Using Formal Transformations to Construct a Component Repository

Dr. Martin Ward
Computer Science Dept
Science Labs
South Rd
Durham DH1 3LE

July 16, 1993

Abstract

This paper discusses how theoretical results from the field of program transformations can be applied to develop a new approach to software reuse. We describe a model for the semantics of nondeterministic programs and specifications and use this model to show how refinements and transformations of programs and specifications can be proved correct by reference to their corresponding *Weakest Preconditions* expressed as formulae in infinitary first order logic. We then show how this theory of program refinements and transformations (which is further developed in [7]) can be applied to the construction of a repository of reusable components consisting of code, specifications, documentation and development methods. These components are linked together in such a way that specifications and their implementations can be extracted easily.

1 Introduction

Production of software is costly and error prone and the most important means of production (good programmers) are scarce. Therefore there exists a need to circumvent this costly manual production process. Analogies from classical engineering suggest that by building up a catalogue of standard components and construction techniques whose characteristics are well documented can greatly reduce the cost of new construction projects. The bridge builder knows under what conditions a "suspension bridge" is the right approach and has a collection of standard girders, cables, nuts, bolts etc. which she can use in the design. This has lead to the notion of a component repository which will reduce the effort involved in constructing new software.

1.1 Current Reuse Technology

The desire to avoid writing the same section of code more than once led to the invention of macros and subroutines. These allow the reuse of common code sequences, but the the reuse is confined to a single author, or at most a single project. This is too restricted to bring relief to the industry.

Standard subroutine libraries have proved a more powerful technique. Packages like SSP and SPSS have had high success because they not only relieve the programmer from the drudgery of coding but also (in their limited domain of application) relieve him from the need to develop an algorithm, or to understand in detail the theory behind it. Unfortunately, only limited progress has been made in this area since the early days.

We could conceive of a high-level language as an attempt at reusability: canonical structures which frequently occur in program, such as looping constructs and methods of procedure call, have been encapsulated into a single command. In addition, common programming *techniques* such as register allocation, loop strength reduction and other optimisations are carried out automatically by the compiling system. In the case of the GNU C compiler [5] "function inlining" can be carried out automatically: ie the distinction between macros and procedures has (for almost all practical cases) been removed. The further extension of this idea is restricted by the perceived need for a compilation to be a totally automatic process.

The module or package concept in languages such as Modula-2 or ADA appears to provide even greater support for reuse. The programmer can define data or procedural abstractions and link code of a reasonably general nature into the code she is writing. However, the module implementations at hand are ofter incompatible with each other since they have been developed independently. They will also be incompatible with the new product under construction. The difficulties involved in re-writing and patching existing code, without introducing bugs are often greater then the cost of starting from scratch.

Certain operating system features, such as pipes, have been considered as a means of supporting reuse [3]. A new system is built by combining existing programs using these features.

A problem with having a large library of modules or components is that for it to be reused effectively the programmer has to know what is available and what each piece of code does, and must be able to combine them on the source code level without any further support from the system. This becomes extremely difficult as the component library gets larger: but that is just when it is becoming useful. As a result these methods have found their greatest success when they are limited to a narrow domain of programs. A 4GL[1] can be seen as an example of a collection of reusable modules for a narrow programming domain, together with the means to compose them into new programs. A further problem is that only a minor part of the program development effort is spent on coding: therefore we want to re-use more than just the code. Development methods, designs and documentation should all be reusable.

1.2 Traditional Development Methods

The traditional development methods can be grouped into four main types:

- The traditional "waterfall" lifecycle which starts with a fixed specification and develops it into a finished product through a number of stages.
- Incremental development: in which a small part of the product is implemented and then enhanced as the specification is developed.
- Rapid prototyping: in which a prototype of the main part of the system is developed and analysed and used as the bases for the next in a series of prototypes. The traditional waterfall method has been described as "slow prototyping".
- A combination of the above.

All of these methods can be seen as applications of reuse: the initial work (a specification, a partial implementation or a prototype) is reused in the later stages. Software maintenance can be seen as the development of enhanced products involving a high degree of reuse of the existing product. However the reuse is almost invariably restricted to a single project, and is often restricted to code reuse.

