SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Conditioned Semantic Slicing for Abstraction; Industrial Experiment[†]

Martin Ward[‡,*], Hussein Zedan[§], Matthias Ladkau[¶] and Stefan Natelberg[‖]

De Montfort University, Gateway House, The Gateway, Leicester, LE1 9BH

SUMMARY

One of the most challenging tasks a programmer can face is attempting to analyse and understand a legacy assembler system. Many features of assembler make analysis difficult, and these are the same features which make migration from assembler to a high level language difficult. In this paper we discuss the application of program transformation technology to assist with analysing and understanding legacy assembler systems. We briefly introduce the fundamentals of our program transformation theory and program slicing which generalises to conditional semantic slicing. These transformations are applied to a large commercial assembler system to automatically generate high-level abstract descriptions of the behaviour of each assembler module, with error handling code sliced away. The assembler system was then migrated to C. The result is a dramatic improvement in the understandability of the programs: on average a 6,000 line assembler listing is condensed down to a 132 line high level language abstraction. A second case study, involving over one million lines of source code from many different assembler systems showed equally dramatic results.

KEY WORDS: Slicing, Program Transformation, FermaT, WSL, Legacy Assembler, Reverse Engineering, Reengineering

Contents

[†]The work is supported by Software Migrations Limited (SML).
[‡]martin@gkc.org.uk
[§]hzedan@dmu.ac.uk
[¶]matthias@ladkau.de
[‖]StefanNatelberg@t-online.de
[*]Correspondence to: De Montfort University, Gateway House, The Gateway, Leicester, LE1 9BH

Received December 3, 2007

1. INTRODUCTION

Over 70% of all business critical software runs on mainframes. If we examine the global distribution of language use, we find that over 10% of all code currently in operation is implemented in assembler. This amounts to 140–220 billion lines of assembler code [1], much of which is running business critical and safety critical systems. The percentage varies in different countries, for example, in Germany it is estimated that about half of all data processing organizations uses information systems written in assembler.

There is a large amount of IBM 370 assembler currently in operation, 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.

Analysing assembler code is significantly more difficult than analysing high level language code. With a typical well-written high level language program it is fairly easy to see the top level structure of a section of code at a glance: conditional statements and loops are clearly indicated, and the conditions are visible. A programmer can glance at a line of code and see at once that it is, say, within a double-nested loop in the ELSE clause of a conditional statement. Assembler code, on the other hand, is simply a list of instructions with labels and conditional or unconditional branches. A branch to a label does not indicate whether it is a forwards or backwards branch, and a backwards branch does not necessarily imply a loop.

If a large body of assembler code can be replaced by a smaller amount of high level language code, without seriously affecting performance, then the potential savings (in the form of software and hardware maintenance costs) are very large.

In this paper we discuss the application of program transformation technology to assist with analysing and understanding legacy assembler systems. We will provide a brief introduction to our program transformation theory and the wide spectrum language WSL. We show that transformation theory provides a unified mathematical framework for program slicing and for transforming low level programs into semantically equivalent higher level programs and illustrate the dramatic improvements in understandability that can be achieved.

1.1. Contributions

In Section 4.4 we give an example which illustrates how a small, but fairly complex assembler module is transformed into a simple, structured WSL program which is easily translated into C or COBOL. The assembler code included a complex control flow structure, with many tests and branches, and two examples of self-modifying code (modifying the end of data address stored in a Data Control Block, and modifying a branch instruction). FermaT was able to generate efficient, structured and maintainable C and COBOL with no human intervention required.

In Section 7 we present two case studies of migrating large-scale assembler systems. These demonstrate that the program transformation technology used in the small example is able to scale up to apply to large bodies of assembler code while still providing a dramatic reduction in complexity. A programmer attempting to understand the function of an unfamiliar module has no option but to examine the listing (the source files usually do not include enough information, since much of the details are hidden in macros and copybooks). In our case studies, the average listing was over 6,000 lines long, while the average abstract WSL program was only 132 lines.

2. THE CHALLENGE FOR AUTOMATED ASSEMBLER ANALYSIS

The technical difficulty of generating a high-level abstract description of assembler code should not be underestimated. Translating assembler instructions to the corresponding HLL (High Level Language) code, and even unscrambling spaghetti code caused by the use of labels and branches, is only a very small part of the analysis task. Other technical problems include:

- Register operations: registers are used extensively in assembler programs for intermediate data, pointers, return addresses and so on. The high-level 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 (Abnormal program termination, or "crash"). Some programmers will write `EX R0,*` (which causes the instruction to execute 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: often 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. A NOP is a "no operation" instruction. This is actually implemented as a "branch never". So by overwriting the part of the instruction which records the conditions under which the branch is taken, a "branch never" can be converted into a "branch always" and vice versa. Less commonly a conditional branch can be modified or created. Overwriting one instruction with a different one is not uncommon, but more general self-modifying code (such as dynamically creating a whole 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 (a software interrupt which invokes an operating system routine) 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 label to branch to when an end of file condition is encountered;

- 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. The simplest solution is to translate the macro expansion and use standard WSL to WSL transformations to restructure the resulting code.
- 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 way 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). For example, the IBM 370 is a "big endian" machine with the most significant byte of a number stored first. The Intel PC architecture is "little endian". So suppose that the assembler program loads the forth byte of a four byte field. If this field contains an integer, then we want to load the low order byte (which is the *first* byte on a little endian machine). But if this field contains a string, then we want the forth character, not the first. There is nothing to stop the assembler programmer from using a four byte character field as an integer, and vice-versa!

Another 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). See [17] for a description of a major migration project where over half a million lines of 16 bit assembler, implementing the core of an embedded system, were migrated to efficient and maintainable C code.

3. OUR APPROACH

Our approach to understanding and migrating assembler code involves four stages:

1. Translate the assembler to WSL;
2. Translate and restructure data declarations;
3. Apply generic semantics-preserving WSL to WSL transformations;
4. Apply task-specific operations as follows:
   (a) For migration: translate the high-level WSL to the target language.
   (b) For analysis: apply slicing or abstraction operations to the WSL to raise the abstraction level even further.

In the following we will describe WSL and the transformation theory and how program slicing can be defined as a transformation within the theory. The mathematical approach to program slicing lends itself naturally to several generalisations, the most important and general of which is conditioned semantic slicing.

4. WSL AND PROGRAM TRANSFORMATION THEORY

The way to get a rigourous proof of the correctness of a transformation is to first define precisely when two programs are "equivalent", and then show that the transformation in question will turn any suitable program into an equivalent program. To do this, we need to make some simplifying assumptions: for example, we usually ignore the execution time of the program. This is not because we don't care about efficiency but because we want to be able to use the theory to prove the correctness of optimising transformations: where a program is transformed into a more efficient version.

More generally, we ignore the internal sequence of state changes that a program carries out: we are only interested in the initial and final states.

Our mathematical model is based on *denotational semantics*. We define the semantics of a program as a function from states to sets of states. A state is simply a function which gives a value to each of the variables in a given set V of variables. The set V is called the *state space*. For each initial state s, the function f returns the set of states $f(s)$ which contains all the possible final states of the program when it is started in state s. A special state $\perp$ indicates nontermination or an error condition. If $\perp$ is in the set of final states, then the program might not terminate for that initial state. If two programs are both potentially nonterminating on a particular initial state, then we consider them to be equivalent on that

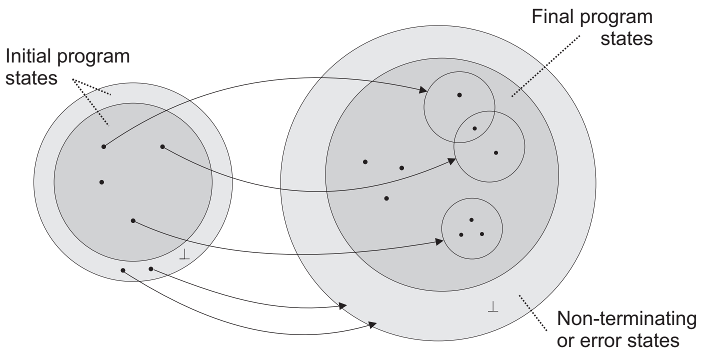

Figure 1: The semantics of a program

state. (A program which might not terminate is no more useful than a program which never terminates: we are just not interested in whatever else it might do). So we define our semantic functions to be such that whenever $\perp$ is in the set of final states, then $f(s)$ must include every other state.

