Legacy Assembler Reengineering and Migration

M. P. Ward, H. Zedan and T. Hardcastle
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
Martin.Ward@durham.ac.uk and {zedan,timh}@dmu.ac.uk

Abstract

In this paper we describe the legacy assembler problem and describe how the FermaT transformation system is used to reengineer assembler systems and migrate from assembler to C and COBOL.

1. Legacy Assembler

A legacy is "something handed down or received from an ancestor or predecessor". A legacy assembler system may be defined as any system which:

1. is implemented in assembler, or has substantial components implemented in assembler;
2. was developed some years ago and has been handed down to the current maintainers;
3. is nevertheless essential to the smooth running of the organisation.

There are estimated to be over 250 million lines of IBM 370 assembler currently in operation at the major IBM mainframe sites in Europe alone, but there is a decreasing pool of experienced assembler programmers. As a result, there is increasing pressure to move away from assembler, including pressure to move less critical systems away from the mainframe platform, so the legacy assembler problem is likely to become increasingly severe.

There are a number of options for dealing with legacy assembler systems:

1. Keep the current application and modify it as needed;
2. Discard the application and replace it with a purchased vendor package;
3. Understand and document the application and then rewrite it in a high level language (HLL);
4. Use automated migration technology to migrate to a HLL.

Each of these options may be suitable in some cases, but also presents problems, which are discussed in the following sections.

1.1. Keep the Current Application

If the application is relatively stable with few enhancement requests and bug fixes, then keeping the current application may be a viable option. If there is a shortage of assembler expertise, then a "wrap and freeze" approach may work: wrap a modern interface around the old application and work around any bugs or limitations, rather than attempting to fix the code. Alternatively, if assembler expertise is still readily available and enhancement requirements are moderate, then it may be possible to modernise the application and continue to use it.

"Do nothing" is always an attractive option in the short term: the short term cost of doing nothing is very low! But in the face of a worsening skills shortage and increasing pressure to modernise the application, or to migrate to a more cost-effective platform, then doing nothing may not be a viable long-term option.

1.2. Replace with a Vendor Package

Brooks [2] comments that "The most radical possible solution for constructing software is not to construct it at all....". The option of replacing a legacy application with a package would appear to solve the maintenance problem at a stroke: but in reality, the closest matching vendor package is still likely to have many differences compared to the existing application [3]. The organisation will need to decide how much the business needs to change to fit the package, and how much the package will need to be tailored to fit the business. More importantly, the differences between *your* application and the package used by everyone else in the industry may be what gives you a competitive edge: it would be foolish to throw away this advantage unless you are forced to do so: one major package tour operator has rejected the "replace with a package" option for precisely this reason. The tour operator believed that their hand-crafted assembler system gave them a substantial edge in the marketplace: why give this up in order to run the same package used by every other tour operator in the business?

1.3. Rewrite in a HLL

One of the biggest problems with many legacy systems is that the code structure no longer matches the program logic: after many patches and fixes, the original program design is inadequate and unsuitable for current requirements. The business rules may be complex, but the program logic complexity may be far greater than is necessary to implement the business rules.

As a result, it is difficult to perform major enhancements: the result is either a huge backlog of change requests or, more probably, major enhancements are not even contemplated. If there is a shortage of assembler expertise, then it may not be possible to deal with this backlog, even if there are programming resources available. A major driver behind the Tenovis migration project (see below) was to enable the desired enhancements to the system to be made, despite the shortage of assembler programmers.

Assembler systems are also more expensive to maintain than equivalent systems written in high level languages. Caper Jones Research computed the annual cost per function point as follows:

Assembler	£48.00
PL/1	£39.00
C	£21.00
COBOL	£17.00

One solution is therefore to rewrite the assembler system in a high level language for which programming expertise is readily available. The design for the new system should be extracted by analysis of the existing assembler since this contains the only complete and reliable description of the business rules embedded in the code. Automated tools to assist with the analysis of the assembler are therefore very important.

Another major driver for rewriting or migrating to a HLL is the need to move less critical systems from the expensive mainframe platform to cheaper PC or Unix hardware. This will free up resources on the mainframe and eliminate the need for expensive upgrades.

