Combining Dynamic and Static Slicing for Analysing Assembler

Martin Ward and Hussein Zedan
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
martin@gkc.org.uk and zedan@dmu.ac.uk

Abstract

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 describe some of the methods used in the FermaT transformation system for analysing and migrating assembler systems. One technique we discuss in detail is to combine a simple dynamic slice, computed with virtually no overhead, and a static slice implemented using program transformation technology, to generate very concise high-level descriptions of the sliced code.

Keywords: Assembler; Slicing; Abstraction; Program transformation; Formal methods; Dynamic slicing; Static slicing.

Contents

1 Introduction **3**

2 The Challenge for Automated Assembler Analysis **4**

2.1 Condition Codes 6
2.2 Jump Tables 6
2.3 Assembler Slicing 7
2.4 Embedded Systems 7

3 Our Approach **7**

4 WSL AND PROGRAM TRANSFORMATION THEORY **8**

4.1 WSL 9
4.1.1 Syntax of the Kernel Language 10
4.1.2 Language Extension by Definitional Transformations 11
4.2 The Specification Statement 12
4.3 Weakest Preconditions 12
4.4 Example Transformations 13

5 Slicing in WSL **13**
5.1 Reduction . . . 14
5.2 Semi-Refinement . . . 15
5.3 Slicing Unstructured Programs . . . 18
5.4 Semantic and Amorphous Slicing . . . 19
5.5 Slicing At Any Position . . . 20
5.6 Dynamic Slicing . . . 21
5.7 Dynamic Slicing of Assembler Modules . . . 21
5.8 Conditioned Slicing . . . 22
5.9 Conditioned Semantic Slicing . . . 23
5.10 Transformations Involving Abort and Assertions . . . 24
5.11 A Minimal Semantic Slice . . . 25
5.11.1 FermaT Implementation of Abstraction and Refinement . . . 26
5.12 Slicing Programs Containing Loops . . . 28
5.12.1 Speculative Unrolling . . . 28

6 Assembler Abstraction and Analysis **31**
6.1 Assembler to WSL Translation . . . 31
6.2 Data Restructuring and Translation . . . 32
6.3 WSL to WSL Transformation . . . 32
6.3.1 Preventing Loops . . . 33
6.3.2 Dataflow Analysis . . . 34

7 First Case Study **36**

8 Second Case Study **40**

9 Mass Migration Exercises **45**

10 Practical Applications **46**
10.1 Debugging . . . 46
10.2 Reengineering to an Object Oriented System . . . 46

11 Related Work **47**
11.1 Assembler Migration . . . 47
11.2 Amorphous Slicing . . . 48

12 FermaT Availability **49**

13 Conclusion **49**

1 Introduction

Over 70% of all business critical software runs on mainframes [25]. 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 [18], 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 [26].

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. Simply finding all the branch instructions which lead to a particular label involves scanning the whole program (and just because an instruction does not have a label, does not mean that it cannot be branched to!)

As well as being more difficult to analyse, for a given functionality there is more code to analyse. A single function point requires on average, about 575 lines of basic assembler or 400 lines of macro assembler to implement, while only 220 lines of C or COBOL are needed. A higher level language such as perl will require only 50 lines on average to implement one function point [19].

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

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

In previous papers [32,39,42] we have described the application of program transformation technology to automated migration from assembler to a high level language. The basic approach follows three stages:

1. Translate the assembler into our internal Wide Spectrum Language (called WSL);
2. Apply correctness-preserving WSL to WSL transformations to the code to restructure, simplify, raise the abstraction level, etc. These may include syntactic and/or semantic code slicing [41];
3. Translate the high-level WSL directly into the target language (currently either C or COBOL).

Since these papers were published there have been many improvements made to FermaT including:

- Improved detection and translation of self-modifying code;
- Extensive jump table detection (see Section 2.2);
- Improved dataflow analysis;
- Array detection and analysis (including detection of arrays of structures);
- Implementation of program slicing for WSL (our internal Wide Spectrum Language) and assembler;
- Static Single Assignment computation.

This paper describes some of the recent developments in the technology, focusing on the specific task of extracting a high-level abstract description of the semantics of an assembler module when executed on a specific input state, or a finite set of different input states.

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 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: some programmers write "`EX R0,*`" (where the instruction executes itself) precisely to achieve an ABEND: so the translator has to take this into account;
- 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 (NOP is a "branch never" instruction) into an unconditional branch, or modifying an unconditional branch into a NOP. 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 EODAD (end of file) address label;

- User macros: users typically write their own macros, and these may include customised versions of system macros. The translation technology needs to be highly customisable to cope with these and to decide in each case whether to translate the macro directly, or translate the macro expansion;
- Structured macros: in the case of so-called "structured macros" (IF, WHILE etc.) it is best simply to translate the macro expansion because there are no restrictions on using structured macros in unstructured ways. 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 a 32 bit integer, a floating point number, a seven digit packed decimal number, a four digit zoned decimal number, a pointer, a pair of 16 bit integers, an array of four characters, or 32 separate one-bit flags. Ideally, the HLL data should be laid out in memory in precisely 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` in terms of `RECSTART`;
- Pointers: these are used extensively in many assembler programs. If the HLL is C then pointers and pointer arithmetic are 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. With COBOL, more work is required to eliminate pointers where possible by accessing data directly (possibly via an index);
- Memory addressing: DSECT data in a 370 assembler program is accessed via a base register which contains the address of the start of the block of data. This is added to a symbolic offset to compute the memory address. 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 C code on a non-IBM 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. Note that this may conflict with the requirement to preserve the memory layout. It may be necessary to implement a pointer stored in memory as a 32 bit offset from a 64 or 128 bit base pointer.
- "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!

Describing how all these challenges are met would take far more space than a single paper, so we focus on two examples in the next two sections.

2.1 Condition Codes

The condition codes are implemented in the initial translation as the variable cc: this is a special variable which can only take one of the four values 0, 1, 2 or 3. An instruction, such as COMPARE, which sets the condition code is translated to WSL code to assign to cc. For example `CR R1,R2` is translated:

if $r_1 = r_2$ **then** $cc := 0$
elsif $r_1 < r_2$ **then** $cc := 1$
else $cc := 2$ **fi**

A conditional branch instruction is translated into code which tests cc, for example, `BNH FOO` (branch on not high) becomes:

if cc <> 2 **then call** FOO **fi**

After the first statement, FermaT knows the condition under which cc is set to each of the possible values. These are listed in Table 1.

Value	Condition
0	$r_1 = r_2$
1	$r_1 < r_2$
2	$r_1 > r_2$
3	**false**

Table 1: Conditions for each cc value

The Constsnt_Propagation transformation scans forwards through the control flow of the program looking for references to cc. Each condition on cc is replaced by the appropriate set of conditions from the table. In our example, cc <> 2 is the same as $\mathsf{cc} = 0 \vee \mathsf{cc} = 1 \vee \mathsf{cc} = 3$ so the condition is replaced by the formula $r_1 = r_2 \vee r_1 < r_2 \vee$ **false** which simplifies to $r_1 \leqslant r_2$. So the code becomes:

if $r_1 = r_2$ **then** $cc := 0$
elsif $r_1 < r_2$ **then** $cc := 1$
else $cc := 2$ **fi**
if $r_1 \leqslant r_2$ **then call** FOO **fi**

In most cases, the conditional branch immediately follows the condition-setting instruction, but this cannot be relied upon. In some cases the two instructions may be some distance apart, and further transformations are needed to bring them together. Also, note that the assignments to cc have not (yet) been deleted: there may be further references to these cc values elsewhere in the program. Once all references to cc have been replaced in this way, all assignments to cc can be deleted.

2.2 Jump Tables

To find jump tables the translator looks for various "candidate" instructions. Among other cases, these include:

- A LOAD instruction which loads from a memory location which contains the address of a code line, where the instruction includes an index register;

- A branch instruction which has a code label and an index register: if the branch is to the start of a sequence of branch instructions, then these are the entries in the table. The index register determines which branch instruction will be executed;
- A LOAD or LOAD ADDRESS of a code label, followed by a branch to register or subroutine call, which uses that register plus an index register;
- A LOAD ADDRESS instruction which adds the address of a code label to the value in a register, and which is followed by a branch to register instruction on that register;

In each case, when a jump table has been detected then a multi-way IF statement is constructed from the list of target addresses.

2.3 Assembler Slicing

The authors are not aware of any practical tools, other than FermaT, which are capable of static or dynamic slicing of IBM assembler modules.

Commercial assembler systems can be very large: applications which comprise of several million lines of code and many thousands of modules are not uncommon. The systems have grown over many years and are highly interdependent, so that it is difficult or impossible to decompose a large system into a number of smaller self-contained subsystems.

The assembler modules are usually "algorithmically straightforward" (compared, for example, to a typical scientific program) but it is hard to see the wood for the trees with lots of code to handle different types of input, many special cases, and thorough error checking and sanity checking of all inputs. There may also be obsolete and redundant code which has been left in because nobody is quite sure what the code does or why it was there in the first place.

2.4 Embedded Systems

A major application for assembler code is in embedded systems. Many embedded systems were developed for processors with limited memory and processing capability, and were therefore implemented in tightly coded hand written assembler. Modern processors are now available at a lower cost which have much more processing and memory capacity and with efficient C compilers. To make use of these new processors the embedded system needs to be re-implemented in a high level language in order to reduce maintenance costs and enable implementation of major enhancements. Many of the challenges with 370 assembler (such as the EXecute instruction and self-modifying code) are not relevant to embedded systems processors, but other challenges become important (such as 16 bit addresses and 8 or 16 bit registers). See [39] 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 the following stages:

1. Execute the assembler program on sample inputs, recording a trace of which basic blocks were executed;
2. Translate the assembler to WSL;
3. Insert **abort** statements in the basic blocks which were never executed;
4. Apply semantics-preserving WSL to WSL transformations to simplify the WSL code using the **abort** statements and raise the abstraction level;
5. Apply semantic slicing on the outputs of interest to generate a high-level abstract representation of the semantics of the program for the given inputs and outputs.

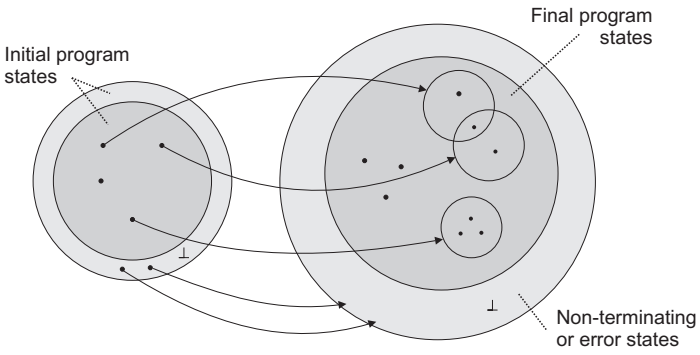

Figure 1: The semantics of a program

We will first describe WSL and the transformation theory, next we will describe how program slicing can be defined as a transformation within the theory. This mathematical approach to program slicing lends itself naturally to several generalisations, the most important and general of which is conditioned semantic slicing. We describe the main transformations related to assembler analysis, including **abort** processing and loop unrolling.

Finally, we present two case studies each consisting of an assembler module which is analysed via combined (dynamic plus static) slicing and transformation to determine a high-level abstract description of the semantics of the module for a given set of inputs and outputs.

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*: a program may have different initial and final state spaces. 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 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 therefore 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 a semantically 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:

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

More examples can be found in [41].

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 exists 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 [24,31,36]). A description of WSL can also be found in [46].

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, not a formula with n as a free variable. 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 a transformation system, called FermaT. See [39] 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.

4.1.1 Syntax of the Kernel Language

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 WSL kernel language requires just four primitive statements and three compound statements. Let $\mathbf{P}$ and $\mathbf{Q}$ be any infinitary logical formulae and $\mathbf{x}$ and $\mathbf{y}$ be any finite, non-empty lists of variables. The primitive statements are:

1. **Assertion:** $\{\mathbf{P}\}$ is an assertion statement which acts as a partial **skip** statement. If the formula $\mathbf{P}$ is true then the statement terminates immediately without changing any variables, otherwise it aborts (we treat abnormal termination and non-termination as equivalent, so a program which aborts is equivalent to one which never terminates);
2. **Guard:** $[\mathbf{Q}]$ is a guard statement. It always terminates, and enforces $\mathbf{Q}$ to be true at this point in the program *without changing the values of any variables*. It has the effect of restricting previous nondeterminism to those cases which will cause $\mathbf{Q}$ to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition (including $\mathbf{Q}$);
3. **Add variables: add**($\mathbf{x}$) first ensures that the variables in $\mathbf{x}$ are in the state space (by adding them if necessary) and then assigns arbitrary values to the variables in $\mathbf{x}$. The arbitrary values may be restricted to particular values by a subsequent guard;
4. **Remove variables: remove**($\mathbf{y}$) ensures that the variables in $\mathbf{y}$ are *not* present in the state space (by removing them if necessary).

The compound statements are:

1. **Sequence:** ($\mathbf{S}_1$; $\mathbf{S}_2$) executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;

2. **Nondeterministic choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution, the choice being made nondeterministically;
3. **Recursion:** $(\mu X.\mathbf{S}_1)$ where X is a *statement variable* (a symbol taken from a suitable set of symbols). The statement $\mathbf{S}_1$ may contain occurrences of X as one or more of its component statements. These represent recursive calls to the procedure whose body is $\mathbf{S}_1$.

A WSL program $\mathbf{S}$ is always interpreted in the context of an initial state space V and a final state space W. We define the trinary relation $\mathbf{S} : V \rightarrow W$ on $\mathbf{S}$, V and W to be true whenever V and W are valid initial and final state spaces for $\mathbf{S}$.

4.1.2 Language Extension by Definitional Transformations

The full WSL language is built up from the kernel by defining new constructs in terms of existing constructs, and ultimately in terms of kernel constructs. It includes low-level statements such as assignments, **if** statements, **while** loops and local variables. The deterministic **if** statement

$$\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

is defined as:

$$(([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

The nondeterministic **if** statement (Dijkstra's "guarded command" [12]):

$$\textbf{if } \mathbf{B}_1\ \rightarrow\ \mathbf{S}_1 \square \mathbf{B}_2\ \rightarrow\ \mathbf{S}_2 \textbf{ fi}$$

is defined as:

$$(\{\mathbf{B}_1\ \vee\ \mathbf{B}_2\};\ (([\mathbf{B}_1];\ \mathbf{S}_1)\ \sqcap\ ([\mathbf{B}_2];\ \mathbf{S}_2)))$$

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 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 :
  $A_1\ \equiv$
  $\mathbf{S}_1$ **end**
  ...
  $A_n\ \equiv$
  $\mathbf{S}_n$ **end endactions**

Here, A_i are the action names and $\mathbf{S}_i$ are the corresponding action bodies. Each $\mathbf{S}_i$ 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 $\mathbf{S}_i$. 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 The Specification Statement

For our transformation theory to be useful for both forward and reverse engineering it is important to be able to represent abstract specifications as part of the language and this motivates the definition of the *Specification statement*. Then the refinement of a specification into an executable program, or the reverse process of abstracting a specification from executable code, can both be carried out within a single language. Specification statements are also used in semantic slicing (see Section 5.4).

