Assembler to C Migration using the FermaT Transformation System

M. P. Ward
Software Migrations Ltd.
Mountjoy Research Centre
Stockton Rd.
Durham, UK
Martin.Ward@durham.ac.uk
Phone: 44+(0)191 386 0420
Fax: 44+(0)191 374 0936

Abstract

The FermaT transformation system, based on research carried out over the last twelve years at Durham University and Software Migrations Ltd., is an industrial-strength formal transformation engine with many applications in program comprehension and language migration. This paper describes one application of the system: the migration of IBM 370 Assembler code to equivalent, maintainable C code. We present an example of using the tool to migrate a small, but complex, assembler module to C with no manual intervention required. We briefly discuss a mass migration exercise where 1,925 assembler modules were sucessfully migrated to C code.

Keywords: Assembler, Migration, Comprehension, Formal Methods, WSL, Wide Spectrum Language, Program Transformation, Legacy Systems, Restructuring.

1 Introduction

There is a vast collection of operational software systems which are vitally important to their users, yet are becoming increasingly difficult to maintain, enhance and keep up to date with rapidly changing requirements. For many of these so called *legacy systems* the option of throwing the system away and re-writing it from scratch is not economically viable. Methods are therefore urgently required which enable these systems to evolve in a controlled manner. In particular, legacy assembler systems have high maintenance costs, and migrating such systems to a different environment (eg. a client-server architecture) is much more difficult than for C or COBOL systems. The FermaT transformation system uses formal proven program transformations, which preserve or refine the semantics of a program while changing its form. These transformations are applied to restructure and simplify the legacy systems and to extract higher-level representations. This paper describes one application of the system: the migration of IBM 370 Assembler code to equivalent, maintainable C code.

By using an appropriate sequence of transformations, the extracted representation is guaranteed to be equivalent to the original code logic. The method is based on a formal wide spectrum language, called WSL, with accompanying formal method. Over the last ten years we have developed a large catalogue of proven transformations, together with mechanically verifiable applicability conditions. These have been applied to many software development, reverse engineering and maintenance problems. In this paper, we focus on the results of using this approach in the migration of IBM Assembler to C. (The same techniques are also being applied to migration to COBOL). We conclude that formal methods have an important practical role in program comprehension and software migration.

2 Theoretical Foundation

The theoretical work on which FermaT is based originated not in software maintenance, but in research on the development of a language in which proofs of equivalence for program transformations could be achieved as easily as possible for a wide range of constructs.

WSL is the "Wide Spectrum Language" used in our program transformation work, which includes low-level programming constructs and high-level abstract specifications within a single language. This has the advantage that one does not need to differentiate between programming and specification languages: the entire transformational development of a program from abstract specification to detailed implementation can be carried out in a single language.

Conversely, the entire reverse-engineering process, from a transliteration of the source program to a high-level specification, can also be carried out in the same language. During either of these processes, different parts of the program may be expressed at different levels of abstraction. So a wide-spectrum language forms an ideal tool for developing methods for formal program development and also for formal reverse engineering (for which we have coined the term *inverse engineering*).

A *program transformation* is an operation which modifies a program into a different form which has the same external behaviour (i.e. it is equivalent under a precisely defined denotational semantics). Since both programs and specifications are part of the same language, transformations can be used to demonstrate that a given program is a correct implementation of a given specification.

A *refinement* is an operation which modifies a program to make its behaviour more defined and/or more deterministic. Typically, the author of a specification will allow some latitude to the implementor, by restricting the initial states for which the specification is defined, or by defining a nondeterministic behaviour (for example, the program is specified to calculate a root of an equation, but is allowed to choose which of several roots it returns). In this case, a typical implementation will be a *refinement* of the specification rather than a strict equivalence. The opposite of refinement is *abstraction*: we say that a specification is an abstraction of a program which implements it. See [5,6] and [1] for a description of refinement.

The syntax and semantics of WSL are described in [8,9, 10,13] so will not be discussed in detail here. Most of the constructs in WSL, for example if statements, while loops, procedures and functions, are common to many programming languages. However there are some features relating to the "specification level" of the language which are unusual. Expressions and conditions (formulae) in WSL are taken directly from first order logic: in fact, an infinitary first order logic is used (see [4] for details), which allows countably infinite disjunctions and conjunctions, but this is not essential for understanding this paper. This use of first order logic means that statements in WSL can include existential and universal quantification over infinite sets, and similar (non-executable) operations.