We want to extend reuse to cross project boundaries and to extend the base of components which can be reused to include all the products of development work

[1] 4^{th} Generation Language

2 The Software Repository

The idea of constructing new software by composition from a collection of reusable components is not new and clearly has many attractions. However it has yet to receive widespread implementation. There appear to be several technical reasons for this (in addition to the managerial issues such as the "Not Invented Here" syndrome):

- The repository must be large enough to contain a useful collection of components, yet each component must be readily accessable.
- The components must be highly reliable since they will (hopefully) be re-used in many applications.
- There must be some means for extracting components from existing code for addition to the archive: writing a complete library of components from scratch would involve a great deal of investment of effort before any return on the investment would be perceived.
- In order to be widely useful the components should be written to handle the most general cases, this means that programs constructed from components can be much less efficient than programs written from scratch which can exploit regularities in the data.

This paper describes how the theory of program refinements and transformations developed in [7,9] can be applied to the construction of a repository of usable components from which new software can be constructed. The repository contains code, specifications and techniques as the components, connected by formal and informal links. The formal links record proven knowledge about the components, for example an abstract specification will be connected via a *refinement link* to its implementation, two algorithms for solving the same problem will be connected via a *transformation link* and an implementation of an abstract data type in terms of concrete data types will be recorded as a *reification link*. Informal links will enable keyword searches and will connect informal text descriptions of components to other components.

3 Theoretical Foundation

In [6,7,9] a formal theory is developed in which it is possible to prove that one program or specification is a refinement or transformation of another (we define a transformation to be a refinement which works in both directions). The language which is developed along with the theory includes both general specifications (expressed in terms of set theory and first order logic) as well as standard programming constructs, hence in the following a "program" can be either a program, or a specification, or a hybrid mixture of program and specification (such as a partially-implemented program). A refinment of a program is another program which is more defined (ie is defined on a larger initial set of states) and more deterministic (ie for each initial state it has a smaller set of potential final states).

The *semantics* of a program is a mathematical object which captures the external behaviour of the program while ignoring its internal details. In [9] we define the semantics of a program to be a pair $\langle d, r \rangle$ where d is the set of initial states for which the program is *defined* and r is a relation which maps defined initial states to potential final states (we define a *state* to be a finite non-empty collection of variables with values assigned to them). If s and t are states such that $\langle s, t \rangle \in r$, ie s and t are related under r, then t is a possible final state for the initial state s. In other words, if we start the program in a state in $s \in d$ then it is guaranteed to terminate and the set of possible final states is the set of all states related to s by r. We write $r(s)$ for this set of states, ie:

$$r(s) = \{ \, t \mid \langle s, t \rangle \in r \, \} \qquad (1)$$

If $\langle d_1, r_1 \rangle$ and $\langle d_2, r_2 \rangle$ define the semantics of two programs $\mathbf{S}_1$ and $\mathbf{S}_2$ then we say $\mathbf{S}_2$ refines

$\mathbf{S}_1$ iff

$$(d_1 \subseteq d_2) \wedge \forall s \in d_1. (r_2(s) \subseteq r_1(s)) \tag{2}$$

If $\mathbf{S}_1$ refines $\mathbf{S}_2$ and $\mathbf{S}_2$ refines $\mathbf{S}_1$ then we say $\mathbf{S}_1$ and $\mathbf{S}_2$ are equivalent. See [9] for the details.

3.1 Weakest Preconditions

We use first order logic to express conditions on states, for example the formula $x \geq y$ expresses the condition that the value of the variable x in the state is greater than or equal to that of y. So a formula is either "true" or "false" for a given state: ie each formula defines a function from the set of states on a given finite non-empty set of variables, to the set of *truth values*, $\{\text{tt}, \text{ff}\}$ with the obvious interpretation.

