Specifications from Source Code—
Alchemists' Dream or Practical Reality?

M.P. Ward
Computer Science Department
Science Labs
South Rd
Durham DH1 3LE

Abstract

We describe a method for extracting high-level specifications from unstructured source code. The method is based on a theory of program refinement and transformation, which is used as the bases for the development of a catalogue of powerful semantics-preserving transformations. Each transformation is an operation on a program which has a mechanically-checkable correctness condition, and which has been rigorously proved to produce a semantically equivalent result. The transformations are carried out in a wide spectrum *programming language (called WSL). This language includes high-level specifications as well as low-level programming constructs. As a result, the formal reverse engineering process (from source code to equivalent specifications) and the redevelopment process (refinement of specifications into source code) can both be carried out within a single language and transformation theory.*

We also discuss a tool (FermaT) which has been developed to support this approach to reengineering. The tool is a program transformation system, largely written in an extension to WSL called $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL. *Thus it is possible to use the tool in the maintenance of its own source code, and this is starting to be the case.*

1 Introduction

This paper describes the application of formal program transformations to uncover specifications from source code. We take as an example a small report writing program. Due to errors in the original design, this program had several bugs which were gradually uncovered and fixed in the usual way (patches, first-time-through switches, duplicated code etc. etc.). The resulting program appears to work, but has a complex and messy structure which makes it extremely difficult to maintain.

The aim of our case study is to restructure the program and extract its specification (a concise, high-level description of *what* the program does which ignores the low-level details of *how* this result is achieved). It should be noted that our aim is emphatically NOT that of "Design Recovery" in the sense of "Recovering the original design"—the original design is full of bugs and not worth recovering! Instead we aim to transform the "base metal" of unstructured code into the "gold" of a high-level specification. (Strictly speaking, this is really a mining operation, rather than a transmutation of elements, since the "gold" is already there—it just needs to be extracted!)

The approach is based on a "Wide Spectrum Language" (called WSL) which includes both high-level abstract specifications and low-level programming constructs within the same language. The language has been developed over the last ten years in parallel with the development of the transformation theory: the catalogue of proven transformations and transformation techniques which form the basis for both refinement and reverse engineering. All the transformations have been proved correct and have mechanically checkable applicability conditions. This makes it possible to "encapsulate" the mathematics in a transformation system: the user does not need to understand how to *prove* the correctness of a transformation, he or she only needs to be able to read WSL and know the sorts of operations that can be applied to it. Since each step in the reverse engineering process consists of the application of a proven transformation, whose applicability condition has been mechanically checked, the extracted specification is guaranteed to be a correct representation of the original program.

Fundamental to our approach is the use of infinitary first order logic (see [11]) both to express the weakest preconditions of programs [6] and to define assertions and guards in the kernel language. Engeler [7] was the first to use infinitary logic to describe properties of programs; Back [1] used such a logic to express the weakest precondition of a program as a logical formula. His kernel language was limited to simple iterative programs. We use a different kernel language which includes recursion and guards, so that Back's language is a subset of ours. We show that the introduction of infinitary logic as part of the language (rather than just the metalanguage of weakest preconditions), together with a combination of proof methods using both denotational semantics and weakest preconditions, is a powerful theoretical tool which allows us to prove some general transformations and representation theorems.

The denotational semantics of the kernel language is based on the semantics of infinitary first order logic. Kernel language statements are interpreted as functions which map an initial state to a set of final states (the *set* of final states models the nondeterminacy in the language: for a deterministic program this set will contain a single state). A program $\mathbf{S}_1$ is a refinement of $\mathbf{S}_2$ if, for each initial state, the set of final states for $\mathbf{S}_1$ is a subset of the final states for $\mathbf{S}_2$. Back and von Wright [2] note that the refinement relation can be characterised using weakest preconditions in higher order logic (where quantification over formulae is allowed). For any two programs $\mathbf{S}_1$ and $\mathbf{S}_2$, the program $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$ if the formula $\forall \mathbf{R}. \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$. This approach to refinement has two problems:

1. It has not been proved that for *all* programs $\mathbf{S}$ and formulae $\mathbf{R}$, there exists a finite formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ which expresses the weakest precondition of $\mathbf{S}$ for postcondition $\mathbf{R}$. Can proof rules justified by an appeal to WP in *finitary* logic be justifiably applied to arbitrary programs, for which the appropriate *finite* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ may not exist? This problem does not occur with infinitary logic, since $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ has a simple definition for all programs $\mathbf{S}$ and all (infinitary logic) formulae $\mathbf{R}$;

2. Second order logic is *incomplete* in the sense that not all true statements are provable. So even if the refinement is true, there may not exist a proof of it.