Informally, a specification describes *what* a program does without defining exactly *how* the program is to work. This can be formalised by defining a specification as a list of variables (the variables whose values are allowed to change) and a formula defining the relationship between the old values of the variables, the new values, and any other required variables.

With this in mind, we define the notation $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ where $\mathbf{x}$ is a sequence of variables and $\mathbf{x}'$ the corresponding sequence of "primed variables", and $\mathbf{Q}$ is any formula. This assigns new values to the variables in $\mathbf{x}$ so that the formula $\mathbf{Q}$ is true where (within $\mathbf{Q}$) $\mathbf{x}$ represents the old values and $\mathbf{x}'$ represents the new values. The formula $\mathbf{Q}$ therefore specifies the program by defining the relationship between the old values of the variables and the new values. If there are no new values for $\mathbf{x}$ which satisfy $\mathbf{Q}$ then the statement aborts. The formal definition is:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q} \;=_{\mathrm{DF}}\; \{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

For example, the specification statement $\langle x \rangle := \langle x' \rangle.(x' = 1)$ defines a program which assigns the value 1 to the variable x, while the specification statement $\langle x \rangle := \langle x' \rangle.(x' = x + 1)$ defines a program which increments the value of x.

If the "unprimed" variables $\mathbf{x}$ do not appear free in $\mathbf{Q}$, then that simply means that the values assigned to the variables in $\mathbf{x}$ do not depend on the original values of those variables $\mathbf{x}$. Conversely, if the primed variables $\mathbf{x}'$ do not appear in $\mathbf{Q}$, then the statement can assign *any* values to the variables in $\mathbf{x}$, provided condition $\mathbf{Q}$ holds initially. In this case, if $\mathbf{Q}$ is not true initially, then the statement aborts since $\mathbf{Q}$ cannot be made true by assigning values to the primed variables. To be precise, if non of the variables in $\mathbf{x}'$ appears free in $\mathbf{Q}$ then:

$$\Delta \vdash \mathbf{x} := \mathbf{x}'.\mathbf{Q} \;\approx\; \{\mathbf{Q}\}$$

4.3 Weakest Preconditions

Dijkstra introduced the concept of weakest preconditions [12] as a tool for reasoning about programs. For a given program $\mathbf{P}$ and condition $\mathbf{R}$ on the final state space, the weakest precondition $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ is the weakest condition on the initial state such that if $\mathbf{P}$ is started in a state satisfying $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ then it is guaranteed to terminate in a state satisfying $\mathbf{R}$.

Given any statement $\mathbf{S} : V \to W$ and any formula $\mathbf{R}$ whose free variables are all in W and which defines a condition on the final states for $\mathbf{S}$, we define the *weakest precondition* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$

to be the weakest condition on the initial states for $\mathbf{S}$ such that if $\mathbf{S}$ is started in any state which satisfies $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ then it is guaranteed to terminate in a state which satisfies $\mathbf{R}$. By using an infinitary logic, it turns out that $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ has a simple definition (as a formula of infinitary logic) for all WSL programs.

4.4 Example Transformations

In this section we introduce some of the basic transformations which are proved using weakest preconditions.. To prove the transformation:

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

we simply need to show that the corresponding weakest preconditions are equivalent. See [41] for the proof.

Another simple transformation is merging two assignments to the same variable:

$$\Delta \vdash x := e_1;\ x := e_2 \ \approx\ x := e_2[e_1/x]$$

Another simple transformation is Expand Forwards:

$$\Delta \vdash \textbf{if } \mathbf{B}_1 \textbf{ then } \mathbf{S}_1 \ldots \textbf{ elsif } \mathbf{B}_n \textbf{ then } \mathbf{S}_n \textbf{ fi};\ \mathbf{S} \ \approx\ \textbf{if } \mathbf{B}_1 \textbf{ then } \mathbf{S}_1;\ \mathbf{S} \ldots \textbf{ elsif } \mathbf{B}_n \textbf{ then } \mathbf{S}_n;\ \mathbf{S} \textbf{ fi}$$

For more complex transformations involving recursive constructs, we have a useful induction rule which is not limited to a single recursive procedure, but can be used on statements containing one or more recursive components (including nested recursion). For any statement $\mathbf{S}$, define $\mathbf{S}^n$ to be $\mathbf{S}$ with each recursive statement replaced by its nth truncation.

Lemma 4.1 *The General Induction Rule for Recursion:* If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that $\Delta \vdash \mathbf{S}^n \ \leq\ \mathbf{S}'$ for all $n < \omega$, then $\Delta \vdash \mathbf{S} \ \leq\ \mathbf{S}'$.

Here, "bounded nondeterminacy" means that in each specification statement there is a finite number of possible values for the assigned variables. See [31] for the proof.

An example transformation which is proved using the generic induction rule is *loop merging*. If $\mathbf{S}$ is any statement and $\mathbf{B}_1$ and $\mathbf{B}_2$ are any formulae such that $\mathbf{B}_1 \Rightarrow \mathbf{B}_2$ then:

$$\Delta \vdash \textbf{while } \mathbf{B}_1 \textbf{ do } \mathbf{S} \textbf{ od};\ \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od } \approx\ \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od}$$

To prove loop merging it is sufficient to prove by induction that for each n there exists an m such that:

$$\Delta \vdash \textbf{while } \mathbf{B}_1 \textbf{ do } \mathbf{S} \textbf{ od}^n;\ \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od}^n \ \leq\ \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od}^m$$

and for each m there exists an n such that:

$$\Delta \vdash \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od}^m \ \leq\ \textbf{while } \mathbf{B}_1 \textbf{ do } \mathbf{S} \textbf{ od}^n;\ \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od}^n$$

(See [31] for the induction proofs). The result then follows from the general induction rule.

5 Slicing in WSL

Weiser [44] defined a program slice $\mathbf{S}$ as a *reduced, executable program* obtained from a program $\mathbf{P}$ by removing statements, such that $\mathbf{S}$ replicates part of the behaviour of $\mathbf{P}$. In the context of this paper, program slicing is a useful tool to assist with understanding the behaviour of an assembler module. In this section we will provide 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. 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.

If we are slicing on the *end* of the program then subset of the behaviour we want to preserve is simply the final values of one or more variables (the variables in the slicing criterion). If we modify both the original program and the slice to delete the unwanted variables from the state space, then the two modified programs *will* be semantically equivalent, provided that they both terminate. (We will see later that *equivalence* is not quite the relation we want. The correct relation is *semi-refinement*. See Section 5.2).

If we are interested in slicing on variables in the middle of a program, then we can "capture" the values of the variables at this point by assigning to a new variable, slice. Preserving the final value of slice ensures that we preserve the values of the variables of interest at the point of interest. See Section 5.5 below for the details.

This discussion suggests that the operation of slicing can be formalised as a combination of two relations: a syntactic relation (statement deletion) and a semantic relation (which shows what subset of the semantics has been preserved).

We start with a formal definition of the concept of statement deletion, by defining a syntactic relation on programs, called *reduction*.

5.1 Reduction

The reduction relation defines the result of replacing certain statements in a program by **skip**, **exit** or **abort** statements. The WSL language includes "unbounded" or "infinite" loops of the form **do** ... **od**. Such a loop can only be terminated by execution of an **exit**(n) statement: where n is a positive integer. The statement **exit**(n) causes immediate terminate of the enclosing nested n loops: so **exit**(1) will terminate the directly enclosing loop, **exit**(2) will terminate a double-nested loop, and so on. The *terminal value* of the statement **exit**(n) is n. More generally, the set of terminal values $\mathsf{TVs}(\mathbf{S})$ for any statement $\mathbf{S}$ is the set of possible loops which can be terminated by executing the statement. This can be computed by taking the set of values $n - d$ such that there is a statement **exit**(n) enclosed in d nested loops within $\mathbf{S}$ which will cause termination of $\mathbf{S}$ if executed. If $\mathbf{S}$ can be terminated by a statement other than an **exit**(n), then zero is also a terminal value for $\mathbf{S}$, i.e. $0 \in \mathsf{TVs}(\mathbf{S})$.

For example, the statement:

do if $n < 0$ **then exit**(3) **fi**;
$n := n + 1$;
if $n = 10$ **then exit**(1) **fi od**

has terminal values $\{0, 2\}$.

We define the relation $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$, read "$\mathbf{S}_1$ is a reduction of $\mathbf{S}_2$", on WSL programs as follows:

$$\mathbf{S} \sqsubseteq \mathbf{S} \quad \text{for any program } \mathbf{S}$$

$$\textbf{skip} \sqsubseteq \mathbf{S} \quad \text{for any proper sequence } \mathbf{S}$$

If $\mathbf{S}$ is not a proper sequence and $n > 0$ is the largest integer in $\mathsf{TVs}(\mathbf{S})$ then:

$$\textbf{exit}(n) \sqsubseteq \mathbf{S}$$

If $\mathsf{TVs}(\mathbf{S}) = \varnothing$ then:

$$\textbf{abort} \sqsubseteq \mathbf{S}$$

If $\mathbf{S}'_1 \sqsubseteq \mathbf{S}_1$ and $\mathbf{S}'_2 \sqsubseteq \mathbf{S}_2$ then:

$$\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}'_1 \textbf{ else } \mathbf{S}'_2 \textbf{ fi} \sqsubseteq \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

If $\mathbf{S}' \sqsubseteq \mathbf{S}$ then:

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S}' \textbf{ od} \sqsubseteq \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}$$

$$\textbf{var } \langle v := e \rangle : \mathbf{S}' \textbf{ end} \sqsubseteq \textbf{var } \langle v := e \rangle : \mathbf{S} \textbf{ end}$$

$$\textbf{var } \langle v := \bot \rangle : \mathbf{S}' \textbf{ end} \sqsubseteq \textbf{var } \langle v := e \rangle : \mathbf{S} \textbf{ end}$$

This last case will be used when the variable v is used in $\mathbf{S}$, but the initial value e is not used. Here we are using $\bot$ to denote an unused *value*: this is different from the $\bot$ in Section 4, which denotes an undefined *state*.

If $\mathbf{S}'_i \sqsubseteq \mathbf{S}_i$ for $1 \leqslant i \leqslant n$ then:

$$\mathbf{S}'_1;\ \mathbf{S}'_2;\ \ldots;\ \mathbf{S}'_n \sqsubseteq \mathbf{S}_1;\ \mathbf{S}_2;\ \ldots;\ \mathbf{S}_n$$

Note that the reduction relation does not allow actual deletion of statements: only replacing a statement by a **skip**, **exit** or **abort**. This makes it easier to match up the original program with the reduced version: the position of each statement in the reduced program is the same as the corresponding statement in the original program. Deleting any extra **skip**s is a trivial step.

In effect, we have expanded Weiser's "reduction" process into a two stage process: reduction followed by deletion of redundant **skip**s.

Three important properties of the reduction relation are:

Lemma 5.1 Transitivity: If $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$ and $\mathbf{S}_2 \sqsubseteq \mathbf{S}_3$ then $\mathbf{S}_1 \sqsubseteq \mathbf{S}_3$.

Lemma 5.2 Antisymmetry: If $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$ and $\mathbf{S}_2 \sqsubseteq \mathbf{S}_1$ then $\mathbf{S}_1 = \mathbf{S}_2$.

Lemma 5.3 The *Replacement Property*: If any component of a program is replaced by a reduction, then the result is a reduction of the whole program.

5.2 Semi-Refinement

In this subsection we will discuss the selection of a suitable semantic relation for the definition of slicing.

Initially we will consider the special case where the slicing point s is the end point of the program, but we will generalise the variable v to a set X of variables. (As we will see in Section 5.5, slicing at a point or points within the program does not introduce any further complications.) If X does not contain all the variables in the final state space of the program, then the sliced program will *not* be equivalent to the original program. However, consider the set $W \setminus X$, where W is the final state space. These are the variables whose values we are *not* interested in. By removing these variables from the final state space we can get a program which is equivalent to the sliced program. If a program $\mathbf{S}$ maps state spaces V to W, then the effect of slicing $\mathbf{S}$ at its end point on the variables in X is to generate a program equivalent to $\mathbf{S}$; $\textbf{remove}(W \setminus X)$.

Binkley et al. [6,7] define an *amorphous slice* to be a combination of a syntactic ordering (any computable, transitive, reflexive relation on programs) and a semantic requirement which is *any* equivalence relation on a projection of program semantics. In WSL terms, this suggests defining a slice of $\mathbf{S}$ on X to be any program $\mathbf{S}' \sqsubseteq \mathbf{S}$, such that:

$$\Delta \vdash \mathbf{S}';\ \textbf{remove}(W \setminus X) \approx \mathbf{S};\ \textbf{remove}(W \setminus X)$$

However, the requirement that the slice be strictly equivalent to the original program is too strict in some cases. Consider the program:

$$\mathbf{S};\ x := 0$$

If we are slicing on x then we would like to reduce the whole of $\mathbf{S}$ to a **skip**: but the two programs

$$\textbf{skip};\ x := 0;\ \textbf{remove}(W \setminus \{x\}) \qquad \text{and} \qquad \mathbf{S};\ x := 0;\ \textbf{remove}(W \setminus \{x\})$$

are only equivalent provided that $\mathbf{S}$ always terminates. But most slicing researchers see no difficulty in slicing away potentially non-terminating code: Weiser [44] says that the slice can do anything in the case where the original program fails to terminate. Any of the standard slicing algorithms which use a dependency graph [17] will delete a loop which contains no assignments to variables of interest, without making any attempt to prove termination of the loop. (In the general case, of course, there is no *algorithm* which can unequivocally determine whether an arbitrary block of code will terminate).

So, WSL equivalence is not suitable for defining program slicing. In fact, there is *no* semantic equivalence relation which is suitable for defining a useful program slice! Consider the two programs **abort** and **skip**. Any possible semantic relation must either treat **abort** as equivalent to **skip**, or must treat **abort** as not equivalent to **skip**.

1. Suppose **abort** is not equivalent to **skip**. Then the slicing relation will not allow deletion of non-terminating, or potentially non-terminating, code. So this is not suitable;
2. On the other hand, suppose **abort** is equivalent to **skip**. Then the slicing relation will allow deletion of statements which turn a terminating program into a non-terminating program. For example, in the program:

   $$x := 0;\ x := 1;\ \textbf{while}\ x = 0\ \textbf{do skip od}$$

   we could delete the statement $x := 1$ to give a syntactically smaller, semantically "equivalent" but non-terminating program. Few slicing researchers are happy to allow a non-terminating program as a valid slice of a terminating program! (Weiser [44], for example, requires the slice to terminate when the original program does).

It would appear that the concept of an "amorphous slice", as described in [7] and elsewhere, is fundamentally flawed.