The weakest precondition was introduced by Dijkstra in [2]. For a given program $\mathbf{S}$ and condition on the final state (expressed as a formula) $\mathbf{R}$, the weakest precondition $\text{WP}\mathbf{S}, \mathbf{R}$ 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: $\text{WP}x := 5, x > y = 5 > y$. In [7,9] we develop a Wide Spectrum Language (WSL) which includes general specifications expressed in first order logic, and imperative programming constructs. We show that the weakest precondition of any program in WSL for any condition on the final state can be expressed as a simple formula of infinitary logic. The infinitary logic we use is a simple extension of standard first order logic which allows the conjunction or disjunction of a countably infinite sequence of formulae as a valid formula. We then go on to prove that the refinement relation between two programs is captured by the implication of their corresponding weakest preconditions, ie if $\mathbf{S}_1$ and $\mathbf{S}_2$ are programs then:

$$\mathbf{S}_1 \leq \mathbf{S}_2 \quad \iff \quad \text{WP}\mathbf{S}_1, \mathbf{R} \Rightarrow \text{WP}\mathbf{S}_2, \mathbf{R} \tag{3}$$

for an arbitrary formula $\mathbf{R}$. This means that the problem of proving a refinement or equivalence on two programs is reduced to proving an implication or equivalence of two formulae, for which all the tools of mathematics are available to assist. This technique has proved highly successful, we have developed a large catalogue of useful transformations and have been able to tackle a diverse range of algorithms and specifications [6,7,9].

3.2 The Atomic Specification

We want the language we are modeling to include general specifications (expressed in terms of mathematical logic) as well as programs. This will reduce the task of proving that a program is a correct implementation of a specification to one of proving that one statement (the program) is a refinement of another statement (the specification). Instead of having two languages (a specification language and a programming language) all our proofs and carried out within a single *wide spectrum language*. When implementing specifications as executable programs we will often need to assign values to temporary variables which are not mentioned in the specification and whose 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 a new primative statement, the *atomic specification* which specifies a program using logical formulae:

Definition 1 *The Atomic Specification: written* $\mathbf{x}/\mathbf{y}.\mathbf{Q}$*, where* $\mathbf{Q}$ *is a formula of first order logic and* $\mathbf{x}$ *and* $\mathbf{y}$ *are sequences of variables, is a form of nondeterministic assignment statement. Its effect is to add the variables in* $\mathbf{x}$ *to the state space, assign new values to them such that* $\mathbf{Q}$ *is satisfied, remove the variables in* $\mathbf{y}$ *from the state and terminate. If there is no assignment to the variables in* $\mathbf{x}$ *which satisfies* $\mathbf{Q}$ *then the Atomic Specification does not terminate (ie it is not defined for those initial states).*

This is based on the "atomic description" of Back [1].

Some examples of Atomic Specifications:

1. $\langle x \rangle / \langle \rangle . (x > y)$
   This sets x to any value greater then the value of y. If there is no such value then the specification does not terminate.
2. $\langle z \rangle / \langle \rangle . (z = x + y)$; $\langle \rangle / \langle z \rangle . (x = z)$
   This sequence implements the assignment statement $x := x + y$ using a temporary variable z.
3. $\mathbf{x}' / . \mathbf{Q}$; $\mathbf{x} / \mathbf{x}' . (\mathbf{x} = \mathbf{x}')$
   Here $\mathbf{x}$ is a sequence of variables and $\mathbf{x}'$ a sequence of temporary variables. These statements implement the general assignment statement: $\mathbf{x} := \mathbf{x}' . \mathbf{Q}$ which assigns new values $\mathbf{x}'$ to $\mathbf{x}$ where $\mathbf{Q}$ gives the relation between $\mathbf{x}$ and $\mathbf{x}'$.
4. $\langle n, x, y, z \rangle / \langle \rangle . (n, x, y, z \in \mathbb{N}^+ \ \wedge \ (n > 2) \wedge \ (z^n = x^n + y^n))$
   This example illustrates the fact that proving the termination of even a single primitive statement of WSL can be quite a challenge!

3.3 The join Construct

Together with the atomic description and more familiar programming constructs the Wide Spectrum Language includes a new construct called *join*. The join of two programs is defined to be the weakest (ie least defined) program which satisfies any specification satisfied by either of the two programs. If one of the component programs does not terminate for a particular initial state then it cannot satisfy any specification defined on that state, so the join of the two programs is identical to the other program on that state. A property of the join construct is that any program which refines both components will also refine their join. This property is very useful in searching the repository: if we have a specification which we wish to implement we first want to search the database for an equivalent (or at least similar) specification which has already been implemented. For a large and complex specification this will give rise to a potentially highly complex matching problem. If, however, the specification is expressed as the join of several simpler specifications then the matching problem for each component will be much easier to solve. Once we have found all the components, we can search through the refinement links to find a common ancestor to all of the components. From the above property of join, this ancestor will be a correct implementation of the full specification. We give an example of this search below.