Over the last twelve years we have been developing the WSL language, in parallel with the development of a transformation theory and proof methods. Over this time the language has developed from a simple and tractable kernel language [9,10] to a complete and powerful programming language. At the "low-level" end of the language there exists automatic translators from IBM Assembler into WSL, and from a subset of WSL into C. At the "high-level" end it is possible to write abstract specifications, similar to **Z** and VDM.

The WSL language includes constructs for loops with multiple exits, action systems, side-effects, etc. and the transformation theory includes a large catalogue of proven transformations for manipulating these constructs. Many of the transformations have been implemented in the FermaT transformation engine developed by Software Migrations Ltd. [11,15,16].

In [12,14] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [12,13,14] the same transformations are used in the reverse direction: using transformations to derive a concise abstract representation of the specification for several challenging programs.

Our aim in this paper is to describe how the transformation theory has been sucessfully applied to the very challenging task of migrating from assembler language to modular and maintainable C code. As far as we know, none of the other researchers in program transformations (for example, [2,7]) have attempted to apply their methods to assembler code.

3 The FermaT Workbench

The FermaT Workbench consists of a collection of tools and databases based around the core technology of program transformations in the WSL language. The objectives of the FermaT Workbench are:

1. Increase the *comprehension* of legacy code to:
   (a) Improve maintenance productivity and hence reduce maintenance costs;
   (b) Improve the quality of maintenance by properly understanding the impact of code changes;
   (c) Enable business enhancements to be delivered faster;
   (d) Support the analysis of existing legacy systems; and
2. Support the *re-engineering* of legacy code to:
   (a) Arrive at better structured programs, and hence increase code quality and subsequent maintainability;
   (b) Ease the burden of code conversion tasks (eg Year 2000, EMU, Product Codes, etc.); through the provision of code change assessment and semi-automated conversion facilities;
   (c) Allow legacy code to be automatically migrated to more mainstream higher-level languages and hence extend the life of existing systems.

The major components in the Workbench are:

- Inventory Gatherer: This contains tools to enable the user to scan, select and collect an inventory of files to create a "project" and populate the FermaT repository;
- Inventory Navigator: These tools provide the user with an overall view of the project and select modules for further processing. The tools include call graph and structure chart generators;
- Program Analysis Environment: This contains tools to display an interactive program flow chart which is linked to an integrated text editor, to determine the impact of changing one or more data fields (this is also used for Year 2000 and EMU analysis) and to display the structure of the data declarations;
- Repository: This stores and controls all input, working, database and output files related to a project. Project files may be at different stages of FermaT processing:

  Level 0 Collected and scanned code;

  Level 1 Current Physical: parsed code and data and unstructured WSL;

  Level 2 Restructured Physical: restructured WSL code and structured data;

  Level 3 Abstracted Business: interactively transformed or verified WSL code and data structures;

  Level 4 Generated Physical: target language source code.
- Code Parsers: include Assembler to WSL translator and a COBOL parser. (The COBOL parser is under development);
- Transformation Engine: This is the heart of the FermaT workbench. The Engine contains an extensive library of proven WSL program transformations, developed over the past twelve years of research, together with heuristics for applying transformations to achieve different goals. For example, there are heuristics for restructuring translated assembler code, removing dispatch calls and condition code references. The transformation engine also handles WSL to target language translation and data flow analysis;
- Data Transformer: analyses data layouts (lengths, types and offsets) to generate structured data declarations;
- Code Generators: take the transformed WSL code and restructured data layouts and generate target language source code and data declarations. These are customisable to meet varying coding standards and the need to trade efficiency and maintainability;
- Document Generators: produce flowcharts, data catalogues, call graphs, structure charts, program listings and CASE tool export files.

4 Modelling Assembler in WSL

Constructing a useful scientific model necessarily involves throwing away some information: in other words, to be useful a model *must* be inaccurate, or at least idealised, to a certain extent. For example "ideal gases", "incompressible fluids" and "billiard ball molecules" are all useful models which gain their utility by abstracting away some details of the real world. In the case of modelling a programming language, such as Assembler, it is theoretically possible to have a perfect model of the language which correctly captures the behaviour of all assembler programs. Certain features of Assembler, such as branching to register addresses, self-modifying code and so on, would imply that such a model would have to record the entire state of the machine, including all registers, memory, disk space, and external devices, and "interpret" this state as each instruction is executed. (Consider the effect of loading some data from a disk file into memory, performing arithmetic at arbitrary places in the data, and then branching to the start of the data block!) Unfortunately, such a model is useless for reverse engineering or migration purposes.