Our approach to program refinement and equivalence solves both of these problems. Using infinitary logic allows us to give a simple definition of the weakest precondition of *any* statement (including an arbitrary loop) for any postcondition. In addition, we have proved that for each pair of statements $\mathbf{S}_1$ and $\mathbf{S}_2$ there is a single postcondition $\mathbf{R}$ such that $\mathbf{S}_1$ is a refinement of $\mathbf{S}_2$ iff $\mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ and $\mathrm{WP}(\mathbf{S}_1, \textsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \textsf{true})$ (see [19]). Another advantage is that the infinitary logic we use is *complete*, so if there is a refinement then there is also guaranteed to be a proof of the corresponding formula—although the proof may be infinitely long! However, it is perfectly practical to construct infinitely long proofs: in fact the proofs of many transformations involving recursion or iteration are infinite proofs constructed by induction. Thus infinitary logic is both necessary and sufficient for proving refinements and transformations.

We consider the following criteria to be important for any practical wide-spectrum language and transformation theory:

1. General specifications in any "sufficiently precise" notation should be included in the language. For sufficiently precise we will mean anything which can be expressed in terms of mathematical logic with suitable notation. This will allow a wide range of forms of specification, for example **Z** specifications [8] and **VDM** [10] both use the language of mathematical logic and set theory (in different notations) to define specifications. The "Representation Theorem" [19] proves that our specification statement is sufficient to specify *any* WSL program (and therefore any computable function, since WSL is certainly Turing complete);

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. The implementor can then use this latitude to provide a more efficient implementation which still satisfies the specification;

3. A well-developed catalogue of proven transformations which do not require the user to discharge complex proof obligations before they can be applied. In particular, it should be possible to introduce, analyse and reason about imperative and recursive constructs without requiring loop invariants;

4. Techniques to bridge the "abstraction gap" between specifications and programs. See [23,30] for examples;

5. 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". See [15,17,24] for examples;

6. Scalable to large programs: this implies a language which is expressive enough to allow automatic translation from existing programming languages, together with the ability to cope with unstructured programs and a high degree of complexity. See [25] for example.

A system which meets all these requirements would have immense practical importance in the following areas:

- Improving the maintainability (and hence extending the lifetime) of existing mission-critical software systems;
- Translating programs to modern programming languages, for example from obsolete Assembler languages to modern high-level languages;
- Developing and maintaining safety-critical applications. Such systems can be developed by transforming high-level specifications down to efficient low level code with a very high degree of confidence that the code correctly implements every part of the specification. When enhancements or modifications are required, these can be carried out on the appropriate specification, followed by "re-running" as much of the formal development as possible. Alternatively, the changes could be made at a lower level, with formal inverse engineering used to determine the impact on the formal specification;
- Extracting reusable components from current systems, deriving their specifications and storing the specification, implementation and development strategy in a repository for subsequent reuse. The use of the **join** construct as an indexing mechanism is discussed in [22].

2 The Example Program

Our example program was selected to illustrate how design errors and poor design, leads to a buggy program with a poor structure. Fixing the bugs in the usual way (patches, "hacks", "programming tricks" etc.) further degrades the structure until the program becomes extremely difficult to understand, despite its short length. We aim to demonstrate the results achievable by applying program transformations, based on formal logic, to such unpromising material.

The program is a simple report printer which prints a management report showing the net changes of stock items in a warehouse. It reads a sorted transaction file, consisting of a list of records, each of which contains the name of the stock item, and the amount brought in or taken out of the warehouse. Receipts are denoted by positive numbers and dispatches by negative numbers. The program should print the name and net change for each item which has experienced a net change in stock, and also note the number of items whose stock levels have changed.