We have used this theory to develop tools for the development of algorithms and programs from specifications, and the derivation of specifications from code (we term this process "inverse engineering"). A large catalogue of practical transformations and refinements has been developed which are being applied to a wide variety of programs.

4 Why Invent *Another* New Language?

There are several reasons why we have invented another language rather than using an existing programming language such as C or ADA:

- We needed a language with a simple semantics and tractable reasoning methods. In particular, our language has been designed from the start with ease of transformation and refinement as a major objective. New constructs are added to the language only if we can show that they will be easy to work with: in particular, we need a useful set of transformations which make use of that construct before it becomes part of the language. This policy has proved very successful and enabled us to avoid some of the problems which can occur when the language definition is the starting point for research.

- We wanted to include the implementation of a (possibly non-executable) specification as an allowable refinement step, we also wanted to be able to write programs using a mixture of specification and programming constructs. This facilitates the stepwise refinement of specifications into programs and the iterative analysis of programs into specifications. No existing implementable programming language includes general specifications in its syntax (for obvious reasons!).
- By expressing our results in a general language we get results which are independent of any particular programming language. Programs in existing programming language can be transcribed into WSL, manipulated as WSL programs, and then re-transcribed, perhaps into a different programming language.
- All existing programming languages have limitations (in particular, the limitation to executable constructs which is intolerable in a specification language). Also many popular languages have a number of quirks and foibles which would greatly complicate the semantics while adding little expressive power.

5 Components

The components in out repository will consist of pieces of Wide Spectrum Language (WSL) code [7,9]: this code could be either a program module, or the specification of a module, or a mixture of programming constructs and specifications. We have extended the WSL language to express meta-programming constructs (ie program editing operations, transformations and refinements). This means that as well as recording the specification and implementation of a module as two components, the sequence of transformations used to derive the implementation from the specification can also be recorded as a third component. These derivation histories can be generalised into derivation *strategies* which can also be transformed by applying meta-transformations written in a meta-meta-programming language. In fact, the meta-meta-programming language is identical with the meta-programming language since this is simply an extension of WSL.

Documentation and informal requirements in the form of text are also included as components of the repository.

6 Component Links

The components are connected together using links to form the repository, there are two different types of link:

- **Formal links** which record proven facts about the components and which are therefore transitive. These are of four different types:
  1. **Change in data representation:** $\twoheadrightarrow$ This links two programs which are equivalent in effect but which use different representations of the data.
  2. **Refinement:** $\longrightarrow$ This links a "less defined" program to a refinement of it.
  3. **Transformation:** $\leftrightarrows$ This links two programs which are equivalent, but which may use different internal data or algorithms for example.
  4. **Reification:** $\xrightarrow{\cong}$ This is similar to transformation in that it links equivalent programs, but the program to the right uses a less abstract internal data representation and less abstract algorithms and is therefore closer to an implementation.
- **Informal links:** $\dashrightarrow$ These connect documentation, informal requirements and keywords to the other components.

7 An Example

In this section we present a small example of a fragment of a repository concerning sorting programs. For the moment we will restrict attention to a subset of the repository components and the formal links which connect them. The components are as follows (here $A[1..n]$ is an array of elements to be sorted in place):

- **abort**: this is the totally undefined program, any program is therefore a refinement of **abort**.
- $\mathit{random\_perm} = A := A'.(\exists \pi \in \mathit{Perms}(n). \forall 1 \leq i \leq n.\, A[\pi(i)] = A'[i])$: here $\mathit{Perms}(n)$ is the set of all permutations of the elements $\{1, \ldots, n\}$. This program nondeterministically permutes the elements of array A.
- $\mathit{random\_inc\_seq} = \langle A \rangle / \langle \rangle.\mathit{Sorted}(A)$: here $\mathit{Sorted}(A) =_{\text{DF}} \forall 1 \leq i < j \leq n.\, A[i] \leq A[j])$ ie it is true iff array A is sorted. This program assigns arbitrary values to the array such that it is sorted.
- $\mathit{SORT} =$ **join** $\mathit{random\_inc\_seq} \sqcup \mathit{random\_perm}$ **nioj**: This is a specification of a program to sort A. Note that it gives no indication of the algorithm to use (testing all possible assignments of increasing sequences to see which are permutations of the original array is not a practical sorting algorithm!). Note the use of **join** to split up the specification into two simpler sub-specifications. (Sort is probably a simple enough concept by itself—this is just an example to illustrate the technique).
- *merge_sort_array*: this is the implementation of a merge sorting algorithm.
- *merge_sort_file*: this is obtained from *merge_sort_array* by changing the data representation: we use an array to represent a file.
- *recursive_quicksort*: this is obtained from *SORT* by "algorithm derivation" (see below).
- *iterative_quicksort*: a reified algorithm obtained from *recursive_quicksort*.
- *ADA_quicksort*: obtained from *iterative_quicksort* by transforming into a form which can be automatically translated into an efficient ADA module.
- *C_quicksort*: see *ADA_quicksort*.

