems for recursion removal. A wide range of sources have been culled for potential theorems for recursion removal ([Arsac 82], [Barron 68], [Bauer & Wossner 82], [Bird 77], [Burstall 69], [Cooper 66], [Knuth 74], [Partsch & Pepper 76], [Pepper 79], [Rice 65], [Vuillemin 74]) and we have generalised these to take account of side-effects in functions and expressions and developed proofs within our system. (See chapters six to eight).

* Techniques which use the theorems above for deriving algorithms from specifications and for analysing programs by transforming them into specifications. The main examples here are the Schorr-Waite Graph Marking algorithm, the treatment of backtracking programs, and an example of the transformation of an unstructured program into a specification (chapters eight and nine).

The "power" of the set of transformations developed is illustrated by the fact that over the last two years a wide range of problems have been tackled using these methods, including IBM 360 Assembler programs (see [Ward 88], [Ward 89a], [Ward 89b], [Ward 89c]) without the need for any fundamental extensions to the theory.

PROBLEMS IN SOFTWARE ENGINEERING

Too much software currently being produced is late, over budget, and does not perform as expected; yet software costs are rising all the time. The fact that the software development industry is in a crisis has been recognised since 1969 [Brooks 69]. Since that time a great deal of research has been carried out on the problems of software development with some progress, but only marginal impact on the crisis itself. F.P.Brooks [Brooks 87] claims that he knows of no single development which will solve the software crisis; no single development offers a gain in software productivity comparable to the gains in hardware brought about by the introduction of transistors, integrated circuits or large-scale integration. He gives the following reasons why he believes that such a development is impossible:

Complexity: This is an essential property of all large pieces of software, "essential" in that it cannot be abstracted away from. This leads to several problems:

- Communication difficulty among team members, leading to product flaws, cost overruns and schedule delays.
- It is impossible to enumerate all the states of the system and therefore it is impossible to understand the system completely.
- The system is difficult to extend without invoking unforseen side-effects.
- Unvisualised states can lead to security loopholes.
- It is difficult to get an overview of the system, so maintaining conceptual integrity becomes increasingly difficult.
- It is hard to ensure that all loose ends are accounted for.
- There is a steep learning curve for new personnel.

Conformity: Many systems are constrained by the need to conform to complex human institutions and systems, for example a wages system is greatly complicated by the need to conform to current tax regulations.

Change: Any successful system will be subject to change as it is used:

-by being applied beyond the original domain,
-surviving beyond the normal life of the machine it runs on,
-being ported to other machines and environments.

Invisibility: With complex mechanical or electronic machines or large buildings the designers and constructors have blueprints and floorplans which provide an accurate overview and geometric representation of the structure. For complex software systems there is no such geometric representation. There are several distinct but interacting graphs of links between parts of the system to be considered; including control flow, data flow, dependency, time sequence etc. One way to simplify these, in an attempt to control the complexity, is to cut links until the graphs become hierarchical structures [Parnas 79].

CURRENT APPROACHES

In this section we outline some current approaches to solving the problems outlined above. We will briefly discuss the merits and limitations with each approach. More "radical" approaches such as the application of Artificial Intelligence, or new programming paradigms such as Object Oriented Programming are not covered by the survey.

Program Verification

The "Program Verification" movement started from the observation that most programs are wrong and that therefore a mathematical proof of the correctness of a program (ie of the fact that the program implements its specification) would be desirable [Hoare 69], [Gries 81]. The goal of a program which not only operates correctly but has been proven to do so under all circumstances is indeed attractive, but so far the results have been very disappointing. Only very small programs have been tackled and the labour required in constructing the proof is very great. A more serious objection however is that most, if not all, large programs still contain bugs, ie they are incorrect, and an attempt to prove the correctness of an <u>incorrect</u> program is doomed to failure. One answer that has been proposed to this question is that the construction of the correctness proof and the program should go hand in hand [Gries 81]. However, although such techniques have their applications, for smaller programs in areas where the vital importance of absolute correctness justifies the high cost (eg "safety-critical" systems such as the control program for a nuclear power plant or an aircraft's automatic pilot), there are difficulties with "scaling up" the methods to deal with large and complex programs. These methods do not deal with modifications to the program once constructed, yet the largest proportion of the resources spent on a program over its lifetime is spent on modification and enhancement [Leintz & Swanson 80]. With current program verification techniques the whole program would have to be re-verified after each modification or enhancement.

Higher-Level Languages

The development of high level languages such as Fortran, Algol and Cobol led to an order of magnitude increase in programmer productivity, proponents of 4GLs and other "higher-level languages" suggest that the next generation of languages will lead to similar gains. But the difficult part of programming is the specification, design and testing of the conceptual construct and not the representation of that construct in a given programming language. Current 4GLs are closer to libraries of commonly used routines together with a simple interface for putting these routines together. This can be a highly productive method of program construction, but it does mean that any particular 4GL is limited in the problem areas to which it can be successfully applied. As languages become more complex they give rise to an increasing burden on the programmer to learn all the constructs of the language and avoid undesired interactions between them. This is a problem with Ada, [Hoare 81b] and was already recognised in PL/1 (Dijkstra [Dijkstra 72] states: "Using Pl/1 must be like flying a plane with 7,000 buttons, switches and handles to manipulate in the cockpit.")

Automatic Programming and Program Plans

The various attempts to automate, partially or totally, the programming process have met with some success, but only in limited areas: for example sorting programs, the integration of differential equations and database management systems (such as Focus). These techniques seem to require a simple problem for which there are several known methods of solution and for which explicit rules can be found which select the appropriate solution method. The semi-automatic plan-based systems of [Rich 81] were intended to provide the programmer with a selection of "plans" (basic building blocks of programs: searching, accumulating, hashing etc.) which he could select and combine in order to construct his program. However the number of plans required for even simple programs is estimated at "several thousand" [Waters 85], and the system would find it difficult to cope with all the potential interactions between plans. The biggest problem with such systems however is that they only tackle the problem of translating a detailed design into program code, and this is only a small part of the programmers task. Hence such systems can have limited impact on programmer productivity.