Another semantic relation which has been proposed [33] is to allow any *refinement* of a program, which is also a reduction, as a valid slice. This would allow slicing away nonterminating code, since **skip** is a refinement of any nonterminating program, and would also disallow a nonterminating slice of a terminating program. But such a definition of slicing is counter-intuitive, in the sense that slicing is intuitively an *abstraction* operation (an operation which throws away information), while refinement is the opposite of abstraction. A more important consideration is that we would like to be able to analyse the sliced program and derive facts about the original program (with the proviso that the original program might not terminate in cases where the slice does). If the sliced program assigns a particular value to a variable in the slice, then we would like to deduce that the original program assigns the *same* value to the variable. But with the refinement definition of a slice, the fact that the slice sets x to 1, say, tells us only that 1 is one of the *possible* values given to x by the original program.

Consider the following nondeterministic program which we want to slice on the final value of x:

$x := 1;$
while $n > 1$ **do**
  if even?(n) **then** $n := n/2$

else $n := 3 * n + 1$ **fi od**;
if true $\rightarrow x := 1$
$\square$ **true** $\rightarrow x := 2$ **fi**

The **while** loop clearly does not affect x, so we would like to delete it from the slice. But if we are insisting that the slice be *equivalent* to the original program (on x), then we have to prove that the loop terminates for all n before we can delete it. The loop generates the Collatz sequence and it is an open question as to whether the sequence always reaches 1. (The problem was first posed by L. Collatz in 1937 [21,27]).

Allowing any refinement as a valid slice (as in [33]) would allow us to delete the **while** loop, but would also allow us to delete the **if** statement, giving $x := 1$ as a valid slice. If the slice is being computed as part of a program analysis or comprehension task, then the programmer might (incorrectly) conclude that the original program assigns the value 1 to x whenever it terminates.

(Note that one should be careful not to confuse the *definition* of a slice with an *algorithm* for computing slices. One such algorithm involves tracking data and control dependencies. But the existence of a dependency on a statement does not necessarily mean that the statement *must* be included in the slice. See Section 5.12.1 for an example).

These considerations led to the development of the concept of a *semi-refinement*:

Definition 5.4 A *semi-refinement* of $\mathbf{S} : V \rightarrow W$ is any program $\mathbf{S}' : V \rightarrow W$' such that

$$\Delta \vdash \mathbf{S} \;\approx\; \{\mathrm{WP}(\mathbf{S}, \mathbf{true})\};\ \mathbf{S}'$$

It is denoted $\Delta \vdash \mathbf{S} \;\preccurlyeq\; \mathbf{S}'$.

A semi-refinement can equivalently be defined in terms of the weakest preconditions:

$$\begin{array}{rlll} \Delta \vdash \mathbf{S} \;\preccurlyeq\; \mathbf{S}' & \text{iff} & \Delta \vdash \mathrm{WP}(\mathbf{S}, \mathbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}', \mathbf{true}) \\ & \text{and} & \Delta \vdash \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \Leftrightarrow (\mathrm{WP}(\mathbf{S}, \mathbf{true}) \;\wedge\; \mathrm{WP}(\mathbf{S}', \mathbf{x} \neq \mathbf{x}')) \end{array}$$

where $\mathbf{x}'$ is a list of the variables in W.

The assertion in the semi-refinement relation shows that program equivalence is only required where the original program terminates. So we define a slice of $\mathbf{S}$ on X to be any reduction of $\mathbf{S}$ which is also a semi-refinement:

Definition 5.5 A *Syntactic Slice* of $\mathbf{S} : V \rightarrow W$ on a set $X \subseteq W$ of variables is any program $\mathbf{S}' : V \rightarrow W$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

$$\Delta \vdash \mathbf{S};\ \mathbf{remove}(W \setminus X) \;\preccurlyeq\; \mathbf{S}';\ \mathbf{remove}(W \setminus X)$$

The assertion $\{\mathrm{WP}(\mathbf{S}, \mathbf{true})\}$ is a **skip** whenever $\mathbf{S}$ is guaranteed to terminate and an **abort** whenever $\mathbf{S}$ aborts. So in the case when $\mathbf{S}$ aborts, $\mathbf{S}'$ can be anything: in particular, setting $\mathbf{S}'$ to **skip** will trivially satisfy $\mathbf{S}' \sqsubseteq \mathbf{S}$. So this definition allows us to slice away nonterminating code while also preserving the nondeterministic behaviour of terminating code.

Semi-refinement also allows deletion of any assertions:

Lemma 5.6 $\{\mathbf{Q}\} \;\preccurlyeq$ **skip**

Proof: $\mathrm{WP}(\{\mathbf{Q}\}, \mathbf{true}) = \mathbf{Q}$ so $\{\mathbf{Q}\} \;\approx\; \{\mathrm{WP}(\{\mathbf{Q}\}, \mathbf{true})\};$ **skip** ■

If $\Delta \vdash \mathbf{S} \;\preccurlyeq\; \mathbf{S}'$ then $\Delta \vdash \mathbf{S} \;\leq\; \mathbf{S}'$ (since deleting an assertion is a valid refinement), but the converse does not hold. So the relationship lies somewhere between a refinement and an equivalence.

As with refinement and reduction, semi-refinement also satisfies the *replacement property*: if any component of a program is replaced by a semi-refinement then the result is a semi-refinement of the whole program.

With this definition of slicing, a slice can be computed purely by applying program transformation operations which duplicate and move the **remove** statements through the program and then use the **remove** statements to transform components of the program to **skip** statements.

5.3 Slicing Unstructured Programs

Note that we can apply arbitrary transformation in the process of slicing, provided that the final program satisfies all the conditions of Definition 5.5 (in particular, the slice is a reduction of the original program: in other words, it can be constructed from the original program by replacing statements by **skip**s, **exit**s or **abort**s). So we can implement a slicing algorithm as the sequence:

$$\mathsf{transform} \;\rightarrow\; \mathsf{reduce} \;\rightarrow\; \mathsf{transform}$$

provided that the reduction step is also a semi-refinement and the final transformation step "undoes" the effect of the initial transformation. This step is facilitated by the fact that the reduction relation preserves the positions of sub-components in the program. In practice, the final transform step is implemented by tracking the movement of components in the initial transform step, noting which components are reduced in the reduce step and replacing these by **skip**s directly in the original program.

In [40] we derive an algorithm for slicing structured WSL programs from a specification of slicing. The above discussion suggests that an algorithm for slicing structured WSL can easily be extended to unstructured code: first restructure the program to use only **if** statements and **while** loops (for example, using Bohm and Jacopini's algorithm [8]), then slice the structured program, then determine which simple statements have been reduced, and apply the corresponding reduction to the original program (or equivalently, undo the restructuring transformation).

The simplest restructuring algorithm converts the whole program to a single **while** loop whose body is a multi-way conditional controlled by a single variable next which stores the number of the next block to be executed:

while next $\neq 0$ **do**
  if next $= 1$ **then** $\mathbf{S}_1$; **if** $\mathbf{B}_1$ **then** next $:= n_{11}$ **else** next $:= n_{12}$ **fi**
  ...
  elsif next $= i$ **then** $\mathbf{S}_i$; **if** $\mathbf{B}_i$ **then** next $:= n_{i1}$ **else** next $:= n_{i2}$ **fi**
  ... **fi od**

At the end of each branch of the conditional, next is conditionally or unconditionally assigned an integer value representing transfer of control to that block. Block i will be followed by block n_{i1} or n_{i2} depending in the result of the condition $\mathbf{B}_i$.

If such a program is sliced on any variable assigned anywhere in the program, then next will have to be included in the slice (since every statement is control dependent on next). All the variables in all of the $\mathbf{B}_i$ conditions are control variables for next (because next is control dependent on all these variables), so these variables will be included in the slice. Then all statements which assign to any variables that affect any $\mathbf{B}_i$ will also be included in the slice. This is likely to be most of the program.

FermaT's solution is to *destructure* the program to an action system in "basic blocks" format. Each basic block consists of a sequence of atomic statements (such as assignments, comments and procedure calls) ending with either an unconditional call to the next block, or a conditional statement with a call in each branch of the conditional. The action system is therefore a *regular* action system which can only terminate via a **call** Z statement. Within each block we allow at most one assignment to each variable.

To slice the action system, FermaT computes the Static Single Assignment (SSA) form of the program, and the control dependencies of each basic block using Bilardi and Pingali's optimal algorithms [5,23]. FermaT tracks control and data dependencies to determine which statements can

be deleted from the blocks. Tracking data dependencies is trivial when the program is in SSA form, and the control dependencies are FermaT links each basic block to the corresponding statement in the original program, so it can determine which statements from the original program have been deleted (in effect, this will "undo" the destructuring step). This algorithm is implemented as the Syntactic_Slice transformation in FermaT.

5.4 Semantic and Amorphous Slicing

The definition of a syntactic slice immediately suggests a generalisation: why not keep the semantic relation and drop the syntactic relation? In other words, why not drop the requirement that $\mathbf{S}' \sqsubseteq \mathbf{S}$?

The relation between a WSL program and its semantic slice is a purely semantic one: compare this with a "syntactic slice" where the relation is primarily a syntactic one with a semantic restriction.

Definition 5.7 A *semantic slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that:

$$\Delta \vdash \mathbf{S};\ \mathbf{remove}(W \setminus X) \preccurlyeq \mathbf{S}';\ \mathbf{remove}(W \setminus X)$$

Note that while there are only a finite number of different syntactic slices (if $\mathbf{S}$ contains n statements then there are at most 2^n different programs $\mathbf{S}'$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$) there are infinitely many possible semantic slices for a program: including slices which are actually *larger* than the original program. Although one would normally expect a semantic slice to be no larger than the original program, there are cases where a high-level abstract specification can be larger than the program while still being arguably easier to understand and more useful for comprehension and debugging. A program might use some very clever coding to re-use the same data structure for more than one purpose. An equivalent program which internally uses two data structures might contain more statements and be less efficient while still being easier to analyse and understand. See [34] and [35] for a discussion of the issues and [34] for an example.

Semantic refinement is implemented in FermaT via a process of *abstraction* and *refinement*. The *Representation Theorem* of WSL shows that for *any* WSL program there exists a WSL specification (containing a single specification statement) which implements that program:

Theorem 5.8 *The Representation Theorem*
Let $\mathbf{S}\colon V \to W$, be any non-null WSL statement and let $\mathbf{x}$ be a list of all the variables in W. Without loss of generality we may assume that $W \subseteq V$ (Any variables added by $\mathbf{S}$ are already in the initial state space). Let $\mathbf{y}$ be a list of the variables removed by $\mathbf{S}$, so $\mathbf{x} \cap \mathbf{y} = \varnothing$ and $\mathbf{x} \cup \mathbf{y} = V$. Then

$$\Delta \vdash \mathbf{S} \approx \mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \wedge \mathrm{WP}(\mathbf{S}, \mathbf{true}));\ \mathbf{remove}(\mathbf{y})$$

A WSL statement is non-null if the set of final states is non-empty for every initial state. Equivalently, a non-null program satisfies Dijkstra's *Law of the Excluded Miracle*:

$$\mathrm{WP}(\mathbf{S}, \mathbf{false}) \Leftrightarrow \mathbf{false}$$

The WSL language is so designed that any WSL program which excludes "naked" guard statements (i.e. guard statements which are not part of an assignment, **if** statement, specification etc.) is guaranteed to be non-null.

In [37] we describe a partial implementation of the representation theorem in the form of a program transformation called Prog_To_Spec. This is used in combination with a syntactic slicing algorithm plus other transformations to develop a powerful conditioned semantic slicing algorithm. A *conditioned slice* [9,15] is a program slice which makes use of assertions added to the code to simplify the slice. The slicer applies the abstraction transformation Prog_To_Spec to blocks of code which do not contain loops, it then uses FermaT's condition simplifier to simplify the resulting

specification. The simplifier can make use of assertions to simplify the specification, thus generating conditioned slices. A syntactic slicing algorithm is applied to the resulting program (with some semantic slicing extensions). Further simplification transformations, such as Constant_Propagation, are applied and any remaining specification statements are refined (using the Refine_Spec transformation) into combinations of assertions, assignments and **if** statements, where possible.

The abstraction and refinement process can be used to simplify statements based on preceding *and subsequent* assertions: semantic slices can delete code which would falsify a later assertion if executed. This will be particularly important when we consider Conditioned Slicing in more detail in Section 5.8.

5.5 Slicing At Any Position

To slice at an arbitrary position in the program we need to preserve the sequence of values taken on by the given variables at that point in the program. To do this, we simply insert an assignment to a new variable slice at the required position which records the current values of the variables on a list. If $X = \{x_1, \ldots, x_n\}$ is the set of variables we are interested in then we insert the statement:

$$\mathsf{slice} := \mathsf{slice} +\!\!+ \langle\langle x_1, \ldots, x_n \rangle\rangle$$

at the point of interest, in order to record the current values of the variables at that point. Then we slice at the end of the program on the single variable slice. (Note that we *append* to the list in slice, rather than simply assign to it, because we are interested in the whole sequence of values, not just the last set of values taken on by the variables.)

This process can be generalised to slicing at several points in the program, perhaps with a different set of "variables of interest" at each point, simply by inserting the slice assignments at the appropriate places.

One peculiarity of this definition is that if we slice at a point in the program which is within a compound statement that does not modify any of the variables in the slicing criteria, then we can end up with slices which appear to be larger than necessary. For example, suppose that we slice on x within this **if** statement at the point just before the assignment to z on line 3:

(1) $x := g(z)$;
(2) $y := f(z)$;
(3) **if** $y = 0$ **then** $z := 1$ **fi**

The annotated program is:

(1) $x := g(z)$;
(2) $y := f(z)$;
(3) **if** $y = 0$ **then** $\mathsf{slice} := \mathsf{slice} +\!\!+ \langle\langle x \rangle\rangle$; $z := 1$ **fi**;
(4) **remove**(x, y, z)

This is equivalent to:

(1) $x := g(z)$;
(2) $y := f(z)$;
(3) **if** $y = 0$ **then** $\mathsf{slice} := \mathsf{slice} +\!\!+ \langle\langle x \rangle\rangle$ **fi**;
(4) **remove**(x, y, z)

We cannot delete the assignment $y := f(z)$ on line 2 because it determines which branch of the **if** statement is taken, and this affects the final value of slice (although it does not affect the value of x). According to our definition, the slice has to preserve the test $y = 0$ and therefore preserve any previous modifications to y. In effect, by slicing at a particular position we are insisting that the given *position* should also appear in the sliced program. This is arguably correct in the sense that, if the slice has to preserve the sequence of values taken on by x at a particular point in the

program, then a *corresponding point* (at which x takes on the *same* sequence of values) must appear in the slice. But if the **if** statement in the above example is deleted, then x may take on a different sequence of values! To be precise, there is *no* point in the new program at which x takes on the *same* sequence of values as at the slice point in the original program.

However, if it is not required to preserve the slice point then a simple solution is to allow the slicing algorithm to move all the assignments to slice upwards out of any enclosing structures as far as possible, before carrying out the slicing operation itself.

5.6 Dynamic Slicing

Although the term "dynamic program slice" was first introduced by Korel and Laski [20], it may be regarded as a non-interactive version of Balzer's notion of flowback analysis [3]. In flowback analysis, one is interested in how information flows through a program to obtain a particular value: the user interactively traverses a graph that represents the data and control dependences between statements in the program.