This restriction also simplifies the definition of semantic equivalence and refinement. If two programs have the same semantic function then they are said to be *equivalent*.

A *transformation* is an operation plus a set of conditions, called the *applicability conditions*. The operation takes any program satisfying the applicability conditions and returns an equivalent program. In the literature, "program transformation" has a very broad and varied meaning: it can be used to refer to just about any operation which takes a program, or program fragment in some language and returns another program or program fragment in the same or a different language. In the context of this paper, a "transformation" is a *denotational semantics preserving WSL to WSL transformation*.

A generalisation of equivalence is the notion of *refinement*: one program is a refinement of another if it terminates on all the initial states for which the original program terminates, and for each such state it is guaranteed to terminate in a possible final state for the original program. In other words, a refinement of a program is *more defined* and *more deterministic* than the original program. If program $\mathbf{S}_1$ has semantic function f_1 and $\mathbf{S}_2$ has semantic function f_2, then we say that $\mathbf{S}_1$ *is refined by* $\mathbf{S}_2$ (or $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$), and write:

$$\mathbf{S}_1 \ \leq \ \mathbf{S}_2$$

if for all initial states s we have:

$$f_2(s) \subseteq f_1(s)$$

If $\mathbf{S}_1$ may not terminate for a particular initial state s, then by definition $f_1(s)$ contains $\perp$ and every other state, so $f_2(s)$ can be anything at all and the relation is trivially satisfied. The program abort (which terminates on no initial state) can be refined to *any* other program. Insisting that $f(s)$ include every other state whenever $f(s)$ contains $\perp$ ensures that refinement can be defined as a simple subset relation.

A *transformation* is any operation which takes a statement $\mathbf{S}_1$ and transforms it into an equivalent statement $\mathbf{S}_2$. A transformation is defined in the context of a set of *applicability conditions*, denoted Δ. This is a (possibly empty) set of formulae which give the conditions under which the transformation is valid. If $\mathbf{S}_1$ is equivalent to $\mathbf{S}_2$ under applicability conditions Δ then we write:

$$\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$$

An example of an applicability condition is a property of the function or relation symbols which a particular transformation depends on. For example, the statements $x := a \oplus b$ and $x := b \oplus a$ are equivalent when $\oplus$ is a commutative operation. We can write this transformation as:

$$\{\forall a, b.\, a \oplus b = b \oplus a\} \vdash x := a \oplus b \approx x := b \oplus a$$

An example of a transformation which is valid under *any* applicability conditions is reversing an if statement:

$$\Delta \vdash \mathsf{if}\ \mathbf{B}\ \mathsf{then}\ \mathbf{S}_1\ \mathsf{else}\ \mathbf{S}_2\ \mathsf{fi} \approx \mathsf{if}\ \neg\mathbf{B}\ \mathsf{then}\ \mathbf{S}_2\ \mathsf{else}\ \mathbf{S}_1\ \mathsf{fi}$$

More examples can be found in [16].

4.1. WSL

Over the last twenty years we have been developing the WSL language, in parallel with the development of a transformation theory and proof methods. In this time the language has been extended from a simple and tractable kernel language to a complete and powerful programming language. At the "low-level" end of the language there exist automatic translators from IBM assembler, Intel x86 assembler, TPF assembler, a proprietory 16 bit assembler and PLC code into WSL, and from a subset of WSL into C, COBOL and Jovial. At the "high-level" end it is possible to write abstract specifications, similar to Z and VDM. WSL and the transformation theory has been discussed in other papers before (see [12, 14, 15]). A description of WSL can also be found in [13].

The main goals of the WSL language are:

- Simple, regular and formally defined semantics
- Simple, clear and unambiguous syntax
- A wide range of transformations with simple, mechanically-checkable correctness conditions
- The ability to express low-level programs and high-level abstract specifications

The WSL language and the WSL transformation theory is based on infinitary logic: an extension of first order logic which allows infinitely long formulae. These infinite formulae are very useful for describing properties of programs: for example, termination of a while loop can be defined as "Either the loop terminates immediately, or it terminates after one iteration or it terminates after two iterations or ...". With no (finite) upper bound on the number of iterations, the resulting description is an infinite formula. (Note that the formula which defines the statement "the loop terminates after n iterations" is a different formula for each n. So it is not possible to combine these into a finitary first order logic formula of the form $\exists n.$ the loop terminates after n iterations).

The use of first order logic means that statements in WSL can include existential and universal quantification over infinite sets, and similar (non-executable) operations. The language includes constructs for loops with multiple exits, action systems, side-effects etc., while the transformation theory includes a large catalogue of proven transformations for manipulating these constructs, most of which are implemented in our transformation system, called FermaT. See [17] for a detailed description of the WSL language and transformation theory.

The transformations can be used to derive a variety of efficient algorithms from abstract specifications or the reverse direction: using transformations to derive a concise abstract representation of the specification for several challenging programs.

A WSL statement is a syntactic object: a collection of symbols structured according to the syntactic rules of infinitary first order logic, and the definition of WSL. There may be infinite formulae as components of the statement. The WSL language is built on a simple and tractable kernel language which is extended into a powerful programming language by means of definitional transformations. These are transformations which define the meaning of new programming constructs by expressing them in terms of existing constructs.

The full WSL language includes low-level statements such as assignments, if statements, while loops and local variables. There are other, more unusual statement types which include the following:

- An external procedure call is written:

  !P foo$(e_1, e_2, \ldots e_n$ var $v_1, v_2, \ldots, v_m)$

  The expressions e_i are value parameters and the lvalues v_j are value-result parameters. An external procedure is assumed to always return and to only affect the values of the var parameters.
- Similarly, the condition !XC foo$(e_1, \ldots, e_n)$ is an external boolean function call.
- An assertion statement: $\{\mathbf{Q}\}$ where $\mathbf{Q}$ is any formula, acts as a partial skip statement. If $\mathbf{Q}$ is true then the statement has no effect, while if $\mathbf{Q}$ is false then the statement aborts. A transformation which inserts an assertion into a program must therefore prove that the corresponding condition is always true at that point in the program. Conversely, deleting an assertion is always a valid program refinement since the resulting program can only be more well-defined. (It will be defined on an identical or larger set of initial states, compared to the original program).
- A loop of the form: do $\mathbf{S}$ od is an unbounded loop which can only be terminated by execution of a statement exit(n). This statement will immediately terminate the n enclosing loops. Here n must be a simple integer, not a variable or an expression, so that it is immediately obvious which statement is executed following the exit(n).
- An *action system* is a collection of mutually-recursive parameterless procedures:

  actions A_1 :
  $A_1 \ \equiv S_1$ end
  . . .
  $A_n \ \equiv S_n$ end endactions

  Here, A_i are the action names and S_i are the corresponding action bodies. Each S_i is a statement and the whole action system is also a statement: so it can be a component of an enclosing statement. The action system is executed by executing the body of the starting action (A_1 in this case). A statement call A_i executes the corresponding body S_i. A special call call Z causes the whole action system to terminate immediately. A *regular* action is one in which every execution of the action eventually leads to another action call. An action system is regular if every action is regular. Such an action system can only be terminated by a call Z. Since no action call can ever return, an action call in a regular action system is equivalent to a goto. The assembler to WSL translator generates a regular action system in which each action contains a complete translation of a single assembler instruction, or macro.

4.2. Conditional Semantic Slicing in FermaT

Weiser [18] (Pages: 352–357) defined a program slice **S** as a *reduced, executable program* obtained from a program **P** by removing statements, such that **S** replicates part of the behaviour of **P**. In the context of this paper, program slicing is a useful tool to assist with understanding the behaviour of an assembler module.