Requirements Refinement and Rapid Prototyping

The hardest part of any large programming project is deciding what to build: errors in the requirements and specification of a program are the most crippling when they occur, and the most expensive to fix. Often the client doesn't know himself what he wants so it is impossible to get the specification right first time. The solution is to lash up a prototype as quickly as possible, which can then be shown to the client to get his reaction. The easier it is to modify and enhance this prototype (ignoring all concerns for efficiency and most concerns for completeness!) the easier it will be to get the specification right later on. Prototypes give a concrete realisation of the conceptual structure of the design, however there can be problems in going from the prototype to the product while preserving the conceptual structure. There is also a temptation to put the prototype into production and skip the final development state. Since the prototype was developed with little regard to efficiency or robustness this can be a false economy.

Transformation of Recursion Equations

In [Burstall & Darlington 75] [Burstall & Darlington 77] programs are written in the form of recursion equations which are transformed into other forms by operations such as definition, folding, unfolding, abstraction, instantiation and redefinition. The aim of this work is to develop programs by writing an inefficient but clear version (in the form of recursion equations) which can be automatically transformed into an efficient iterative program. This work arose from attempts to understand and systematise the mechanical program transformation system described in [Darlington & Burstall 76]. Their transformations are based on the theory of recursion equations, but there is no attempt to find the conditions under which the transformations will work–it is left for the user to avoid, for example, folding a definition to give $\mathbf{P(x)=P(x)}$ as the definition of $\mathbf{P}$.

Strong, [Strong 71] showed how recursion equations can be translated into flowcharts. He proved that in the general case two stacks are needed, one to save the values of parameters and local variables over inner recursive calls of the procedure (the parameter stack) and the other to record the current point in the execution of each invocation of the procedure (the protocol stack). For example if the body of a procedure contains two recursive calls then when an inner recursive call terminates the protocol stack will show whether it was the first or second call in the body.

Griffiths [Griffiths 76], [Griffiths 79], suggests that the problem of proving that a program satisfies its specification can be tackled by starting with the specification and transforming it into a program. Provided that the transformations preserve correctness we can be confident that the final program satisfies the specification. This should also help to provide a general method for developing algorithms. He suggests that we "consider programs much more as manipulable objects which exist in different forms and which do well-defined things". He stresses the importance of the re-usability of results that such an approach allows: different versions of the specification and program can be stored, and a library of general transformations of specifications built up which can be used in different programming languages.

This approach needs to be based on a formal system in which the transformations have been proved and so can be relied on to work–Bauer [Bauer 76], [Bauer 76a] also exposes the need for some sort of unified conceptual basis for programming.

The work of developing a program by successive transformation can be made much easier and less error-prone if an interactive system is provided which can carry out the transformations, perhaps check that they are used in a valid way, and keep a record of the various versions of the program. The Cornell Program Synthesiser of [Teitelbaum & Reps 81] [Barstow et al 84] is an example of an interactive system for program writing and editing which acts directly on the <u>structure</u> of the program by inserting and deleting structures in such a way as to ensure that the edited program is always syntactically correct. Arsac [Arsac 82] describes using a simple automatic system to carry out transformations of a program and store the various versions. His system makes no attempt to check that the correctness conditions of a transformation hold though. Kibler [Kibler et al 77] describes another such system which is based on the work of Loveman [Loveman 77] and Burstall and Darlington [Burstall & Darlington 77]. Loveman [Loveman 77] lists over 40 program transformations which could be applied automatically–these are mainly simple modifications of the program aimed at improving the efficiency.

The Harvard PDS (Program Development System) is a system for creating and applying transformation rules. In [Townley 82] is used to implement an algorithm for reducing a deterministic automaton. Wegbreit [Wegbreit 76] notes that languages such as LISP and APL encourage programmers to write programs in a form which is very inefficient to execute. He uses program analysis to find the critical areas of a program (where efficiency improvements are most valuable) and gives some examples of improving programs in LISP and APL.

Axiomatic Data Types

Henderson & Snowdon [Henderson & Snowdon 72] show that in using the "top-down" method of program development the data structures need to be precisely defined at each level of detail–and this requires a very general and powerful notation in which to express the data structures.

One way of describing a data structure and the operations on it is to give a complete and consistent set of "axioms" which the data structure is required to satisfy. For example the operation of pushing an element onto a stack results in a non-empty stack, and so on. Provided a program using the data structure does not assume anything about it which cannot be derived from these axioms then such a program should work correctly with any implementation of the data structure which satisfies the axioms. This method is used by Hoare [Hoare 72] and Bauer [Bauer 79]. A similar approach is the "Algebraic" method of Guttag and Horning [Guttag 77] [Guttag & Horning 78], these two approaches are compared in [Guttag 79]. Both of these "non-constructive" methods have their problems. Guttag [Guttag 79] says of the problem of finding a suitable set of axioms for a data structure, ". . . it is not always clear how to attack the problem. Nor once one has constructed an axiomatisation, is it always easy to ascertain whether or not the axiomatisation is consistent and sufficiently complete. It is extremely easy to overlook one or more cases."

Majester [Majester 77] points out the difficulties with non-constructive methods, and suggests that a constructive or model-based method in which the data structure is defined in terms of known mathematical objects is better. For example, a stack can be described simply as a finite sequence of elements and the operations **push**, **pop** etc. defined as functions acting on sequences. He gives an example of a data structure which has a simple constructive definition yet which would need several infinite sets of axioms to define algebraically. In order to get a non-constructive definition one usually has to start with a constructive definition anyway in order to find the axioms and relations which apply. Also, the usual way of proving that an algebraic specification is consistent is to provide a constructive definition of a data type which satisfies all the axioms. This would indicate that working directly with a representative model would be more productive. In order for this to work, techniques are needed which can demonstrate that two different models of a data structure are isomorphic; and other techniques which demonstrate that one data structure is a correct implementation of another.