What we need is a practical model for assembler programs which is suitable for reverse engineering, and is accurate enough to deal with all the programming constructs which are likely to be encountered.

4.1 Assembler to WSL Translation

The assembler to WSL translator works from a listing file, rather than a source file, in order to make as much information available as possible. For example: the listing will usually contain macro expansions, it will show the base and index registers determined for each instruction, it will list the offset of each instruction and data item, and any conditional assembly instructions will have been expanded already. The translator makes use of *all* this information, so while it would be possible to write a translator which works from source files, such a translator would have to duplicate much of the functionality of an assembler. The translator generates two output files:

<**file**>**.wsl** contains the WSL translation of all the executable code;

<**file**>**.dat** contains information about each symbol declared or referenced in the listing: the length, offset, type, initial value, and the DSECT or CSECT to which it belongs. Separate programs will restructure the data file into hierarchical structures and unions. Other

programs generate C header files or COBOL data divisions.

The assembler to WSL translator includes the following features:

- Standard opcodes: Each assembler instruction is translated into WSL statements which capture *all* the effects of the instruction. The machine registers and memory are modelled as arrays, and the condition code as a variable. Thus, at the translation stage we don't attempt to recognise "if statements" as such, we translate into statements which assign to `cc` (the condition code variable), and statements which test `cc`.

- Standard system macros for file handling etc. When translating a `GET` macro, for example, the system determines the error label (if any) and end of file condition label (by searching for the data control block declaration) and inserts the appropriate tests and branches.

- User macros can be added to the translation table, with an appropriate WSL translation. If a macro is found which is not in the translation table, then the macro expansion is translated. If there is no macro expansion, then a suitable procedure call is generated.

- All structured macros are handled by simply translating the macro expansion: this replaces the structure by equivalent branches and labels, but our restructuring transformations are powerful enough to recover the original structure in each case.

- The condition code is implemented as a variable (`cc`): this is because when a condition code is set it is not always obvious exactly where it will be tested, and it may be tested more than once. Specialised transformations convert conditional assignments to `cc` followed by tests of `cc` into simple conditional statements.

- BAL/BAS (Branch and Save), and branch to register: this is handled by attempting to determine all possible targets of any branch to register instruction by determining all the places where a return address could be saved, or where a modified return address could end up at. Each label is turned into a separate action with an associated value (the relative address). A "store return address" instruction stores the *relative* address in the register. A "branch to register" instruction passes the relative address to a "dispatch" action which tests the value against the set of recorded values, and jumps to the appropriate label. This can deal with simple cases of address arithmetic (including jump tables) but may theoretically be defeated if more complex address manipulations are carried out before a branch to register instruction is executed.

- Simple external branches (external subroutine calls) are detected.

- Simple jump tables are detected: the code for detecting jump tables can be customised and extended as necessary.

- EXecute statements are detected and generate the appropriate code (the executed statement is translated and then modified appropriately). The "Execute" (EX) instruction in IBM assembler is a form of self-modifying code: it takes two parameters, a register number and an address of the actual instruction to be executed. If the register number is non-zero, then the actual instruction is modified by the register contents before being executed. Execute instructions are typically used to create a variable-length move or compare operation (by overwriting the length field of a normal move or compare instruction).

- Data Declarations: all assembler data (EQUates, DS, DC, DCB etc.) are parsed and restructured into C unions and structs, where appropriate.

- DSECTs are converted into pointers to structs (whenever the DSECT's base register is modified, the appropriate pointer is modified to keep it in step).

- EQUates are translated as #defines, apart from: (a) "`EQU *`" in a data area, which is translated as an appropriate data element, and (b) "`FOO EQU BAR`" which is recorded as declaring `FOO` as a synonym for `BAR`. (If the C translation of `BAR` is `baz.bar`, for example, then the C translation for `FOO` will be `baz.foo`).

- Self-modifying code: cases where a NOP or branch is modified into a branch or NOP are detected and translated correctly (using a generated flag).

- C header files are generated automatically: one for the main program and separate header files for each DSECT referenced.

- Structured and unstructured CICS calls (eg `HANDLE AID`, `HANDLE CONDITION`) are translated into the appropriate code. Unstructured CICS calls are translated into equivalent structured code through a mechanism which can be extended to other macro packages, eg databases, SQL, etc.

