The FermaT Assembler Re-engineering Workbench

M. P. Ward
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
`Martin.Ward@durham.ac.uk`

Abstract

Research into the working practices of software engineers has shown the need for integrated browsing and searching tools which include graphical visualisations linked back to the source code under investigation. In addition, for assembler maintenance and re-engineering there is an even greater need for sophisticated control flow analysis, data flow analysis, slicing and migration technology. All these technologies are provided by the FermaT Workbench: an industrial-strength assembler re-engineering workbench consisting of a number of integrated tools for program comprehension, migration and re-engineering. The various program analysis and migrations tools are based on research carried out over the last sixteen years at Durham University, De Montfort University and Software Migrations Ltd., and make extensive use of program transformation theory.

Keywords

Assembler, Re-engineering, Reverse Engineering Migration, Comprehension, Formal Methods, WSL, Wide Spectrum Language, Program Transformation, Legacy Systems, Restructuring.

1 Introduction

Recent research into the activities of software engineers [10] has shown the need for tools capable of *both* semantic-based searching and browsing through hierarchical structures. Other studies [3,6,9] provide strong evidence that software engineers desire tools to help them explore software. They use such tools heavily already and want improvements (the main search tools currently in use are text editors and regular expression search utilities such as `grep`). Top-down program comprehension requires browsing, while bottom-up comprehension required searching: and programmers use both strategies, and frequently switch between them. The four most common search targets are: function definitions, all uses of a function, variable definitions, and all uses of a variable. The most common search motivations are: defect repair, code reuse, program understanding, feature addition, and impact analysis [9].

In [8] a "design browser" tool is described, for flexible browsing of a system's design level representation and for information exchange with a suite of program comprehension tools, complemented with a "retriever" supporting full-text and structural searching. Source code is parsed to an intermediate ASCII representation, imported into a repository based on the UML metamodel, and accessed through an OO database management system (Poet 6.0). The elements in the database can be accessed like normal Java objects and used to build graphical representations in form of diagrams (information views).

The FermaT Workbench is an industrial-strength assembler re-engineering workbench consisting of a number of integrated tools for program comprehension, migration and re-engineering. It differs from these other tools in that FermaT is capable of a much deeper semantic analysis of the assembler source code.

2 Theoretical Foundation

The core of the FermaT Workbench is the FermaT transformation engine. This 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 sixteen years in parallel with the development of the transformation theory: the catalogue of proven transformations and transformation techniques which form the basis for both forward 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, and with the FermaT Workbench, it is not even essential for the user to be able to read and understand WSL. Since each step in the reverse engineering process consists of the application of a proven transformation, whose applicability condition has been mechanically checked, the transformed program is guaranteed to be a correct representation of the original program.

The syntax and semantics of WSL are described in [7,11,12,17] 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. 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.

In [14,19] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [14,17,19] the same transformations are used in the reverse direction: using transformations to derive a concise abstract representation of the specification for several challenging programs.

In [15] program transformations in WSL are used to migrate from assembler to C. In [16] an assembler module is transformed into an equivalent high-level abstract specification in WSL.

3 Previous Transformation Tools

The first tool to be developed as a result of the author's work on WSL and program transformation theory, was the "Maintainer's Assistant" (MA). This was a joint project involving the University of Durham, the Centre for Software Maintenance Ltd., and IBM United Kingdom Laboratories Ltd. MA is implemented in Lisp and includes an X windows based front end (xma) which displays formatted WSL code. The user can select any point in the program and see a list of all the transformations that are applicable at that point. The user can then select a transformation fro the list and see the result immediately.

MA includes a large number of transformations, but is very 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 into equivalent high-level language programs.