Denotational Semantics

In order to prove that one program is equivalent to another a formal specification of the programming language is required. In this area much progress has been achieved since the pioneering work of McCarthy and Landin. In this early work only a few simple concepts of programming languages could be specified: the methods of denotational and axiomatic semantics have now been applied to produce formal specifications of complete programming languages such as PASCAL [Tennet 77], [Hoare & Wirth 73]. These two methods are valuable to both the language designer and implementor since they provide a thorough description of the language. The designer has the means to formulate his intentions precisely and the implementor is able to show that his implementation meets those requirements.

Despite these advantages, such methods are less useful for programmers because the semantics of statements in real programming languages and hence the semantics of real programs quickly tend to become complicated and unwieldy. This is because the design of every construct in the language has to be as complicated as the most complex one–thus the initial success experienced when specifying simple languages is not carried through when attempts are made to tackle "real" programming languages. Although the specifications are useful to compiler implementors they are unsuitable for reasoning about all but the smallest programs. An example is the introduction of continuations in denotational semantics: they are needed to handle **goto**s, but once introduced they have to be added to the definitions of all other language constructs. A precise specification of a language is still useful however when transcribing a program from that language to a different notation and vice-versa.

Algebraic Transformation Systems

A way to reduce the complexity of semantic specifications is to start with a simple language and add more complex constructs to it by defining them in terms of the simple ones. The new constructs may be regarded as "notational variants" of ones already present even though they may be used to introduce new concepts. (For instance we will introduce an **exit** statement which causes termination of the enclosing loop by defining it in terms **while** and **if** statements). The technique of notational variants has been used in the definition of ALGOL 58, ALGOL 68 and PASCAL.

Such an approach is even more attractive in the context of a system for transforming programs into equivalent forms: it is used for example in the project CIP (Computer-aided, Intuition-guided Programming) in [Bauer 79], [Bauer et al 79]. Similar methods are used in [Broy et al 79] and in [Pepper 79], [Wossner et al 79] and [Bauer & Wossner 82]. A simple "kernel" language is presented from which a sequence of more elaborate languages can be defined by providing the "definitional transformations" which define the new constructs of a higher level language in terms of constructs already present.

All of these workers take an <u>applicative</u> language as their kernel which uses only function calls and the conditional (**<u>if</u>** ... **<u>then</u>**) statement. This language is provided with a set of "axiomatic transformations" consisting of: α-, β-and η-reduction of the Lambda calculus [Church 51], the definition of the **<u>if</u>**-statement, and some error axioms. Two programs are considered "equivalent" if one can be reduced to the other by a sequence of axiomatic transformations. This core language can be extended until it resembles a functional programming language. Imperative constructs (variables, assignment, procedures, **<u>while</u>**-loops etc.) are introduced by defining them in terms of this "applicative core" and giving further axioms which enable the new constructs to be reduced to those already defined. However this approach leads to severe problems, particularly as imperative constructs are added to the system:

(1) It is difficult to show that the transformations specify the semantics in a unique way.

(2) The method can only deal with equivalence of full programs and not arbitrary program parts ([Pepper 79] P.17).

(3) A large number of axioms are needed as the usual imperative constructs are introduced, including "implicit axioms" such as $(\mathbf{S}_1;\mathbf{S}_2);\mathbf{S}_3 \approx \mathbf{S}_1;(\mathbf{S}_2;\mathbf{S}_3)$. The reduction of **<u>goto</u>** statements to parameterless procedures requires "a large number of axioms" and the introduction of functions with side effects would increase the complexity to nightmare proportions!

(4) As more axioms are added, more conditions seem to be needed on each transformation; and these all need to be checked before the transformation can be applied.

(5) It is not possible to represent general specifications within the language and therefore no solution can be offered to the general problem of proving that a given program refines a given specification.

A similar approach is presented in [Hoare et al 87] in which program equivalence is defined by giving a set of algebraic laws. However, more than one hundred laws are required to deal with a very simple programming language, the authors point out the problems with presenting them all as self-evident axioms; the completeness and consistency of the set of axioms needs to be demonstrated within some theoretical framework. As the language is extended to other constructs, still more axioms are required.

The system presented in this thesis avoids all these problems by taking as the starting point a simple <u>imperative</u> language which includes a special kind of assignment statement, the **<u>if</u>** statement, a nondeterministic statement and a simple form of recursive procedure. We are able to extend this core to include loops with multiple **<u>exit</u>**s, **<u>goto</u>**s, procedures, functions and abstract data types by using definitional transformations <u>without</u> obscuring the basic simplicity of the language. We give a simple denotational semantics for the core language which enables us to develop a proof technique to prove the transformations and refinements using weakest preconditions. As the catalogue of proven transformations is built up these can be used as lemmas in the proofs of more complex transformations.

Algebraic Data Types

The "algebraic" specification of abstract data types leads to similar problems as the algebraic specification of program equivalence. In [Bauer 79], [Bauer & Wossner 82] data types are introduced by giving their "characteristic predicate" which lists all the properties of functions which act on that type. This requires a great deal of effort in proving that the characteristic predicate is complete and consistent since the predicates can get very long (the definition of a "flexible array" for example takes 24 lines). Our approach to data types is "constructive" rather than axiomatic: data types are defined in terms of well-known mathematical objects rather than by listing a set of axioms which they are required to satisfy. Since our system is based on a variant of mathematical logic we find that we can incorporate any results about these objects into our system.