1.4. Use Automated Migration Technology

In a survey of potential solutions to the legacy assembler problem, Ehrman [3] dismisses the option of automated migration, claiming that the technology is insufficiently well-developed to meet the huge challenge of migrating from assembler to maintainable and efficient high level language code. Until recently, most automated migration offerings have taken the form of a "brute force" conversion, which simply maps each assembler instruction to corresponding code in the HLL. Registers, flags and storage are mapped into HLL variables that mimic the original machines architecture and instructions and each assembler instruction is translated directly to the equivalent HLL code. Brute force translations:

- Are typically seven times slower than the assembler [4];
- Result in highly obscure code: the complexity of the original program is preserved with complexities of the translation added;
- Are difficult to modify and maintain: the maintainers will probably need to be familiar with both the original source language and the target language in order to make sense of the code;
- Do not make use of the facilities of the target language which improve maintainability and performance (structuring concepts, subroutine libraries etc.)

Brute force translations may work for small applications with limited maintenance requirements where it is essential to migrate to a different platform. But even in this situation, performance and maintainability of the code may be little better than can be achieved by running the original code under an emulator.

The FermaT migration technology takes a different approach: the assembler instructions are translated into an intermediate language (a wide-spectrum language called WSL) which is uniquely suitable for applying program transformations. Many thousands of WSL to WSL transformations are applied to restructure and simplify the program and raise the level of abstraction. The restructured WSL is than translated into maintainable and readable HLL code. In the next section we discuss some of the challenges faced by a practical migration technology, and the rest of the paper will describe FermaT's solution in more detail.

2. The Challenge for Automated Migration

The technical difficulty of automated migration should not be underestimated. Translating assembler instructions to the corresponding HLL code, and even unscrambling spaghetti code caused by the use of labels and branches, is only a very small part of the translation task. Other technical problems include:

- Register operations: registers are used extensively in assembler programs for intermediate data, pointers, return addresses and so on. The migrated code should eliminate the use of registers where possible;
- Condition codes: test instructions set a condition code or flags which can then be tested by conditional branch instructions. These need to be combined into structured branching statements such as **if** statements or **while** loops: note that the condition code may be tested more than once, perhaps at some distance from the instruction which sets it. So it is not sufficient simply to look for a compare instruction followed by a conditional branch;
- Subroutine call and return: in IBM 370 assembler a subroutine call is implemented as a BAL (Branch And Link) instruction which stores the return address in a register and branches to the subroutine entry point (there is no hardware stack). To return from the subroutine the program branches to the address in the register via a BR (Branch to Register) instruction. Return addresses may be saved and restored in various places, loaded into a different register, overwritten, or simply ignored. Also, a return address may be incremented (to branch over parameter data which appears after the BAL instruction). Merely determining which instructions form the body of the subroutine can be a major analysis task: there is nothing to stop the programmer from branching from the middle of one subroutine to the middle of another routine, for example;
- The 370 instruction set includes an EX (EXecute) instruction which takes a register number and the address of another instruction. The referenced instruction is loaded and then modified by the value in the register, and then the modified instruction is executed. This can be used to implement a "variable length move" instruction, by modifying the length field of a "move characters" instruction, but any instruction can be EXecuted. EXecuting another EX instruction causes an ABEND: some programmers do "EX R0,*" (where the instruction executes itself) precisely to achieve an ABEND: so the translator has to take this into account also;
- Jump tables: these are typically a branch to a computed address which is followed by a table of unconditional branch instructions. The effect is a multi-way branch, similar to the "computed GOTO" in FORTRAN. There are many ways to implement a jump table in assembler: typically the branch into the table will be a "branch to register" instruction which must be distinguished from a "branch to register" used as a subroutine return;
- Self-modifying code: a common idiom is to implement a "first time through switch" by modifying a NOP instruction into an unconditional branch, or modifying an unconditional branch into a NOP. Less commonly a conditional branch can be modified or created. More general self-modifying code (such as dynamically creating a block of code and then executing it) is rare in 370 assembler systems;
- System macros: the macro expansion for a system macro typically stores values in a few registers and then either executes an SVC call or branches to the operating system. It does not make sense to translate the macro expansion to HLL, so the macros should be detected and translated separately. Some macros may cause "unstructured" transfer of control: for example the system GET macro (which reads a record from a file) will branch to a label on reaching the end of the file. The end of file label is not listed in the macro, but in the DCB (Data Control Block) which itself may only be indirectly indicated in the GET macro line. The DCB itself may refer to a DCBE macro which records the EODAD (end of file) address label;
- User macros: users typically write their own macros, and these may include customised versions of system macros. The translation technology needs to be highly customisable to cope with these and to decide in each case whether to translate the macro directly, or translate the macro expansion;
- Structured macros: in the case of so-called "structured macros" (IF, WHILE etc.) it is best simply to translate the macro expansion because there are no restrictions on using structured macros in unstructured ways;
- Data translation: all the assembler data declarations need to be translated to suitable HLL data declarations. Assembler imposes no restrictions on data types: a four byte quantity can be used interchangeably as an integer, a floating point number, a pointer, an array of four characters, or 32 separate one-bit flags. Ideally, the HLL data should be laid out in memory in the same was as the assembler data: so that accessing one data element via an offset from the address of another data element will work correctly. Reorganising the data layout (if required) is a separate step that should be carried out *after* migration, rather than attempting to combine two complex operations (migration and data reorganisation) into a single process. Symbolic data names and values should be preserved where possible, for example:

```
RECLEN   EQU *-RECSTART
```