The (fictional) history of this program, from the initial impressive-looking five-level hierarchical functional decomposition, through four "quick fixes" including a first-time-through switch, a patch on top of a patch, and an example of "defensive programming" (set the switch twice in a row, just to be sure it is *really* set) is eloquently described in [3], so we will only present the final version:

```
proc Management_Report ≡
  var SW1 := 0, SW2 := 0 :
    Produce_Heading;
    read(stuff);
    while NOT eof(stuff) do
      if First_Record_In_Group
        then if SW1 = 1
                then Process_End_Of_Prev_Group
             fi;
             SW1 := 1;
             Process_Start_Of_New_Group;
             Process_Record;
             SW2 := 1
        else
           Process_Record; SW2 := 1
      fi;
      read(stuff)
    od;
    if SW2 = 1 then Process_End_Of_Last_Group
    fi;
    Produce_Summary
  end.
```

Although this program is quite small, it is extremely difficult to follow the logic and convince yourself that it is correct. In the next section we introduce our approach to this sort of problem, which is based on *Inverse Engineering*.

3 Program Refinement and Transformation

The WSL language includes both specification constructs, such as the general assignment, and programming constructs. One aim of our program transformation work is to develop programs by refining a specification, expressed in first order logic and set theory, into an efficient algorithm. This is similar to the "refinement calculus" approach of Morgan et al [9,12]; however, our wide spectrum language has been extended to include general action systems and loops with multiple exits. These extensions are essential for our second, and equally important aim, which is to use program transformations for reverse engineering from programs to specifications.

Refinement is defined in terms of the denotational semantics of the language: the semantics of a program **S** is a function which maps from an initial state to a final set of states. The set of final states represents all the possible output states of the program for the given input state. Using a set of states enables us to model nondeterministic programs and partially defined (or incomplete) specifications. For programs $\mathbf{S}_1$ and $\mathbf{S}_2$ we say $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ (or $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$), and write $\mathbf{S}_1 \leq \mathbf{S}_2$, if $\mathbf{S}_2$ is more defined and more deterministic than $\mathbf{S}_1$. If $\mathbf{S}_1 \leq \mathbf{S}_2$ and $\mathbf{S}_2 \leq \mathbf{S}_1$ then we say $\mathbf{S}_1$ is equivalent to $\mathbf{S}_2$ and write $\mathbf{S}_1 \approx \mathbf{S}_2$. Equivalence is thus defined in terms of the external "black box" behaviour of the program. A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. See [14] and [16] for a description of the semantics of WSL and the methods used for proving the correctness of refinements and transformations. We use the term *abstraction* to denote the opposite of refinement: for example the "most abstract" program is the non-terminating program **abort**, since any program is a refinement of **abort**.

A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. See [14] and [16] for a description of the semantics of WSL and the methods used for proving the correctness of refinements and transformations.

4 Inverse Engineering

Inverse engineering is the process of extracting high-level abstract specifications from source code using program transformations. By developing our program transformation theory in a wide spectrum language, we are able to use transformations not only for restructuring at the same abstraction level, but also for crossing levels of abstraction: transforming from low-level code to high-level abstract specifications (see [23]). Because the transformations are proved to be correct, they can be applied without needing to understand the program first, and with a very high degree of confidence that the resulting specifications correctly represent the initial program. Similar transformations can be used to refine the specifications (either immediately or after modifications and enhancements) back to executable source code, which may be in a different programming language. It is therefore possible to use this method to migrate code between programming languages: including migrating from Assembler code to a high-level language (see [25]). The refinement of extracted specifications back to executable code can be made largely automatic, and this means it is possible to maintain the *specifications* rather than the source code. Changes and enhancements can be applied at the right level of abstraction, without diving into details of the implementation.

5 FermaT: A tool for Inverse Engineering

FermaT is a program transformation system based on the theory of program refinement and equivalence developed in [14,19] and applied to software development in [13,24] and to reverse engineering in [23,25]. The transformation system is intended as a practical tool for software maintenance, program comprehension, reverse engineering and program development.

The first prototype transformation system, called the "Maintainer's Assistant", was written in LISP [5, 28]. It included a large number of transformations, but was vey much an "academic prototype" whose aim was to test the ideas rather than be a practical tool. In particular, little attention was paid to the time and space efficiency of the implementation. Despite these drawbacks, the tool proved to be highly successful and capable of reverse-engineering moderately sized assembler modules (up to 80,000 lines) into equivalent high-level language programs.

The system is based on semantic preserving transformations in a wide spectrum language (called WSL). The language includes both low-level programming constructs and high-level non-executable specifications. This means that refinement from a specification to an implementation, and reverse-engineering to determine the behaviour of an existing program can both be carried out by means of semantic-preserving transformations within a single language.