Top-Down Design

Strict top-down design, or stepwise refinement, is an attempt at mastering the complexity of large programs by constructing them according to a rigid hierarchical structure. The top level structure of the program is written first, with gaps to be filled in later. Each gap is treated in turn, its structure is elaborated and its function re-expressed in terms of operations with simpler functionality. The idea is that only a small part of the program is worked on at each stage, there are only a few details to keep track of so it is easy to ensure that each structure is correct. If each component is elaborated correctly at each stage then (in theory) the whole program will be correct by construction. This method has been used successfully with some small examples [Wirth 71], [Wirth 73]; however, some severe problems emerge as larger programs are attempted:

(1) It is not always obvious what is a "simpler" operation: for example a program to find the largest element of an array could be expressed as: "sort the array"; "pick the first element". Experienced programmers usually have a rough idea of the relative complexity involved in implementing a given operation, however with a large program on which many people are working there is always the danger that a simple operation will be implemented in terms of a complex one.

(2) The correctness of the final program depends on the implementation of each operation precisely matching its use, if (as tends to be the case) the high level operations are expressed in informal English there is a danger that the user and the implementor will place different interpretations on the words used. This problem can appear even with small programs developed with careful attention to detail. For example in [Henderson & Snowdon 72] this led to the production

(and publication) of an incorrect program. This problem is a major source of errors in large programs.

(3) It is not always easy at each stage to discover what the "right" structure to use is. If a mistake is made then the whole development has to be scrapped and repeated from that point on–and choosing the wrong structure is highly likely in the early stages of development. A simple example is the development of a program to print the first 1000 primes, there are several candidates for the top-level structure:

```
n:=1; printed:=0;
while printed<1000 do
   n:=n+1;
   if "n is prime" then "print n"; printed:=printed+1 fi od.
```

OR:

```
n:=1;
while n⩽1000 do
   "calculate nth prime number and store in prime"
      "print prime"
      n:=n+1 od.
```

OR:

```
n:=1; printed:=1; prime:=2;
  "print 2";
while n<1000 do
   "calculate the smallest prime larger than prime, store in prime";
      "print prime";
      n:=n+1 od.
```

Each of these is a possible candidate: until they are developed further it is not easy to see which is the best candidate. Note that these all print the primes as they are calculated. Other possible first refinements might calculate all the primes and store them in a table and then print them all.

Structured Programming

The controversy over "structured programming" was sparked off by [Dijkstra 68] which suggested that the indisciplined use of **goto** statements was a harmful programming practice as it led to programs which were very hard to understand and modify. This has led to a great deal of discussion over what constitutes "good practice" in programming and what program constructs may be considered "harmful". In particular there is the question of whether program constructs which allow a loop to terminate from the middle are good or bad [Knuth 74]. Calliss [Calliss 88] discusses some of the problems with the automatic translation of programs to remove the constructs considered harmful.

Peterson et al [Peterson et al 73] shows that any program using **goto** statements can be translated into one which uses only statements of the form **exit(n)** (which will terminate the **n** enclosing loops). Such statements:

(1) Cannot create a loop since they can only transfer control forwards. Hence all loops are clearly indicated by **do** ... **od** pairs.

(2) Cannot be used to branch into a control structure since they always branch out of a structure.

Buhr [Buhr 85] presents a case for teaching loops with multiple exits to beginning programmers because most programmers talk about, document and think about the situations that cause a loop to stop; not the situations that cause a loop to continue, so it is more natural to program with loops that continue automatically until an **exit** statement is executed. Buhr suggests that **exit** should not be treated as a separate statement since this leads to the possibility of having statements in a program which can never be executed, for example: **exit; x:=1**–the assignment to **x** is never executed. However, this is possible anyway–for instance with: **if 1=0 then x:=1 fi**, and it turns out that the separate **exit** statement is very useful for program transformation purposes and is much easier to work with.

Covington [Covington 84] also notes the advantages of the **exit** statement for teaching programming. He finds that one disadvantage with the usual **while** and **repeat** statements is that beginners tend to think that they involve continuous testing of the condition. The empirical work of Soloway et al [Solway et al 83] suggests that people will write a correct program more often if they use a language which is appropriate to their cognitive strategy–allowing **exit**s from the middle of a loop improves the probability of a correct program being written.

Whitelaw [Whitelaw 85] notes that in cases where there are several invariants to be maintained in a loop it makes sense to test each one in turn and leave the loop as soon as one fails. He gives the following example of a loop which sums the elements of array **a[1..n]** of positive integers where if there are fewer than **n** elements to be summed the set ends with a zero and the loop is also required to avoid numerical overflow.

Compare the two versions:

```
sum:=0; i:=1; sum:=0; i:=1;
do m:=a[i]; m:=a[i];
  if m=0 then exit fi; cont:=true;
  if maxint−m<sum then exit fi; while m≠0 ∧ cont do
  sum:=sum+m; if maxint−m⩾sum
  if i=n then exit fi; then sum:=sum+m;
  i:=i+1 od. if i<n
                                          then i:=i+1;
                                                 m:=a[i]
                                          else cont:=false fi
                                      else cont:=false fi od.
```

Note in the second version:
(a) The deeply-nested **if** statements.
(b) The use of a Boolean variable **cont** to control the execution.
(c) The need for two copies of one statement (**m:=a[i]**).
(d) The complexity of the invariant which would be needed to prove this version correct.

All these are features of programs which rely entirely on **if** and **while** constructs. Both Williams and Amit [Williams 84] [Amit 84] note that the "invented" Boolean variables needed to map a problem onto the available control structures are as confusing as the **goto** statements they are trying to avoid.

Other looping constructs have been suggested, [den Hartog 83], [Newman 83], [Bochmann 73] and [Yuan 84] none of these suggestions however has the elegance and simplicity of the **exit** statement.