Slicing only a program without any additional assumptions is nowadays known as "static slicing". The term "dynamic program slice" was first introduced by Korel and Laski [9]. A dynamic slice of a program P is a reduced executable program S which replicates part of the behaviour of P on a particular initial state. This initial state can be defined by means of an assertion. Some researchers allow furthermore a finite set of initial states, or a partial initial state which restricts a subset of the initial variables to particular values (see [11]). Later researchers have generalised dynamic slicing and combined static and dynamic slicing in various ways. One is "conditioned slicing" first presented in [4] which is a generalisation of both static and dynamic slicing. Our approach, which we call "conditioned semantic slicing", can be seen as a refinement of "conditioned slicing".

In previous publications (see [16]) we provided a unified mathematical framework for program slicing which places all slicing work, for sequential programs, on a sound theoretical foundation. The main advantage to a mathematical approach is that it is not tied to a particular representation. In fact the mathematics provides a sound basis for *any* particular representation. This mathematical representation lends itself naturally to several generalisations, of which *conditioned semantic slicing* is the most general and most useful (see [16] for further details).

A conditioned semantic slice produces a concise, abstract representation of the behaviour of a program with respect to one or more outputs of interest, and under the assumption that certain conditions hold: for example, that no error occurs. Such a representation is very valuable to a programmer who is unfamiliar with the program in question and who needs to work out what the program does under normal operation.

Many large commercial systems contain a lot of error handling code: in some cases much of the code in a module is for error handling and this can obscure the algorithms computed by the module. In addition, many modules produce more than one output. By inserting abort statements at all the points where an error has been detected, and slicing on each individual output it is possible to compute a concise representation of the algorithm (the "business rule") for each output of the module under normal (non error) conditions.

4.3. Automated Migration in Practice

The usual automated migration of assembler to C takes the following steps:

1. Translate the assembler listing to "raw WSL". This translation aims to capture the full semantics of the assembler program without concern for efficiency or redundancy. Typically, each

assembler instruction is translated to a block of WSL code which captures all the effects of the instruction;

2. The data layout of the assembler program is analysed and converted to the equivalent structured data using records, fields, and possibly unions if necessary; (Note that the data layout is **not** changed)
3. The raw WSL is restructured and simplified by applying a large number of correctness preserving WSL transformations. These restructure the control flow to generate structured if statements while loops and so on. They also remove redundant code and use dataflow analysis to remove register usage where possible. The most difficult part of analysing assembler code is tracking return addresses through subroutines to determine subroutine boundaries;
4. Finally, the restructured WSL is translated to the target language. This is a fairly simple transliteration process since the code is already structured and simplified.

For semantic slicing analysis, the restructured WSL code is edited to insert abort statements at the points where error handling code appears, for example, a macro or subroutine call to display an error message would be replaced by an abort. This allows the semantic slicer to "slice away" all code related to error handling: including code both before and after the abort and any tests which branch to error handing code. The result is a dramatically reduced program which includes only the pure "business logic".

4.4. Example of an Automated Migration of Assembler to C

In this section we give an example of how assembler code is transformed into small and easy-to-understand C code. FermaT's semantic slicer is then applied to transform the high-level WSL code into a more compact abstract representation.

```
*********************************************************************
*  REPORT PROGRAM                                                   *
*********************************************************************
*
*        PRINT NOGEN
         REGEQU
         CSECT
         DCBD
START    CSECT
         STM   R14,R12,12(R13)         SAVE ALL REGISTERS
         LR    R3,R15                  COPY R15 TO R3
         USING START,R3                SET UP ADDRESSABILITY
         ST    R13,WSAVE+4             SAVE R13
         LA    R14,WSAVE               SET UP REGISTER SAVE AREA
         ST    R14,8(R13)              SAVE RETURN ADDRESS (R14)
         LA    R13,WSAVE               LOAD R13
         OPEN  (DDIN,(INPUT))          OPEN INPUT FILE
         OPEN  (RDSOUT,(OUTPUT))       OPEN OUTPUT FILE
         NI    LAB140+1,X'0F'          CLEAR THE BRANCH
```

```
         GET   DDIN,WREC                READ FIRST RECORD
         LA    R15,LABEOF               GET THE ADDRESS OF THE CODE
         STCM  R15,B’0111’,DDIN+33      MODIFY DCB FOR NEW EODAD
         B     LAB140                   PROCESS FIRST RECORD
LAB100   GET   DDIN,WREC                READ A RECORD
         CLC   WLAST,WORD               COMPARE TWO STRINGS
         BE    LAB160                   BRANCH ON EQUAL TO LAB160
LAB170   BAL   R10,PUTREC               CALL SUBROUTINE
LAB140   NOP   LAB999                   MODIFIED BRANCH INSTRUCTION
         MVC   PWORD,WORD               STORE INDEX WORD IN PRINT LINE
         PACK  WORKP,NUM                CONVERT STRING TO PACKED DECIMAL
         ZAP   TOTAL,WORKP              COPY WORKP TO TOTAL
LAB120   B     LAB130
LAB160   PACK  WORKP,NUM                CONVERT STRING TO PACKED DECIMAL
         AP    TOTAL,WORKP              ADD NUMBER
         B     LAB100
LAB130   MVC   WLAST,WORD               STORE LAST WORD
         B     LAB100
LAB999   CLOSE DDIN
         CLOSE RDSOUT
         L     R13,WSAVE+4
         LM    R14,R12,12(R13)
         SLR   R15,R15
         BR    R14                      RETURN FROM MODULE
*
LABEOF   OI    LAB140+1,X’F0’           SET THE BRANCH
         B     LAB170                   CONTINUE
*
PUTREC   MVC   PNUM,=X’402020202020202020202120’
         ED    PNUM,TOTAL               CONVERT TOTAL TO STRING IN PNUM
         PUT   RDSOUT,WPRT              WRITE OUTPUT RECORD
         MVI   WPRT,C’ ’                CLEAR PRINT LINE
         MVC   WPRT+1(79),WPRT          CLEAR PRINT LINE
         BR    R10                      RETURN TO CALLER
*
WSAVE    DS    18F                      REGISTER SAVE AREA
WREC     DC    CL80’ ’                  INPUT RECORD AREA
         ORG   WREC
WORD     DS    CL20
         DS    C’ ’
NUM      DS    CL11
         DS    CL48
WPRT     DC    CL80’ ’
         ORG   WPRT
```


Prepared using speauth.cls

```
64 PWORD    DS    CL20
65 PNUM     DS    CL12
66          DS    CL48
67 WLAST    DC    CL20' '
68 TOTAL    DC    PL6'0'
69 WORKP    DC    PL6'0'
70 DDIN     DCB   DDNAME=DDIN,                                   *
71                DSORG=PS,                                      *
72                EODAD=LAB999,                                  *
73                MACRF=GM,RECFM=FT,LRECL=80
74 RDSOUT   DCB   DDNAME=RDSOUT,                                 *
75                DSORG=PS,                                      *
76                MACRF=PM,RECFM=FT,LRECL=80
77          LTORG
78 *
79          END
```

When translating assembler to WSL every instruction or macro is translated into a separate action in a WSL action system. The name of the action is the label of the instruction or a generated name of the form A_xxxxxx where xxxxxx is the hex value of the offset of that instruction from the start of the module.

The compare instruction (`CLC`) for example, translates to code which tests the condition and sets variable cc, representing the "condition code" in the CPU, to 0, 1 or 2 as appropriate. The conditional branch instruction expands to WSL code which tests cc and branches to the appropriate next action.

Unlike modern microprocessors, the IBM 370 does not have a function call stack. Instead, when a module is called it is the caller's responsibility to provide a register save area. Lines 10–16 save the registers in the caller's save area and set up a new save area for any called modules (as it happens, there are no called modules). The FermaT migration process has to detect this register chaining code since the migrated code will save and restore registers automatically on the stack, and therefore R13 will not necessarily point to a valid memory location. These lines also set up R3 to be the base register for the module.

