EXECUTIVE SUMMARY

FermaT Transformation System

The **FermaT Transformation System** can help organizations transform their legacy Assembler systems and applications to better support the current and future needs of the business. The FermaT system has evolved from SML's experience in the development and implementation of its FermaT tools for Assembler documentation, transformation and migration. The FermaT Workbench and FermaT Migration Engine have been developed specifically to support Assembler code documentation, transformation and migration. Current versions of the tool provide developers support for Assembler code documentation, logic, data analysis,, business rule identification and code migration from Assembler to C and COBOL.

FermaT Migration Engine Overview

The objective of the FermaT Migration Engine is to enable the migration of large, highly complex legacy systems from Assembler to higher-level language such as C or COBOL. Once migrated, these systems are substantially easier to maintain and can evolve faster to meet the changing needs of the business they support.

Because of FermaT's use of a unique, formally defined high-level language (WSL) and its specifically designed code transformations, the migration process can be automated. As a result, large legacy systems can be migrated quickly, requiring a fraction of the resources necessary to migrate the code manually.

The WSL FermaT transformation system is built on the transformation theory that has the following objectives.

- Improving the maintainability (in particular, flexibility and reliability, and hence extending the lifetime) of existing mission-critical software systems;

- Translating programs to modern programming languages;

- Developing and maintaining safety-critical applications;

- Extracting reusable components from current systems, deriving their specifications, and storing the specification, implementation, and development strategy in a repository for subsequent reuse;

- Reverse engineering from existing systems to high-level specifications, followed by subsequent reengineering and evolutionary development;

Unlike simple line by line language migration technologies, the FermaT Migration Engine's unique semantics preserving code transformations enable the original application to be automatically cleaned-up, simplified and restructured to its optimum state for migration to the chosen new language. This ensures that only functional code is migrated to the new language, helping to ensure that the migrated code is significantly easier to maintain and adapt than the original.

The process used by the FermaT Migration Engine consists of three basic steps:

Step 1   **Translation from Assembler to WSL**

A sophisticated Assembler parser is used to capture the entire functionality of the Assembler code. This is then automatically converted to SML's own intermediate Wide Spectrum Language (WSL) designed specifically to support code transformation.

Step 2   **Transformation of the WSL**

Once the entire functionality of the Assembler code has been replicated within WSL a series of sophisticated code transformations are automatically applied to the code to restructure and simplify the code to its optimum logical state prior to migration to the chosen target language.

Step 3   **Translation from WSL to either C or COBOL**

Once the automatic restructuring has been completed, an additional set of language specific transformations are applied which convert the restructured WSL representation of the Assembler code to either C or COBOL.

FermaT Transformation System

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.

By using an appropriate sequence of transformations, the extracted representation is guaranteed to be equivalent to the original code logic. The Wide Spectrum Language, called WSL is a logic-based formal method used for the transformation. Over the last sixteen years a large catalogue of proven transformations has been developed, together with mechanically verifiable applicability conditions. These have been applied to many software development, reverse engineering and maintenance problems.

Theoretical Foundation

The theoretical work, on which FermaT is based, originated 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 to support program transformation and includes low-level programming constructs and high-level abstract specifications within a single language. This has the advantage that it is not necessary 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. During this process, 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.

A *program transformation* is an operation that modifies a program into a different form that has the same external behavior (i.e. it is equivalent under precisely defined denotation 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 behavior more defined and/or more deterministic. Typically, the author of a specification will allow some latitude to the developer, by restricting the initial states for which the specification is defined, or by defining a nondeterministic behavior. 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.

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: 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 sixteen years SML have been developing the WSL language, in parallel with the development of transformation theory and proof methods. Over this time the language has developed from a simple and tractable kernel language to a complete and powerful programming language. 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.

Modeling 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 idealized, to a certain extent. In the case of modeling a programming language, such as Assembler, it is theoretically possible to have a perfect model of the language, which correctly captures the behavior 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. However, such a model is useless for migration purposes. What is needed is a practical model for Assembler programs, which is accurate enough to deal with all the programming constructs which are likely to be encountered.

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.

Translation of Standard Assembler Constructs

The following describes how the FermaT Migration Engine handles some standard Assembler constructs with particular reference to C migration.

- Standard opcodes: Each assembler instruction is translated into WSL statements that 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 typical jump tables, incremented return addresses and other simple cases of address arithmetic.

- ❑ 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 (when ever 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 (e.g. HANDLE AID, HANDLE CONDITION) are translated into the appropriate code. Unstructured CICS calls are translated into equivalent structured code through a mechanism that can be extended to other macro packages, e.g. databases, SQL, etc.

The aim of the assembler to WSL translator is to generate WSL code which models as accurately as possible the behavior 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.

As a result, the restructured WSL will be smaller than the original Assembler, with the complexity metrics of the restructured WSL being reduced by a factor of 10 from the raw WSL. Meaning the resultant code is more structured and less complex than the original Assembler.

Mathematically Proven

A fundamental attribute of the FermaT Migration Engine 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.

Additional detailed information can be obtained by reading, "Successful Evolution of Software Systems" written by Martin Ward and Hongji Yang published by Artech House, Inc. in 2003 (www.artechhouse.com). Martin Ward is the chief architect of the FermaT Transformation System.

For further detailed information regarding the FermaT Transformation System contact:
Software Migrations Limited
sales@smltd.com
Tel +44 (0) 8703 898699