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. We can define this initial state by means of an assertion. Suppose $V = \{v_1, v_2, \ldots, v_n\}$ is the set of variables in the initial state space for **P**, and V_1, V_2, ..., V_n are the initial values of these variables in the state of interest. Then the condition

$$\mathbf{A} =_{\mathrm{DF}} v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n$$

is true for this initial state and false for every other initial state. The assertion $\{\mathbf{A}\}$ is **abort** for every initial state other than the specified one. So we define:

Definition 5.9 A *Dynamic Syntactic Slice* of **S** with respect to a formula **A** of the form

$$v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n$$

where $V = \{v_1, v_2, \ldots, v_n\}$ is the initial state space of **S** and V_i are constants, and the set of variables X is a subset of the final state space W of **S**, is any program $\mathbf{S}' \sqsubseteq \mathbf{S}$ such that:

$$\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \mathbf{remove}(W \setminus X) \preccurlyeq \{\mathbf{A}\};\ \mathbf{S}';\ \mathbf{remove}(W \setminus X)$$

5.7 Dynamic Slicing of Assembler Modules

Traditional dynamic slicing algorithms incur a high runtime overhead due to the tracing and dataflow information that is collected during the program's execution. In [28] the authors describe a method for computing statement coverage (i.e. determining which statements are executed during one run of the program) for various microprocessors (Alpha, Sparc, Power, Mips, IA-64 and x86). This involves replacing certain instructions by a branch to a base trampoline which can call mini-trampolines both before and after executing the relocated instruction. The instrumentation code (but not the base trampoline) can be removed when it is no longer required: even so, the slowdown in execution speed can still be a factor of two or more.

We have developed a method for recording execution paths in IBM mainframe code which has virtually zero overhead. This method works as follows:

1. Insert breakpoints at the start of each basic block in the assembler module. FermaT already computes all the potential targets of branch instructions (these include branch to register instructions) as part of the assembler to WSL translation.
2. The breakpoint handler records that this basic block was executed, restores the original instruction, and branches back into the program.
3. Subsequent executions of the same basic block will therefore execute at full speed with no performance penalty.

With this slicing technique the overhead is proportional to the size of the program, and not to the execution time.

The resulting slice will include all executed code: including code which does not affect the variables of interest. However, a subsequent static slice has the opportunity to remove this code.

As with any slicing algorithm, it is easy to find cases which are not optimal. (It can be proved that there is no optimal slicing algorithm, therefore this is true for *any* possible slicing algorithm!) For example, consider the following loop:

```
while B1 do
  if B2 then x := p fi;
  S;
  if B3
    then if x = 1 then S1 else S2 fi fi od
```

Suppose that on every iteration, if $\mathbf{B}_2$ is true in the first **if** statement, then $\mathbf{B}_3$ is true in the second, and if $\mathbf{B}_2$ is false then $\mathbf{B}_3$ is also false. Suppose that the only occurrences of x in the loop are those given explicitly. Then the dynamic slice will include all the statements, and the subsequent static slice will conclude that the initial value of x is required. A traditional dynamic slice which recorded dataflow information for every iteration of the loop would be able to determine that the value of x in the test $x = 1$ in every iteration in which the test is executed comes from the assignment $x := p$ and therefore that there are no dataflows before the loop, and that assignments to x before the loop body are not needed.

These cases are rare in practice, but if a more accurate slice is required then it can be achieved using our algorithm by expanding the first **if** statement over the next two statements:

```
while B1 do
  if B2
    then x := p;
         S;
         if B3
           then if x = 1 then S1 else S2 fi fi
   else S;
         if B3
           then if x = 1 then S1 else S2 fi fi fi od
```

Now the dynamic slice produces this result:

```
while B1 do
  if B2
    then x := p;
         S;
         if B3
           then if x = 1 then S1 else S2 fi fi
   else S;
         {¬B3} fi od
```

and the subsequent static slice deduces that the initial value of x is not required.

5.8 Conditioned Slicing

Researchers have generalised dynamic slicing and combined static and dynamic slicing in various ways. For example: some researchers allow a finite set of initial states, or a *partial* initial state which restricts a subset of the initial variables to particular values [30]. In our formalism, all of these generalisations are subsumed under the obvious generalisation of dynamic slicing: why restrict the initial assertion to be of the particular form $\{v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n\}$?

If we allow any initial assertion, then the result is called a *conditioned slice*:

Definition 5.10 A *Conditioned Syntactic Slice* of $\mathbf{S}$ with respect to any formula $\mathbf{A}$ and set of variables X is any program $\mathbf{S}' \sqsubseteq \mathbf{S}$ such that:

$$\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \mathsf{remove}(W \setminus X) \ \preccurlyeq\ \{\mathbf{A}\};\ \mathbf{S}';\ \mathsf{remove}(W \setminus X)$$

Conditioned slicing is thus a generalisation of both static slicing (where the condition $\mathbf{A}$ is **true**) and dynamic slicing (where $\mathbf{A}$ takes on the form $v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n$ and the set $\{v_1, v_2, \ldots, v_n\}$ lists all the variables in the program). One algorithm for computing a conditioned slice is to use the initial condition to simplify the program before applying a syntactic slicing algorithm. Danicic et al [15] describe a tool called ConSIT, for slicing a program at a particular point, given that the initial state satisfies a given condition.

The ConSIT tool works on an intraprocedural subset of C using a three phase approach:

1. Symbolically Execute: to propagate assertions through the program where possible;
2. Produce Conditioned Program: eliminate statements which are never executed under the given conditions;
3. Perform Static Slicing: using a traditional (syntactic) slicing method.

In ConSIT, the slicing condition can be given in the form of `ASSERT` statements scattered through the program: the authors [15] claim that these `ASSERT` statements are equivalent to a single condition on the initial state, but in general this requires assertions to be formulae of *infinitary* logic. This is because the general case of moving an assertion "backwards" over or out of a loop breaks down into a countably infinite sequence of cases depending on the number of possible iterations of the loop. Fortunately, the assertion statements in WSL are already expressed in infinitary logic, so this is not a problem in our framework.

In our transformation framework, the `ASSERT` statements are simply WSL assertions. Symbolic execution and producing a conditioned program (by using assertions to delete statements which cannot be executed) are examples of transformations which can be applied to the WSL program. In [31] we provide a number of transformations for propagating assertions and eliminating dead code.

Theorem 5.11 *A set of assertions scattered through a program can be replaced by an equivalent assertion at the beginning of the program (in the sense that the two programs are equivalent).*

Proof: Let $\mathbf{S}$ be any program and let $\mathbf{S}'$ be constructed from $\mathbf{S}$ by deleting assertions (i.e. replacing the assertions with **skip** statements). Each assertion deletion is a semi-refinement, so by the Replacement Property (Section 5), $\mathbf{S}$' is a semi-refinement of $\mathbf{S}$. So, by the definition of semi-refinement (Definition 5.4):

$$\Delta \vdash \mathbf{S} \ \approx\ \{\mathrm{WP}(\mathbf{S}, \mathbf{true})\};\ \mathbf{S}'$$

which proves the theorem. ■

5.9 Conditioned Semantic Slicing

Again, a generalisation is suggested: why restrict ourselves to the assertion moving and dead code removal transformations of ConSIT? A conditioned semantic slice can be defined by simply removing the syntactic condition from the definition of a syntactic slice (i.e. the condition that $\mathbf{S}' \sqsubseteq \mathbf{S}$):

Definition 5.12 Suppose we have a program $\mathbf{S} : V \to W$ and a slicing criterion, defined from $\mathbf{S}$ by inserting assertions and assignments to the slice variable to form $\mathbf{S}'$. A *conditioned semantic slice* of $\mathbf{S}$ with respect to this criterion is any program $\mathbf{S}''$ such that:

$$\Delta \vdash \mathbf{S}';\ \mathsf{remove}(W) \ \preccurlyeq\ \mathbf{S}'';\ \mathsf{remove}(W)$$

Any syntactic slice is also a semantic slice (but not vice versa), so the conditioned semantic slice is a generalisation of syntactic, semantic, dynamic, conditioned and operational slicing in the sense that any of these slices is also a conditioned semantic slice.

5.10 Transformations Involving Abort and Assertions

A particularly useful form of conditioned slicing, which is especially effective when applied to commercial assembler systems, is the removal of error handling code. In a typical commercial assembler module, much of the code involves testing for and handling errors: this can amount to more than half of the lines of code in the whole module. This error handling code is scattered throughout the program and can make it much more difficult to work out what the program actually does under normal (non-error) operation.

The basic approach is to insert an **abort** at any point in the module where it is known for certain that we are in error handling code. For commercial assembler systems there are three cases to consider:

1. Any `ABEND` instruction can be replaced by **abort**, since an `ABEND` causes an abnormal termination of the program. These are trivial to implement: we simply change the translation of `ABEND` in the translation table to be an **abort** statement;
2. Any macro which expands to an unconditional `ABEND` and where FermaT is translating the macro directly rather than translating the macro expansion should also be replaced by an **abort**. Again this is a trivial change to the translation table, and these macros are well known: since they are already in the translation table;
3. A call to an external module which handles error processing. This is quite rare in commercial assembler systems since error processing is handled separately by each module: common code is usually handled via a macro rather than a call. If there are external modules which handle error processing, then these will be familiar to the maintenance programmers, and again the solution simply involves an addition to the translation table. (A `CALL` to an external module will be translated directly if the module name is found in the translation table).

The *translation table* is a text file which lists all the assembler instructions with their corresponding WSL translations. Macros and external calls can also be included in the translation table. If a macro is found in the table, then the corresponding WSL code is generated and the macro expansion is skipped. Otherwise, the macro expansion is translated.

The following transformations can then be applied to remove the error handling code:

Theorem 5.13 $\Delta \vdash (\mathbf{S};\ \textbf{abort}) \approx \textbf{abort} \approx (\textbf{abort};\ \mathbf{S})$

Theorem 5.14 $\Delta \vdash \textbf{if } \mathbf{B} \textbf{ then abort else } \mathbf{S} \textbf{ fi} \approx \{\neg\mathbf{B}\};\ \mathbf{S}$

Theorem 5.15 $\Delta \vdash \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S} \textbf{ else abort fi} \approx \{\mathbf{B}\};\ \mathbf{S}$

FermaT will also automatically unfold any procedure whose body consists of an **abort** or an assertion: so a procedure which simply checks for and handles error conditions will be transformed to an assertion and unfolded everywhere.

To see how this works in practice, consider the following typical, but somewhat abbreviated, error checking situation. The main program, or a subroutine, tests for an error and branches to a specific error handler:

if r3 = 0 **then** ERR123() **fi**

The specific error handler sets an error code, prints a message, and calls the generic error handler:

proc ERR123() ≡ *code* := 123; WTO("`Error message...`"); GENERR()**.**

The generic error handler executes cleanup code (closing files, perhaps creating a memory dump etc.) and then finally does an `ABEND`:

proc GENERR() $\equiv$ cleanup_code; ...; ABEND; **abort.**

FermaT applies the above transformations repeatedly in a loop. The body of GENERR is transformed to a single **abort**. All calls to GENERR are unfolded. The body of ERR123 is then transformed to an **abort** and unfolded everywhere. Finally, the **if** statement in the mainline code is simplified to the assertion $\{\mathsf{r3} \neq 0\}$.

5.11 A Minimal Semantic Slice

Recall that any non-null program $\mathbf{S} : V \to W$ where $V \subseteq W$ is equivalent to the specification

$$\mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \ \wedge \ \mathrm{WP}(\mathbf{S}, \mathbf{true}))$$

where $\mathbf{x}$ is a list of all the variables modified in $\mathbf{S}$. The variables in $\mathbf{x}'$ do not appear in $\mathrm{WP}(\mathbf{S}, \mathbf{true})$ so this is equivalent to:

$$\{\mathrm{WP}(\mathbf{S}, \mathbf{true})\}; \ \mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$$

Then, by the definition of semi-refinement:

$$\mathbf{S} \preccurlyeq \mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$$

This is clearly a *minimal* semantic slice (counting statements) since it only contains a single statement, and by definition no WSL program can be smaller than a single statement. (It is not necessarily minimal if we are counting the total number of symbols: if statement $\mathbf{S}$ contains loops or recursion then the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ is infinitely long!) So we have:

Theorem 5.16 *The Minimal Semantic Slice Theorem*
Let $\mathbf{S}$, be any null-free statement and let $\mathbf{x}$ be any list of variables in the final state space. Then the statement $\mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$ is a minimal semantic slice of $\mathbf{S}$ on the final values of $\mathbf{x}$.

This may appear to contradict Weiser's theorem on the non-computability of minimal slices, but Weiser's theorem only applies to algorithms for computing minimal *syntactic* slices. The construction of $\mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$ from $\mathbf{S}$, while being well defined, is not an algorithm in the usual sense because the formula $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}')$ may be infinitely long. An infinite specification statement is not directly executable, so this result is only practical for statements which contain no loops or recursion, but it does show that *no semantic slice need be larger than a single statement.*

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 [29] 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. He suggests that such a slicer ought to be capable of producing the following slice:

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

With semantic slicing we can, of course, produce a smaller slice. Theorem 5.16 gives the following slice:

$y := y'.(\neg\mathrm{WP}(\textbf{if } p = q \textbf{ then } x := 18$
$\qquad\qquad\qquad\qquad \textbf{else } x := 17 \textbf{ fi};$
$\qquad\qquad \textbf{if } p \neq q \textbf{ then } y := x$
$\qquad\qquad\qquad\qquad \textbf{else } y := 2 \textbf{ fi}, y \neq y'))$

This simplifies to:

$y := y'.(\neg\mathrm{WP}(\textbf{if } p = q \textbf{ then } x := 18$
$\qquad\qquad\qquad\qquad \textbf{else } x := 17 \textbf{ fi},$
$\qquad\qquad (p \neq q \Rightarrow y' \neq x)$
$\qquad\qquad\ \wedge\ (p = q \Rightarrow 2 \neq y')))$

which in turn simplifies to:

$y := y'.(\neg((p = q \Rightarrow 2 \neq y')$
$\qquad\qquad \wedge\ (p \neq q \Rightarrow 17 \neq y')))$

which is equivalent to:

$y := y'.((p = q \Rightarrow 2 = y')\ \wedge\ (p \neq q \Rightarrow 17 = y'))$

This can be expressed as a simple assignment on a conditional expression:

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

FermaT's semantic slicer produces the equivalent **if** statement:

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

5.11.1 FermaT Implementation of Abstraction and Refinement

For WSL programs with no loops or recursion (and where all the formulae are finite). Theorem 5.16 *does* give an algorithm for computing a minimal semantic slice on any given slicing criterion.

We have implemented a function @WP in the FermaT transformation system which computes the weakest precondition for any program which does not include loops or procedure calls. (The implementation could be extended to non-recursive procedures and functions in the obvious way: by unfolding all procedures and functions in the main body of the program). With the aid of this function, we have implemented a transformation called Prog_To_Spec which can transform any non-recursive and non-iterative program into an equivalent specification statement. The implementation of @WP required less than 100 lines of $\mathcal{META}$WSL code, and the body of Prog_To_Spec is only 32 lines of code (including comments), demonstrating that $\mathcal{META}$WSL is the ideal language for implementing program transformations! (See Section 6.3 for a brief description of $\mathcal{META}$WSL).