The next tool, GREET (Generic REverse Engineering Tool) [1] was a complete reimplementation of the transformation engine using Lisp and a commercial CASE tool builder. The transformations in GREET are implemented in $\mathcal{METAWSL}$, an extension of WSL which includes high-level features for writing program transformations [12,13,18]. 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 GREET is implemented entirely in $\mathcal{METAWSL}$. The implementation of $\mathcal{METAWSL}$ involves a translator from $\mathcal{METAWSL}$ to LISP, a small LISP runtime library (for the main abstract data types) and a WSL runtime library (for the high-level $\mathcal{METAWSL}$ constructs such as **ifmatch**, **foreach**, **fill** etc.).

GREET contains parsers for IBM 370 Assembler and JOVIAL and can generate JOVIAL and C code as well as WSL. The user interface is similar to MA in that the user is presented with formatted WSL code and can click on a section of code and apply transformations.

One of the claims made in [13] is that implementing a large system in a very high level domain-specific language (such as $\mathcal{METAWSL}$) will greatly simplify maintenance and portability. This has proved to be the case with GREET: the entire transformation engine was ported to a very different version of Lisp (Scheme) by a single programmer in a few weeks. Several factors prompted this porting exercise:

- The transformation technology had reached such a level of maturity that the whole transformation process, from the "raw" WSL generated directly from the parsed assembler to high-level WSL suitable for translation to C or COBOL, could be carried out automatically with no human intervention;
- As a result, we could do away with the CASE tool technology on which GREET was built and so

avoid the maintenance overhead, and the memory consumption, at the same time vastly improving portability;

- Transferring from the proprietary Lisp to Scheme meant that we could port the transformation engine from Solaris to AIX, Linux and Windows 95 in a matter of days;
- Eliminating the user interface meant that we could "hide" WSL from the maintenance programmer: no longer is training in WSL and transformation theory a prerequisite for using the tool. Although programmers could see the utility of GREET, there was some resistance to its deployment due to the (perceived) steep learning curve involved in gaining familiarity with WSL and the transformation technology.

The new transformation engine is called FermaT and forms a central component of the FermaT Workbench.

4 Analysing IBM Assembler Code

Assembler code presents a number of unique challenges to automated (and human!) analysis. The code is typically completely unstructured with branches and labels allowed in arbitrary positions. Even where "structured macros" are in use (IF...THEN...ELSE, WHILE...DO etc.) there are no restrictions on branching into or out of structures: so the apparent "surface structure" provided by the macros cannot be relied upon. Subroutines are called by storing a return address in a register and then branching to the start of the subroutine. A subroutine returns by loading the register and branching to the address is contains: but there is nothing to stop the programmer from overwriting or modifying the return address, or branching from the middle of one subroutine to the middle of another, or branching directly back to the main program or any number of other practices. As a result, even determining the boundaries of a subroutine body can be a challenge! Jump tables are yet another problem: the program carries out some computation and then treats the result as an address and branches to it. Self-modifying code is commonly used in legacy assembler code: rather than "wasting" a byte by using a flag, clever programmers would overwrite a branch instruction with a NOP instruction, or vice versa. The IBM 370 architecture also includes an "execute" instruction (EX): this contains the address of an instruction elsewhere in the program and a register which is used to modify the target instruction before executing the modified instruction.

These difficulties also show why assembler, especially legacy assembler, is so much more difficult and costly to maintain, compared to modern high-level languages. All of these complications need to be addressed by any commercial tool for assembler re-engineering. In addition, the need for comprehensive semantic analysis tools is much greater for assembler than for high-level languages. For example: a crude form of data flow analysis is possible in COBOL simply by searching for names of variables. If a variable `FOO` is referenced in one statement, then a search for all assignments to `FOO` will quickly enable the programmer to determine where `FOO` gets its value. But the heavy use of registers and work areas in Assembler, and the lack of data type enforcement, combined with the lack of control flow structure, make these scanner based techniques much less useful. A search for all references to `R3` might return hundreds of hits, almost all of which are irrelevant. But is very difficult to determine if there is an execution path from one line of assembler to another distant line. What is required is a detailed and thorough data flow analysis of the whole program. Such an analysis will also require a detailed and thorough control flow analysis of the whole program: for example to determine all possible return points for a subroutine call.