should translate to code which defines `RECLLEN` symbolically in terms of `RECSTART`;

- Pointers: these are used extensively in many assembler programs. If the HLL is C then pointers and pointer arithmetic is available: for COBOL it is still possible to emulate the effect of pointer arithmetic, but the code is less intuitive and less familiar to many COBOL programmers;
- Memory addressing: DSECT data in a 370 assembler program is accessed through a base register which contains the address of the start of the block of data. If the base register is modified, then the same symbolic data name will now refer to a different memory location;
- Packed Decimal Data: 370 assembler (and COBOL also) have native support for packed decimal data types. IBM's mainframe C compiler also supports packed decimal data, but if the migration is to a different platform then either the data will need to be translated, or the packed decimal operations will have to be emulated;
- Pointer lengths may be different in the source and target languages;
- "Endianness": when migrating to different hardware platforms, the two systems might store multi-byte integers in different orders (most significant byte first vs least significant byte first).

2.1. Embedded Systems

A major application for assembler code is in embedded systems. Many embedded systems were developed for processors with limited memory and processing capability, and were therefore implemented in tightly coded hand written assembler. Modern processors are now available at a lower cost which have much more processing and memory capacity and with efficient C compilers. To make use of these new processors the embedded system needs to be re-implemented in a high level language in order to reduce maintenance costs and enable implementation of major enhancements. Many of the challenges with 370 assembler (such as the EXecute instruction and self-modifying code) are not relevant to embedded systems processors, but other challenges become important (such as 16 bit addresses and 8 or 16 bit registers).

3. Our Approach

Over the last 20 years the authors have developed a wide spectrum language (WSL) and transformation theory which has formed the basis for a successful assembler analysis and migration system. WSL and the transformation theory has been discussed in other papers [6,7,11] so will not be described in detail here. See Chapter 3 of [12] for a recent description of WSL.

The translation of an assembler modules is carried out in four phases:

1. Translation of the assembler to WSL;
2. Data restructuring and translation;
3. WSL to WSL transformations;
4. Translation of the WSL to HLL.

These phases are described in the following sections.

3.1. Assembler to WSL Translation

The assembler to WSL translator is designed to translate an assembler module to WSL as accurately as possible: capturing every detail of the behaviour of the system without worrying about the size, efficiency or complexity of the resulting code. This is because it is anticipated that phase 3 (WSL to WSL translation) will remove inefficiencies and redundant operations. As a result, we can separate the two, apparently conflicting, requirements of correctness and efficiency/maintainability into separate phases, and therefore meet both requirements.

Perfect correctness is not possible: any scientific model of something *must* be an imperfect representation(i.e. an approximation): otherwise it would not be a model, but the thing itself! Our approach therefore is to model as accurately as possible everything that *can* be modelled, and to detect and flag cases that cannot be modelled.

The translator is table-driven: each assembler instruction or macro is listed in a configuration table, along with its WSL translation. If a macro is listed in the table then the corresponding WSL code is generated and the macro expansion is skipped, otherwise the macro expansion is translated. The translator works from an assembly listing with all macros expanded, so it has available the offset address of each instruction and data element, the object code generated, and the full expansion of each macro.

The translator parses the listing into an internal data format and then makes several passes over the data to determine jump tables, all the possible targets for branch to register instructions, relative branch targets, self-modifying code, CICS calls, DSECT names and so on. A final pass over the internal data generates a WSL file and a data file (which lists all data declarations).

3.2. Data Restructuring and Translation

A separate process parses the data file and restructures the data declarations into nested structures. IBM assembler offers several ways to create overlapping data structures: declaring a symbol with a type and length but with a "repeat count" of zero will not allocate any data, so subsequent data declarations will overlap the symbol. An ORG directive can be used to redefine the same area of memory with two or more different layouts.

The data restructuring process analyses the length, repeat count, offset and type of each symbol to determine the nesting of data structures. Where structures cannot be properly nested it generates unions of structures.