In the rest of the paper, all slicing examples were computed by FermaT in a single step and the output copied into the paper.

Applying Prog_To_Spec to the assignment

$x := 2 * x + 1$

gives the specification:

$x := x'.(x' = 2 * x + 1)$

Another example:

if true $\rightarrow p := 1$
$\square$ **true** $\rightarrow p := 2$ **fi**

gives:

$p := p'.(p' = 1\ \vee\ p' = 2)$

The statement to be specified may include assertions, local variables, nested **if** statements and so on. FermaT's simplifier will use the assertions to simplify other parts of the generated specification, eliminate local variables and so on, automatically. For example:

var $\langle x := y \rangle$:
  if $p > q$ **then** $x := x + 2$
    else $x := x - 2$ **fi**;
  $\{x = 10\}$ **end**

is transformed to the assertion:
$\{y = 8 \ \wedge \ p > q \ \vee \ y = 12 \ \wedge \ p \leqslant q\}$

Note that the assertion $\{x = 10\}$ has been used to simplify the preceding code.

A simple **if** statement such as:
if $x > y$ **then** $z := 1$ **else** $z := 2$ **fi**

is transformed to the specification:
$z := z'.(z' = 1 \ \wedge \ x > y \ \vee \ z' = 2 \ \wedge \ x \leqslant y)$

while a nested **if** statement such as:
if $x = 1$ **then** $y := 2$
elsif $x = 2$ **then** $y := 3$
  else $y := 4$ **fi**

becomes:
$y := y'.(y' = 4 \ \wedge \ x \neq 1 \ \wedge \ x \neq 2$
  $\vee \ y' = 2 \ \wedge \ x = 1$
  $\vee \ y' = 3 \ \wedge \ x = 2)$

Generally, programmers find that a compound statement with assertions, **if** statements and simple assignments to be easier to read and understand than the equivalent single specification statement. So we have implemented another transformation Refine_Spec which analyses a specification statement and carries out the following operations:

1. Factor out any assertions;
2. Expand into an IF statement: for example, the specification $\mathbf{x} := \mathbf{x}'.(\mathbf{Q} \ \vee \ (\mathbf{B} \ \wedge \ \mathbf{P}))$ where $\mathbf{B}$ does not contain any variables in $\mathbf{x}'$, is equivalent to:

   if $\mathbf{B}$ **then** $\mathbf{x} := \mathbf{x}'.(\mathbf{Q}' \ \vee \ \mathbf{P}')$ **else** $\mathbf{x} := \mathbf{x}'.\mathbf{Q}''$ **fi**

   where $\mathbf{Q}'$ and $\mathbf{P}'$ are the result of simplifying $\mathbf{Q}$ and $\mathbf{P}$ under the assumption that $\mathbf{B}$ is true, and $\mathbf{Q}''$ is the result of simplifying $\mathbf{Q}$ under the assumption that $\mathbf{B}$ is false. These sub-specifications are then recursively refined;
3. Finally, any simple assignments or parallel assignments are extracted.

For example, the statement:
var $\langle x := x \rangle$:
  if $p = q$
    then $x := 18$
    else $x := 17$ **fi**;
  if $p \neq q$
    then $y := x$
    else $y := 2$ **fi end**

is abstracted to the specification:
$y := y'.(y' = 2 \ \wedge \ p = q \ \vee \ y' = 17 \ \wedge \ p \neq q)$

Applying Refine_Spec produces:
if $p = q$ **then** $y := 2$ **else** $y := 17$ **fi**

The above example shows one way in which abstraction and refinement can be applied to construct a semantic slice: simply convert all the assigned variables that we do *not* want to slice on (x in this case) into local variables, and apply the abstraction and refinement transformations to the result!

5.12 Slicing Programs Containing Loops

As explained above, the simple approach of abstraction to a specification followed by refinement, cannot be applied to programs containing loops or recursion. This is not such a crippling disadvantage as it might seem: many large commercial systems actually contain very few loops, compared to the total number of lines of code. For example: we analysed a fairly large commercial assembler system and found that over 55% of the modules (1,083 out of 1,945 code modules) contained *no* loops or recursion. Many modules contained one or two loops: in fact there was a total of 2,287 loops in the whole system (averaging 1.18 loops per module).

In a selection of 1749 modules supplied by twelve different organisations, 35% of the modules contained no loops or recursion. (This selection was partly biased towards larger code modules since much of the code was from pilot projects to test the capabilities of the migration engine).

FermaT's semantic slicer is *not* restricted to programs with no loops. This is because abstraction and refinement forms only one component of the general slicing algorithm. Abstraction and refinement is applied to fragments of the program (such as the bodies of the innermost loops) that are loop free.

Another feature of many commercial systems is that many of the assignment statements assign a constant value: either a numeric or string constant, or a value which depends on variables which are not assigned in the module, or at least in the enclosing loop statement. For example, in the system mentioned earlier, 44% of all assignment statements are of constant values.

If a variable in the slicing criterion is assigned in the body of a loop, then the loop itself must be included in any syntactic slice. With a dependence-based slicing algorithm, including the loop in the slice may also bring in other statements which are not actually needed.

However, if the assignment is of a constant value then only the first execution of the assignment statement is needed, since any further assignments will give the variable the same value. So we are only interested in the *first* iteration in which the assignment is executed. In particular, if the assignment is executed on the first iteration of the loop, then we can unroll the first iteration and "slice away" the rest of the loop.

5.12.1 Speculative Unrolling

This is the motivation for the "speculative unrolling" transformation which is another component of FermaT's semantic slicer:

1. If a loop is encountered in which a variable in the slicing criterion is only assigned a constant value, then unroll the first iteration of the loop;
2. Next, apply simplification transformations to the result. These will delete subsequent redundant assignments to the variable;
3. Next, recursively apply Semantic_Slicing to the result (with speculative unrolling disabled). This will delete the rest of the loop if there are no further assignments to variables in the slicing criterion;
4. If the result is smaller than the original (either in statement count, or the same number of statements but fewer expressions), then keep it, otherwise restore the original loop.

A simple example to test speculative unrolling is the following program:

while $p?(i)$ **do**

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

Table 2: Control and data dependencies in the example program

if $q?(c)$
  then $x := f;\ c := g$ **fi**;
$i := h(i)$ **od**

where we are slicing on the final value of x. This program is based on an example in [11].

Any dataflow-based slicing algorithms (such as [17]) 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 2 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. For example, the FermaT syntactic slicer will return the whole program when asked to slice on the final value of x since it uses just such a dependency tracking algorithm.

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

It then applies Fully_Absorb_Right to the inner **if** statement:

if $p?(i)$
  then if $q?(c)$
    then $\{q?(c)\}$;
      $\langle c := g, x := f \rangle$;
      $i := h(i)$;
      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
```

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 simple 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 simple 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:

$$\textbf{if } p?(i) \ \wedge \ q?(c) \textbf{ then } x := f \textbf{ fi}$$

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.

In the general case, the constant assignment may not occur on the first iteration of the loop. Here, we need to determine on which iteration of the loop the assignment takes place (if at all). If we can analyse the loop body and determine the condition under which the assignment takes place, then we can apply a general purpose transformation, Entire_Loop_Unrolling to split the loop into two.

Theorem 5.17 *Entire Loop Unrolling: for any statement* **S** *and any conditions* **B** *and* **Q***:*

$$\Delta \vdash \textbf{while B do S od} \ \approx \ \textbf{while B} \wedge \textbf{Q do S od}; \textbf{ while B do S od}$$

This transformation is valid for *any* **while** loop and *any* condition **Q**. Let **Q** be the negation of the condition under which the assignment takes place. Then, the assignment will never take place in the first **while** loop. If **B** is true on termination of the first **while** loop, then **Q** must be false, and the first iteration of the second loop will assign to x. Any remaining iterations of the second loop can only assign the same value to x, so these are redundant and can be transformed away.

We call this technique "Speculative Entire Unrolling".

6 Assembler Abstraction and Analysis

The next part of this paper describes the application of WSL and transformation theory to the analysis of assembler programs. As discusses in Section 3, the translation and analysis of an assembler modules is carried out in four phases:

1. Translation of the assembler to WSL;
2. Translate and restructure data declarations;
3. Apply semantics-preserving WSL to WSL transformations;
4. (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.

These phases are described in the following sections.

6.1 Assembler to WSL Translation

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

Perfect correctness is not possible: any scientific model of something *must* be an imperfect representation (i.e. an approximation): otherwise it would not be a model, but the thing itself! In modelling assembler code, consider the following situation: a bug in the program results in an out of range value being passed to a jump table. The assembler computes the address to branch to: this happens to be in the middle of another instruction (rather than being one of the branches in the jump table). But the "instruction" at that point is benign and control eventually reaches the right

place without doing irreparable damage. In one sense the program is "correct" (it works perfectly), but in another sense, there is a bug waiting to happen. The only way such a program can be modelled is via some form of emulator which keeps track of the contents of each memory location (including the memory holding the program's executable code) and interprets the execution of each instruction.

Our approach therefore is to model as accurately as possible everything that *can* be modelled, and to detect and flag cases that cannot be modelled.

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

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

6.2 Data Restructuring and Translation

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

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

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

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

6.3 WSL to WSL Transformation

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

The transformation engine is based on the WSL transformation theory, as described in the first part of this paper, which provides methods to prove the correctness of a WSL to WSL transformation. As a result we can have a very high degree of confidence in the accuracy of the results, despite the fact that an average of over 1,800 transformations are applied to *each* assembler

module during the migration process. If the average transformation were "only" 99.9% accurate, then after applying 1,800 transformations the probability of a correct result falls to about 16.5%. So it is vitally important to be confident of the correctness of each transformation step.

Most transformations fit into one of the two main classes:

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

There are a total of 148 different transformations implemented in FermaT (not all of which are used for assembler migrations). These are implemented in $\mathcal{META}$WSL: an extension of WSL specifically designed for writing program transformations. See [38] for a description of $\mathcal{META}$WSL. The transformations range from 29 lines of $\mathcal{META}$WSL code to 3,381 lines with a median size of 105 lines: so most transformations are quite simple and easy to prove correct.

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

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

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

With simple assembler programs the order of applying transformations is not critical: since the transformations are applied repeatedly until no further improvement can be achieved. With more complex programs, especially highly unstructured programs, applying the transformations in the right order can be important id FermaT is to fully restructure the program. The Fix_Assembler transformation has been developed and refined over many years experience with analysing assembler, and has been tested on millions of lines of code from dozens of different companies.

6.3.1 Preventing Loops

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

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

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

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

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

6.3.2 Dataflow Analysis

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

For example, suppose we have the following assembler code:

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

This translates to the following WSL code:

```
A1 ≡
  r12 := 234; call FOO end
A2 ≡
  ... end
FOO ≡
  SAVER12 := r12;
  r12 := 0;
  ...
  r13 := SAVER12;
```

```
  destination := r13;
  call dispatch end
...
dispatch ≡
  if destination = 0 then call Z
  ...
  elsif destination = 234 then call A2
  ... fi end
```

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

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

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

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

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

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

This approach only works for non-recursive subroutines. In practice, recursion within a single assembler module is very rare and almost certainly a bug. This is because the usual (almost universal) practice is for an assembler subroutine to store its return address in a static location. So any recursion (including mutual recursion) within the same module will overwrite the original return address and prevent the outermost call from returning properly. Recursion between modules, or recursive calls to other entry points in the same module (where return addresses are stored in the savearea chain) are possible, and these are handled correctly by FermaT.

Many assembler subroutines include tests for error conditions which will branch to an error routine, leading ultimately to an `ABEND` instruction, or to a return from the module, instead of

returning to the caller. A WSL procedure, on the other hand, must always return to the caller. The solution is for FermaT generate code which will set a special flag variable exit_flag and then return. If exit_flag = 0 then the return was a normal return, if exit_flag = 1 then the program must immediately terminate. Other values of exit_flag may be used when one subroutine branches directly into another subroutine, without going via a normal call and return.

7 First Case Study

For the first case study we are using an assembler module, `UDATECNV` which is a date conversion module developed by Micro Focus Ltd. We are interested in how the variable `WRKMTH` is calculated for a particular set of input data.

The source code given in Appendix 1 has had some lines elided for brevity, but contains all the code of interest.

The module takes a pointer to a string, which contains both the input data and output data area. For this case study we modified the module to set up the parameter string directly from local storage, and to print the result before returning. This turns the module into a self-contained program and avoids the need for a test harness.

We did not have access to a mainframe for this case study, so the dynamic slicing method describe in Section 5.7 could not be used. Instead the module was executed using the `z390` Portable Mainframe Assembler and Emulator from Automated Software Tools ⟨http: // www. automatedsoftwaretools. com/ z390⟩ with the `TRACE` option turned on. This generates a trace of all executed instructions. A utility program `mark-traced` takes the trace file, the assembler listing file, and the raw WSL translation of the assembler and generates a new WSL file with added **abort** statements. From the trace file, the program computes the offsets of all executed instructions. From the listing, it then computes the line numbers of executed instructions. These line numbers are included in the WSL file as FermaT comments, **abort** statements can be inserted at all points in the program where the translation of a non-executed instruction appears.

The resulting WSL program is guaranteed to be semantically equivalent to the original program *when executed on the given initial state*. It is also possible to execute the program for several different initial states and combine the traces to get a program which is equivalent to the original for *all* the given states.

The resulting WSL program is transformed and simplified: the transformations include operations which make use of the **abort** statements to simplify the program (see Section 5.10).

Here is the result of this "dynamic slice" applied to `UDATECNV`:

```
!P chain_reg( var r13, os);
r12 := 6;
r11 :=!XF address_of(SAVEAREA);
r13 :=!XF address_of(SAVEAREA);
r2 :=!XF address_of(CDATEDIN);
!P WTO("Here is the input data:" var os);
!P WTO(CDFMT[5..CDFMT + 4] var os);
!P WTO(CDMSG[5..CDMSG + 4] var os);
r0 :=!XF system_time(os);
SYSTIME :=!XF system_time(os);
!P unpk(SYSTIME var DBLEWORD[1..7]);
LOCALTIM := DBLEWORD[1..6];
SYSDATE :=!XF system_date(os);
!P unpk(SYSDATE var DBLEWORD[1..7]);
LOCALDAT := DBLEWORD[1..5];
```