The links which connect these components are shown diagrammatically in Figure 1. Let us suppose that the user of the repository wishes to implement a sorting algorithm. He writes his specification in the form of a **join** of smaller specifications which he then searches the repository to see if implementations already exist. The initial stages of the search could make use of informal links: for instance an analysis of the specifications would suggest that "Array" would be a suitable keyword to restrict the search to specifications of array manipulation programs. Theorem-proving techniques can be applied to prove, for instance, that *random_perm* is a refinement of one component of the specification, and that *random_inc_seq* is a refinement of the other component. Once refinements of all components have been found then the refinement and reification links can be searched automatically to find a common descendant of all the refinements. In this case *SORT* will be found immediately.

Note that the process of finding a common ancestor could fail: for example the system might notice that **skip** is a refinement of *random_perm*, but the **join** of **skip** and *random_inc_seq* is not fully defined (in fact it is the guard $[\mathit{Sorted}(A)]$. If a guard or other partially-domained statement (or "miracle" as some auther's refer to it, for example [4]) is reached in the search for a common descendant then the search has failed since such a specification cannot be implemented.

Once *SORT* has been found, the various implementations of the specification can be extracted by following the reification links. For example the "quicksort" implementation could be selected. This has previously been transformed into an efficient iterative algorithm which has been massaged into

$$\begin{tikzcd}
 & \textbf{abort} \arrow[dl] \arrow[dr] & \\
\mathit{random\_perm} \arrow[dr] & & \mathit{random\_inc\_seq} \arrow[dl] \\
\mathit{sort\_file} \arrow[d, harpoon, "\cong"] & \mathit{SORT} \arrow[l, "|" marking] \arrow[d, harpoon, "\cong"] \arrow[dr, "\cong"] & \\
\mathit{merge\_sort\_file} & \mathit{merge\_sort\_array} \arrow[l, "|" marking] & \mathit{recursive\_quicksort} \arrow[d, harpoon, "\cong"] \\
 & & \mathit{iterative\_quicksort} \arrow[dl, "\cong"] \arrow[d, harpoon, "\cong"] \\
 & \mathit{ADA\_quicksort} & \mathit{C\_quicksort}
\end{tikzcd}$$

Figure 1: A Fragment of a Repository

two forms, one appropriate for translation to C and the other for ADA. See [8] for the derivation of the programs from the *SORT* specification. Alternatively a file sorting algorithm could be extracted by following the "change data representation" links.

This process is analogous to the process we go through when selecting a purchase from the set of manufacturer's offerings. We have a rough idea of what we want, which is still precise enough for us to check it against the given supply. Our requirements are frequently expressed as a set of objectives which we require to be simultaneously achieved. (This is analogous to writing a specification as the join of a set of incomplete specifications. A suitable implementation has to satisfy all the component specifications). As we narrow down the set of possibilities we add more details to our specification and make more precise discriminations. If no ready-built product is suitable then we may choose to get one specially manufactured, the manufacturing process will throw up requirements for components which will have to be searched for in turn.

8 Adding Existing Code

One problem with current research on reuse is that several people have produced prototype component repositories but nobody wants to start using them because of the enormous effort involved in developing a large enough set of components for the repository to be useful. With the system presented here this problem is much less acute: existing code can be placed in the repository, initially with informal links only. Later, as the code is analysed using code analysis and specification tools such as the Maintainer's Assistant [6,10]. This process can be carried out in conjunction with normal maintenance, as the specifications of code are extracted they can be placed in the repository. In addition the transformational development of new code from specifications and components will provide new components and links for the repository.