Lines 17–18 open the input and output files. Line 19 modifies the branch instruction at `LAB140` to turn it into a `NOP` (branch never). Line 20 reads a record from the input file `DDIN` into the data area `WREC`. The `DCB` (Data Control Block) on lines 71–74 tells the system about the file. In particular, the `EODAD` parameter tells the system where to branch to on end of file. So there is a "hidden" transfer of control from the `GET` macro to the label `LAB999`. Lines 21–22 load the address of label `LABEOF` into R15 and store it 33 bytes into the `DDIN` data control block. This overwrites the end of data address with a new label. The migrated code will not have data control blocks, or labels, so FermaT has to detect that the `STCM` instruction is actually changing the end of file address and generate an appropriate translation. FermaT creates a new integer variable EODAD_DDIN which records the offset from the start of the module of the current address to branch to on an end of file condition. The translation of the `GET` macro includes a test for end of file. If the test succeeds, then the current value of EODAD_DDIN is copied to a special variable called destination and the dispatch action is called. This action tests destination against each possible branch target offset and branches to the corresponding action. The same mechanism is used to translate subroutine calls: the `BAL` (Branch And Link) instruction is translated to WSL code which stores the offset of the return point in the given register and calls the action with the given label. A `BR` (Branch to Register) instruction is translates to code which copies the register value into destination and calls dispatch.

The program reads an input file consisting of a sequence of records each consisting of a string and a number. A sample input file looks like this:

```
aardvark          1
aardvark         10
aardvark         17
aardvark        123
     bat          2
     bat          3
     cow         99
```

Each contiguous set of records with the same string is summarised into a single output record consisting of the same string plus the sum of all the numbers. In this case, we should get:

```
aardvark        151
     bat          5
     cow         99
```

The following program is part of the raw WSL translation of the assembler module, after the Data_Translation transformation has been applied:

```
var ⟨cc := 0,
  destination := 0,
  EODAD_DDIN := 144,
  EODAD_UNKNOWN := 144⟩ :
  actions _enter_ :
  _enter_ ≡ C :  <ENTRY POINT> ;
             !P init_NOP_flag(0 var F_LAB140);
             call START_0 end
  ...
  LAB100 ≡ C : READ A RECORD;
             !P GET_FIXED(DDIN var WREC, result_code, os);
             r15 := DDIN_STATUS;
             if !XC end_of_file(DDIN)
               then destination := EODAD_DDIN; call dispatch fi;
             call A_00004E end
  A_00004E ≡ C : COMPARE TWO STRINGS;
               if WLAST = WREC.WORD
                 then cc := 0
               elsif WLAST < WREC.WORD
                    then cc := 1
                    else cc := 2 fi;
```

```
               call A_000054 end
A_000054 ≡ C : BRANCH ON EQUAL TO LAB160;
               if cc = 0 then call LAB160 fi;
               call LAB170 end
LAB170 ≡ C : CALL SUBROUTINE;
             r10 := 92;
             call PUTREC end
...
A_0000D0 ≡ C : RETURN TO CALLER;
               destination := r10;
               call dispatch end
dispatch ≡ if destination = 0
              then call Z
           elsif destination = 92
                 then call LAB140
           elsif destination = 144
                 then call LAB999
           elsif destination = 168
                 then call LABEOF
                 else C : Unknown destination ; call Z fi end
endactions end
```

A sequence of WSL transformations are applied to this "raw WSL" which restructure and simplify the action system into procedures and loops. The transformations also automatically remove the cc variable. Removing the cc references significantly reduces the cyclometric complexity of the resulting code. For example, actions A_00004E and A_000054 above will simplify to a single action:

```
A_00004E ≡ C : COMPARE TWO STRINGS;
           if WLAST = WREC.WORD
             then call LAB160
             else call LAB170 fi end
```

reducing the complexity from 5 to 2. Other transformations can dramatically improve the comprehensibility of the code while having no effect on cyclometric complexity: for example, eliminating an unconditional branch (by replacing a call statement by the action body for an action which is only called once) does not change the cyclometric complexity, but "spaghetti code" with many branch statements can be very difficult to understand.

The restructured WSL code looks like this:

```
!P init_NOP_flag(0 var F_LAB140);
C : SAVE ALL REGISTERS;
!P push_regs(r0, r1, r2, ..., r14 var reg_stack);
!P chain_reg( var r13, os);
C : COPY R15 TO R3;
C : SAVE R13;
C : SET UP REGISTER SAVE AREA;
```

```
C : SAVE RETURN ADDRESS (R14);
C : LOAD R13;
C : OPEN INPUT FILE;
!P OPEN(DDIN, "INPUT" var result_code, os);
C : OPEN OUTPUT FILE;
!P OPEN(RDSOUT, "OUTPUT" var result_code, os);
C : CLEAR THE BRANCH;
F_LAB140 := 0;
C : READ FIRST RECORD;
!P GET_FIXED(DDIN var WREC, result_code, os);
if ¬(!XC end_of_file(DDIN))
  then C : GET THE ADDRESS OF THE CODE;
       C : MODIFY DCB FOR NEW EODAD;
       C : PROCESS FIRST RECORD;
       do C : MODIFIED BRANCH INSTRUCTION;
          if F_LAB140 = 1
            then exit(1)
            else C : STORE INDEX WORD IN PRINT LINE;
                 WPRT.PWORD := WREC.WORD;
                 C : CONVERT STRING TO PACKED DECIMAL;
                 !P pack(WREC.NUM var WORKP);
                 C : COPY WORKP TO TOTAL;
                 !P zap(WORKP var TOTAL);
                 C : STORE LAST WORD;
                 WLAST := WREC.WORD;
                 do C : READ A RECORD;
                    !P GET_FIXED(DDIN var WREC, result_code, os);
                    r15 := DDIN_STATUS;
                    if WLAST ≠ WREC.WORD ∨ !XC end_of_file(DDIN)
                      then exit(1)
                      else C : CONVERT STRING TO PACKED DECIMAL;
                           !P pack(WREC.NUM var WORKP);
                           C : ADD NUMBER;
                           !P ap(WORKP var TOTAL) fi od;
                 if !XC end_of_file(DDIN)
                   then C : *;
                        C : SET THE BRANCH;
                        F_LAB140 := 1;
                        C : CONTINUE;
                        C : COMPARE TWO STRINGS;
                        C : BRANCH ON EQUAL TO LAB160 fi;
                 C : CALL SUBROUTINE;
                 C : *;
                 C : CONVERT TOTAL TO STRING IN PNUM;
```

```
        !P ed(TOTAL, "hex 0x4020202020202020202020202120"
         var WPRT.PNUM, cc1, wedit_addr);
        C : WRITE OUTPUT RECORD;
        !P PUT_FIXED(RDSOUT, WPRT var result_code, os);
        C : CLEAR PRINT LINE;
        C : CLEAR PRINT LINE;
        !P fill(!XF mvi(" ") var WPRT);
        C : RETURN TO CALLER;
        exit_flag := 0 fi od fi;
!P CLOSE(DDIN var result_code, os);
!P CLOSE(RDSOUT var result_code, os);
!P pop_regs(
 var r0, r1, r2, ..., r14, reg_stack);
r15 := 0;
C : RETURN FROM MODULE
```

Points to note in the WSL code:

- The code to save registers and set up the register chain has been recognised and translated to external procedure calls push_regs and chain_regs. Similarly, the code to restore registers has been translated to pop_regs. These calls are purely for documentation and are ignored in translations from WSL to C or COBOL.
- The destination variable, dispatch action and related code have all been eliminated automatically by the restructuring transformations. Dataflow analysis has enabled FermaT to determine the target of each end of file branch, which is then converted to structured code. Also, the PUTREC subroutine has been inlined since it is only called once. In the general case, assembler subroutines will be translated into WSL single-entry, single-exit procedures.
- The modified branch instruction has been translated to a conditional branch using a new flag variable F_LAB140. Code which sets and clears the branch has been converted to code to set and clear the flag.