C or COBOL data declarations are generated from the restructured data file: "filler" data items are inserted where necessary to ensure that the layout of the C or COBOL data is identical to the original assembler data. This is important for two reasons:

1. If the migrated code is to be executed on the mainframe, then it may be necessary to share data structures with existing assembler or HLL programs: it will be essential to ensure that the data layouts match in this case;
2. Even if the intention is to migrate to a different platform, the code may expect a certain layout of data and may fail if the data is reorganised. For example, an offset from one data element may be used to access a different element, or pointers may be moved around in the data structures. The migration process can detect and report on places where data items are accessed outside their declared length: this report will indicate potential failure points if the data were to be reorganised.

3.3. WSL to WSL Transformation

This is the heart of the transformation system: a large number of correctness-preserving transformations are applied to the WSL code in order to remove redundant statements, restructure the code, determine procedure boundaries and so on.

The transformation engine is based on the WSL transformation theory which provides methods to prove the correctness of a WSL to WSL transformation. As a result we can have a very high degree of confidence in the accuracy of the results, despite the fact that an average of over 4,000 transformations are applied to *each* assembler module during the migration process. If the average transformation were "only" 99.9% accurate, then after applying 4,000 transformations the probability of a correct result falls to less than 2%. So it is vitally important to be confident of the correctness of each transformation step.

Most transformations fit into one of the two main classes:

1. Small, localised transformations. These are applied to a localised region of the program to make a small improvement: such as merging two **if** statements, removing a single register reference or removing a local variable. Repeated application of localised transformations over the whole program can have a dramatic effect on the structure, efficiency and maintainability of the program.
2. Whole-program analysis transformations: these include Constant Propagation, Dead Code Removal, Data Translation and transformations based on constructing the Static Single Assignment (SSA) form of the program. See Section 4 for details on the construction and applications of SSA form.

A control program selects the order of executing the transformations: this control program is simply another transformation (Fix Assembler) which invokes other transformations via calls to @Trans? and @Trans. The function @Trans? test whether the give transformation is applicable at the current position in the current program. The procedure @Trans applies the given transformation to the current program at the current position. Typically, the localised transformations are iterated over every applicable position in the syntax tree of the WSL program: this is easily achieved with the **foreach** and **ateach** looping constructs of $\mathcal{METAWSL}$. (See [8] for a description of $\mathcal{META}$WSL and these constructs). $\mathcal{META}$WSL is an extension of WSL specifically designed for writing program transformations. An example of such an iteration is the following code from Fix Assembler:

```
foreach Statement do
  if @ST(@I) = T_Cond ∧ @Size(@I) ⩽ 20
    then if @Trans?(TR_Join_All_Cases)
           then PRINFLUSH("+");
                @Trans(TR_Join_All_Cases, "")
         fi fi od;
```

Within the **foreach** loop the function @I returns the currently selected item (in this case, the currently selected statement). Function @ST returns the *specific type* of its argument, so if the current item is an **if** statement, then @ST($@I$) returns T_Cond. @Size(@I) is the number of components of the current item: for an **if** statement this is the number of branches. So this loop will test and apply the transformation TR_Join_All_Cases to every **if** statement which has no more than 20 branches. It prints a "+" each time a transformation is applied. Originally, such messages were necessary to inform the user that the system was still running and applying transformations (and had not got stuck in a loop). With the dramatic improvements in CPU speeds and improvements in the efficiency of FermaT, these messages are not really necessary: in fact, if printed to the screen they scroll up far too fast to read!

3.3.1 Preventing Loops

There are two cases where this strategy can cause looping: both of which must be avoided:

1. A transformation could make the WSL program larger. Later on (perhaps as a result of other transformations) the same transformation could become applicable to a component of the expanded program. The result is that the program size grows indefinitely without converging to a solution;
2. A transformation could "undo" the effect of a previous transformation: the program could then oscillate between two different versions, again without converging to a solution.

Broadly speaking, our solution to both of these problems is to only apply a transformation when it makes the program "better" according to some complexity metric. If the complexity metric is integer valued and every program reduces complexity (according to the metric), then the transformation process is guaranteed to converge.

Unfortunately, there is currently no universal integer valued complexity metric which is monotonically reduced under every transformation application (this is a topic for further research). However, the vast majority of transformations do reduce the *size* of the program: overall, the WSL to WSL transformation step reduces the size to between a third to a half of the original. A few transformations do increase the size of the program, but the result is "obviously" an improvement (in the opinion of the developers anyway!) and the migration engine is prevented from applying the inverse of these transformations.