SUMMARY

Many of the methods and techniques described above have been hailed as the solution to the software crisis; however in practice only small gains in productivity have been achieved. Brooks [Brooks 87] suggests that a combination of different methods will be needed to produce the order of magnitude increase in productivity which he believes is required. However, few of these methods pay any attention to the problems of maintaining and enhancing the developed software and none address the problems of maintaining existing systems, many of which have been developed using traditional (non-formal) methods and heavily modified over the years. Frequently there is little or no accurate documentation available. Program verification techniques are rarely of any help in locating bugs in an existing program; it can be difficult to enhance programs developed by top-down design without either destroying the structure (by traditional maintenance methods) or repeating much of the work that went into the original design. Automated programming methods such as the Programmers Apprentice [Waters 85] also offer little assistance with program maintenance.

The result is that even if the promised large improvements in development speed by the use of new methods do eventually appear they will have little impact on total software costs since any gain from increased development will be swallowed up by the increased maintenance costs. In fact if the resulting programs are harder to maintain there will be a net loss in cost effectiveness if the new methods are used.

OUR APPROACH

To make it possible to prove that a program implements its specification, we include general specifications (expressed using mathematical logic) as primitive statements in the kernel programming language. This means that the specification of a program will itself be a program, and therefore the proof that the program implements its specification is a proof of program refinement. This makes it possible to develop an algorithm by transformation of the specification. We will give some examples of the rigorous development of algorithms by transformation of their specifications (see Chapter 9).

The program transformations we develop can also be used to develop systems by a form of rapid prototyping in which a specification is transformed into an executable prototype which can be easily modified to meet the users changing requirements. This prototype can then be developed into a production implementation using transformations. This avoids changing the "conceptual structure" of the prototype, since the use of proven transformations will preserve the external effects of the prototype in the production version.

The methods developed in this thesis can be used to develop interactive program analysers which can be used to reveal the structure of a program and extract its specification from the code. This is a sort of inverse operation to stepwise refinement and is hence given the name "inverse engineering". An example of inverse engineering applied to a published program can be found in Chapter 9. With suitable extensions for different programming languages the transformations can assist in porting a program from one language to another or one operating system to another. The author is currently involved in a project which aims to build a prototype tool to assist in extracting the specification (expressed in **Z** notation) of a large program written in IBM Assembler.

Often programmers are faced with the problem of having to modify a program to meet a different specification to the one they were originally presented with. The only option presented by some "structured programming" methods (eg strict stepwise refinement) is to throw the old program away and start from scratch with the new specification, hoping that one will be able to re-use much of one's previous work! The approach being suggested here is to reverse the refinements used to generate the program (or in the case of an existing program to find such inverse refinements) to create an abstraction of the program at a sufficiently high level such that the proposed modification is trivial. The modified abstract program can then be refined back into a concrete program, making use of previous results where applicable.

This suggests that programming is a process of transformation starting from the problem description. Such a process would need to be carried out at several different levels of abstraction, from recursive procedures and recursive data structures down to global variables, assignments and branches. The transformations should ideally be carried out within a single language which incorporates all the levels–a Wide Spectrum Language such as that described in [Baur 79]. This is not a particular notation but rather a general language for expressing algorithms–an "algorithmic language" analogous to mathematical language [Bauer & Wossner 82].

Programming often means to meliorate the efficiency of algorithms and sometimes this means going right back to the problem description at the abstract level and doing mathematics.

Ideally this should be possible without leaving the language. This implies that mathematical specifications should form part of the language. Having the whole spectrum of operations, data structures and programming constructs contained within a single language which also includes all possible specifications, allows us to develop different parts of a program at different rates; different parts of the same program can be expressed at different levels of abstraction. The more time-critical parts of a program can be refined down to a lower level of abstraction (which should allow the maximum efficiency to be obtained) while the less critical parts can be left at a higher level of abstraction. This is useful because in most large systems there are a few "hot spots" (small areas of code which take up the lion's share of the processor's time) and programming effort may be concentrated on these. [Knuth 72] recommends this approach.

Criteria for Success

For our approach to be successful it must be capable of dealing with the following problems:

(1) General specifications in any "sufficiently precise" notation. For sufficiently precise we will mean anything which can be expressed in terms of mathematical logic with suitable notation. This allows a wide range of forms of specification, for example **Z** specifications [Hayes 87] and **VDM** [Jones 86] both use the language of mathematical logic and set theory (in different notations) to define specifications.

(2) Nondeterministic programs–since we do not want to have to specify everything about the program we are working with (certainly not in the first versions) we need some way of specifying that some executions will not necessarily result in a particular outcome but one of an allowed range of outcomes. For example a sorting program may not specify the effect on items with identical keys, these items may be placed in any order. The implementor can then use this latitude to provide a more efficient implementation which still satisfies the specification.

(3) Reasoning methods which are extensive enough to incorporate results from other approaches: we don't want to restrict programmers to just one way of reasoning about programs. For some programs a direct verification of the final program using pre-and post-conditions is easy, for others a "proof by construction" which starts with a specification and derives the program by successive (proven) transformations, is a more appropriate technique.

(4) Applicable to real programs–not just those in a "toy" programming language with few constructs. This is achieved by the (programming) language independence and extendibility of the notation via "definitional transformations".

(5) Scalable to large programs–although space precludes a full demonstration of this.

(6) Automatable where possible–this relates to (5), tedious transformations must be capable of being automated with confidence that the semantics of the program will not be affected.

We will show how these criteria lead naturally to the use of weakest preconditions expressed as formulae of infinitary first order logic.

In the next section, we discuss weakest preconditions in more detail. We describe the kernel language we will be using and define weakest preconditions of the kernel language structures in terms of infinitary first order logic.