The aim of the assembler to WSL translator is to generate WSL code which models as accurately as possible the behaviour of the original assembler module: without worrying too much about the size, efficiency or complexity of the resulting code. Typically, the raw WSL translation of an assembler module will be three to five times bigger than the source file and have a very high McCabe cyclomatic

complexity (typically in the hundreds, often in the thousands). This is, in part, because every "branch to register" instruction branches to the dispatch routine, which in turn contains branches to every possible return point.

However, the FermaT transformation engine includes some very powerful transformations for simplifying WSL code, removing redundancies, tracking dispatch codes, and so on. In most cases FermaT can automatically unscramble the tangle of "branch and save" and "branch to register" code to extract self-contained, single-entry single-exit procedures and so eliminate the dispatch procedure. In addition, FermaT can nearly always eliminate the cc variable by constructing appropriate conditional statements.

5 The Sample Program

Our sample program was taken from "A Guided Tour of Program Design Methodologies", by G. D. Bergland [3] who in turn took it from a story called "Getting it Wrong" that has been related by Michael Jackson on numerous occasions:

```
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_Previous_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.
```

The program is a simple report generator which reads a sorted transaction file: each transaction contains the name of an item and the amount received or distributed from the warehouse. The program generates a report showing the net change in inventory for each item in the transaction file.

Our resident assembler guru was given the above pseudocode and asked to write an assembler implementation which uses as many "features" of assembler as possible. The result is given in Section 11 (I should like to point out on his behalf that this is *not* his normal coding style!) The program includes self-modifying code (the "first time through switch" `SW1` is implemented by modifying the branch labelled `LAAA` to a `NOP` in the instruction labelled `LAB`), and an EXecute statement has been used to get a variable length move.

The following is an extract of the "raw" WSL code generated by the assembler to WSL translator:

```
LAAA ≡
  if F_LAAA = 1 then call LAB fi;
  call A_00006C end
A_00006C ≡
  r10 := 108 + 4; call ENDGROUP;
  call LAB end
LAB ≡
  F_LAAA := 0;
  call A_000074 end
A_000074 ≡
  !P mvc(a[db(WRITEM, r3), 3 + 1]
          var a[db(WLAST, r3), 3 + 1]);
  call A_00007A end
A_00007A ≡
  a[db(WNET, r3), 3 + 1] :=!XF zap(!XF p_lit(1, 1, "0"));
  if !XC dec_eq(a[db(WNET, r3), 3 + 1], 0)
    then cc := 0
  elsif !XC dec_less(a[db(WNET, r3), 3 + 1], 0)
       then cc := 1
       else cc := 2 fi;
  call A_000080 end
A_000080 ≡
  r10 := 128 + 4; call PROCGRP;
  call A_000084 end
A_000084 ≡
  a[db(XSW1, r3), 1] :=!XF x_lit(1, 1, "FF");
  call A_000088 end
A_000088 ≡
  if true then call LAA fi;
  call LAC end
LAC ≡
  r10 := 140 + 4; call PROCGRP;
  call A_000090 end
A_000090 ≡
  a[db(XSW1, r3), 1] :=!XF x_lit(1, 1, "FF");
  call A_000094 end
```

Note that each instruction expands into several WSL statements. Each symbol reference is implemented as an array access with the base register plus offset. The modified branch instruction has been implemented as a conditional branch on a machine generated flag (F_LAAA). A `BAL` instruction is implemented by storing the return address in a register and branching to the label. A branch to register (for example, to return from a subroutine) is implemented

by loading the special variable destination with the value in the register, and branching to a special dispatch routine. dispatch is generated automatically by the WSL translator and looks, in part, like this:

```
dispatch ≡
  if destination = 0
    then call Z
         ...
  elsif destination = 112
       then call LAB
  elsif destination = 132
       then call A_000084
  elsif destination = 144
       then call A_000090
            ...
       else !P external_branch( var a); call Z fi end
```

Figure 5 lists the metrics for the raw WSL translation and after automatic restructuring and simplifying transformations have been applied. The meaning of these metrics

Metric	Raw WSL	Structured WSL
Statements	561	106
Expressions	1,589	210
McCabe	184	17
Control/Data Flow	520	156
Branch–Loop	145	17
Structural	6,685	751

Figure 1. Metrics Before and After Transformation

is as follows:

Statements total number of WSL statements, including compound statements;

Expressions total number of WSL expressions, including compound expressions (these two actually count the number of nodes in the parse tree);

McCabe Cyclomatic Complexity measures the complexity of a module's decision structure. It is the number of linearly independent paths through the program;

Control/Data Flow total number of variable accesses and updates plus procedure calls and branches;

Branch–Loop total number of loops (do ... od, while and for loops) plus procedure calls and branches;

Structural a "weighted sum" over the parse tree where nodes are given weights ranging from 1 (for simple expressions) to 10 (for branches).