Now it is possible to translate everything to a higher level language such as C:

```
#include "fmt006.h"
  FILE * ddin;
FILE *
  rdsout;
static FWORD
  f_lab140 = 0;

/* End of declarations */

#include "fmt006_init.h"

int
start_0(regs_t * p_regs)
```

```
{
  regs = *p_regs;
  exit_flag = 0;
/* SAVE ALL REGISTERS */
/* COPY R15 TO R3 */
/* SAVE R13 */
/* SET UP REGISTER SAVE AREA */
/* SAVE RETURN ADDRESS (R14) */
/* LOAD R13 */
/* OPEN INPUT FILE */
  OPEN(ddin, "INPUT", &result_code);
/* OPEN OUTPUT FILE */
  OPEN(rdsout, "OUTPUT", &result_code);
/* CLEAR THE BRANCH */
  f_lab140 = 0;
/* READ FIRST RECORD */
  GET_FIXED(ddin, (BYTE *) & wrec, &result_code);
  if (!(end_of_file(ddin))) {
                                            /* GET THE ADDRESS OF THE CODE */
                                               /* MODIFY DCB FOR NEW EODAD */
                                                   /* PROCESS FIRST RECORD */
    for (;;) {                /* DO loop 1 */
                                            /* MODIFIED BRANCH INSTRUCTION */
      if (f_lab140 == 1) {
        break;
      } else {
                                         /* STORE INDEX WORD IN PRINT LINE */
        memmove(wprt.pword, wrec.word, 20);
                                       /* CONVERT STRING TO PACKED DECIMAL */
        pack(workp, 6, wrec.num, 11);
                                                    /* COPY WORKP TO TOTAL */
        zap(total, 6, workp, 6);
                                                        /* STORE LAST WORD */
        memmove(wlast, wrec.word, 20);
        for (;;) {            /* DO loop 2 */
                                                          /* READ A RECORD */
          GET_FIXED(ddin, (BYTE *) & wrec, &result_code);
          if ((end_of_file(ddin)
               || memcmp(wlast, wrec.word, 20))) {
            break;
          } else {
                                       /* CONVERT STRING TO PACKED DECIMAL */
            pack(workp, 6, wrec.num, 11);
                                                             /* ADD NUMBER */
```

```
            ap(total, 6, workp, 6);
          }
        }                       /* OD */
        if (end_of_file(ddin)) {
/* */
                                                          /* SET THE BRANCH */
          f_lab140 = 1;
                                                                /* CONTINUE */
                                                     /* COMPARE TWO STRINGS */
                                               /* BRANCH ON EQUAL TO LAB160 */
        }
                                                         /* CALL SUBROUTINE */
/* */
                                         /* CONVERT TOTAL TO STRING IN PNUM */
        ed(wprt.pnum, 12, &cc1, &wedit_addr, total, 6,
           "\x40\x20\x20\x20\x20\x20\x20\x20\x20\x20\x21\x20",
           12);
                                                     /* WRITE OUTPUT RECORD */
        PUT_FIXED(rdsout, (BYTE *) & wprt, &result_code);
                                                        /* CLEAR PRINT LINE */
                                                        /* CLEAR PRINT LINE */
        memset((BYTE *) & wprt, ' ', 80);
                                                        /* RETURN TO CALLER */
        exit_flag = 0;
      }
    }                           /* OD */
  }
  CLOSE(ddin, &result_code);
  CLOSE(rdsout, &result_code);
  regs.r15 = 0;
/* RETURN FROM MODULE */
  exit_flag = 0;
  return (regs.r15);
}
```

This C code generates identical results to the original assembler.

4.5. Raising the Abstraction Level

In this section we will apply transformations to raise the abstraction level of the restructured WSL program. The Raise_Abstraction transformation carries out the following steps:

1. abort processing: this will apply simplification transformations around abort statements: for example, any code after an abort can be deleted, and an if statement where one branch consists of an abort can be converted to an assertion. In this example, there are no aborts.

2. Delete dead code: normally in a migration, blocks of dead code are converted to procedures in case they are needed for debugging. For raising the abstraction level we want to delete this code.
3. Delete all comments.
4. Convert NOP flags to local variables: this will allow the Flag_Removal transformation to have an opportunity to remove the flags.
5. Delete calls to support functions which are only left in for documentation purposes (these are ignored in a C of COBOL migration anyway). These functions include push_regs, pop_regs, chain_reg and spm. The first three are concerned with register saving, restoring and savearea chaining. spm represents the instruction to Set the Program Mask (the Program Mask contains various flags which control interrupt handling, exponent underflow processing and so on).
6. Unfold procedures which are only called once (there may have been other calls in dead code).
7. Delete redundant statements and simplify all statements and expressions.
8. Apply Flag_Removal (see below).
9. Convert do ... od loops to while loops.

The Flag_Removal transformation attempts to remove flag variables from a program by moving code which tests the flag value closer to code which sets the flag. In this context, a "flag" is a variable which is only ever set to one of two distinct constant values. Flag_Removal checks for candidate flags and carries out the following transformations:

1. Flag tests are converted to test for equality against one of the two known possible values. For example, if the values are 0 and 1 then a test flag > 0 is converted to the equivalent test flag $= 1$.
2. Statement sequences are searched for a statement which sets the flag, followed by a statement (later in the same sequence) which tests the flag. If the statement which sets the flag is a loop, then the first iteration of the loop is unrolled. If the statement is an if statement, then subsequent statements are absorbed into it until the flag test is reached. For example, the program:

   if $x > 0$ then flag $:= 0$ else flag $:= 1$ fi;
   $\mathbf{S}_1$;
   $\mathbf{S}_2$;
   if flag $= 0$ then $\mathbf{S}_3$ fi

   becomes:

   if $x > 0$
     then flag $:= 0$; $\mathbf{S}_1$; $\mathbf{S}_2$; if flag $= 0$ then $\mathbf{S}_3$ fi
     else flag $:= 1$; $\mathbf{S}_1$; $\mathbf{S}_2$; if flag $= 0$ then $\mathbf{S}_3$ fi fi

3. Dataflow analysis is used to determine if flag references can be removed. In the above example (assuming that $\mathbf{S}_1$ and $\mathbf{S}_2$ do not modify the flag) we get:

   if $x > 0$
     then flag $:= 0$; $\mathbf{S}_1$; $\mathbf{S}_2$; $\mathbf{S}_3$
     else flag $:= 1$; $\mathbf{S}_1$; $\mathbf{S}_2$ fi

4. Opportunities for "entire loop unrolling" are searched for. This makes use of the transformation:

   while $\mathbf{B}_1$ do $\mathbf{S}_1$ od $\approx$ while $\mathbf{B}_1 \wedge \mathbf{B}_2$ do $\mathbf{S}_1$ od; while $\mathbf{B}_1$ do $\mathbf{S}_1$ od

At the end of the first while loop, $\neg(\mathbf{B}_1 \wedge \mathbf{B}_2)$ is true, therefore if $\mathbf{B}_1$ is true then $\mathbf{B}_2$ is false. So if the second loop is entered, $\mathbf{B}_2$ must be false on the first iteration. The aim is to split the loop into two and unroll the first iteration of the second loop in order to isolate the code which sets the flag.

5. Potential "first time through" flags are searched for: this is where a flag is tested in a loop body, and subsequently set. If this is found, then "speculatively unroll" to see if the loop body can be simplified.