Weakest Preconditions and the Kernel Language

We will be making use of the concept of a **weakest precondition** which was introduced by Dijkstra in [Dijkstra 76]. The weakest precondition for a given program **S** and condition on the final state **R,** written **WP(S,R)** is defined as the weakest condition on the initial state such that starting the program in a state satisfying that condition results in the program terminating in a final state satisfying the given postcondition. For example, the statement **x:=5** will terminate in a state satisfying **x>y** iff it is started in a state satisfying **5>y**, hence: **WP(x:=5, x>y) = 5>y**
An example using the **if** statement is:

$$\begin{aligned}&\mathbf{WP}(\underline{\mathbf{if}}\ \mathbf{x{=}1}\ \underline{\mathbf{then}}\ \mathbf{x{:=}5}\ \underline{\mathbf{else}}\ \mathbf{x{:=}6}\ \underline{\mathbf{fi}},\ \mathbf{x>y})\\ &\quad = ((\mathbf{x{=}1}) \Rightarrow \mathbf{WP}(\mathbf{x{:=}5},\ \mathbf{x>y})) \wedge ((\mathbf{x{\neq}1}) \Rightarrow \mathbf{WP}(\mathbf{x{:=}6},\ \mathbf{x>y}))\\ &\quad = ((\mathbf{x{=}1}) \Rightarrow \mathbf{5>y}) \wedge ((\mathbf{x{\neq}1}) \Rightarrow \mathbf{6>y})\end{aligned}$$

Weakest preconditions are useful because we often express the specification of a program in terms of a condition which the program is required to "make true" by assigning values to some of the variables. For example the specification of an exponentiation program might be: "Assign a value to variable **z** such that $\mathbf{z=x}^y$." A division program might have the specification: "Assign values to **q** and **r** such that $(\mathbf{a=q.b+r} \wedge \mathbf{0 \leqslant r<b})$".

When implementing specifications as executable programs we will often need to assign values to "temporary" variables which are not mentioned in the specification so their final values do not matter. To express this we need a notation for adding and removing variables from the set of active variables (called the "state space"). Both of these concepts are combined in the notion of an "Atomic Description" which was introduced by Back in [Back 80]. If **x** and **y** are lists of variables and **Q** is any formula then the Atomic description:

$$\mathbf{x/y.Q}$$

adds the variables in **x** to the state space (if they are not already there), assigns values to these variables such that **Q** is true and then removes the variables in **y** from the state space. If there is no assignment to variables in **x** which will make **Q** true then the statement **x/y.Q** is considered not to have terminated.

The weakest precondition of the atomic description $\mathbf{x/y.Q}$ for the condition $\mathbf{R}$ on the final state is therefore:

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

Here the first part $\exists\mathbf{x.Q}$ is required because the atomic description will not terminate if there is no assignment of values which satisfies $\mathbf{Q}$ (so the $\mathbf{WP}$ must be false in this case) and the second part is required so that the $\mathbf{WP}$ is only true when any assignment of values to $\mathbf{x}$ which satisfies $\mathbf{Q}$ will also satisfy $\mathbf{R}$.

It turns out that this statement will be the <u>only</u> primitive statement we will need to express any program or specification–for example a statement such as $\mathbf{x{:=}x{+}y}$ can be expressed in terms of atomic descriptions as:

$$\langle\mathbf{z}\rangle/\langle\rangle.\mathbf{(z{=}x{+}y)};\ \langle\mathbf{x}\rangle/\langle\mathbf{z}\rangle.\mathbf{(x{=}z)}$$

where $\mathbf{z}$ is a temporary variable and $\langle\mathbf{x}\rangle$ is the list containing the single variable $\mathbf{x}$.

Dijkstra found it impossible to express the weakest precondition of a general loop as a single formula since a loop may terminate immediately and satisfy the post-condition, or it may terminate after one iteration and satisfy the post-condition, or ... and so on. This problem was solved by Back in [Back 80] by expressing weakest preconditions as formulae of <u>infinitary</u> logic. This is an extension of ordinary first order logic which allows infinitely long formulae. The weakest precondition of a loop which aborts automatically if it attempts to carry out more than $\mathbf{n}$ iterations (where $\mathbf{n}$ is a given integer) <u>can</u> be expressed as a finite combination of the weakest precondition of the body. Such a loop is called the "**n**th truncation" of the general loop (The general loop can execute arbitrarily many iterations before terminating). We construct the weakest precondition of a loop by ORing together the weakest preconditions of <u>all</u> the **n**th truncations of the general loop. This is because if the general loop terminates then one of the **n**th truncations must terminate and if the general loop does not terminate then neither will any truncation of it. We extend this idea to define the **n**th truncation of a recursive procedure. The "Fixed Point Theorem" of Denotational Semantics [Stoy 77] provides the link between the semantics of each truncation and the semantics of the general loop.

Thus, infinitely long formula are needed to express the weakest preconditions of loops and recursive procedures. They are also used to prove that certain assignments may choose to assign from only a <u>finite</u> set of values, and to give a particularly clean and simple way of expressing the concept of a recursively defined data type within our programming language. These infinite formulae consist of a (possibly infinite) sequence of formulae ANDed or ORed together (for example $\mathbf{(x{=}1)}\ \vee\ \mathbf{(x{=}2)}\ \vee\ \mathbf{(x{=}3)\ ...}$), where each component is either finite or another such sequence. The formulae are built up from a <u>finite</u> number of applications of the rules–this means that constructions such as $\mathbf{(Q_1 \wedge (Q_2 \vee (Q_3 \wedge (Q_4 \vee ...))))}$ are excluded. This allows us to use induction on the structure of formula in

our proofs. Note that $\mathbf{Q_1 \wedge Q_2 \wedge ...}$ is true iff $\mathbf{Q}_n$ is true for all $\mathbf{n}$ and is false otherwise, however $\mathbf{(true \wedge (false \vee (true \wedge (false \vee ...))))}$ cannot be evaluated as **true** or **false** since it is not a well-formed formula.