6 Formal Program Transformation

The first stage in the transformation process is Data Translation. This transformation uses the restructured data file to change the data representation in the program. Initially all data is accessed directly from memory (represented as the byte array a) by adding the base register to the displacement to get an address. The restructured data file gives the layout of all data in memory, so by making some reasonable assumptions about non-overlapping DSECTS etc., FermaT is able to transform the program into an equivalent program where the data is accessed directly through variables and structures. For example, the "raw WSL" statement:

```
!P mvc(a[db(WRITEM, r3), 3 + 1]
        var a[db(WLAST, r3), 3 + 1]);
```

is transformed into the simple assignment:

```
WLAST := WREC.WRITEM;
```

In the case of our simple program, there is only one structure to uncover: the WREC print record which contains fields WRITEM, WRTYPE and WRQTY plus some unnamed fillers. See the generated C header file in Section 12.

The next stage is control flow restructuring: eliminating non-essential labels and branches, introducing loops. This is carried out in a series of passes through the program, at each iteration the program is searched for points where a simplifying transformation (such as loop insertion or branch merging) can be applied. The iteration is continued until no further improvement can be achieved.

The system then analyses the remaining actions to determine which actions may form the body of a simple procedure. To do this it uses both control flow and data flow analysis. If it determines that a collection of actions form a procedure, then these actions are extracted out as a sub-action system in the body of the procedure.

After control flow restructuring we have data flow analysis: in particular an extended form of constant propagation which can propagate return addresses through procedure calls. If a dispatch call is encountered with a known destination value, then it can be unfolded and simplified. The same transformation also deals with conditional assignments to the condition code (cc) in order to remove references to cc where possible.

FermaT was able to extract a collection of actions to form the ENDGROUP procedure, so that action A_00006C becomes:

```
A_00006C ≡
  r10 := 112; ENDGROUP(); call dispatch end
```

FermaT determines that the value in r10 will be copied into destination by the body of ENDGROUP, so this call dispatch can be replaced by call LAB.

The control flow and data flow restructuring transformations are iterated until no further improvement is possible. The result is typically a dramatic improvement in all the metrics, for our sample program Figure 5 compares the before and after values of the metrics. This order of magnitude improvement in most of the metrics is typical for all sizes of assembler module.

A fundamental attribute of the FermaT workbench is that its transformations are all mathematically proven to preserve the semantics of the subject program. The programmer can be confident that the WSL program after transformation is functionally equivalent to its original form. Redundant code and variables can safely be removed, "spaghetti" code can be straightened out, and the program simplified and its maintainability improved. Given the large number of transformations applied in the migration process (typically in the hundreds if not thousands), confidence in the correctness of each transformation is essential.

7 WSL to C Translation

The final step is to generate C code from the structured WSL. This may involve further transformations to eliminate WSL features which cannot be directly implemented in C, or to meet customers' requirements on the C layout or features used. For example, some customers dislike `break` statements. These are introduced to implement the exit from the middle of a do ... od loop. If they are not required, then a transformation can be applied to transform the do ... od loop to an equivalent while loop, if necessary with an associated flag. These transformations also implement assignments and conditions as `memmove` and `memcmp` calls where necessary and converted some function calls to procedure calls which return the result via a pointer passed as a parameter.

See Section 13 for the C code of our example. It should be emphasized at this point that the C code in Section 13 was generated directly from the assembler, with no manual intervention required.

Note that two procedures (void functions) have been extracted: `endgroup_p` and `writeone_p`. Their names are derived from the original assembler labels. FermaT has been able to restructure the code so that there is only one place where the flag `xsw1` is set (which represents the flag SW2 in the pseudocode), and one place where the code to process a record is called. Hence there was no need to create a C function for the assembler subroutine `PROCGRP`. Note that the dispatch routine has been eliminated, as has the cc variable. All branches have been eliminated and replaced by structured code. Many register operations (such as saving and restoring return addresses) and other redundant operations have been removed.