For the next version of the tool (i.e. FermaT itself) we decided to extend WSL to add domain-specific constructs, creating *a language for writing program transformations*. This was called $\mathcal{METAWSL}$. The extensions include an abstract data type for representing programs as tree structures and constructs for pattern matching, pattern filling and iterating over components of a program structure. The "transformation engine" of FermaT is implemented entirely in $\mathcal{METAWSL}$. The implementation of $\mathcal{METAWSL}$ involves a parser for $\mathcal{METAWSL}$ (written in `rdp`, a public domain recursive descent parser package), an interpreter for $\mathcal{META}$WSL written in C, a translator from $\mathcal{META}$WSL to C (written in $\mathcal{META}$WSL and interpreted to translate itself), a small C runtime library (for the main abstract data types) and a WSL runtime library (for the high-level $\mathcal{META}$WSL constructs such as **ifmatch**, **foreach**, **fill** etc.). One aim was so that the tool could be used to maintain its own source code: and this has already proved possible, with transformations being applied to simplify the source code for other transformations! Another aim was to test our theories on language oriented programming (see [18]): we expected to see a reduction in the total amount of source code required to implement a more efficient, more powerful and more rugged system. We also anticipated noticeable improvements in maintainability and portability. These expectations have been fulfilled, and we are achieving a high degree of functionality from a small total amount of easily maintainable code: the current prototype consists of around 16,000 lines of $\mathcal{META}$WSL and C code, while the previous version required over 100,000 lines of LISP.

The FermaT design is based on a recursive application of language oriented programming, involving two "layers" of domain-specific languages. These are:

1. A fairly high-level, general purpose language, based on the executable constructs of WSL [14,19] together with an abstract data type (ADT) for recording, analysing and manipulating programs and fragments of programs. This is implemented in LISP, using a WSL to LISP translator together with a "LISP runtime library" of functions and procedures to implement the ADT. The ADT includes facilities for loading and saving programs, moving around (it records the "current selection" within the current program), and editing operations. This consists of less than 2000 lines of LISP: the entire rest of the transformation engine is written in WSL and $\mathcal{META}$WSL code. Hence porting the system to another language (and a C version is currently under development) would entail writing at the most a few thousand lines of code;

2. On top of the "base" WSL language we have implemented a very high-level, domain-specific language for writing program transformations, called $\mathcal{META}$WSL. This includes high level constructs which do most of the work involved in writing transformations: see below for an example. $\mathcal{META}$WSL is implemented almost entirely in WSL; there are a few extensions to the WSL to LISP translator and a number of WSL libraries which are compiled into LISP and linked to the translated $\mathcal{META}$WSL.

$\mathcal{META}$WSL encapsulates much of the expertise developed over the last 10 years of research in program transformation theory and transformation systems. As a result, this expertise is readily available to the programmers, some of whom have only recently joined the project. Working in $\mathcal{META}$WSL, it takes only a small amount of training before new programmers become effective at implementing transformations and enhancing the functionality of existing transformations.

6 Inverse Engineering the Example Program

In this section we show how FermaT copes with the example program of Section 2. The first three stages were carried out on a prototype of FermaT, starting with the original program and applying general-purpose transformations. The final stage (going to an abstract specification) was carried out manually because the necessary transformations are still being implemented in the tool.

6.1 First Stage: Restructure and Simplify

The first stage involves applying some general-purpose restructuring and simplification transformations. The selection of the transformations by the user is based on some simple heuristics, developed from our experience of using the tool. It is likely that these heuristics could be largely automated in the future: in fact it is a feature of the development of the various prototypes which led up to FermaT that experience with each version enabled us to elicit more knowledge about the process of restructuring and reverse engineering through formal transformations. This knowledge was then incorporated in the next version of the tool, in the form of more powerful transformations and program manipulation functions.