```
exit_flag_1 := 0;
DDTIMEO := "hh:mm:s";
NUMAREA := ZEROS_0;
{DDENV = ENVCICS};
NUMAREA[10..15] := DDTIMEI[2..7];
r1 := 15;
r15 :=!XF address_of(NUMAREA);
do {a[r15] ⩾ "0"};
   {a[r15] ⩽ "9"};
   r15 := r15 + 1;
   r1 := r1 − 1;
   if r1 = 0 then exit(1) fi od;
if NUMAREA[10..11] ⩽ "23"
  ∧ NUMAREA[12..13] ⩽ "59"
  ∧ NUMAREA[14..15] ⩽ "59"
  then DDTIMEO[1..2] := NUMAREA[10..11];
       DDTIMEO[3] := ":";
       DDTIMEO[4..5] := NUMAREA[12..13];
       DDTIMEO[6] := ":";
       DDTIMEO[7..8] := NUMAREA[14..15] fi;
DDODATA := "  ";
{DDITYPE = "0"};
{DDOTYPE ≠ "1"};
{DDOTYPE = "2"};
DDODATA.DDO2C := DDI0C;
DDODATA.DDO2Y := DDI0Y;
WRKMM := DDI0M;
{WRKMM[1] ⩾ "0"};
{WRKMM[1] ⩽ "1"};
{WRKMM[2] ⩾ "0"};
{WRKMM[2] ⩽ "9"};
{WRKMM ⩾ "01"};
{WRKMM ⩽ "12"};
DBLEWORD :=!XF pack(WRKMM);
r1 := 3 ∗ (!XF cvb(DBLEWORD) − 1);
r15 := 3 ∗ (!XF cvb(DBLEWORD) − 1)+!XF address_of(MONTHS);
WRKMTH := MONTHS[3 ∗ (!XF cvb(DBLEWORD) − 1), 3];
DDODATA.DDO2M := WRKMTH;
DDODATA.DDO2D := DDI0D;
DDODATA.DDO2S1 := "-";
DDODATA.DDO2S2 := "-";
exit_flag := 0;
r13 := SAVEAREA[5..8];
!P WTO("Here is the result:" var os);
!P WTO(CDFMT[5..CDFMT + 4] var os);
!P WTO(CDMSG[5..CDMSG + 4] var os);
r15 := 0
```

The resulting program has had much irrelevant code deleted: all statements which were not executed in the program run have been removed and many tests have been converted to assertions.

Note that code which is executed but which does not contribute to the output variables of interest is still included in the slice: for example, this includes the loop which checks that all the

characters in NUMAREA are digits. The code which is executed when an non-digit is found is not included in the slice (since during the actual execution, all the characters are digits), but the tests are included as is the loop which moves the pointer in r15

Traditional dynamic slicing algorithms have the opportunity to remove this code, if they can determine (via dynamic dataflow analysis) that there are no dataflow links to the code. However, FermaT can also remove this code via the static slicing step which follows.

The next step is a backwards static slice on the final value of WRKMTH. The result consists of just three statements:

WRKMM := DDI0M;
DBLEWORD :=!XF pack(WRKMM);
WRKMTH := MONTHS[3 ∗ (!XF cvb(DBLEWORD) − 1), 3]

Applying a semantic slice collapses the result to a single assignment:
WRKMTH := MONTHS[3 ∗ (!XF cvb(!XF pack(DDI0M)) − 1), 3]

It is now immediately obvious how the final value of WRKMTH is calculated: the initial value of the string DDI0M is first converted to a packed decimal number, and then to a binary number. This number is used as an index into the string MONTHS to extract a three byte substring. Checking the source code, we see that MONTHS indeed consists of twelve three byte strings "JAN", "FEB", ..., "DEC". So it is clear that the program converts a month number DDI0M (in the range 1–12) to a three character abbreviated month name.

By contrast, a static slice of the original program, on the final value of WRKMTH, looks like this:

```
begin
  if DDITYPE ≠ "0"
    then if DDITYPE ≠ "1"
           then if DDITYPE ≠ "2"
                  then if DDITYPE = "3" ∧ DDOTYPE = "1"
                         then CONVDDD(); MM2MTH()
                       elsif DDITYPE = "3"
                         then if DDOTYPE = "2" then CONVDDD(); MM2MTH() fi fi
                elsif DDOTYPE = "1"
                  then WRKMM := DDI2M; MM2MTH()
                  else if DDOTYPE = "2"
                         then WRKMM := DDI2M; MM2MTH() fi fi
         elsif DDOTYPE = "1"
           then WRKMM := DDI1M; MM2MTH()
           else if DDOTYPE = "2"
                  then WRKMM := DDI1M; MM2MTH() fi fi
  elsif DDOTYPE = "1"
    then WRKMM := DDI0M; MM2MTH()
    else if DDOTYPE = "2"
           then WRKMM := DDI0M; MM2MTH() fi fi
where
proc CONVDDD() ≡
  WRKDAYS :=!XF pack(DDI3DDD);
  WRKMTH := "hex 0x1C";
  r15 :=!XF address_of(MTHDAYS);
  do if WRKDAYS ⩽ a[r15, 2]
       then exit(1)
```

```
          else !P ap("hex 0x1C" var WRKMTH);
               !P sp(a[r15, 2] var WRKDAYS);
               r15 := r15 + 2 fi od;
  !P unpk(WRKMTH[2..3] var WRKMM) end
proc MM2MTH() ≡
  if WRKMM[1] < "0"
    then WRKMTH := "***"
  elsif WRKMM[1] > "1"
       then WRKMTH := "***"
  elsif WRKMM[2] < "0"
       then WRKMTH := "***"
  elsif WRKMM[2] > "9"
       then WRKMTH := "***"
  elsif WRKMM < "01"
       then WRKMTH := "***"
        else if WRKMM > "12"
               then WRKMTH := "***"
               else DBLEWORD :=!XF pack(WRKMM);
                    WRKMTH := MONTHS[3 ∗ (!XF cvb(DBLEWORD) − 1), 3]
             fi fi end
end
```

This version of the program contains much extraneous code concerned with error checking and the various algorithms for handling different types of input. This is the best result we can get via a purely static analysis of the program, without providing further information. If sufficiently detailed conditioning assertions are inserted, then a conditioned semantic slice will produce the same result as the combined slice. In practice, however, it is difficult to determine what these assertions should be, and where they should be inserted, without a detailed knowledge of the behaviour of the program. But "a detailed knowledge of the behaviour of the program" is just what we are trying to ascertain!

The dynamic slicing step can make use of the results of two or more executions of the module. For example, combining two traces, one with a valid month number and one with an invalid month number, followed by a static slice, we get:

```
WRKMM := DDI0M;
if WRKMM[1] < "0"
  then WRKMTH := "***"
elsif WRKMM[1] > "1"
      then WRKMTH := "***"
elsif WRKMM[2] < "0"
      then WRKMTH := "***"
elsif WRKMM[2] > "9"
      then WRKMTH := "***"
elsif WRKMM < "01"
      then WRKMTH := "***"
elsif WRKMM > "12"
      then WRKMTH := "***"
       else DBLEWORD :=!XF pack(WRKMM);
            WRKMTH := MONTHS[3 ∗ (!XF cvb(DBLEWORD) − 1), 3]
fi
```

This combines the behaviour of the program ion both valid and invalid data into a concise abstract representation.

8 Second Case Study

For the second case study we take a typical assembler module which reads a file and generates a report. See Appendix 2 for the source code.

After translating from assembler to WSL and transforming the WSL code to restructure and simplify, and raise the abstraction level, we get this result:

```
begin
  !P chain_reg( var r13, os);
  r12 := 6;
  r11 :=!XF address_of(SAVEAREA);
  r13 :=!XF address_of(SAVEAREA);
  !P OPEN("INPUT" var PARMS);
  r15 := result_code;
  !P OPEN("OUTPUT" var TEXTOUT);
  r15 := result_code;
  RESULTLN := " REPORT ON OUT-OF-RANGE DATA ";
  !P PUT_FIXED(" REPORT ON OUT-OF-RANGE DATA " var TEXTOUT);
  r15 := TEXTOUT_STATUS;
  do r0 := NOT_USED;
     r1 := NOT_USED;
     r14 := 96;
     !P GET_FIXED( var PARMS, PARMSIN);
     r15 := PARMS_STATUS;
     if !XC end_of_file(PARMS)
       then RESULTLN.RESULTMS := "TOTALS ";
            !P ed(SUM1[1..8], "hex 0x40402020202020202020202020202020"
                  var RESULTLN.RESULTC1, cc1, wedit_addr);
            !P ed(SUM2[1..8], "hex 0x40402020202020202020202020202020"
                  var RESULTLN.RESULTC2, cc1, wedit_addr);
            !P PUT_FIXED(RESULTLN var TEXTOUT);
            r15 := TEXTOUT_STATUS;
            if FOUNDL = 1
              then RESULTLN := "A LOW VALUE WAS FOUND ";
                   !P PUT_FIXED("A LOW VALUE WAS FOUND "
                                var TEXTOUT);
                   r15 := TEXTOUT_STATUS fi;
            if FOUNDH = 1
              then RESULTLN := "A HIGH VALUE WAS FOUND ";
                   !P PUT_FIXED("A HIGH VALUE WAS FOUND "
                                var TEXTOUT);
                   r15 := TEXTOUT_STATUS fi;
            RESULTLN := " END OF REPORT ";
            !P PUT_FIXED(" END OF REPORT "
                         var TEXTOUT);
            r15 := TEXTOUT_STATUS;
            exit(1)
       else NUMAREA := PARMSIN.CODE1;
            NUMCHECK();
            if r15 ≠ 0
```

```
                then !P WTO("Non-numeric data in file" var os); exit(1)
                 else NUMAREA := PARMSIN.CODE2;
                      NUMCHECK();
                      if r15 ≠ 0
                        then !P WTO("Non-numeric data in file" var os); exit(1)
                         else if PARMSIN.CODE1 < LOWVAL
                                then LOWVAL := PARMSIN.CODE1;
                                     RESULTLN.RESULTMS := "NEW LOW VALUE ";
                                     RESULTLN.RESULTC1 := PARMSIN.CODE1;
                                     RESULTLN.RESULTC2 := PARMSIN.CODE2;
                                     !P PUT_FIXED(RESULTLN var TEXTOUT);
                                     r15 := TEXTOUT_STATUS;
                                     FOUNDL := 1 fi;
                              if CDC1 > HIGHVAL
                                then HIGHVAL := CDC1;
                                     RESULTLN.RESULTMS := "NEW HIGH VALUE ";
                                     RESULTLN.RESULTC1 := PARMSIN.CODE1;
                                     RESULTLN.RESULTC2 := PARMSIN.CODE2;
                                     !P PUT_FIXED(RESULTLN var TEXTOUT);
                                     r15 := TEXTOUT_STATUS;
                                     FOUNDH := 1 fi;
                              VALUE_0 :=!XF pack(PARMSIN.CODE1);
                              !P ap(VALUE_0 var SUM1[1..7]);
                              VALUE_0 :=!XF pack(PARMSIN.CODE2);
                              !P ap(VALUE_0 var SUM2[1..7]) fi fi fi od;
  exit_flag := 0;
  !P CLOSE( var PARMS);
  r15 := result_code;
  !P CLOSE( var TEXTOUT);
  r15 := 0
where
proc NUMCHECK() ≡
  r1 := 15;
  r15 :=!XF address_of(NUMAREA);
  do if (a[r15] < "0" ∨ a[r15] > "9") ∧ a[r15] ≠ " "
       then if a[r15] < "0"
              then r15 := 4; exit_flag := 0
               else r15 := 4; exit_flag := 0 fi;
            exit(1)
     elsif a[r15] ≠ " "
       then r15 := r15 + 1; r1 := r1 − 1
        else r1 := r1 − 1 fi;
     if r1 = 0
       then r15 := 0; exit_flag := 0; exit(1) fi od end
end
```

Figure 2 shows a sample input file, and Figure 3 shows the corresponding report file.

We are interested in the code which computes the final values of FOUNDL and FOUNDH under normal processing conditions. A static slice of FOUNDL looks like this:

```
begin
```

```
  8          888
  7          777
 12          212
120          120
121          121
114          114
```

Figure 2: Sample input file

```
          REPORT ON OUT-OF-RANGE DATA
NEW LOW VALUE              8          888
NEW HIGH VALUE             8          888
NEW LOW VALUE              7          777
TOTALS                   382         2232
A LOW VALUE WAS FOUND
A HIGH VALUE WAS FOUND
          END OF REPORT
```

Figure 3: Sample output file

```
  !P OPEN("INPUT" var PARMS);
  do !P GET_FIXED( var PARMS, PARMSIN);
     if !XC end_of_file(PARMS)
       then exit(1)
        else NUMCHECK();
             if r15 ≠ 0
               then exit(1)
                else NUMCHECK();
                     if r15 ≠ 0
                       then exit(1)
                        else if PARMSIN.CODE1 < LOWVAL
                                then LOWVAL := PARMSIN.CODE1;
                                     FOUNDL := 1 fi fi fi fi od
where
proc NUMCHECK() ≡
  r1 := 15;
  r15 :=!XF address_of(NUMAREA);
  do if (a[r15] < "0" ∨ a[r15] > "9") ∧ a[r15] ≠ " "
       then if a[r15] < "0"
               then r15 := 4 else r15 := 4 fi;
            exit(1)
     elsif a[r15] ≠ " "
          then r15 := r15 + 1; r1 := r1 − 1
           else r1 := r1 − 1 fi;
     if r1 = 0 then r15 := 0; exit(1) fi od end
end
```

This has to include the calls to NUMCHECK as well as the body of the procedure, even though (as it happens) this procedure is wholly concerned with error checking.

A dynamic slice was computed by executing the program on the sample input file. A syntactic slice of this dynamic slice produced this result:

```
!P OPEN("INPUT" var PARMS);
do !P GET_FIXED( var PARMS, PARMSIN);
   if !XC end_of_file(PARMS)
     then exit(1)
      else if PARMSIN.CODE1 < LOWVAL
             then LOWVAL := PARMSIN.CODE1; FOUNDL := 1 fi fi od
```

This is a good result, but FermaT's semantic slicer can do even better. First, it converts the **do** ... **od** loop to an equivalent **while** loop:

```
!P OPEN("INPUT" var PARMS);
!P GET_FIXED( var PARMS, PARMSIN);
while ¬!XC end_of_file(PARMS) do
  if PARMSIN.CODE1 < LOWVAL
    then LOWVAL := PARMSIN.CODE1; FOUNDL := 1 fi;
  !P GET_FIXED( var PARMS, PARMSIN) od
```

Next, FermaT notices the **if** statement in the loop body, and determines that the condition:

$$\neg !\text{XC end\_of\_file(PARMS)} \wedge \text{PARMSIN.CODE1} \geqslant \text{LOWVAL}$$

would be a suitable candidate for speculative entire loop unrolling:

```
!P OPEN("INPUT" var PARMS);
!P GET_FIXED( var PARMS, PARMSIN);
while ¬!XC end_of_file(PARMS) ∧ PARMSIN.CODE1 ⩾ LOWVAL do
  if PARMSIN.CODE1 < LOWVAL
    then LOWVAL := PARMSIN.CODE1; FOUNDL := 1 fi;
  !P GET_FIXED( var PARMS, PARMSIN) od;
while ¬!XC end_of_file(PARMS) do
  if PARMSIN.CODE1 < LOWVAL
    then LOWVAL := PARMSIN.CODE1; FOUNDL := 1 fi;
  !P GET_FIXED( var PARMS, PARMSIN) od
```

The first loop simplifies to:

```
while ¬!XC end_of_file(PARMS) ∧ PARMSIN.CODE1 ⩾ LOWVAL do
  !P GET_FIXED( var PARMS, PARMSIN) od;
```

On termination of this loop, the assertion:

$$!\text{XC end\_of\_file(PARMS)} \vee \text{PARMSIN.CODE1} < \text{LOWVAL}$$

is true. FermaT unrolls the first step of the second loop and uses the assertion to simplify the loop body:

```
if ¬(!XC end_of_file(PARMS))
  then ⟨LOWVAL := PARMSIN.CODE1, FOUNDL := 1⟩;
       !P GET_FIXED( var PARMS, PARMSIN);
       while ¬(!XC end_of_file(PARMS)) do
         if PARMSIN.CODE1 < LOWVAL
           then ⟨LOWVAL := PARMSIN.CODE1, FOUNDL := 1⟩ fi;
         !P GET_FIXED( var PARMS, PARMSIN) od fi
```

Constant_Propagation determines that the second assignment to FOUNDL is redundant:

```
if ¬(!XC end_of_file(PARMS))
  then ⟨LOWVAL := PARMSIN.CODE1, FOUNDL := 1⟩;
```

```
      !P GET_FIXED( var PARMS, PARMSIN);
      while ¬(!XC end_of_file(PARMS)) do
        if PARMSIN.CODE1 < LOWVAL
          then LOWVAL := PARMSIN.CODE1 fi;
        !P GET_FIXED( var PARMS, PARMSIN) od fi
```

Now the whole of the **while** loop is redundant, as is the assignment to LOWVAL and the call to GET_FIXED:

```
if ¬(!XC end_of_file(PARMS))
  then FOUNDL := 1 fi
```

The resulting program is:

```
!P OPEN("INPUT" var PARMS);
!P GET_FIXED( var PARMS, PARMSIN);
while ¬!XC end_of_file(PARMS) ∧ PARMSIN.CODE1 ⩾ LOWVAL do
  !P GET_FIXED( var PARMS, PARMSIN) od;
if ¬(!XC end_of_file(PARMS))
  then FOUNDL := 1 fi
```

This result is found to be simpler than the original loop, so our "speculation" has succeeded and this program is returned as the result.

The syntactic slice for FOUNDH (applied to the dynamic slice) is very similar to that for FOUNDL:

```
!P OPEN("INPUT" var PARMS);
do !P GET_FIXED( var PARMS, PARMSIN);
   if !XC end_of_file(PARMS)
     then exit(1)
    else if CDC1 > HIGHVAL
           then HIGHVAL := CDC1; FOUNDH := 1 fi fi od
```

so we might expect the semantic slice to be similar. In fact, the semantic slice is:

```
!P OPEN("INPUT" var PARMS);
!P GET_FIXED( var PARMS, PARMSIN);
if ¬(!XC end_of_file(PARMS)) ∧ CDC1 > HIGHVAL
  then FOUNDH := 1 fi
```

The difference is due to the fact that LOWVAL is compared against PARMSIN.CODE1 which depends on the value of the record read from the file, while HIGHVAL is compared against CDC1 which is a constant value.

In this case, selective entire loop unrolling does not simplify the program, but selective unrolling of the first iteration of the **while** loop produces this result:

```
if ¬(!XC end_of_file(PARMS))
  then if CDC1 > HIGHVAL
         then ⟨FOUNDH := 1, HIGHVAL := CDC1⟩;
              !P GET_FIXED( var PARMS, PARMSIN) fi;
       while ¬(!XC end_of_file(PARMS)) do
         if CDC1 > HIGHVAL
           then ⟨FOUNDH := 1, HIGHVAL := CDC1⟩ fi;
         !P GET_FIXED( var PARMS, PARMSIN) od fi
```

FermaT then expands the inner **if** statement forwards and inserts assertions. The assertions are used to simplify the branches of the **if** statement:

```
if ¬(!XC end_of_file(PARMS))
  then if CDC1 > HIGHVAL
```

```
      then {CDC1 > HIGHVAL};
           ⟨FOUNDH := 1, HIGHVAL := CDC1⟩;
           !P GET_FIXED( var PARMS, PARMSIN);
           while ¬(!XC end_of_file(PARMS)) do
             if CDC1 > HIGHVAL
               then ⟨FOUNDH := 1, HIGHVAL := CDC1⟩ fi;
             !P GET_FIXED( var PARMS, PARMSIN) od
       else {CDC1 ⩽ HIGHVAL} fi fi
```

As before, Constant_Propagation determines that the second assignment to FOUNDH is redundant, whereupon the inner **while** loop and the call to GET_FIXED become redundant:

```
if ¬(!XC end_of_file(PARMS))
  then if CDC1 > HIGHVAL
         then {CDC1 > HIGHVAL};
              FOUNDH := 1
         else {CDC1 ⩽ HIGHVAL} fi fi
```

This is abstracted into the specification statement:

```
FOUNDH := FOUNDH′.(
((FOUNDH′ = 1 ∨ CDC1 ⩽ HIGHVAL) ∧ FOUNDH′ = FOUNDH
∨ FOUNDH′ = 1 ∧ CDC1 > HIGHVAL)
∧ ¬(!XC end_of_file(PARMS))
∨ !XC end_of_file(PARMS) ∧ FOUNDH′ = FOUNDH)
```

And then refined into a simple **if** statement:

```
if ¬(!XC end_of_file(PARMS)) ∧ CDC1 > HIGHVAL
  then FOUNDH := 1 fi
```

9 Mass Migration Exercises

We have also applied the semantic slicer to two mass migration case studies. 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.

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. 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 3 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

	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

Table 3: Lines of Code and complexity metrics for raising abstraction level

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!

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).

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 4 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.

	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

Table 4: Lines of Code and complexity metrics for raising abstraction level

These two case studies are described in more detail in [43].

10 Practical Applications

There are two main practical applications for the combined slicing technique: debugging and program comprehension and reengineering. These will be discussed in the following sections.

10.1 Debugging

Typically, when faced with a debugging problem a programmer will have test data which can reproduce the bug. This is not always the case however: sometimes bugs appear sporadically and are difficult to reproduce. Since the dynamic slicing method has such a small impact on performance, it is quite practical to leave it "switched on" all the time. The next time the bug manifests itself, the programmers can examine the dynamic slice to see which instructions were actually executed, and apply backwards static slicing to the result, using the incorrect output as the slicing criteria.

Thanks to the combined slicing technique, the programmer can extract just the code which:

1. Was actually executed when the bug manifested, and
2. Contributed to the incorrect output value.

This leads to a dramatic reduction in debugging effort.

10.2 Reengineering to an Object Oriented System

In this section we outline a method for reengineering a legacy system to an object oriented system. This necessarily requires more work than a simple migration, but FermaT transformations can

provide a lot of assistance with defining an object structure and determining which code belongs in each object.

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

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

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

11 Related Work

11.1 Assembler Migration

Feldman and Friedman [13] describe an automated assembler to C migration project which involved the migration of a large database system and application generator written in IBM 370 assembly language. They developed a "literal" translator which translated each instruction separately into C code with no optimisation. In effect, the result of the translation was an IBM 370 simulator. When became clear that this approach would not be sufficient, a new translator (called Bogart) was developed based on abstraction and re-implementation. Bogart produced code which was between half and three quarters as large and more than twice as fast as the literal translator's output. However, the translator required extensive manual modification of the assembler code before it could be applied. Experienced programmers could process about 3600 lines of code per person-month. As a result, "Manual preparation of the code has probably damaged the code's quality. Programmers estimate that the code is less efficient after standardization, and, naturally, new bugs were introduced. ... two versions now had to be debugged, tested, and maintained" [13]. In addition "Readability was only a secondary goal in this case, because the target code was not meant to be handled by human programmers"

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

1. The absolute minimum of manual modification to the assembler code before migration: our aim is always for 100% automated migration, but there will always be a handful of constructs which appear so rarely in the code that it is easier to fix the original assembler than to program a special-purpose transformation rule.
2. Generate HLL code which is both efficient to execute and maintainable by programmers unfamiliar with IBM 370 assembler;
3. No manual modification of the HLL code after migration.

With all our migration projects to date, we have been able to fix any problems either by adjusting the migration process (usually by updating translation tables) or as a last resort by rewriting parts of the original assembler. So far we have not needed to carry out *any* manual modification of the generated code. This is important because it allows us to regenerate the code at any time simply by re-running the migration process, without having to redo any manual fixes. Combined with the sheer speed of the migration process (around 10 seconds per module), this allows an almost interactive edit/assemble/migrate/compile/test cycle during the initial phase where the transformation rules are being "fine tuned" to produce the best possible output code.

FermaT has been used for a number of successful migration projects. One early project involved migrating an embedded system consisting of over 544,000 lines of 186 assembler to efficient and maintainable C code. Another successful migration project involved migrating over 750,000 lines of IBM assembler to efficient and maintainable cross-platform C code. The migrated C code runs on the mainframe (a big-endian, EBCDIC machine) and on Windows and Linux PCs (little-endian, ASCII machines).

11.2 Amorphous Slicing

Harman, Binkley and Danicic [7] define a slice in terms of a *simplicity measure*, which is a syntactic relation that defines that which is allowed to change, and a *preservation requirement*, which is a semantic relation which captures that which must remain invariant. Unfortunately, their theoretical foundation has a fundamental flaw: the preservation requirement is defined to be an *equivalence relation*. When applied to slicing nonterminating programs there are exactly two possibilities:

1. A nonterminating program can be equivalent to a terminating program;
2. A nonterminating program is never equivalent to a terminating program.

If (2) is the case, then according to their framework, no slicing algorithm is allowed to delete a nonterminating loop. More seriously, the algorithm is not allowed to delete a loop which does not affect the slicing criteria unless it can prove that the loop always terminates! Such a proof is not always easy, see the example in Section 5.2 for an example where mathematicians suspect that the loop always terminates, but so far have failed to prove it.

In [7] the authors take case (1) and use the lazy semantics of Cartwright and Felleisen [10]. This allows a nonterminating program to be equivalent to a terminating program in certain situations, and therefore allows a terminating program to be a valid slice of a nonterminating program. But because the semantic relation is an *equivalence*, it also allows a nonterminating program to be a valid slice of a terminating program! This is something that Weiser [45], for example, explicitly excluded in his discussion of program slicing. Consider the following program:

$x := 1$;
$x := 0$;
while $x = 1$ **do** $y := y + 1$ **od**;
$x := 2$

where we are slicing on the value of x at the end of the program. This program always terminates and sets x to 2. According to [7], the following program is a valid syntactic slice (since it meets both the syntactic and semantic constraints):

$x := 1$;
while $x = 1$ **do** $y := y + 1$ **od**;
$x := 2$

But this program will never terminate!

Cartwright and Felleisen [10] give this example program Q_2:

$y := 1$; **while true do** $x := 0$ **od**

and state that "From the perspective of program optimization, the semantics of sequential execution is too restrictive. ... in the program Q_2 the **while** loop is superfluous if the value of x is never demanded." What they fail to mention is that the assignment to y is *also* superfluous *even if the value of y is demanded*. This is because execution can never get past the **while** loop, so the value assigned to y can never be used. In fact, the **while** loop is still superfluous, even if the value of x *is* demanded: because the value assigned to x inside the loop can never be used elsewhere. A slicing theory based on semantic equivalence using this lazy semantics would not allow one to delete either the assignment to y (when y is demanded) or the **while** loop (when x is demanded), since the lazy

semantics of the two programs is not the same. This contradicts Weiser's assertion that anything is allowed as a slice of a nonterminating program.

For these reasons, we have defined a semantic relation (semi-refinement) which is not an equivalence relation. Semi-refinement allows the following:

1. If the program does not terminate for some initial state, then the slice can do anything;
2. If the program terminates for some initial state then the slice must be semantically equivalent (for the subset of the final state space that we are interested in);
3. In particular, the slice must terminate whenever the original program terminates.

It is not possible to meet all these conditions if we use an equivalence relation as the semantic constraint.

12 FermaT Availability

The FermaT transformation system implements all of the transformations described in this paper, including syntactic and semantic slicing, constant propagation, abstraction and refinement.

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

http://www.gkc.org.uk/fermat.html
http://www.cse.dmu.ac.uk/~mward/fermat.html

13 Conclusion

In this paper we have described an approach to assembler analysis which combines a highly efficient form of dynamic slicing with static semantic slicing based on WSL transformation theory to derive concise, abstract descriptions of the semantics of the program for a given set of inputs and outputs. Two case studies of typical assembler modules show the dramatic improvements in understandability which can be achieved using these methods.

Appendix 1: Source code for First Case Study

```
*********************************************************************
*                                                                   *
*  Copyright (C) 1998-1999 Micro Focus. All Rights Reserved.        *
*  This demonstration program is provided for use by users of       *
*  Micro Focus products and may be used, modified and distributed   *
*  as part of your application provided that you properly           *
*  acknowledge the copyright of Micro Focus in this material.       *
*                                                                   *
*********************************************************************
UDATECNV CSECT
         SAVE  (14,12),,*
         EQUREGS
         BALR  R12,R0                  ESTABLISH
         USING *,R12                    ADDRESSABILITY
         LA    R11,SAVEAREA            DO
         ST    R11,8(R0,R13)            NORMAL
         ST    R13,4(R0,R11)            SAVEAREA
         LR    R13,R11                  CHAINING
*
*        L     R2,0(R0,R1)             LOAD PASSED PARM
***
*** Set up R2 from local data
         LA    R2,CDATEDIN
         WTO   'Here is the input data:'
         WTO  MF=(E,CDFMT)
```

```
         WTO  MF=(E,CDMSG)
***
***
         USING CDATED,R2               ADDRESS PASSD PARMS
*
         TIME  DEC                     GET SYSTEM DATE & TIME
         ST    R0,SYSTIME              STORE SYSTEM TIME (HHMMSSDD)
         OI    SYSTIME+3,X'0F'         SET ZONE SIGN ON TIME
         UNPK  DBLEWORD(7),SYSTIME     UNPACK TIME
         MVC   LOCALTIM,DBLEWORD       STOTE MACHINE TIME
*
         ST    R1,SYSDATE              STORE SYSTEM DATE (00YYDDDS)
         OI    SYSDATE+3,X'0F'         SET SIGN BITS ON DATE
         UNPK  DBLEWORD(7),SYSDATE     UNPACK DATE
         MVC   LOCALDAT,DBLEWORD + 2
*
         BAL   R10,CONVTIM             LINK TO TIME RTN
         BAL   R10,CONVDAT             LINK TO DATE RTN
*
GOBACK   EQU   *                       COMMOM EXIT POINT
         L     R13,SAVEAREA+4           RESTORE R13
***
*** Print the result
         WTO   'Here is the result:'
         WTO  MF=(E,CDFMT)
         WTO  MF=(E,CDMSG)
***
         RETURN (14,12),RC=0           RETURN TO CALLER
*
*********************************************************************
* Convert any supplied time to hh:mm:dd format.                     *
* If ENV is CICS then we have the time as OHHMMSS.                  *
* If ENV is IMS then we have date as HHMSSD                         *
* If ENV is nulls or INET we will use system time                   *
* Any other value of ENV is invalid so return 'hh:mm:ss' rather than *
*  a real time                                                      *
*********************************************************************
CONVTIM  EQU   *
         MVC   DDTIMEO,=CL7'hh:mm:ss'  SET DEFAULT RESPONES
         MVC   NUMAREA,ZEROS           CLEAR NUMERIC WORK AREA
         CLC   DDENV,ENVCICS           IS ENV = CICS
         BNE   CONVTIMA                NO - SKIP TO CONVTIMA
         MVC   NUMAREA+9(6),DDTIMEI+1  MOVE IN CICS TIME
         B     CONVTIMX                SKIP TO FORMAT PART
...
CONVTIMX EQU   *
         BAL   R14,NUMCHECK            CHECK TIME FOR NUMERIC
         LTR   R15,R15                 WAS IT NUMERIC?
         BNZ   CONVTIMZ                NO - SKIP TO EXIT
         CLC   NUMAREA+9(2),=CL2'23'   IS HH GREATER THAN 23?
         BH    CONVTIMZ                YES - SKIP TO EXIT
         CLC   NUMAREA+11(2),=CL2'59'  IS MM GREATER THAN 59
         BH    CONVTIMZ                YES - SKIP TO EXIT
         CLC   NUMAREA+13(2),=CL2'59'  IS SS GREATER THAN 59?
         BH    CONVTIMZ                YES - SKIP O EXIT
CONVTIMY EQU   *                       REFORMAT THE TIME
         MVC   DDTIMEO(2),NUMAREA+9    MOVE IN THE HH
         MVI   DDTIMEO+2,C':'          MOVE IN A SEPERATOR