As a result of these precautions, the only infinite loops we have seen have been the result of bugs in the implementation of a transformation.

3.3.2 Dataflow Analysis

The raw WSL (before transformation) accurately models the control flow of the original assembler but only allows a very crude dataflow analysis since it contains control flow paths from the end of *every* assembler subroutine (ie from every Branch to Register instruction) to *every* possible return point from a subroutine. But an accurate dataflow analysis is required in order to extract WSL procedures from the unstructured code: to decide if a particular set of WSL actions can be used to form a WSL procedure body FermaT needs to determine that the return address passed in via a register at the top of the subroutine is preserved through the subroutine body (although it may be incremented) and is finally passed to the dispatch action in the destination variable. If this is the case, then FermaT can create a WSL procedure from the set of actions, and convert the action calls to procedure calls and remove some dispatch calls.

For example, suppose we have the following assembler code:

```
A1  BAL R12,FOO     SUBROUTINE CALL
A2  ...
FOO ST  R12,SAVER12 SAVE RETURN ADDRESS
    LA  R12,0       RE-USE R12
    ...
    L   R13,SAVER12 RESTORE RETURN ADDR
    BR  B13         RETURN FROM SUBR
```

This translates to the following WSL code:

```
A1  ≡ r12 := $234$; **call** FOO **end**
A2  ≡ ... **end**
FOO ≡ SAVER12 := r12;
      r12 := $0$;
      ...
      r13 := SAVER12;
      destination := r13;
      **call** dispatch **end**
...
dispatch ≡ **if** destination $= 0$ **then call** Z
             ...
           **elsif** destination $= 234$ **then call** A2
           ... **fi end**
```

Here, the return address stored in r12 (the address of the label A2) is represented as the "dispatch code" 234. This is calculated as the decimal offset of label A2 from the start of the program (the offset of each instruction is given in the assembler listing).

If the dataflow analysis on the body of FOO is successful, then the Fix_Dispatch transformation will transform the code into this:

```
**begin**
  A1 ≡ FOO(); **call** A2 **end**
  A2 ≡ ... **end**
  ...
  dispatch ≡ **if** destination $= 0$ **then call** Z
               ... **fi end**
**where**
**proc** FOO() ≡ SAVER12 := r12;
  r12 := $0$;
  ...
  r13 := SAVER12 **end**
**end**
```

We have created a new procedure FOO, removed the FOO action from the action system and removed a control flow link from the dispatch action. The result is a simplified control flow, from which a more accurate dataflow analysis can be constructed.

FermaT does not depend on a single, initial, dataflow analysis but iteratively improves the dataflow analysis as the control structure is improved: the simplified control structure makes possible a more accurate dataflow analysis which, in turn, leads to further simplifications in the control structure.

The first iteration can process assembler subroutines which call no other (internal) subroutines (external calls are handled separately: FermaT recognises when a return address is passed to an external routine and assumes that control will return via that return address). In the next iteration, FermaT can process subroutines which only call subroutines processed in the first iteration, and so on.

Many assembler subroutines include tests for error conditions which will branch to an error routine, leading ultimately to an ABEND instruction instead of returning to the caller. A WSL procedure, on the other hand, must always return to the caller. The solution is to set a special flag variable exit_flag and then return. If $\text{exit\_flag} = 0$ then the return was a normal return, if $\text{exit\_flag} = 1$ then the program must immediately ABEND.

3.4. WSL to HLL Translation

As with the assembler to WSL translator, our aim is for the WSL to C and WSL to COBOL translators to be a simple as possible: all the work should be done by WSL to WSL translators with the final WSL being as close as possible to the target language. To achieve this, each target language translator includes a series of language-specific transformations: for example, assignments which move more than four bytes (or exactly three bytes) will need to be converted to memmove calls before translating to C. On the other hand, before migrating to COBOL, functions need to be converted to procedures and pointer operations will need converting to code which uses the SET ADDRESS OF feature.

The WSL to COBOL translator is also table-driven, this is mainly to allow users to customise the COBOL code generated for their customer-specific assembler macros. The translation table is also used for IBM system macros. So, for example, the WSL statement:

!P GET(FILE1 VAR os, r_0, r_1, r_{15}, BUF1)

is translated to the COBOL code:

```
READ FILE1 INTO BUF1
  AT END MOVE 1 TO EOF-FLAG
END-READ
```

The WSL to HLL translators also generate a migration report for each module, listing the following information:

- Dead Code Candidates: procedures which do not appear to be called anywhere;
- Memory Overflows: variables which are accessed via an offset or length which exceeds their defined length. These will need special attention if the memory layout is changed;
- Metrics for each extracted procedure;
- Variables referenced and updated;
- EXecute statements and their targets (this information is used by the dead code marking routine to avoid marking EXecuted statements as dead code);
- Dead Code Candidate Line Numbers: also used by the dead code marking routine.