Data flow analysis is needed for:

1. Debugging: search backwards through data flow from the point where the value of an item is known to be invalid in order to find the code which sets the value; and
2. Enhancement: search forward from an area of code which is about to be changed in order to determine the impact of the proposed change.

These are some of the considerations which led to the development of the FermaT Workbench.

5 The FermaT Workbench

The various tools comprising the FermaT Workbench are accessed via a toolbar and consist of:

- Function Catalogue;
- Function Call Graph;
- Text Editor;
- Program Flowchart;

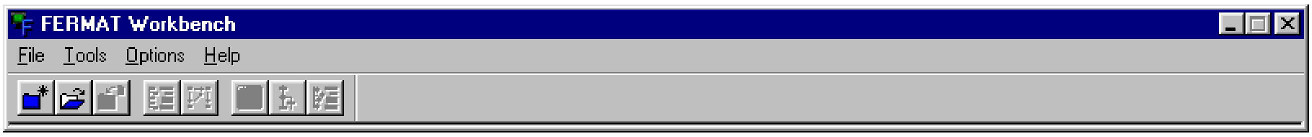

Figure 1: The FermaT Workbench Toolbar

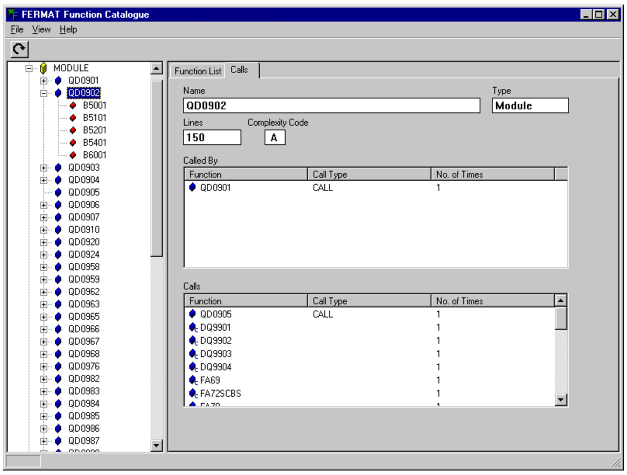

Figure 2: The Function Catalogue

- Data Catalogue;
- Control Flow Analyser;
- Data Flow Analyser;
- Program Slicer;
- Migration Tools;

Each tool is an independent executable, or set of executables, which communicate with the "thin client" workbench toolbar via TCP/IP connections and shared data files (see Figure 1. This design has several advantages:

- The tools do not all need to run on the same machine: for example, the processor-intensive analysis tools can run on a separate high power workstation and communicate with the workbench across a local network (or even across the Internet);
- One tool will not "freeze" the whole workbench while carrying out a time-consuming activity. The user can switch to another tool and carry on working while waiting for output from the first tool;
- Tools can be tested independently of the rest of the workbench via a direct TCP connection (such as `telnet`). This also provides a simple way to automate regression testing.

Source files in FermaT are organised into directories called *Projects*. Each FermaT project consists of a collection of assembler source files, typically representing an assembler program or sub-system. The project also contains all the working files produced by the Workbench. A new project can be created at any time and source files can be imported to the project via a simple list selection.

5.1 The Function Catalogue

A *module* is either a source file, a macro file or a copybook file. Modules are grouped into *functions* and functions can be nested inside other functions. The Function Catalogue shows a hierarchical view of the function tree, with modules as the leaves of the tree, plus a detailed view of the currently selected function. The detailed view shows which functions call this function, plus which functions are called by this function (including external modules: ie calls to modules which are not available in the current project or macro or copybook library).