         MVC   DDTIMEO+3(2),NUMAREA+11 MOVE IN THE MM
         MVI   DDTIMEO+5,C':'          MOVE IN A SEPERATOR
         MVC   DDTIMEO+6(2),NUMAREA+13 MOVE IN THE SS
         B     CONVTIMZ
CONVTIMZ EQU   *
```

```
         BR    R10
*
*********************************************************************
* Convert date from supplied format to requested format.            *
*********************************************************************
CONVDAT  EQU   *
         MVC   DDODATA,=CL20' '        CLEAR THE OUTPUT DATA AREA
*                                      CHECK OUTPUT TYPE
         CLI   DDITYPE,C'0'            IS INPUT TYPE 0 (YYYY-MM-DD)
         BE    CONVDAT0                YES - SKIP TO CONVDAT0
...
CONVDAT0 EQU   *
         CLI   DDOTYPE,C'1'            IS OUTPUT TYPE 1 (DD-MMM-YY)?
         BE    CONVDT01                YES - SKIP TO CONVDT01
         CLI   DDOTYPE,C'2'            IS OUTPUT TYPE 2 (DD-MMM-YYYY)?
         BE    CONVDT02                YES - SKIP TO CONVDT02
...
CONVDT02 EQU   *                       TYPE 2 (YYYY-MM-DD=>DD-MMM-YYYY)
         MVC   DDO2C,DDI0C             MOVE CC TO OUTPUT
         MVC   DDO2Y,DDI0Y             MOVE YY TO OUTPUT
         MVC   WRKMM,DDI0M             MOVE MM TO WORKAREA
         BAL   R14,MM2MTH              LINK TO CONVERSION RTN
         MVC   DDO2M,WRKMTH            MOVE MMM TO OUTPUT
         MVC   DDO2D,DDI0D             MOVE DD TO OUTPUT
         MVI   DDO2S1,C'-'             MOVE IN SEPERATOR
         MVI   DDO2S2,C'-'             MOVE IN SEPERATOR
         BR    R10                     RETURN
...
MM2MTH   EQU   *                       CONVERT MM TO MMM (ALPHA)
         CLI   WRKMM,C'0'              CHECK MM IS VALID NUMERIC AND
         BL    MM2MTHE                  IN RANGE 01 TO 12
         CLI   WRKMM,C'1'
         BH    MM2MTHE
         CLI   WRKMM+1,C'0'
         BL    MM2MTHE
         CLI   WRKMM+1,C'9'
         BH    MM2MTHE
         CLC   WRKMM,=CL2'01'
         BL    MM2MTHE
         CLC   WRKMM,=CL2'12'
         BH    MM2MTHE
         PACK  DBLEWORD,WRKMM          CONVERT MM TO BINARY
         CVB   R1,DBLEWORD
         BCTR  R1,R0                   NOW REDUCE IT BY 1 AND
         MH    R1,=H'3'                MULT BY TABLE ENTRY
         LA    R15,MONTHS(R1)          TO GET ADDRESS IN TABLE
         MVC   WRKMTH,0(R15)           MOVE TABLE ENTRY TO WORK AREA
         B     MM2MTHX                 SKIP TO EXIT POINT
...
MM2MTHX  EQU   *
         BR    R14                     RETURN FROM SUB ROUTINE
*
*********************************************************************
* Check the characters in NUMAREA for being numeric                 *
*********************************************************************
NUMCHECK EQU   *                       CHECK NUMAREA FOR NUMERICS
         LA    R1,15(R0,R0)            LENGTH OF NUMAREA
         LA    R15,NUMAREA             POINT TO NUMAREA
NUMCHCKA EQU   *
         CLI   0(R15),C'0'             IS CHAR LESS THAN 0?
         BL    NUMCHCKY                YES - SKIP TO ERROR
         CLI   0(R15),C'9'             IS CHAR GREATER THAN 9?
         BH    NUMCHCKY                YES - SKIP TO ERROR
         LA    R15,1(R0,R15)           INCREMENT POINTER
```

```
         BCT   R1,NUMCHCKA             LOOP BACK
NUMCHCKX EQU   *
         LA    R15,0(R0,R0)            CLEAR RETURN CODE
         B     NUMCHCKZ                SKIP TO EXIT
...
NUMCHCKZ EQU   *
         BR    R14
*
DBLEWORD DS    D
SAVEAREA DS    18F
R14STORE DS    F
NUMAREA  DS    CL15
ZEROS    DC    CL15'000000000000000'
SYSDATE  DC    F'0'
LOCALDAT DC    CL5'YYDDD'
SYSTIME  DC    F'0'
LOCALTIM DC    CL6'HHMMSS'
ENVCICS  DC    CL4'CICS'
ENVIMS   DC    CL4'IMS '
ENVINET  DC    CL4'INET'
ENVNULL  DC    XL4'00000000'
WRKMM    DS    CL2
WRKMTH   DS    CL3
WRKDD    DS    CL2
WRKDAYS  DS    CL3
WRKYY    DS    CL3
MONTHS   DS    0CL36
         DC    CL3'JAN'
         DC    CL3'FEB'
         DC    CL3'MAR'
         DC    CL3'APR'
         DC    CL3'MAY'
         DC    CL3'JUN'
         DC    CL3'JUL'
         DC    CL3'AUG'
         DC    CL3'SEP'
         DC    CL3'OCT'
         DC    CL3'NOV'
         DC    CL3'DEC'
...
*** Local data for testing:
UDATECNV CSECT
CDFMT    DC    H'61'
         DC    H'0'
         DC    CL61'CICS0hhmmssTIME OUTfccyy-mm-dd          fDATE OUT'
CDMSG    DC    H'61'
         DC    H'0'
CDATEDIN DC    CL61'CICS0124500        02007-03-22          2'
*** End of data
```

Appendix 2: Source code for Second Case Study

```
*********************************************************************
*  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
PWORD    DS    CL20
PNUM     DS    CL12
         DS    CL48
WLAST    DC    CL20' '
TOTAL    DC    PL6'0'
WORKP    DC    PL6'0'
DDIN     DCB   DDNAME=DDIN,                                          *
               DSORG=PS,                                            *
               EODAD=LAB999,                                        *
               MACRF=GM,RECFM=FT,LRECL=80
RDSOUT   DCB   DDNAME=RDSOUT,                                       *
               DSORG=PS,                                            *
```

```
              MACRF=PM,RECFM=FT,LRECL=80
         LTORG
*
         END
```

References

[1] H. Agrawal & J. R. Horgan, "Dynamic Program Slicing," *SIGPLAN Notices* 25 (June, 1990), 246–256.

[2] R. J. R. Back, "Proving Total Correctness of Nondeterministic Programs in Infinitary Logic," *Acta Informatica* 15 (1981), 233–249.

[3] R. Balzer, "EXDAMS – EXtendable Debugging And Monitoring System," Proceedings of the AFIPS SJCC, 1969.

[4] Árpád Beszédas, Csaba Faragó, Zsolt Mihály Szabó, János Csirik & Tibor Gyimothy, "Union slices for program maintenance.," *18th International Conference on Software Maintenance (ICSM), 3rd–6th October 2002, Montreal Quebec* (2002).

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

[6] Dave Binkley, Sebastian Danicic, Tibor Gyimothy, Mark Harman, Akos Kiss & Lahcen Ouarbya, "Formalizing Executable Dynamic and Forward Slicing," *Fourth International Workshop on Source Code Analysis and Manipulation*, Los Alamitos, California, USA (2004).

[7] Dave Binkley, Mark Harman & Sebastian Danicic, "Amorphous Program Slicing," *Journal of Systems and Software* 68 (Oct., 2003), 45–64.

[8] C. Bohm & G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Comm. ACM* 9 (May, 1966), 366–371.

[9] G. Canfora, A. Cimitile & A. De Lucia, "Conditioned program slicing," *Information and Software Technology Special Issue on Program Slicing* 40 (1998), 595–607.

[10] Robin Cartwright & Matthias Felleisen, "The Semantics of Program Dependence," *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* 24 (1989), 13–27, Publishes as SIGPLAN Notices.

[11] Sebastian Danicic, "Dataflow Minimal Slicing," London University, PhD Thesis, 1999.

[12] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

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

[14] Mark Harman & Sebastian Danicic, "Amorphous Program Slicing," *5th IEEE International Workshop on Program Comprehesion (IWPC'97), Dearborn, Michigan, USA* (May 1997).

[15] Mark Harman, Sebastian Danicic & R. M. Hierons, "ConSIT: A conditioned program slicer," *9th IEEE International Conference on Software Maintenance (ICSM'00), San Jose, California, USA*, Los Alamitos, California, USA (Oct., 2000).

[16] Mark Harman, Lin Hu, Malcolm Munro & Xingyuan Zhang, "GUSTT: An Amorphous Slicing System Which Combines Slicing and Transformation," *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, Los Alamitos, California, USA (2001).

[17] Susan Horwitz, Thomas Reps & David Binkley, "Interprocedural slicing using dependence graphs," *Trans. Programming Lang. and Syst.* 12 (Jan., 1990), 26–60.

[18] Capers Jones, *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences.*, Addison Wesley, Reading, MA, 1998.

[19] Capers Jones, "Backfiring: Converting Lines of Code to Function Points," *IEEE Computer* 28 (Nov., 1995), 87–88.

[20] Bogdan Korel & Janusz Laski, "Dynamic program slicing," *Information Processing Letters,* 29 (Oct., 1988), 155–163.

[21] J. C. Lagarias, "The $3x + 1$ Problem and Its Generalizations," *American Mathematical Monthly* 92 (1985), 3–23, ⟨http://www.cecm.sfu.ca/organics/papers/lagarias/⟩.

[22] Andrea De Lucia, Mark Harman, Robert Hierons & Jens Krinke, "Unions of Slices are not Slices," *7th European Conference on Software Maintenance and Reengineering Benevento, Italy March 26-2* (2003).

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

[24] H. A. Priestley & M. Ward, "A Multipurpose Backtracking Algorithm," *J. Symb. Comput.* 18 (1994), 1–40, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/backtr-t.ps.gz⟩.

[25] J. Scott, "The e-Business Hat Trick — Adaptive Enterprises, Adaptable Software, Agile IT Professionals," *Cutter IT Journal* 13 (Apr. 2000), 7–12.

[26] Harry Sneed & Chris Verhoef, "Reengineering the Corporation–A Manifesto for IT Evolution," ⟨http://www.cs.vu.nl/~x/br/br.html⟩.

[27] B. Thwaites, "Two Conjectures, or How to Win £1100," *Mathematical Gazette* 80 (1996), 35–36.

[28] Mustafa M. Tikir & Jeffrey K. Hollingsworth, "Efficient Online Computation of Statement Coverage," *The Journal of Systems and Software* 78 (2005), 146–164.

[29] F. Tip, "Generation of Program Analysis Tools," Cantrum voor Wiskunde en Informatica, PhD Thesis, Amsterdam, 1995.

[30] G. A. Venkatesh, "The semantic approach to program slicing.," *SIGPLAN Notices* 26 (1991), 107–119, Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 26-28.

[31] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989, ⟨http://www.cse.dmu.ac.uk/~mward/martin/thesis⟩.

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

[33] M. Ward, "The Formal Transformation Approach to Source Code Analysis and Manipulation," *IEEE International Workshop on Source Code Analysis and Manipulation Florence, Italy, 10th November*, Los Alamitos, California, USA (2001).

[34] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/prog-spec.ps.gz⟩.

[35] M. Ward, "A Definition of Abstraction," *J. Software Maintenance: Research and Practice* 7 (Nov., 1995), 443–450, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/abstraction-t.ps.gz⟩.

[36] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.* 22 (Sept., 1996), 665–686, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/sw-alg.ps.gz⟩.

[37] M. P. Ward, H. Zedan & T. Hardcastle, "Conditioned Semantic Slicing via Abstraction and Refinement in FermaT," *9th European Conference on Software Maintenance and Reengineering (CSMR) Manchester, UK, March 21–23* (2005).

[38] Martin Ward, "Language Oriented Programming," *Software—Concepts and Tools* 15 (1994), 147–161, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.ps.gz⟩.

[39] Martin Ward, "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations," *Science of Computer Programming, Special Issue on Program Transformation* 52 (2004), 213–255, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/migration-t.ps.gz⟩.

[40] Martin Ward & Hussein Zedan, "Deriving a Slicing Algorithm via FermaT Transformations," Submitted to IEEE TSE, 2008.

[41] Martin Ward & Hussein Zedan, "Slicing as a Program Transformation," *Trans. Programming Lang. and Syst.* 29 (Apr., 2007), 1–52, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/slicing-t.ps.gz⟩.

[42] Martin Ward, Hussein Zedan & Tim Hardcastle, "Legacy Assembler Reengineering and Migration," *20th IEEE International Conference on Software Maintenance, 11th-17th Sept Chicago Illinois, USA.* (2004).

[43] Martin Ward, Hussein Zedan, Matthias Ladkau & Stefan Natelberg, "Conditioned Semantic Slicing for Abstraction; Industrial Experiment," *Software Practice and Experience* 38 (Oct., 2008), 1273–1304, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/slicing-paper-final.pdf⟩.

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

[45] Mark Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," University of Michigan, PhD Thesis, Ann Arbor, 1979.

[46] H. Yang & M. Ward, *Successful Evolution of Software Systems*, Artech House, Boston, London, 2003, ISBN-10 1-58053-349-3 ISBN-13 978-1580533492.

[47] Xingyuan Zhang, Malcolm Munro, Mark Harman & Lin Hu, "Mechanized Operational Semantics of WSL," *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, Los Alamitos, California, USA (2002).