4. Assembler Analysis and Reengineering

In cases where automated migration is *not* the preferred solution to a legacy assembler problem, each of the other solutions (keep the current application, replace with a package or rewrite manually) will require a detailed analysis of the assembler.

The starting point for assembler analysis is the restructured WSL file since this will provide a much more accurate dataflow analysis. The raw WSL contains links back to the assembler listing in the form of FermaT comments containing a line number and a list of the symbols which appear on that line of the listing. These comments are carried through the transformation process.

The next phase is to compute a “basic blocks” file from the WSL. The WSL program is divided into blocks of code: each block has a single entry point at the top, one or more exit points at the bottom and no more than one assignment to any variable. The blocks are combined into a control flow graph (CFG) which includes links to the WSL program and the assembler listing.

There are unique start and end nodes: every node in the CFG is reachable from the start, and the end node is reachable from every other node in the CFG. From the CFG we compute control dependencies and control dependence equivalences in linear time using the algorithm in [5]. The control dependencies of node A are all the nodes which *must* be executed from one exit out of A but which *may* be avoided by another path out of A. In other words, B is a control dependency of A if every path from one exit of A to the end node must pass through B, while there exists a path from another exit of A to the end node which avoids B. The control variables in A are therefore important in determining whether B is executed or not.

The nodes in the CFG can be partitioned into control dependence equivalence classes: all the nodes in each partition have the same control dependencies. This means that the set of all control dependencies can be represented in a form which is linear in the size of the program.

We compute the Static Single Assignment (SSA) form of the CFG using the near-linear time algorithm in [1]. The SSA form renames all variables so that each assignment is to a unique variable. At places where control flow paths join, "phi functions" are inserted to combine the dataflows from different variables which were originally the same variable. For example, in the program:

if $z = 0$ **then** $x := 1$ **else** $x := 2$ **fi**;
$y := x$

the two assignments to x are renamed to x_1 and x_2 respectively. But then what name is used for the reference to x in the assignment to y? The solution is to insert a phi function:

if $z = 0$ **then** $x_1 := 1$ **else** $x_2 := 2$ **fi**;
$x_3 := \phi(x_1, x_2)$;
$y := x_3$

The SSA form represents all the dataflows in the program in a concise format: typically linear in the size of the program, but quadratic in the worst case. However, the worst case only occurs when the entire program is a deeply-nested loop with a large number of variables modified in the innermost loop. This is very unlikely to arise in practice as the FermaT transformations attempt to reduce the level of nesting of loops where possible.

All the control dependence and data dependence information is then written out in a concise form which relates the dependencies back to the assembler program. From this dataflow (df) file we can compute forwards and backwards slices directly on the assembler source file.

The following shows an example of a backwards slice on the symbol `NDATE` at line 452:

```
GPASS    EQU   *                                     189
*        <FERMAT><SB><1><==========================> 190
         MVI   PASOPT,C'N'       DEFAULT NO PASSWORD 190
         ICM   R1,B'1111',IKJPW         SOURCE FIELD 191
*        <FERMAT><EB><1><==========================> 191
         BZ    GRESU               NOT PRESENT, CONT 192
...
         RACROUTE REQUEST=EXTRACT,WORKA=(9),       X 404
               TYPE=ENCRYPT,ENCRYPT=((6),INST),...   405
*        <FERMAT><SB><1><==========================> 428
         LTR   R15,R15           TEST RACX           428
*        <FERMAT><EB><1><==========================> 428
         BNZ   ENDBRAC           BAD                 429
         MVC   RXPWNEWP,PASSWORK MOVE BACK AGAIN     430
*                                                    431
* PERFORM UPDATE ON LOCAL SYSTEM WITH ICHEINTY.      432
*                                                    433
         LA    R4,RXPWUSER       ADDR USERNAME       434
         LA    R5,8              MAX LENGTH          435
         LA    R6,0              COUNTER             436
LOOPU    EQU   *                                     437
         CLI   0(R4),C' '        END YET?            438
         BE    ENDU              YES                 439
         LA    R4,1(R4)          UP PTR              440
         LA    R6,1(R6)          UP COUNTER          441
         BCT   R5,LOOPU          LOOP IF NOT END     442
ENDU     EQU   *                                     443
         STC   R6,NUSER          SET LENGTH OF UID   444
         MVC   NUSER+1(8),RXPWUSER SET VALUE OF UID  445
*                                                    446
* UPDATE PASSWORD IF SPECIFIED                       447
*                                                    448
*        <FERMAT><SB><1><==========================> 449
         CLI   PASOPT,C'Y'       PASSWORD OPERAND    449
*        <FERMAT><EB><1><==========================> 449
         BNE   RESREV            NO                  450
*                                                    451
         MVC   NDATE,=XL4'0000000F' ZEROS (PACK DEC) 452
```