9 Problems and Benefits

9.1 Problems

- **Specification Matching:** Any repository or component library is only as good as the technique for matching specifications and extracting components. This is a difficult problem in general: the problem gets more difficult as the library gets larger, but this is just when it is becoming more useful. We believe that the technique of including a large collection of "partial" or "generic" specifications which can be composed using the **join** operator will greatly assist in finding the required component and eliminating unsuitable matches. With a large set of generics there will be a number of paths through the repository, from the results of an initial informal keyword browse to the desired component. Thus the large size of the library actually *assists* in the search rather than hindering it. Developing a "standard style" for writing specifications and a standard set of generics for composing larger specifications will greatly assist the theorem proving specification matcher and improve the ease with which specifications and other components can be extracted.
- **Size of Repository:** The more components and links (especially formal links) there are in the repository, the more useful it will be in the construction of new software. Many of the components will be substantial pieces of code or documentation, including perhaps many different versions of the same piece of code tailored for different purposes. Thus the overall size of the repository is likely to be very large. This can be alleviated with the use of optical WORM[2] storage since most of the operations will consist of reading from and adding to the repository with only very occasional deletions.

[2]Write Once Read Many

- **Efficiency:** Constructing a program from a set of general-purpose reusable components can often generate a highly inefficient result. There may be extra layers of procedure calls plus the general-purpose modules are not able to make use of regularities in the data for the current program to carry out their actions more efficiently. This problem can be avoided by the application of *efficiency improving transformations* to the generated code. Optimising compilers carry this out at a very low level: they construct the program from standard code blocks which implement the high-level constructs and then optimise the result to try and remove the inefficiencies introduced. The transformations we have developed work at a higher level than any optimising compiler, they include removing unnecessary procedure calls, migrating code between modules, adding data structures to store intermediate results rather than re-calculating, changing the representation of data structures, etc. Because the transformations have been proven to preserve the effect of the program, and because they can be applied and the correctness conditions checked automatically, there is no chance of introducing clerical or logical errors in a long series of transformations. Hence they can be used freely wherever appropriate to improve the efficiency of the final product to a sufficient degree. The resulting modules can in turn be added to the repository and reused, as can any new efficiency improving techniques which are developed.

9.2 Benefits

- Recording formal as well as informal links in the repository means that the work involved in proving that a module correctly implements its specification is not lost but is repaid many times over.
- The repository records specifications and development methods as well as code, so these can be reused in the same way.
- Maintenance work carried out using tools such as the "Maintainer's Assistant" [10] generates new components with validated high-level specifications as a by-product. These can be incorporated in the repository so that the existing development and maintenance investment can be made greater use of.
- The creation of formal links means that there is a high degree of confidence that the components in the repository meet their specifications, hence new programs constructed from these components will be correspondingly reliable.
- The efficiency improving transformations make it possible to construct efficient programs out of general purpose components.
- The derivation of specifications from old code undergoing maintenance means that such code can be brought into a CASE strategy.

10 Conclusion

We have described a theory for program transformation and refinement which has proved very powerful for the derivation of programs from specifications and the analysis of existing programs in software maintenance [10]. This, together with the **join** concept for composing specifications and programs, and the "meta-programming" language for describing program developments, forms the foundation for the construction of a repository of reusable components which can be used in the construction of new software with greater reliability at greatly reduced cost.

Bibliography

[1] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[2] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[3] B. W. Kernighan, "The UNIX system and Software Reusability"," *IEEE Trans. Software Eng.* SE-10 (Sept., 1984), 513–528.

[4] C. C. Morgan, "The Specification Statement," *Trans. Programming Lang. and Syst.* 10 (1988), 403–419.

[5] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Inc., Sept., 1989.

[6] M. Ward, "Transforming a Program into a Specification," Durham University, Technical Report 88/1, 1988.

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

[8] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990.

[9] M. Ward, "Specifications and Programs in a Wide Spectrum Language," Submitted to J. Assoc. Comput. Mach., Apr., 1991.

[10] M. Ward, F. W. Calliss & M. Munro, "The Maintainer's Assistant," *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (Oct., 1989).