As a result of these transformations, FermaT is able to eliminate the variable F_LAB140 from the program and restructure it as follows:

```
!P OPEN(DDIN, "INPUT" var result_code, os);
!P OPEN(RDSOUT, "OUTPUT" var result_code, os);
!P GET_FIXED(DDIN var WREC, r0, r1, result_code, os);
while ¬(!XC end_of_file(DDIN)) do
      WPRT.PWORD := WREC.WORD;
      !P pack(WREC.NUM var WORKP);
      TOTAL := WORKP;
      WLAST := WREC.WORD;
      !P GET_FIXED(DDIN var WREC, r0, r1, result_code, os);
      while WLAST = WREC.WORD ∧ ¬(!XC end_of_file(DDIN)) do
            !P pack(WREC.NUM var WORKP);
            !P ap(WORKP var TOTAL);
            !P GET_FIXED(DDIN var WREC, r0, r1, result_code, os)
      od;
      !P ed(TOTAL, "hex 0x40202020202020202020202120"
       var WPRT.PNUM, cc1, wedit_addr);
      !P PUT_FIXED(RDSOUT, WPRT var result_code, os);
      !P fill(" " var WPRT) od;
!P CLOSE(DDIN var result_code, os);
!P CLOSE(RDSOUT var result_code, os);
```

This translates to the following C code:

```
#include "fmt006.h"
  FILE * ddin;
FILE *
  rdsout;

/* End of declarations */

#include "fmt006_init.h"

int
```

```
start_0(regs_t * p_regs)
{
  regs = *p_regs;
/*  <NAME=START_0>  */
  OPEN(ddin, "INPUT", &result_code);
  OPEN(rdsout, "OUTPUT", &result_code);
  GET_FIXED(ddin, (BYTE *) & wrec, &result_code);
  while (!(end_of_file(ddin))) {
    memmove(wprt.pword, wrec.word, 20);
    pack(workp, 6, wrec.num, 11);
    memmove(total, workp, 6);
    memmove(wlast, wrec.word, 20);
    GET_FIXED(ddin, (BYTE *) & wrec, &result_code);
    while ((!(end_of_file(ddin))
            && !(memcmp(wlast, wrec.word, 20)))) {
      pack(workp, 6, wrec.num, 11);
      ap(total, 6, workp, 6);
      GET_FIXED(ddin, (BYTE *) & wrec, &result_code);
    }                          /* OD */
    ed(wprt.pnum, 12, &cc1, &wedit_addr, total, 6,
       "\x40\x20\x20\x20\x20\x20\x20\x20\x20\x20\x21\x20", 12);
    PUT_FIXED(rdsout, (BYTE *) & wprt, &result_code);
    memset((BYTE *) & wprt, ' ', 80);
  }                            /* OD */
  CLOSE(ddin, &result_code);
  CLOSE(rdsout, &result_code);
  return (0);
}
```

This C code also produces identical results to the original assembler.

The C code contains several function calls for handling packed decimal operations: `pack`, `ap` and `ed`. The C compiler for the IBM mainframe includes native packed decimal data types, so the `pack` and `ap` calls are implemented as macros which use native packed decimal operations on this platform. On other platforms the calls are implemented as function calls to emulation functions.

The COBOL language includes native packed decimal data types, and converting a string to a packed decimal is implemented as a simple `MOVE` statement with appropriately declared source and target data elements. Similarly, the edit operation (which converts a packed decimal number to a string with leading zeros suppressed) can be implemented as a simple `MOVE` to a data element declared with the appropriate PICTURE clause. So the COBOL migration of the abstract WSL program is as follows:

```
        PROCEDURE DIVISION.

       *=================================================================
        MAIN                           SECTION.
```


Prepared using speauth.cls

```
*====

      OPEN INPUT DDIN
      OPEN OUTPUT RDSOUT
      SET NOT-END-OF-FILE TO TRUE
      READ DDIN INTO WREC
        AT END SET END-OF-FILE TO TRUE
      END-READ
      PERFORM UNTIL END-OF-FILE
        MOVE WORD TO PWORD
        MOVE NUM TO WORKP
        MOVE WORKP TO TOTAL
        MOVE WORD TO WLAST
        SET NOT-END-OF-FILE TO TRUE
        READ DDIN INTO WREC
          AT END SET END-OF-FILE TO TRUE
        END-READ
        PERFORM UNTIL (WLAST NOT = WORD
          OR END-OF-FILE)
          MOVE NUM TO WORKP
          ADD WORKP TO TOTAL
          SET NOT-END-OF-FILE TO TRUE
          READ DDIN INTO WREC
            AT END SET END-OF-FILE TO TRUE
          END-READ
        END-PERFORM

* #SMLED# EDMSK-BZZZZZZZZZZ9
        MOVE TOTAL TO EDMSK-BZZZZZZZZZZ9
        MOVE EDMSK-BZZZZZZZZZZ9 TO PNUM
        WRITE RDSOUT-RECORD FROM WPRT
        MOVE SPACES TO WPRT
      END-PERFORM

      CLOSE DDIN
      CLOSE RDSOUT
      .
 MAIN-EXIT.
*==========
      PERFORM QUIT-PROGRAM
      .
```

The special comment “`#SMLED# EDMSK-BZZZZZZZZZZ9`” tells the COBOL data generator to declare a data item of the given name with the appropriate PICTURE.

5. RELATED WORK

This section will compare our slicing approach to the work of other slicing researchers. For a deeper discussion the interested reader is referred to [16].

5.1. Amorphous Program Slicing

Harman and Danicic [5, 8] coined the term "amorphous program slicing" for a combination of slicing and transformation of executable programs. To date, the transformations have been restricted to restructuring snd simplifications, but the definition of an amorphous slice allows any transformation (in any transformation theory) of executable programs.

We define a "semantic slice" to be any semi-refinement in WSL, so the concepts of semantic slicing and amorphous slicing are distinct but overlapping. A semantic slice is defined in the context of WSL transformation theory, while an amorphous slice is defined in terms of executable programs (WSL allows nonexecutable statements including abstract specification statements and guard statements). Also, amorphous slices are restricted to finite programs, while WSL programs (and hence, semantic slices) can include infinitary formulae. To summarise:

1. Amorphous slicing is restricted to finite, executable programs. Semantic slicing applies to any WSL programs including non-executable specification statements, non-executable guard statements, and programs containing infinitary formulae;
2. Semantic slicing is defined in the particular context of the WSL language and transformation theory: amorphous slicing applies to any transformation theory or definition of program equivalence on executable programs.

5.2. Slicing and Program Transformation

Tip [10] suggested the computation of slices using a mixture of slicing and transformation in which a program is translated to an intermediate representation (IR), the IR is transformed and optimised (while maintaining a mapping back to the source text), and slices are extracted from the source text.

As an example, consider the following program, where we are slicing on the final value of y:

if $p = q$
  then $x := 18$
  else $x := 17$ fi;
if $p \neq q$
  then $y := x$
  else $y := 2$ fi

Tip's slicer ought to be capable of producing the following slice:

if $p = q$
  then skip
  else $x := 17$ fi;
if $p \neq q$

$$
\begin{array}{rcl}
x := f & \xrightarrow{\text{ctrl}} & q?(c) \\
c := g & \xrightarrow{\text{ctrl}} & q?(c) \\
q?(c) & \xrightarrow{\text{data}} & c := g \\
x := f & \xrightarrow{\text{ctrl}} & p?(i) \\
c := g & \xrightarrow{\text{ctrl}} & p?(i) \\
i := h & \xrightarrow{\text{ctrl}} & p?(i) \\
q?(c) & \xrightarrow{\text{ctrl}} & p?(i) \\
p?(i) & \xrightarrow{\text{data}} & i := h(i)
\end{array}
$$

Table I. Control and data dependencies in the example program

then $y := x$
else $y := 2$ fi

The FermaT toolset with its semantic slicer produces a significantly smaller slice:

$y :=$ if $p = q$ then 2 else 17 fi

5.3. Semantic Slicing Versus Dataflow-Based Slicing

To give a good example of the progressiveness of the semantic slicing consider the example where we are slicing on the final value of x:

while $p?(i)$ do
    if $q?(c)$
      then $x := f$; $c := g$ fi;
    $i := h(i)$ od

Any dataflow-based slicing algorithms (such as [2]) will observe that there is a data dependency between the final value of x and the assignment $x := f$. There is a control dependency between the test $q?(c)$ and the assignment $x := f$ and there is a data dependency between $c := g$ and $q?(c)$. Similarly, there is a control dependency between $x := f$ and $p?(i)$ and a data dependency between $i := h(i)$ and $p?(i)$. Table I summarises the dependencies. If the algorithm simply follows all dependencies in order to determine what statements to include in the slice, then it will conclude that all the statements affect x. FermaT includes implementations for several different slicing algorithms, one of these (the *Syntactic Slicer*) uses just such a dependency tracking algorithm. The syntactic slicer divides the WSL program into "basic blocks" and determines control and data dependencies between the blocks. These dependencies are tracked to compute a slice. So this slicer returns the whole program when asked to slice on the final value of x.

It should not be surprising that the dataflow algorithm sometimes produces a less than minimal slice, since the task of determining a minimal slice is noncomputable in the general case: so there can be no algorithm which always returns a minimal syntactic slice.