Our kernel language is constructed from the atomic description as the only primitive statement, a set of <u>statement</u> <u>variables</u> (which represent the recursive calls of recursive statements) and the following compound statements:

If $\mathbf{S}_1$, $\mathbf{S}_2$ are statements, $\mathbf{B}$ a formula of infinitary logic and $\mathbf{X}$ a statement variable then the following are statements:

(i) $\mathbf{S}_1$; $\mathbf{S}_2$ **Sequential Composition**; first $\mathbf{S}_1$ is executed and then $\mathbf{S}_2$.
(ii) **<u>oneof</u>** $\mathbf{S}_1 \square \kappa\mathbf{S}_2$ **<u>foeno</u>** **Nondeterministic Choice;** one of the statements $\mathbf{S}_1$ or $\mathbf{S}_2$ is chosen for execution, the choice is completely nondeterministic.
(iii) **<u>if</u>** $\mathbf{B}$ **<u>then</u>** $\mathbf{S}_1$ **<u>else</u>** $\kappa\mathbf{S}_2$ **<u>fi</u>** **Conditional Choice;** if the condition $\mathbf{B}$ is true then $\mathbf{S}_1$ is executed, otherwise $\mathbf{S}_2$ is executed.
(iv) **<u>proc</u>** $\mathbf{X} \equiv \mathbf{S}_1$. **Recursive procedure;** within the body $\mathbf{S}_1$, $\mathbf{X}$ represents a recursive call to the procedure.

We are able to avoid the problems which arise in attempting to find a complete and sufficient set of axiomatic transformations by defining the "meaning" or interpretation of a program to be a particular kind of function (called a "state transformation") which maps an initial state to a set of possible final states. The set of final states may include the special "state" $\perp$ (pronounced "bottom") which represents nontermination or error. A state is defined to be a collection of variables with values assigned (so one talks about the value of variable $\mathbf{x}$ in state $\mathbf{s}'$ which is a possible final state for state transformation $\mathbf{f}$ applied to the initial state $\mathbf{s}$). Programs are regarded as equivalent if they have the same meaning function. This interpretation is based on the one used in [Back 80] which has been extended to allow the inclusion of recursive procedures in programs. See [Ward 89b] for a detailed description of our extensions to Back's work.

allowed final states for
given initial state

defined set

initial state
space

final state space

The value of any state transformation applied to $\perp$ is defined as $\{\perp\}$ (ie the set with $\perp$ as its only element). This ensures that if one component of a sequence of statements does not terminate then the whole sequence will not terminate (and similarly if an error condition arises then the whole program returns an error).

We define the formula $\mathbf{WP(S,R)}$ for a given statement $\mathbf{S}$ and formula $\mathbf{R}$ as follows:
(i) $\mathbf{WP(x/y.Q,R)} = \exists \mathbf{x.Q} \ \wedge \forall \mathbf{x.}(\mathbf{Q} \Rightarrow \mathbf{R})$
(ii) $\mathbf{WP(S_1;S_2,R)} = \mathbf{WP(S_1,WP(S_2,R))}$
(iii) $\mathbf{WP(\underline{oneof}\ S_1 \ \Box \ S_2\underline{foeno}\ ,R)} = \mathbf{WP(S_1,R)} \ \wedge \ \mathbf{WP(S_2,R)}$
(iv) $\mathbf{WP(\underline{if}\ B\ \underline{then}\ S_1\ \underline{else}\ S_2\underline{fi}\ ,R)} = (\mathbf{B} \Rightarrow \mathbf{WP(S_1,R)}) \ \wedge \ (\neg \mathbf{B} \Rightarrow \mathbf{WP(S_2,R)})$
(v) $\mathbf{WP(\underline{proc}\ X \equiv S_1.\ ,R)} = \bigvee_{n<\omega} \mathbf{WP(\underline{proc}\ X \equiv S_1.}^{n}\ \mathbf{,R)}$
(vi) $\mathbf{WP(X,R)} = \mathbf{X}$
where in (i) we use the convention that $\exists \mathbf{x.Q}$ and $\forall \mathbf{x.}(\mathbf{Q} \Rightarrow \mathbf{R})$ mean $\mathbf{Q}$ and $(\mathbf{Q} \Rightarrow \mathbf{R})$ respectively when $\mathbf{x} = \langle\rangle$ (the empty sequence of variables). Note that the atomic description may not be allowed to assign values to <u>any</u> variables.

In (v) the $\mathbf{WP}$ for the recursive statement is formed by ORing together the $\mathbf{WP}$'s of the "finite truncations" of the recursion defined thus:

$\mathbf{\underline{proc}\ X \equiv S_1.}^{0} = \mathbf{abort}$

where $\mathbf{abort} = \langle\rangle/\langle\rangle\mathbf{.false}$ is a statement that can never terminate.

$\mathbf{\underline{proc}\ X \equiv S_1.}^{n+1} = \mathbf{S'[X/\underline{proc}\ X \equiv S'.}^{n}\mathbf{]}$

which is $\mathbf{S}_1$ with all occurrences of $\mathbf{X}$ replaced by $\mathbf{\underline{proc}\ X \equiv S_1.}^{n}$.

Note that: $\mathbf{WP(abort,R)} = \mathbf{false}$ for any $\mathbf{R}$

Outline of the Thesis

In Chapter one we present the "Fundamental Theorem of the Weakest Precondition" (an extension of the theorem in [Back 80] to include recursion). This exhibits the link between the formula **WP(S,R)** and the weakest precondition of the state transformation corresponding to **S**. This theorem provides the basis of our proof rule for refinement: if we can prove that the weakest precondition of one program implies the weakest precondition of a second program (as formulae of infinitary first order logic) then the second program is a refinement of the first.

We will show that this rule is complete in the sense that for every such refinement there exists a proof, and sufficient in that if you discover a proof then the corresponding refinement is valid. This means that we are able to prove all the results in succeeding chapters using this rule without having to worry about state transformations any more. We go on prove some fundamental theorems for proving the termination of iterative and recursive programs and for the implementation of specifications as recursive procedures.