Note that `NDATE` is being initialised with a constant value: so there are no direct dataflows to this point in the program. There are three control dependencies in the extract: the conditional branches on lines 191/192, 428/429 and 449/450. Note that the branches on lines 438/439 and 442 are *not* control dependencies because the two control paths from the branches rejoin before reaching line 452.

The control dependency on line 449 has a data dependency to line 190 (`PASOPT` is assigned on line 190 and tested on line 449 which is a control dependency for line 452.

With this simple program it is fairly easy to determine all the dependencies manually: but a typical legacy assembler system involves a large amount of "spaghetti" code with complex control flow, extended dataflows (where a reference to a data item is a considerable distance from the corresponding assignment) and complex control dependencies. In these cases, automated analysis is essential.

4.0.1 Reengineering to an Object Oriented System

The first step in the analysis phase of any reengineering project is to determine the top level structure of the system: i.e. the set of programs executed and the data files they operate on. The order in which programs are invoked is determined by the operator instructions and the JCL (Job Control Language) files. FermaT includes a sophisticated JCL parser which processes all the JCL files to determine:

- The linkage between logical file names and physical file names at each program invocation;
- The order in which programs are invoked

The next step is to determine the major inputs and outputs for each module. The individual modules are then restructured and analysed. For each output a backwards slice is computed: this slice contains all the code needed to compute this output of the module. Overlapping slices can be factored out into shared subroutines. The resulting analysis can be used to develop an object structure for the reengineered program, and to implement the methods for each object.

5. Organisation of a Migration Project

A typical migration project is carried out in two phases: the first phase is a pilot migration of a sample of code. The sample should contain about 10% of the total lines of code in the system and should include examples of all the kinds of code and programming techniques used in the system. It should ideally be a self-contained subsystem which can be tested in isolation (or with a minimal development of test harnesses). The purpose of phase one is to allow rapid turnaround of the code and rapid customisation of the source and target translators and the transformation process.

A recent migration project involved translating over half a million lines of 186 assembler to high-level, structured, maintainable C code, suitable for porting to a more modern processor and also suitable for implementing a backlog of enhancements. The migrated system is a PBX (Private Branch eXchange) running on four different hardware platforms and installed in sites spread across 18 countries. The system contains about 800,000 lines of C code, and 544,000 lines of 186 assembler: with the assembler split over 318 source files.

For this project, a 186 to WSL translator needed to be developed from scratch. An initial case study involved migrating a single 3,000 assembler source file to C. A simple x86 to WSL translator was developed and a standard set of restructuring transformations were applied: these were initially developed for IBM assembler to C migration but required only slight changes to cope with the WSL generated from x86 code. The resulting structured WSL code was translated to C using an existing WSL to C translator. This translator was developed for IBM assembler to C migration [9,10] but again, only needed slight modifications to cope with the different register and flag names.

The next stage in the project was the migration of a subset of the system, which formed a "mini" call control. This consisted of 67,000 lines of assembler in 41 source files.

A number of additional, largely customer-specific, requirements were identified at this stage. The x86 to WSL translator was completely rewritten and made table-driven. The WSL to C translator was also rewritten (in about five man days) since the common code was determined to be only a small part of the translator.

Altogether there were five iterations of the mini call control code with the customer examining the code after each iteration and giving feedback. After implementing various customer-specific enhancements to the translators and transformations, and making various changes to the style of the generated C code, the code from the fifth iteration was compiled and installed on the hardware where it performed flawlessly.

The final stage was to migrate the entire system. Only three iterations (out of the planned five) were required to iron out any remaining issues which were not exposed by the mini call control migration.

The final migration process was carried out on a 2.6GHz PC with 512Mb of RAM. All 318 source files were processed successfully in 1 hour, 27 minutes of CPU time (1 hour 30 minutes elapsed time) for an average of 16.5 seconds per file. A total of 1,436,031 transformations were applied, averaging 4,500 transformations per file and 275 transformations per second.