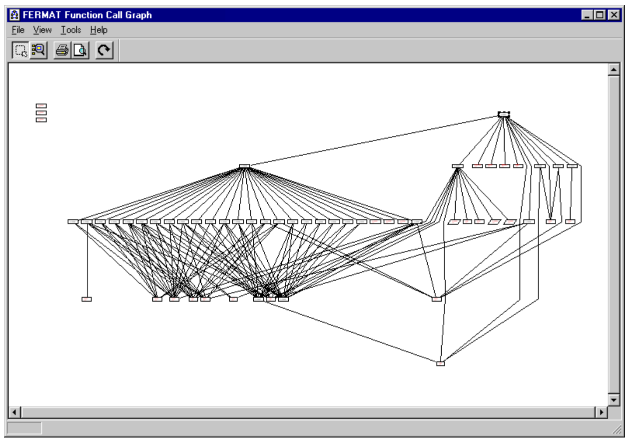

Figure 3: The Function Call Graph

5.2 The Function Call Graph

The function call graph provides a graphical view of the calling relationships between modules. Copybooks and macros can be included or excluded from the graph for clarity.

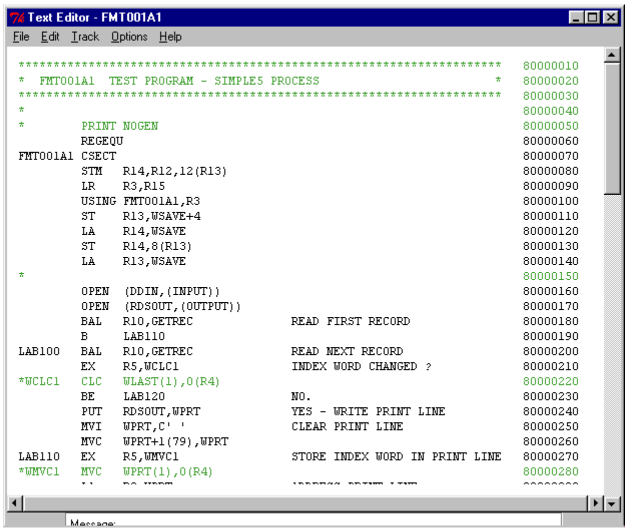

Figure 4: The Text Editor

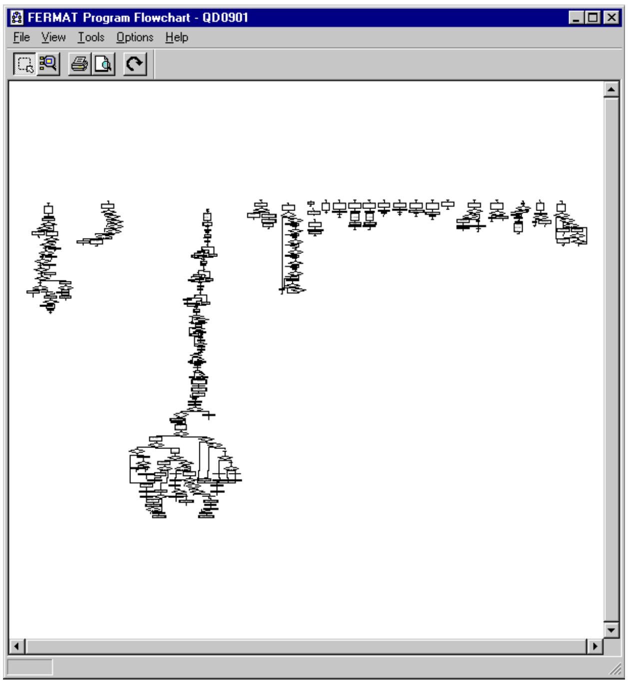

Figure 5: Flowchart (whole program)

5.3 The Text Editor

The text editor is a fully-featured assembler-aware editor which is closely integrated with the other tools in the workbench. Comments are shown in green and other lines may be highlighted in different colours to show the result of a search or other action. The text editor parses each line of assembler and knows which symbols are data items (variables, constants, data structures etc.) and which are not. Any data item can be selected for tracking via the data tracker.