For semantic slicing, speculative unrolling can be applied. The transformation system unrolls the first step of the loop to give:

```
if p?(i)
  then if q?(c)
          then ⟨c := g, x := f⟩ fi;
       i := h(i);
       while p?(i) do
              if q?(c)
                then ⟨c := g, x := f⟩ fi;
              i := h(i) od fi
```

(Here, the statement $\langle c := g, x := f \rangle$ is a parallel assignment statement).

It then applies Fully_Absorb_Right to the inner if statement:

```
if p?(i)
  then if q?(c)
          then {q?(c)};
               ⟨c := g, x := f⟩;
               i := h(i);
               while p?(i) do
                      if q?(c)
                        then ⟨c := g, x := f⟩ fi;
                      i := h(i) od
          else {¬q?(c)};
               i := h(i);
               while p?(i) do
                      if q?(c)
                        then ⟨c := g, x := f⟩ fi;
                      i := h(i) od fi fi
```

The statement $\{\neg q?(c)\}$ is an assertion. The assertion $\{\mathbf{Q}\}$ acts as a skip statement (which has no effect) when the condition $\mathbf{Q}$ is true, and aborts when the condition is false. A transformation which introduces an assertion at a point in the program corresponds to proving that the given condition is always true at that point in the program.

Assertions are used to simplify the inner while loops:

```
if p?(i)
  then if q?(c)
          then {q?(c)};
               ⟨c := g, x := f⟩;
               i := h(i);
               while p?(i) do
```

```
                if q?(c)
                  then ⟨c := g, x := f⟩ fi;
                i := h(i) od
      else {¬q?(c)};
           i := h(i);
           while p?(i) do
                i := h(i) od fi fi
```

The semantic slicing function produces this result:

```
if p?(i)
  then if q?(c)
         then {q?(c)};
              ⟨c := g, x := f⟩;
              i := h(i);
              while p?(i) do
                   if q?(c)
                     then ⟨c := g, x := f⟩ fi;
                   i := h(i) od
         else {¬q?(c)}; fi
```

Constant_Propagation simplifies this to:

```
if p?(i)
  then if q?(c)
         then {q?(c)};
              ⟨c := g, x := f⟩;
              i := h(i);
              while p?(i) do
                   i := h(i) od
         else {¬q?(c)}; fi
```

Another call to the semantic slicing function produces:

```
if p?(i)
  then if q?(c)
         then {q?(c)};
              x := f
         else {¬q?(c)}; fi
```

And finally, abstraction and refinement gives this result:

```
if p?(i) ∧ q?(c) then x := f fi
```

This is smaller than the original program, so it is returned as the result of the transformation. The above analysis and transformation steps were all carried out automatically by FermaT's semantic slicer to produce the given result.

5.4. CodeSurfer

CodeSurfer (see: http://www.grammatech.com/products/codesurfer/overview.html) is a C and C++ code browser which includes pointer analysis, call graphs and slicing. It is based on the interprocedural slicing algorithm using dependence graphs published in [2] together with the improvements published in [3]. The product therefore has the limitations comon to all dataflow based slicers (see the previous subsection) and is aimed at syntactic, rather then semantic slicing.

5.5. Slicing Nondeterministic Programs

Binkley and Gallagher's definition of a slice [7] is as follows:

Definition 5.1. For statement s and variable v, the slice **S** of program **P** with respect to the slicing criterion $\langle s;\ v\rangle$ is any executable program with the following properties:

1. **S** can be obtained by deleting zero or more statements from **P**.
2. If **P** halts on input I, then the value of v at statement s each time s is executed in **P** is the same in **P** and **S**. If **P** fails to terminate normally s may execute more times in **S** that in **P**, but **P** and **S** compute the same values each time s is executed by **P**.

This definition does not work with nondeterministic programs. Consider the program **P**:

if true $\rightarrow x := 1$
$\square$ true $\rightarrow x := 2$ fi;
$y := x$

where we are slicing on the value of x at the assignment to y. The if statement in this example is a Dijkstra "guarded command"

According to Binkley and Gallagher's definition, there are *no* valid slices of **P**! Even **P** itself is not a valid slice: since the value of x at $y := x$ may be different each time $y := x$ is executed. Our definition of slicing avoids this flaw and is capable of handling nondeterministic statements. This might appear to be unimportant in practice (since most executable programs are also deterministic), but in the context of program analysis and reverse engineering it is quite common to "abstract away" some implementation details and end up with a nondeterministic abstraction of the original program. If one wishes to carry out further abstraction on this program via slicing, then it is essential that the definition of slicing, and the algorithms implementing the definition, are able to cope with nondeterminism.

6. THE FermaT TECHNOLOGY

6.1. The Transformation Engine

The FermaT Transformation Engine implements the transformation theory described in this paper. The basic version of the FermaT Engine (called fermat3) is free software (available under the GNU General Public Licence). It includes a wide range of transformations operating on the WSL language

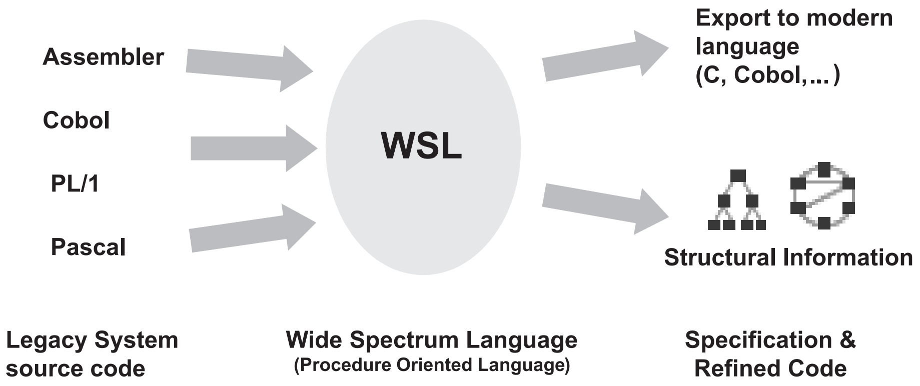

Figure 2: The intention of WSL

but without source and target language translators (e.g. the IBM 360 assembler to WSL translator) and some other extensions and migration-specific transformations.

The WSL language can express many common constructs of procedure oriented imperative languages like Cobol or C as well as assembler languages like the IBM 360 assembler. Once a program has been transferred to WSL the included transformations can be used for restructuring and to raise the abstraction level. If the process is successful the program can be translated into a language which is on a higher level that the source language. For analysis purposes of a given program the engine includes also special tools for control flow and data flow analysis so the internal structure of a legacy system can be presented in diagrams.

Once the source of a legacy system can be transferred into the WSL language the FermaT Transformation Engine can use all its power to raise the abstraction level of the code. The only part of the process where assumptions have to be made is in the translation process from and to the WSL language. All the WSL to WSL transformations on the other hand are mathematically proven to be correct. The advantage of this system is that the most work is being done automatically by the transformation rules. Further details are given in [13].

7. CASE STUDIES

In this section we describe an application for semantic slicing to a full commercial assembler system and a random example of modules from twelve different organisations. Both studies consist of hundred thousands of lines of IBM assembler in over thousand modules. A commercial company, Software Migrations Ltd., have used the FermaT transformation system to migrate an entire assembler system to structured and maintainable C code running on Intel hardware under both Windows and Linux operating systems. The C code was also required to work under the original mainframe environment.

Table II. Case Study 1: Lines of Code and complexity metrics for migration to C

	Total LOC	Per module	McCabe
Original Listings	11,959,084	6,149	—
Raw WSL	2,109,906	1,085	135
Transformed WSL	1,205,766	620	27
WSL without comments	528,440	272	27
C Code including comments	1,120,449	576	—

Our aim for this paper is to describe how FermaT was used to automatically analyse the assembler and generate high-level WSL abstractions for each module, with all error handling code removed.

7.1. Case Study 1: A Full Assembler System

The first case study consisted of a complete assembler system comprising a total of 2,296 modules. The purpose of this case study was to examine FermaT's ability to restructure executable code and remove error handling code. 351 of the modules consisted entirely of data declarations, so these were excluded from the study. The remaining 1,945 modules each contained a mixture of code and data declarations, these totalled 11,959,084 lines of listing (average 6,149 per module).