Prior to the project the customer had received a quote for a complete manual translation of the software, after the data structures had been designed, which came to 67 man months. The total effort for the automated migration was less than 6 1/2 man months: less than 10% of the estimated effort for a manual translation.

6. Related Work

Feldman and Friedman [4] describe an automated assembler to C migration project which involved the migration of a large database system and application generator written in IBM 370 assembly language. They developed a "literal" translator which translated each instruction separately into C code with no optimisation. In effect, the result of the translation was an IBM 370 simulator. When became clear that this approach would not be sufficient, a new translator (called Bogart) was developed based on abstraction and re-implementation. Bogart produced code which was between half and three quarters as large and more than twice as fast as the literal translator's output. However, the translator required extensive manual modification of the assembler code before it could be applied. Experienced programmers could process about 3600 lines of code per person-month. As a result, "Manual preparation of the code has probably damaged the code's quality. Programmers estimate that the code is less efficient after standardization, and, naturally, new bugs were introduced. . . . two versions now had to be debugged, tested, and maintained" [4]. In addition "Readability was only a secondary goal in this case, because the target code was not meant to be handled by human programmers"

In contrast, our goals with the FermaT migration system are:

1. No manual modification of the assembler code required before migration;

2. Generate HLL code which is both efficient to execute and maintainable by programmers unfamiliar with IBM 370 assembler;
3. Reduce the amount of modification of the HLL code to the absolute minimum: our aim is always for 100% automated migration, but there will always be a handful of constructs which appear so rarely in the code that it is easier to fix the generated code than program a special-purpose transformation rule.

The core transformation engine of FermaT (without the source and target translators) is available under the GNU GPL (General Public Licence) from the following web sites:

http://www.dur.ac.uk/∼dcs6mpw/fermat.html
http://www.cse.dmu.ac.uk/∼mward/fermat.html

7. Future work

We are currently exploring the integration of FermaT with AnaTempura for source code analysis.

AnaTempura is a tool for the runtime verification of systems using Interval Temporal Logic (ITL) and its executable subset Tempura. The runtime verification technique uses assertion points to check whether a system satisfies timing, safety or security properties expressed in ITL. The assertion points are inserted in the source code of the system and will generate a sequence of information (system states), like values of variables and timestamps of value change, while the system is running. Since an ITL property corresponds to a set of sequences of states (intervals), runtime verification is just checking whether the sequence generated by the system is a member of the set of sequences corresponding to the property we want to check.

The integration between AnaTempura and FermaT will allow us to:

- Check (both statically and dynamically) that, for example the migration process is correct with respect to some properties that are preserved by the original system; and
- Perform various source code analysis, especially if we perform slicing first using FermaT.

8. Conclusion

Program transformations form the basis for a practical solution to assembler reengineering and migration projects. Near 100% automated migration can be achieved once the transformation engine has been properly tuned to match the source system.

9. References

[1] Gianfranco Bilardi & Keshav Pingali, "The Static Single Assignment Form and its Computation," Cornell University Technical Report, July, 1999, ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/ssa.ps⟩.

[2] F. P. Brooks, "No Silver Bullet," *IEEE Computer* 20 (Apr., 1987), 10–19.

[3] John R. Ehrman, "Continuing to Profi t from Legacy Assembler Code," *SHARE 100, Feb 2003, Session 8132* (2003).

[4] Yishai A. Feldman & Doron A. Friedman, "Portability by Automatic Translation: A Large-Scale Case Study," *Proc. Tenth Knowledge-Based Software Engineering Conference, Boston, Mass.* (Nov., 1995).

[5] Keshav Pingali & Gianfranco Bilardi, "Optimal Control Dependence Computation and the Roman Chariots Problem," *Trans. Programming Lang. and Syst.* (May, 1997), ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/toplas97.ps⟩.

[6] 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⟩.

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

[8] 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⟩.

[9] M. Ward, "Assembler to C Migration using the FermaT Transformation System," *International Conference on Software Maintenance, 30th Aug–3rd Sept 1999, Oxford, England* (1999).

[10] M. Ward, "Reverse Engineering from Assembler to Formal Specifi cations via Program Transformations," *7th Working Conference on Reverse Engineering, 23-25th November*, Brisbane, Queensland, Australia (2000), ⟨http://www.dur.ac.uk/∼dcs0mpw/martin/papers/wcre2000.ps.gz⟩.

[11] 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⟩.

[12] H. Yang & M. Ward, *Successful Evolution of Software Systems*, Artech House, 2003, ISBN 1-58053-349-3.