The Data Tracking facilities in the text editor allow the user to select any data item, search for and mark all lines containing that data item and browse through the list of marked lines.

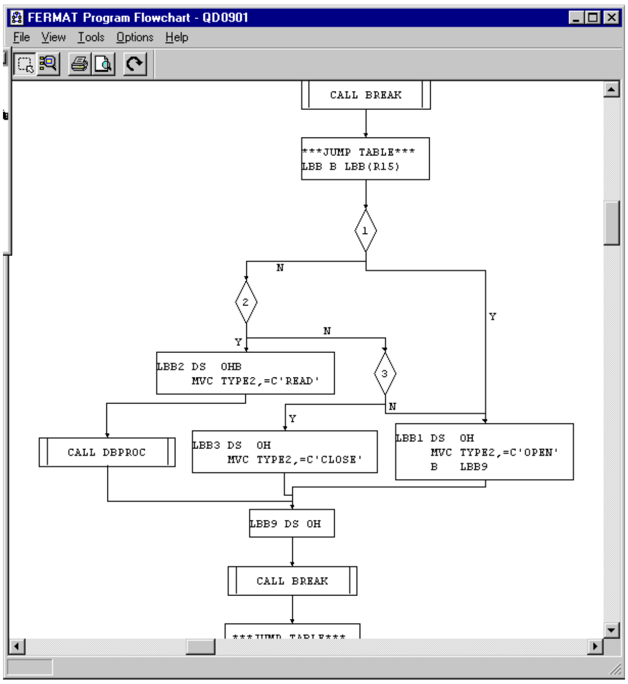

Figure 6: Flowchart (zooming in on part of the program)

5.4 The Program Flowchart

The flowchart tool depicts the control flow of a module in a familiar graphical form. The user can jump from a selected line in the editor to the corresponding node in the flowchart and vice versa. In addition, a block of code can be selected in the editor and highlighted, and the corresponding flowchart nodes will be highlighted. A set of nodes in the flowchart can be highlighted and the corresponding lines in the editor will be highlighted.

Assembler-specific features in the flowcharter include:

- EX instructions: the executed instruction is found and copied in after the line containing the EX instruction;
- Relative branches are computed and the branch target determined where possible;
- Jump tables are detected automatically and converted to a list of conditional branches;
- Subroutines (internal and external) are detected automatically and highlighted;
- Data declarations are ignored in the flowchart;
- Structured macros are interpreted directly.

The user can add their own macros, (including structured macros and subroutine call macros) to the macros table and these will be correctly interpreted by the flowchart tool.

Any block of code in either the editor or flowchart can be highlighted and annotated. The annotation might be a reminder to do something, a warning to a fellow programmer, or a way of documenting a piece of code. In the latter case, the block of code may be "collapsed" to a single node in the flowchart. This is used for incremental redocumentation of the source.

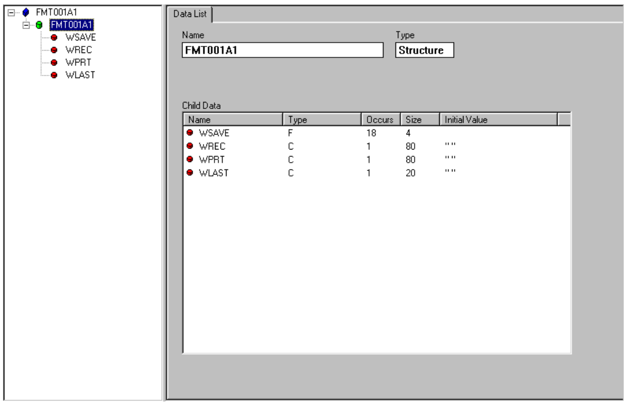

Figure 7: The Data Catalogue

5.5 The Data Catalogue

Like the Function Catalogue, the Data Catalogue shows a hierarchical view of the data layout for the current module, showing which data items are *structures* containing other data items and which are atomic data elements. If the relevant macros and copybooks have been imported to the project, or are present in a library, then the Data Catalogue can also show the data structures which are external to the current module. The Data Catalogue also displays the details of the currently selected data items.