Chapter two introduces our first set of notational variants, or extensions to the kernel language. These add to the expressive power of the language without losing the previous results (contrast this with the denotational semantics approach [Stoy 77] where each language extension means one has to prove all the equivalences again from scratch). The extensions include Dijkstra's Guarded Commands [Dijkstra 76] and statements which introduce new local variables and allow changes of data representation. Changing the data representation is a very important type of program transformation since the development of several important algorithms hinges on the choice of data representation; and this choice may be changed several times as the algorithm is developed. One often needs to change the representation from one in which the algorithm is easy to specify to one in which it is efficient to implement.

Chapter three proves some basic transformations, including transformations for introducing assertions, simple manipulations and simplifications. These basic transformations are used extensively in subsequent chapters as more complex transformations are developed. One disadvantage of this method is that it takes a lot of work to "get off the ground" as every transformation needs to be proved from the weakest preconditions. However this effort only has to be expended once and is quickly repaid as a number of powerful transformations are developed. As more transformations are developed the proofs become easier and easier, even though the transformations are no longer intuitively obvious, and this is a strong indication of the power of the methods.

In Chapter four we introduce **exit** statements which cause termination of a loop from within the body of the loop. We replace such a statement by a statement which changes the value of a new integer variable **depth**. The value of **depth** represents the depth of the loop currently being executed, thus if **depth=4** then **exit(3)** will set **depth** to **1**. Each individual statement is preceded by a test of this variable which ensures that once **depth** is changed no further statements are executed at that level and the appropriate loop is terminated.

This provides a rigorous foundation for reasoning about programs which include **exit** statements. Hitherto the only formal system which encompasses such statements is the technique of "continuations" in denotational semantics. Here in order to add the new statement one has to change the semantics of all the statements defined so far to include the continuation. Hence all the results proved so far have to be re-proved in the new extended system. With this method the kernel language and its semantics stay the same so all previous results carry over to the new system and can be used in the proofs of further transformations.

Chapter five defines another notational variant, which also introduces a new concept, the "action system", which is a collection of (mutually) recursive parameterless procedures. Our notation is developed from that in [Arsac 79] [Arsac 82] where some simple transformations of deterministic actions are discussed. We provide a rigorous system for reasoning about actions which can involve nondeterministic statements and general specifications, this will enable us to prove that a given action system implements a given specification. Note that the simple recursive statement in the kernel language will not allow mutually recursive procedures since all the calls of a recursive procedure must be within its body. We can get round this by putting a collection of procedures together and using the new variable **action** to distinguish which is being called.

In the process of reducing a recursive set of actions to an iterative statement we will need to add a "terminating" action **Z** which causes immediate termination of the whole action system (even if there are recursive calls waiting to be finished). We do this by "guarding" each statement with a test of the variable **action** so that the statements are only executed when **action**$\neq$**"Z"**.

Chapter six proves some basic transformations of action systems, including general recursion removal theorems. Action systems are a useful intermediate step in transforming recursive procedures to iterative forms which, as we will demonstrate, is not only useful for improving the efficiency but also for devising new algorithms. Action systems are also used in "transcribing" programs in other languages (including those with labels and **goto** statements) into our notation, whence they can be analysed to discover their structure and carry out program modification and enhancement. We will give several examples of such program analysis.

Procedures with parameters and local variables are added in Chapter seven by replacing each parameter or local variable by a stack which records the values of the parameter or variable in each "incarnation" of a recursive procedure.

Chapter eight introduces functions and generalised expressions. Functions are introduced by defining them in terms of a "procedural equivalent" which is a procedure which sets a variable to the result returned by a function call. One function is a refinement of another if its procedural equivalent is a refinement of the other function's procedural equivalent. This method is developed as we provide techniques for transforming recursive procedures and functions into efficient iterative algorithms.

Chapter nine gives some examples of simple program developments using our methods, and some applications of the methods to software maintenance. The reader who is not familiar with this method of program development will probably find it useful to skim through this chapter first to gain a feel for the kinds of results the earlier chapters are aiming to prove. This chapter summarises the theory which has been developed and describes potential applications of the results, including the current work in Durham which uses this theory as the foundation for the development of a tool to assist software maintainers.

Our method of program construction turns out to be very similar to the process of deriving theorems in mathematics–in fact proving a refinement amounts to proving a theorem of (infinitary) first order logic. The premise is the initial specification, previous results can be used as lemmas to justify a deduction step, and the final program is the result of the theorem. The final program is guaranteed to be correct if all the transformations used have been proved and have been applied correctly. This can be achieved with the aid of an interactive system which will check the conditions for correct application of each transformation and then perform the transformation and record the results. This process is quite different from program verification where one starts with a program and tries to prove that it is correct: you need a program before you can begin and if (as is the case with most commercial programs) the program isn't correct then the whole exercise is fruitless.

The intermediate steps between the specification and program form an important part of the documentation of the program. Firstly many of the results generated are likely to be useful in other programs (this implies that specifications can be re-used as well as code) and secondly because they facilitate later modification of the program–simply "back up" the levels of refinement until you reach a stage where the modification is trivial and then refine the new version down to the bottom level again; making use of previous results where appropriate.

The transformations can also be used in the reverse direction; in this case one starts with a program for which there may be little reliable documentation and which may have a tangled structure (as a result of a history of modifications and patches) and applies transformations to reveal the structure and derive the specification of those parts of the program of interest. For a large program this leads to an incremental improvement and redocumentation of the program, starting with those areas where modification is required. The Centre for Software Maintenance is currently developing a system to aid a maintenance programmer in understanding and modifying an initially unfamiliar system given only the source code. The system will use knowledge based heuristics to analyse large programs and suggest suitable transformations as well as carrying out the transformations and checking the applicability conditions. Presenting the programmer with a variety of different but equivalent representations of the program can greatly aid the comprehension process, making best use of human problem solving abilities (visualisation, logical inference, kinetic reasoning etc).