Other features of the generated C code which are important for maintainability:

- `goto`s are eliminated where possible, without increasing the complexity of the code. If required by the customer, any remaining `goto`s can also be eliminated automatically by introducing extra variables or function calls.
- Data is accessed by simple operations (with casts added where necessary), more complex data operations are implemented as `memmove`/`memcmp` calls.
- C control structures are used wherever appropriate (`if` statements, `while` and `repeat` loops etc.)
- The module is automatically restructured into a collection of procedures, where each procedure has a single entry point and a single exit point. Each procedure always returns to the call site: non-standard termination of a procedure is indicated by setting an exit_flag variable.
- A "translation report" is generated for each module listing metrics for each function, dead code candidates, variables which are accessed outside their declared size (these accesses may fail to work if the data structures are reordered), and other useful information.
- `EXEC CICS` calls are translated into the appropriate C code.
- Appropriate C code can be generated for user and system macros (in the case where the system does not simply translate the macro expansion)
- Pointer addressing, casting and dereferencing operators are added automatically: this ensures that the C code will compile correctly with no errors or warnings. For example, if we add 3 to register 4 and load the fullword stored at that address, the C code will read: `*(FWORD *)(regs.r4 + 3)`, i.e. add 3 to register 4, cast the result to a fullword pointer, and dereference it. On the other hand, if `savearea` is declared as an array of fullwords and the assembler loads bytes 9 to 12 inclusive of `savearea` into register 5 then the C code will read `regs.r5 = savearea[2]`.
- Fields in a DSECT are accessed via the appropriate DSECT pointer, eg `foo->bar.baz`

8 A Mass Migration Exercise

As an experiment to test the scalability of the transformation approach to industrial code we selected at random, 1,925 assembler listing files for a mass migration exercise.

Apart from a handful of test files, these are all live code taken from more than a dozen large commercial assembler systems, mostly from large financial institutions. The files contained containing a total of 5,884,620 lines, of which 3,090,548 were source, copybook or macro expansion lines and the remainder were page and file headers and cross reference tables.

The files were processed automatically by a control program which used two 200MHz Sparc processors to translate each module to WSL, apply transformations to the WSL code and translate the WSL to C. The experiment completed successfully after 4 days and 18 hours of elapsed time. *Every* module was successfully migrated to C, and *every* generated C file compiled with no warnings or errors. We did not, however, manually check the semantics of each generated file against the original assembler!

The generated C files would still require some work regarding file handling etc., depending on whether the customer wanted to migrate to a different environment. Much of this work can be automated. In addition, the user needs to check for `FIXME` comments in the generated C code which indicate areas where the translated code may be incorrect (for example, an EXecute instruction where FermaT cannot determine at compile time which instruction will be executed).

The overall performance was about 600 KLOC/day per CPU, or about 7 minutes CPU time per assembler module. Note however that processing times vary widely, depending on the file contents. Short files and files consisting mostly of data declarations can take less than a minute each, while larger files with lots of executable code can take an hour or more. In our case, the times ranged from 2 seconds to 20,473 seconds (5 hours 41 minutes) with an average of 398 seconds (6 minutes 40 seconds).

A total of 1,132,278 lines of C code were generated, of which 179,138 lines are data initialisation code, plus a further 1,232,156 lines of header files in 7,793 C header files. Each assembler module generated a header file for local data plus header files for each of the DSECTs it referenced.

9 Conclusion

This work clearly show that Assembler to C migration using the FermaT Workbench is a practical solution to the high costs and skills shortage in assembler maintenance and to the problem of migrating legacy systems away from the mainframe environment.

10 References

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

[2] F. L. Bauer, B. Moller, H. Partsch & P. Pepper, "Formal Construction by Transformation—Computer Aided Intuition Guided Programming," *IEEE Trans. Software Eng.* 15 (Feb., 1989).

[3] G. D. Bergland, "A Guided Tour of Program Design Methodologies," *Computer* 14 (Oct., 1981), 18–37.

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

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

[6] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[7] H. A. Partsch, *Specification and Transformation of Programs*, Springer-Verlag, New York–Heidelberg–Berlin, 1990.

[8] 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`⟩.

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

[10] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/foundation2-t.ps.gz`⟩.

[11] 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`⟩.

[12] M. Ward, "Program Analysis by Formal Transformation," *Comput. J.* 39 (1996).

[13] 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`⟩.

[14] 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`⟩.

[15] 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`⟩.

[16] 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`⟩.