In the first stage we are able to remove both switches and re-express the convoluted control flow as a simple double-nested loop. It is an important feature of our method that the user of the system does not need to understand the purpose of a block of code before transforming it. The system takes care of all correctness conditions and clerical details of applying the transformation, as well as automatically pretty-printing the result. The system and supporting theoretical work together guarantee the semantic equivalence of each version of the program: thus, once an understandable version has been produced, the user can understand *that* version of the program, and be confident that the original program has the same semantics.

```
proc Management_Report ≡
  Produce_Heading;
  read(stuff);
  while NOT eof(stuff) do
    Process_Start_Of_New_Group;
    do Process_Record;
       read(stuff);
       if eof(stuff) OR First_Record_In_Group
         then exit
       fi
    od;
    Process_End_Of_Prev_Group
  od;
  Produce_Summary.
```

6.2 Second Stage: Abstract Data Types

The second stage, after restructuring, is to examine the low-level procedures and data structures and re-express them at a higher level of abstraction, making use of abstract data types where appropriate. This step can also be carried out via formal transformations: some human input is required in selecting the abstract equivalents, though the simpler cases (stacks, sequential and random access files etc.) can be automated. The next version of the program replaces each procedure call with the abstract specification of the procedure body. This version is therefore more complete than the previous one (the "missing" code in the procedure bodies is now included):

```
proc Management_Report ≡
  var i := 0, last := "", record := "", changed := 0 :
    write("Management Report ... ");
    last := record; i := i + 1; record := records[i];
    while i ⩽ ℓ(records) do
      total := 0;
      do total := total + record.number;
         last := record;
         i := i + 1; record := records[i];
         if i > ℓ(records) OR
              last.name ≠ record.name
           then exit
         fi
      od;
      if total ≠ 0
        then write(last.name, total);
             changed := changed + 1
      fi
    od;
    write("Changed items:", changed);
  end.
```

6.3 Third Stage: Restructure and Simplify

Having moved to a higher level of abstraction, some further simplification and restructuring becomes possible. We can get rid of another local variable, and convert both loops to **while** loops. This version is at a intermediate level of abstraction, it uses a higher level data structure than the original source code, but still carries out roughly the same operations.

```
proc Management_Report ≡
  var i := 0, changed := 0 :
    write("Management Report ... ");
    while i ⩽ ℓ(records) do
      total := records[i].number;
      i := i + 1;
      while i ⩽ ℓ(records) AND
            records[i − 1].name = records[i].name do
        total := total + records[i].number;
        i := i + 1
      od;
      if total ≠ 0
        then write(records[i − 1].name, total);
             changed := changed + 1
      fi
    od;
    write("Changed items:", changed);
  end.
```

6.4 Fourth Stage: Specification Level

For the final step, we replace the loops by higher-level operators which describe the effect of the double loop more clearly. Our notation for list operations is based on [4]. $*$ is a "map" operator which takes a unary function and applies it to each element of a list. / is a "reduce" operator which takes a binary function and applies it to a list, for example $+/l$ returns the sum of the elements of list l. The split function takes a list and a binary condition and returns a list of lists, formed by splitting the original list into non-empty sections at the points where the condition is false.

```
proc Management_Report ≡
  begin
    var q := split(records, same_name?) :
      q := summarise ∗ q;
      q := filter(q, change?);
      write("Management Report ... ");
```

```
      write * q;
      write("Changed items:", ℓ(q))
    end
  where
  funct same_name?(x, y) ≡
    x.name = y.name.
  funct summarise(g) ≡
    ⟨g[1].name, +/(.number * g)⟩.
  funct change?(g) ≡
    g[2] ≠ 0.
  end
```

Our specification may therefore be written informally as follows:

1. First split the list of records into sections, where the records in each section all have the same name;
2. Then summarise each section to produce a list of summary pairs. The summarise function returns a pair consisting of the name (common to all records in the section) and the sum of all the number components of the records;
3. Then filter out the summaries with zero totals (these should not appear on the report);
4. The print a report consisting of a header, the list of summaries (one per line) and the number of summaries listed.

This specification illustrates very clearly the relationship between the input data and the final report. It also shows explicitly that the "Changed items" number is precisely the number of summaries listed, which in turn is the number of items whose stock has changed since the last report. This fact would require a careful analysis of the program plus some inductive reasoning if it was to be deduced from the original version of the program.