Any data item selected in the text editor can be made the current item in the data catalogue.

5.6 Analysis Tools

The next four tools (control flow analysis, data flow analysis, slicing and migration) require a much more detailed semantic analysis of the assembler. Because of this, these tools require an assembler listing file as input, (rather than just the source file) since the listing contains macro expansions, copybook expansions, relative addresses for all code and data labels, and other important information. For our tools to derive all this information directly from the source files they would need to replicate much of the functionality of an assembler: so it makes better sense to reuse existing technology.

The Analysis tools make use of the FermaT transformation engine: assembler code is translated into WSL, then a sequence of transformations is applied to restructure and simplify the WSL code and remove low-level assembler features. The resulting high-level WSL code is then analysed for control flow and data flow and is sufficiently high level that it can be translated directly into C or COBOL. See [15] for a case study of the automated migration of IBM assembler to efficient and maintainable C code. The high-level WSL also forms the basis for a transformation reverse engineering to an abstract specification in [16].

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 action, 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 action. In addition, FermaT can nearly always eliminate the cc variable by constructing appropriate conditional statements.

The resulting WSL code, after automatic transformation, can then be processed by several analysis tools. Analysis of the transformed WSL code provides much more information, and more accurate information, than could be provided by a direct analysis of the original assembler. For a start, there are fewer nodes in the control flow graph for the WSL code. There are also considerably fewer edges in the control flow graph: for example, the raw WSL contains edges from every "branch to register" instruction to the dispatch procedure, which in turn has an edge to every possible return point. The transformed CFG has usually eliminated the dispatch procedure and replaced all the "save return address" and "branch to register" code by a hierarchy of single-entry single-exit subroutines. The result is much more accurate control and data flow information.

5.6.1 Control Flow Analysis

The Control Flow Analysis tool breaks up the structured WSL into basic blocks and uses these to construct the nodes of the control flow graph. From this graph we can calculate the dominator tree [5] and control dependence information. The "control dependencies" of an instruction are those branch statements which control whether or not the given instruction is executed. To be precise: if one arm of the branch is taken, then the given instruction will eventually be executed (provided the program terminates at all), while if the other branch is taken then the program may terminate without executing the given instruction.

The control dependence information is then transferred back to the assembler listing and recorded as comments. The user can then see a graphical display of the dominator tree and control dependence graph, as well as displaying and browsing control dependence information in the editor.

5.6.2 Data Flow Analysis

The dominator tree is used to compute the Static Single Assignment (SSA) form of the WSL code [2]. From this, use-def and def-use chains can be computed with ease. Again, this information is recorded in the assembler source file ready for browsing via the editor and other tools.

The user can click on any data element and instantly find all the places where this data element gets assigned (showing *only* those assignments which reach the current position in the program), and all those places where the current value of the data element gets used. This sort of information is extremely important for debugging and for impact analysis.

5.6.3 Program Slicer

Any instruction or data item in the program can be selected and a "program slice" [20] computed and displayed. This may be either a forward slice or backward slice.

5.6.4 Migration Tools

See [15,16] for a description of the FermaT migration technology.

Note: at the time of writing (January 2001), the analysis tools (apart from the migration tools) are still being implemented and integrated with the rest of the Workbench.

6 Results

The results from using the FermaT Workbench on major re-engineering projects have so far been very encouraging. The tool has recently been used successfully on two Euro assessment projects.

6.1 Euro Assessment

With the introduction of the Euro currency throughout much of Europe, banks and other financial organisations will have to make some major enhancements to their software systems. A Euro project involves much more than simply adding another currency to the system: there are complex rules to determine how to convert to and from the Euro, and these rules are enforced by legislation. As a result, a Euro conversion is likely to be an order of magnitude more complex than a Y2K conversion.