11 The Assembler Source

```
****************************************
*  TST004A0  SAMPLE PROGRAM (MCDONALDS) *
****************************************
*
         REGEQU
*
*        PRINT NOGEN
TST004A0 CSECT
         STM   R14,R12,12(R13)
         LR    R3,R15
         USING TST004A0,R3
         ST    R13,WSAVE+4
         LA    R14,WSAVE
         ST    R14,8(R13)
         LA    R13,WSAVE
*
         OPEN  (DDIN,(INPUT))
         OPEN  (RDSOUT,(OUTPUT))
*
         MVC   WPRT(17),=CL17'MANAGEMENT REPORT'
         BAL   R10,WRITE1
         BAL   R10,WRITE1
         MVC   WPRT(20),=CL20'ITEM      NET CHANGE'
         BAL   R10,WRITE1
         BAL   R10,WRITE1
*
         MVI   XSW1,0
LAA      EQU   *
         GET   DDIN,WREC
         CLC   WRITEM,WLAST
         BE    LAC
LAAA     B     LAB
         BAL   R10,ENDGROUP
LAB      MVI   LAAA+1,0
         MVC   WLAST,WRITEM
         ZAP   WNET,=P'0'
         BAL   R10,PROCGRP
         MVI   XSW1,X'FF'
         B     LAA
LAC      BAL   R10,PROCGRP
         MVI   XSW1,X'FF'
         B     LAA
*
LAD      CLI   XSW1,X'FF'
         BNE   LADA
         BAL   R10,ENDGROUP
LADA     EQU   *
         MVC   WPRT(17),=CL17'NUMBER CHANGED = '
         ED    WORKB,WCHANGE
         LA    R4,WORKB
         LA    R1,9
LADB     CLI   0(R4),C' '
         BNE   LADC
         LA    R4,1(R4)
         BCT   R1,LADB
LADC     EX    R1,WMVC1
*WMVC1   MVC   WPRT+17(1),0(R4)
         BAL   R10,WRITE1
*
         CLOSE DDIN
         CLOSE RDSOUT
*
         L     R13,WSAVE+4
         LM    R14,R12,12(R13)
         SLR   R15,R15
         BR    R14
*
PROCGRP  EQU   *
         ST    R10,WST10A
         PACK  WORKA,WRQTY
         CLI   WRTYPE,C'R'
         BNE   LBA
         AP    WNET,WORKA
         B     LBB
LBA      SP    WNET,WORKA
LBB      L     R10,WST10A
         BR    R10
*
ENDGROUP EQU   *
         ST    R10,WST10A
         MVC   WPRT(4),WLAST
         MVI   WSIGN,C'+'
         CP    WNET,=P'0'
         BNL   LCA
         MVI   WSIGN,C'-'
LCA      EQU   *
         MVC   WPRT+7(10),=X'40206B2020206B202120'
         EDMK  WPRT+7(10),WNET
         BCTR  R1,0
         MVC   0(1,R1),WSIGN
         BAL   R10,WRITE1
         BAL   R10,WRITE1
         AP    WCHANGE,=P'1'
         L     R10,WST10A
         BR    R10
*
WRITE1   EQU   *
         PUT   RDSOUT,WPRT
         MVC   WPRT,WSPACES
         BR    R10
*
WMVC1    MVC   WPRT+17(1),0(R4)
*
WSAVE    DC    18F'0'
WST10A   DS    F
WREC     DS    0CL80
WRITEM   DS    CL4
         DS    CL1
WRTYPE   DS    CL1
         DS    CL1
WRQTY    DS    CL3
         DS    CL70
WPRT     DC    CL80' '
WSPACES  DC    CL80' '
WLAST    DC    CL4'****'
WCHANGE  DC    PL4'0'
WNET     DC    PL4'0'
WORKA    DC    PL2'0'
WORKB    DC    XL10'40206B2020206B202120'
WSIGN    DC    CL1' '
XSW1     DC    X'00'
*
         LTORG
*
DDIN     DCB   DDNAME=DDIN,
               DSORG=PS,
               EODAD=LAD,
               MACRF=GM
RDSOUT   DCB   DDNAME=RDSOUT,
               DSORG=PS,
               MACRF=PM
*
         END
```