7 Conclusions

Our approach to reengineering, based on inverse engineering followed by formal refinement, has proved very successful with a number of challenging small-scale case study programs [13,20,21,23,24]. The theoretical work has been developed further to accommodate real time and parallel programs with some success [29,30]. More recently, the development of industrial-strength tool support has allowed us to tackle large JOVIAL restructuring projects, IBM 370 Assembler restructuring projects for modules of up to 20,000 lines, and Assembler to COBOL migration projects [26,27]. Durham Software Engineering Ltd and Durham University are actively developing the tool with translators to and from various languages (including C and COBOL) which will extend the migration and reengineering capabilities of the tool. For the Assembler restructuring and migration projects we have developed techniques for unscrambling the (often convoluted) control flow between subroutines: this has to cope with subroutines which overwrite or modify the stored return address, routines which call other subroutines directly (without storing a return address) and so on. In addition, we have developed techniques for changing the data model from a monolithic block of memory, accessed via pointers (which is essentially how assembler code treats its data) to the equivalent high-level data structures and operations.

References

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

[2] R. J. R. Back & J. von Wright, "Refinement Concepts Formalised in Higher-Order Logic," *Formal Aspects of Computing* 2 (1990), 247–272.

[3] G. D. Bergland, "A Guided Tour of Program Design Methodologies," *Computer* 14, 18–37.

[4] R. Bird, "Lectures on Constructive Functional Programming," Oxford University, Technical Monograph PRG-69, Sept., 1988.

[5] T. Bull, "An Introduction to the WSL Program Transformer," *Conference on Software Maintenance 26th–29th November 1990, San Diego* (Nov., 1990).

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

[7] E. Engeler, *Formal Languages: Automata and Structures*, Markham, Chicago, 1968.

[8] I. J. Hayes, *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[9] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 (Aug., 1987), 672–686.

[10] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[11] C. R. Karp, *Languages with Expressions of Infinite Length*, North-Holland, Amsterdam, 1964.

[12] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[13] H. A. Priestley & M. Ward, "A Multipurpose Backtracking Algorithm," *J. Symb. Comput.* 18 (1994), 1–40, ⟨http: // www. dur. ac. uk/ ~dcs0mpw/ martin/ papers/backtr-t.ps.gz⟩.

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

[15] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨http: // www. dur. ac. uk/ ~dcs0mpw/ martin/ papers/ sorting-t.ps.gz⟩.

[16] M. Ward, “Specifications and Programs in a Wide Spectrum Language,” Submitted to J. Assoc. Comput. Mach., 1991.

[17] M. Ward, “A Recursion Removal Theorem,” Springer-Verlag, Proceedings of the 5th Refinement Workshop, London, 8th–11th January, New York–Heidelberg–Berlin, 1992, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/ref-ws-5.ps.gz`⟩.

[18] M. Ward, “Language Oriented Programming,” *Software—Concepts and Tools* 15 (1994), 147–161, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/middle-out-t.ps.gz`⟩.

[19] M. Ward, “Foundations for a Practical Theory of Program Refinement and Transformation,” Durham University, Technical Report, 1994.

[20] M. Ward, “Reverse Engineering through Formal Transformation Knuths “Polynomial Addition” Algorithm,” *Comput. J.* 37 (1994), 795–813, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/poly-t.ps.gz`⟩.

[21] M. Ward, “Program Analysis by Formal Transformation,” *Comput. J.* 39 (1996).

[22] M. Ward, “Using Formal Transformations to Construct a Component Repository,” in *Software Reuse: the European Approach*, Springer-Verlag, New York–Heidelberg–Berlin, Feb., 1991, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/reuse.ps.gz`⟩.

[23] M. Ward, “Abstracting a Specification from Code,” *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz`⟩.

[24] M. Ward, “Derivation of Data Intensive Algorithms by Formal Transformation,” *IEEE Trans. Software Eng.* 22 (Sept., 1996), 665–686, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/sw-alg.ps.gz`⟩.

[25] M. Ward & K. H. Bennett, “A Practical Program Transformation System For Reverse Engineering,” *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (1993), ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/icse.ps.gz`⟩.