The first stage in a Euro conversion project is the Assessment Phase: where the aim is to determine precisely which lines of code need to be changed. Assessment involves the following steps:

1. First collect the source and run an automatic inventory report. Depending upon how many missing dependencies there are this may take several passes to obtain a full inventory;

2. Then, scan the copybooks/macros for details of all data declarations;

3. Then enter the Seek Table Utility which uses details of the data declarations to assist the user to dynamically (i.e. without requiring a rescan of the source) produce a base Seek Table, based upon comments and data types;

4. The rest of the source modules are scanned for data declarations and this information is again passed onto the Seek Table Utility. Where there are distinct business areas (with few shared data names and structures), the rest of the assessment project can be conducted in parallel for each business area;

5. The base Seek Table is then further tweaked for the source modules in each business area;

6. The Seek Table Utility then exports a matched field list for every module, to take into account fields with the same names but different uses within different modules (e.g. work fields). The Data Impact Scanner then reads in these "matched field lists" and finds every instance of every required field in every module. Reporting information is output that can be imported into databases/spreadsheets.

In our experience, the whole assessment process can be completed in about five man days for a typical 500K LOC system. Larger systems do *not* require proportionately more effort because there are usually common library modules containing a large proportion of data declarations.

To put this into perspective, one customer stated that what was achieved in one morning using the FermaT Workbench (producing a good quality base seek table) had taken them several weeks of on and off work to do manually: this is equivalent to several days of full time work. We estimate that using the FermaT Workbench reduces the total effort by an order of magnitude, whilst improving the consistency and quality of the results.

7 Availablilty

The FermaT Workbench is currently available for both commercial and academic use. Commercial users should contact Simon Grant of Software Migrations Ltd (`Simon.Grant@SMLtd.com`), academic users should contact the author (`Martin.Ward@SMLtd.com`). At the time of writing (January 2001) the control flow, data flow and slicing tools are currently being implemented and integrated with the Workbench and will appear in a later version.

8 References

[1] K. H. Bennett, H. Yang & T. Bull, "A Transformation System for Maintenance—Turning Theory into Practice," *Conference on Software Maintenance, Orlando, Florida* (1992).

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman & F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependance Graph," *Trans. Programming Lang. and Syst.* 13 (July, 1991), 451–490.

[3] S. Elliott, C. L. A. Clarke, R. C. Holt & A. M. Cox, "Browsing and Searching Software Architectures.

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

[5] T. Lengauer & R. E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *Trans. Programming Lang. and Syst.* 1 (July, 1979), 121–141.

[6] T. C. Lethbridge & J. Singer, "Understanding Software Maintenance Tools: Some Empirical Research," *IEEE Workshop on Empirical Studies of Software Maintenance (WESS'97)*, Bari, Italy (Oct., 1997).

[7] 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⟩.

[8] S. Robitaille, R. Schauer & R. K. Keller, "Bridging Program Comprehension Tools by Design Navigation," *IEEE International Conference on Software Maintenance*, San Jose, CA (Oct., 2000).

[9] S. E. Sim, C. L. A. Clarke & R. C. Holt, "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers," *International Workshop on Program Comprehension* (1998).

[10] J. Singer, T. C. Lethbridge, N. Vinson & N. Anquetil, "An Examination of Software Engineering Work Practices," *Proceedings of CASCON '97*, Toronto, Canada (1997).

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

[12] 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⟩.

[13] 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⟩.

[14] M. Ward, "Program Analysis by Formal Transformation," *Comput. J.* 39 (1996), ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/topsort-t.ps.gz⟩.

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

[16] M. Ward, "Reverse Engineering from Assembler to Formal Specifications 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⟩.

[17] 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⟩.

[18] M. Ward, "Specifications from Source Code—Alchemists' Dream or Practical Reality?," *4th Reengineering Forum, September 19-21, 1994, Victoria, Canada* (Sept., 1994).

[19] 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⟩.

[20] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.* 10 (July, 1984), 352–357.