12 The Generated C Header File

The C header and source files are as generated by the FermaT system, with no manual editing other than minor reformatting to fit the page size.

```
#include <assem.h>

/* EQUates table */

#define laa      78
#define lada     164
#define procgrp  242
#define endgroup 282
#define lca      310
#define write1   350

/*           --> CSECT: TST004A0 <-- */
static FWORD wsave[18] = {0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0};
static FWORD wst10a;
static struct { /* wrec */
  BYTE  writem[4];
  BYTE  UNNAMED001;
  BYTE  wrtype;
  BYTE  UNNAMED002;
  BYTE  wrqty[3];
  BYTE  UNNAMED003[70];
} wrec;
static BYTE  wprt[80] = "                \
                         ";
static BYTE  wspaces[80] = "             \
                         ";
static BYTE  wlast[4] = "****";
static DECIMAL(4, wchange) = "\x00\x00\x00\x0C";
static DECIMAL(4, wnet) = "\x00\x00\x00\x0C";
static DECIMAL(2, worka) = "\x00\x0C";
static BYTE  workb[10] = {0x40, 0x20, 0x6B, 0x20, 0x20,
0x20, 0x6B, 0x20, 0x21, 0x20};
static BYTE  wsign = ' ';
static BYTE  xsw1 = 0x00;
```

13 The Generated C Code

```
#include "tst004a0.h"

void endgroup_p();
void write1_p();

FILE *ddin;
FILE *rdsout;
FWORD f_laaa;

void
main()
{
  regs.r3 = regs.r15;
  wsave[1] = 0;
  regs.r14 = (FWORD) & wsave;
  regs.r13 = (FWORD) & wsave;
  OPEN(ddin, input);
  OPEN(rdsout, output);
  memmove(wprt, "MANAGEMENT REPORT", 17);
  write1_p();
  write1_p();
  memmove(wprt, "ITEM      NET CHANGE", 20);
  write1_p();
  write1_p();
  xsw1 = 0;
  for (;;) {                    /* DO loop 1 */
    regs.r0 = 0;
    regs.r1 = 0;
    regs.r15 = 0;
    GET(ddin, &regs.r0, &regs.r1, &regs.r15, &wrec);
    if (end_of_file(ddin)) {
      break;
    } else if ((*(FWORD *)wrec.writem == *(FWORD *)wlast
              || f_laaa != 1)) {
      if (*(FWORD *)wrec.writem == *(FWORD *)wlast) {
      } else {
        endgroup_p();
        f_laaa = 0;
        *(FWORD *)wlast = *(FWORD *)wrec.writem;
        zap(wnet, 4, "\x0C", 1);
      }
    } else {
      f_laaa = 0;
      *(FWORD *)wlast = *(FWORD *)wrec.writem;
      zap(wnet, 4, "\x0C", 1);
    }
    wst10a = regs.r10;
    pack(worka, 2, wrec.wrqty, 2);
    if (wrec.wrtype != 'R') {
      sp(wnet, 4, wnet, 4, worka, 2);
    } else {
      ap(wnet, 4, wnet, 4, worka, 2);
    }
    exit_flag = 0;
    xsw1 = 0xFF;
  }                             /* OD */
  if (xsw1 == 0xFF) {
    endgroup_p();
  }
  memmove(wprt, "NUMBER CHANGED = ", 17);
  ed(workb, 10, wchange, 10);
  regs.r4 = (FWORD)workb;
  regs.r1 = 9;
  for (;;) {                    /* DO loop 2 */
    if (*(BYTE *)regs.r4 != ' ') {
      break;
    }
    regs.r4++;
    regs.r1--;
    if (regs.r1 == 0) {
      break;
    }
  }                             /* OD */
  memmove((BYTE *)((FWORD)wprt + 17), (BYTE *)regs.r4,
    ((regs.r1 & 0xFF) + 1));
  write1_p();
  CLOSE(ddin);
  CLOSE(rdsout);
  return;
}

void
endgroup_p()
{
  wst10a = regs.r10;
  *(FWORD *)wprt = *(FWORD *)wlast;
  wsign = '+';
  if (dec_less(wnet, 4, "\x0C", 1)) {
    wsign = '-';
  }
  memmove((wprt + 7),
    "\x40\x20\x6B\x20\x20\x20\x6B\x20\x21\x20", 10);
  edmk((wprt + 7), 10, &regs.r1, wnet, 10);
  regs.r1--;
  *(BYTE *)regs.r1 = wsign;
  write1_p();
  write1_p();
  ap(wchange, 4, wchange, 4, "\x1C", 1);
  regs.r10 = wst10a;
  exit_flag = 0;
  return;
}

void
write1_p()
{
  PUT(rdsout, *wprt);
  memmove(wprt, wspaces, 80);
  exit_flag = 0;
  return;
}
```