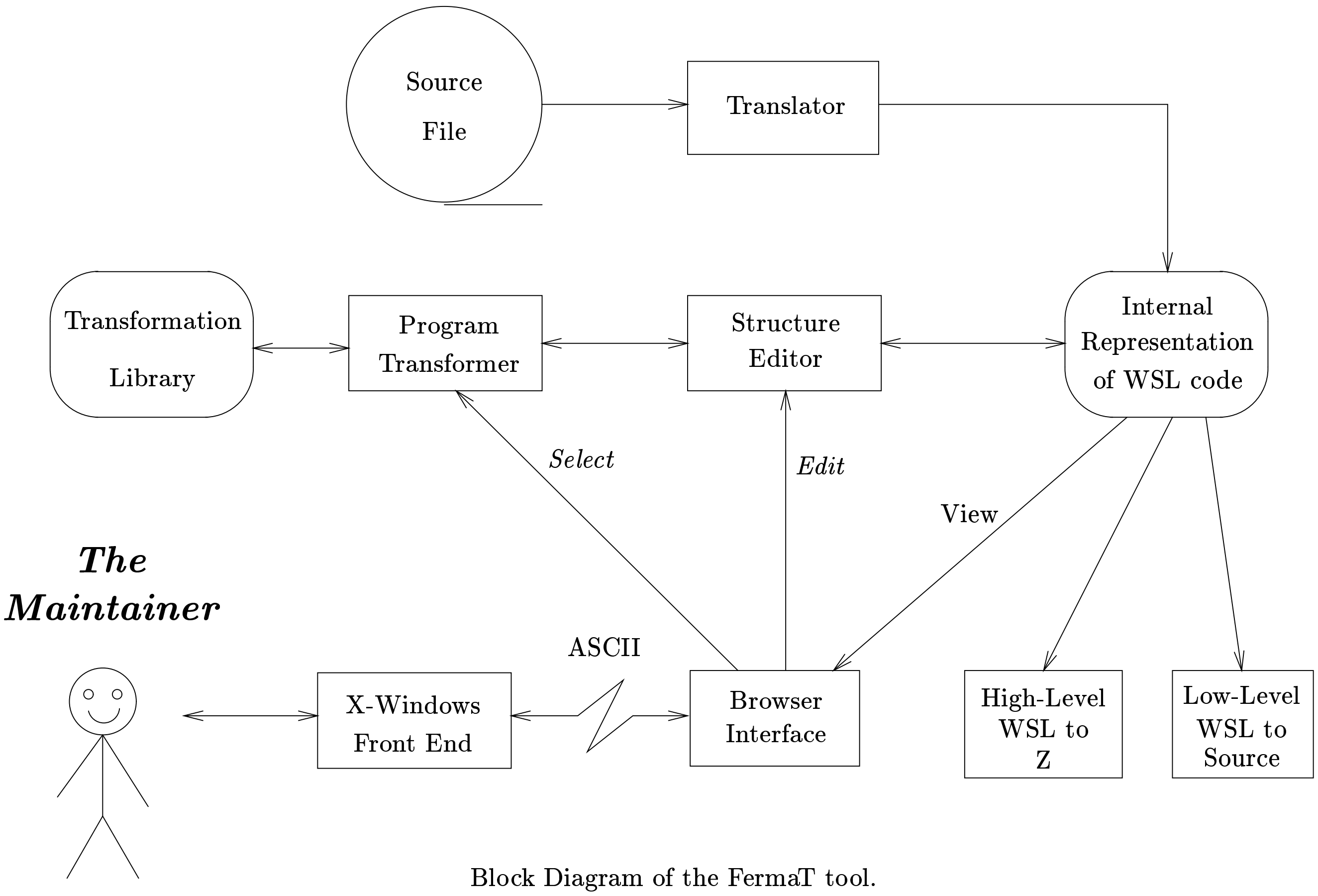
[26] M. Ward & K. H. Bennett, “Formal Methods to Aid the Evolution of Software,” *International Journal of Software Engineering and Knowledge Engineering* 5 (1995), 25–47, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/evolution-t.ps.gz`⟩.

[27] M. Ward & K. H. Bennett, “Formal Methods for Legacy Systems,” *J. Software Maintenance: Research and Practice* 7 (May, 1995), 203–219, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/legacy-t.ps.gz`⟩.

[28] M. Ward, F. W. Calliss & M. Munro, “The Maintainer’s Assistant,” *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (1989), ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/MA-89.ps.gz`⟩.

[29] E. J. Younger & M. Ward, “Understanding Concurrent Programs using Program Transformations,” *Proceedings of the 1993 2nd Workshop on Program Comprehension, 8th-9th July*, Capri, Italy (1993), ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/cap.ps.gz`⟩.

[30] E. J. Younger & M. Ward, “Inverse Engineering a simple Real Time program,” *J. Software Maintenance: Research and Practice* 6 (1993), 197–234.



Block Diagram of the FermaT tool.