For the first test we carried out a complete migration to C for all 1,945 modules. This took 7 hours 38 minutes CPU time, averaging just over 14 seconds per module, running FermaT on a 2.6GHz PC. FermaT applied a total of 4,167,286 transformations, averaging 2,143 transformations per module and 152 transformations per second.

Table II records the lines of code and McCabe complexity metrics for the raw WSL (as translated from the assembler), the transformed WSL and the generated C code. The McCabe complexity for the C code was not measured directly but should be identical to that for the transformed WSL since the WSL to C translation is a simple translation of "C like" WSL to C code. Over 43% of the modules (846 modules) contained *no* loops.

Note that in assembler code, the majority of comments are on the same line as an executable instruction. In WSL, each comment is on a separate line. So counting these comments inflates the number of lines of WSL, without affecting the number of lines of assembler. As can be seen in Table II, well over half the lines of transformed WSL are comment lines.

For the second test we made some changes to the assembler to WSL translation table which inserted ABORT statements at the points where the program ABENDs or calls a macro to print an error message, or where the program returns with an error code. The final step was to apply a number of abstraction transformations to the transformed WSL code to generate a high-level abstract equivalent.

Complete analysis of the entire system (all 1,945 code modules) including removal of error handling code and abstraction to high level WSL took 5 hours 10 minutes CPU time on a 2.6GHz P4 processor.

Table III. Lines of Code and complexity metrics for raising abstraction level

	Total LOC	Per module	McCabe
Original Listings	11,959,084	6,149	—
Raw WSL	2,109,704	1,085	135
Transformed WSL	513,616	264	25
Abstract WSL	256,853	132	23

This is an average of under 10 seconds CPU time per module. FermaT applied a total of 3,876,378 transformations, averaging 1,993 transformations per module and 208 transformations per second.

Table III records the lines of code and McCabe complexity metrics for the raw WSL (as translated from the assembler), the transformed WSL (which in this case, also has comments deleted) and the abstract WSL code. A total of 410 modules contained error handling code that was detected and removed. This code amounted to 16% of all the code in the modules. For 40 of the modules, error handling code amounted to over half the executable code in the module. Over the entire system, removing error handling code produced about a 10% reduction in complexity.

For a programmer who needs to understand the main functions of a module, and the algorithms it implements, reading a 132 line abstract WSL program should be much simpler than trying to make sense of a 6,000 line assembler listing!

7.2. Case Study 2: A Random Sample of Assembler Modules

The second case study consists of a (fairly) random sample of 1,905 assembler modules taken from twelve different organisations, and representing approximately one million lines of source code. Of these, 203 consisted entirely of data declarations, so these were ignored for the code analysis tests. The remaining 1,702 modules totalled 5,377,163 lines of listing (average 3,159 per module).

A complete migration to C for all modules took 9 hours 21 minutes CPU time (19.1 seconds per module) on the same PC as the first case study. FermaT applied a total of 10,376,842 transformations, averaging 6,097 transformations per module and 308 transformations per second.

Table IV records the lines of code and McCabe complexity metrics for the raw WSL (as translated from the assembler), the transformed WSL and the generated C code. Note that these listings were shorter on average than those in the first case study, but each listing expanded into more lines of WSL code. This was mainly because the modules in the first case study were taken from an actual migration project. The macros had been analysed and 53 additional macros were added to the translation table: these macros were translated directly to WSL rather than having their macro expansions translated. These were macros which expanded into a significant block of code and which were called throughout the system. Directly translating the macro to the corresponding WSL code, rather than translating the macro expansion, resulted in a significant reduction in the total amount of WSL code generated. For case study 2, all macros other than standard system macros were fully expanded and the macro

Table IV. Case Study 2: Lines of Code and complexity metrics for migration to C

	Total LOC	Per module	McCabe
Original Listings	5,377,163	3,159	—
Raw WSL	4,047,261	2,378	372
Transformed WSL	1,890,376	1,111	65
WSL without comments	1,315,920	773	65
C Code with comments	1,658,236	974	—

Table V. Lines of Code and complexity metrics for raising abstraction level

	Total LOC	Per module	McCabe
Original Listings	5,377,163	3,159	—
Raw WSL	4,047,258	2,378	373
Transformed WSL	736,816	433	62
Abstract WSL	442,764	260	50

expansion was translated to WSL. In addition, the selection of modules used in case study 2 was partly biased towards larger code modules since much of the code was from pilot projects to test the capabilities of the migration engine.

Over 40.5% of the code modules (690 modules) contained *no* loops.

For the second test we made some changes to the assembler to WSL translation table which inserted ABORT statements at the points where the program is definitely known to ABEND. These points were:

- ABEND and PDUMP macros;
- EXEC CICS ABEND calls;
- SVC 13 instructions; and
- SERRC calls which have parameter C (indicating a catastrophic error).

Then we applied a number of abstraction transformations to the WSL code to generate a high-level abstract equivalent for each module. This took a total of 12 hours 58 minutes CPU time.

FermaT applied a total of 10,318,338 transformations, averaging 6,062 transformations per module and 221 transformations per second.

Table V records the lines of code and McCabe complexity metrics for the raw WSL (as translated from the assembler), the transformed WSL and the abstract WSL code.

8. CONCLUSION

In this paper we presented the abilities of the conditional semantic slicing technique and its use for abstraction. Our approach has been demonstrated on industrial case studies and has shown the increase of abstraction levels. The approach has been fully implemented in an industrial strength system known as FermaT, which is currently in use in several organisations.

The FermaT engine can be found on:

http://www.cse.dmu.ac.uk/∼mward/

The F-UML and F-DOC can be found on:

The University Technology Centre in Software Evolution (UTC) of the STRL
http://www.cse.dmu.ac.uk/STRL/research/utc/index.html

REFERENCES

1. Capers Jones. The Year 2000 Software Problem — Quantifying the Costs and Assessing Addison Wesley, 1998.
2. Susan Horwitz Thomas Reps David Binkley. Interprocedural slicing using dependence graphs. *Trans. Programming Lang. and Syst.*, 12:26–60, 1990.
3. Susan Horwitz Thomas Reps M. Sagiv G. Rosay Speeding up slicing *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 11–20, 1994.
4. G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40:595–607, 1998.
5. Mark Harman & Sebastian Danicic. Amorphous program slicing. In *5th IEEE International Workshop on Program Comprehesion (IWPC'97)*, Dearborn, Michigan, May 1997.
6. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
7. David W. Binkley & Keith Gallagher. A survey of program slicing. *Advances in Computers*, 1996.
8. Mark Harman, Lin Hu, Malcolm Munro, and Xingyuan Zhang. Gustt: An amorphous slicing system which combines slicing and transformation. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, Los Alamitos, California, 2001.
9. Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1998.
10. F. Tip. *Generation of Program Analysis Tools*. PhD thesis, Cantrum voor Wiskunde en Informatica, Amsterdam, 1995.
11. G. A. Venkatesh. The semantic approach to program slicing. *SIGPLAN Notices*, 26(6):107–119, June 1991.
12. H. A. Priestley & M. Ward. A multipurpose backtracking algorithm. *J. Symb. Comput. 18*, pages 1–40, 1994.
13. Hongji Yang & Martin Ward. *Successful evolution of software systems*. Artech house, INC., 2003.
14. M. Ward. *Proving Program Refinements and Transformations*. Dphil thesis, Oxford University, 1989.
15. M. Ward. Derivation of data intensive algorithms by formal transformation. *IEEE Trans. Software Eng. 22*, pages 665–686, Sept. 1996.
16. M. P. Ward and H. Zedan. .Slicing as a Program Transformation *ACM Transactions On Programming Languages and Systems*, 29(2), April 2007.
17. Martin Ward. Pigs from sausages? reengineering from assembler to c via fermat transformations. *Science of Computer Programming, Special Issue on Program Transformation 52*, pages 213–255, 2004.
18. M. Weiser. Program slicing. *IEEE Trans. Software Eng. 10*, July 1984.