Symbolic Interpretation of Legacy Assembly Language

Jacques Carette, Pulak Kumar Chowdhury
Dept. of Computing and Software
McMaster University
Hamilton, Ontario L8S 4K1

Abstract

We apply static analysis and symbolic interpretation techniques to reverse engineering the semantics of legacy assembler code. We examine the case of IBM-1800 programs in detail. From the documented operational semantics of the IBM-1800, we simultaneously obtain an emulator and a symbolic analysis program. Augmented with some control flow information, we can use the symbolic analysis to provide both complete and generic semantics for some interesting code sequences. Based on this, path conditions and generated "correctness" conditions allow us to derive a mixture of denotational and axiomatic semantics for some interesting subsets of assembly programs.

1 Introduction

Symbolic analysis of a program is a static analysis technique that executes the instructions of a program with some of the values (of registers, input channels or memory location) as unknown symbols. This generates an ordered sequence of dataflow equations which, if solved, gives a precise mathematical representation of the computations done in that program. Existing symbolic analysis tools do not always give accurate representations of the meaning of the program because they tend to introduce approximations to the semantics very early in the processing. By working with systems of symbolic, conditional dataflow equations instead of with sets of solutions, we can be more accurate.

While various static analyses, including abstract and symbolic interpretation, have been successfully used for high-level programming languages, the problem becomes considerably more difficult for assembly language programs, and even more so for legacy software. In particular, while the control structures in most modern high-level languages (sequencing, `if-then-else`, `while`, etc) have very well understood semantics and effect the control flow in predictable fashion, assembly programs liberally use gotos. Frequently, branching code is written that cannot be easily translated to a sequence of high-level structures, at least not without code duplication. Other complications (to be detailed later) include lack of data/code seperation, frequent computed gotos, and even some (relatively mild) instances of self-modifying code.

What we are attempting to do here is, via symbolic interpretation, control flow analysis, and condition propagation, to represent a program's semantics by a system of conditional symbolic dataflow equations. If this system of equations can then be solved in a space of semantically meaningful expressions, this gives an understandable representation of the underlying semantics. To a certain extent, we are free to choose our solution space; this allows us to choose spaces with very rich semantics. In particular, instead of choosing a high-level programming language (which would only "move" our understanding problem up some levels instead of resolving it), we choose a variety of specification languages and mathematical languages. In particular, we are looking at producing output that can be read natively by both PVS [12] and Maple [8].

This work is just one part of a bigger project at McMaster of reverse engineering the requirements of some legacy safety-critical real-time software. Our part in this project limits itself strictly to understanding what a program *does*, and ignores completely the larger subjective issue of what the program is *supposed* to do.

2 Problem Definition

Given a (legacy) assembly code, with complex control flow, no data/code separation, etc, we want to understand what the program does. For this work, we shall take for granted that a formal specification, written in a specification language with consistent semantics (including the language of mathematics), is a definite step forward in "understanding" what a program does.

In our approach, we have an IBM-1800 assembly language program, given as assembler source, and we would like to automatically understand what the program does. We have an extremely complete and detailed description of

the operational semantics of the machine language [1]. The source code of the assembler program contains the following information:

- op codes and corresponding data (symbolic or immediate, as appropriate),
- relative addresses of the instructions,
- names for code blocks,
- names for "data" memory locations (in comments).

One important aspect of those source programs is that they are heavily commented; this is extremely helpful for the larger reverse engineering effort. Unfortunately, little automated use can be made of these comments, as:

- when the programs were maintained, the corresponding comments were not always updated,
- block comments are not always in meaningful locations, so that they cannot be used to identify meaningful blocks of code (i.e. functions),
- line comments do not always correspond to the corresponding instruction.

Needless to say, with the notable exception of "data" memory locations, the comments do not exhibit enough structure to be reliably used in an automated process.
A human reading of those programs and operational specifications finds that

- there is no separation between code and data;
- there are many indirect (computed) jumps;
- there is no "subroutine" concept;
- there is self modifying code, which however only modifies the content of addresses or registers, in other words operands of the instructions;
- there is no stack, only memory;
- there is no exception handling (carry, overflow etc.);
- there is fixed, known data size (16 bits, 32 bits).

The first 4 items are definite complications for program understanding. The last 3 items are certainly a definite impediment to writing programs, but turn out to be quite useful in program understanding! They provide hard, definite constraints that must hold true for the program to be meaningful. For example, as there is no carry or overflow check, then it must be the case that all arithmetic operations must not cause either carries or overflows; this implies that some side predicates must always be true for the program to be meaningful. We will later see that this turns out to be quite helpful in generating pre- and post-conditions.

To make the discussion more precise, here is a small code segment of IBM-1800 assembly which will be used as a running example:

```
 0676 * FDBCK DELX = K(TB)*(ERR(N)*1.067-ERR(N-3))/5861
0ADDR REL OBJ. S.NO. LABEL OPCD FT OPRNDS
 35B6 0  C129  0677  TRBFB  LD   1  41   ERR         B15
 35B7 0  A12A  0678         M    1  42   ERR*1.067   B17
 35B8 0  1082  0679         SLT      2               B15
 35B9 0  912B  0680         S    1  43   ERR*1.067 -
                                         LAST ERR    B15
 35BA 0  A12C  0681         M    1  44   *GAIN K     B31
 35BB 0  108F  0682         SLT      15              B16
 35BC 0  A92D  0683         D    1  45   NORMALISED
                                         DELX(FB)    B00
 35BD 0  D12E  0684         STO  1  46
```

Ultimately, we are aiming to (automatically) identify functions, extract nice closed-form formulas and pre/post-conditions that express the actual (or idealized) semantics of those functions.

3 Process overview

Figure 1 gives the steps of our overall symbolic interpretation process. More specifically, these are:

- Use the complete operational semantics of IBM-1800 to derive (human assisted)
  - an emulator, as an explicit state transformer. This emulator will take a `.lst` code file as input, a starting state, and finds the final state of the machine after the execution of that code file.
  - a one-step symbolic emulator. This finds the complete symbolic interpretation of any instructions, given as the state transformer induced by the operational semantics.
- Use the `.lst` code file to derive an approximated Control Flow Graph (CFG).
- Combine the CFG and one-step symbolic emulator to derive a marked-up CFG. In this derived graph, each edge of the CFG will contain the complete one-step symbolic interpretation of all the instruction contained in the source node of the corresponding edge.
- Find execution paths in the CFG.
- Combine the marked-up CFG and the execution paths to find the dataflow equations (DFE) for the assembler program. In this combination process, we find all the splits and joins in the paths to find the high level control structure of the code.
- Simplify the DFEs.

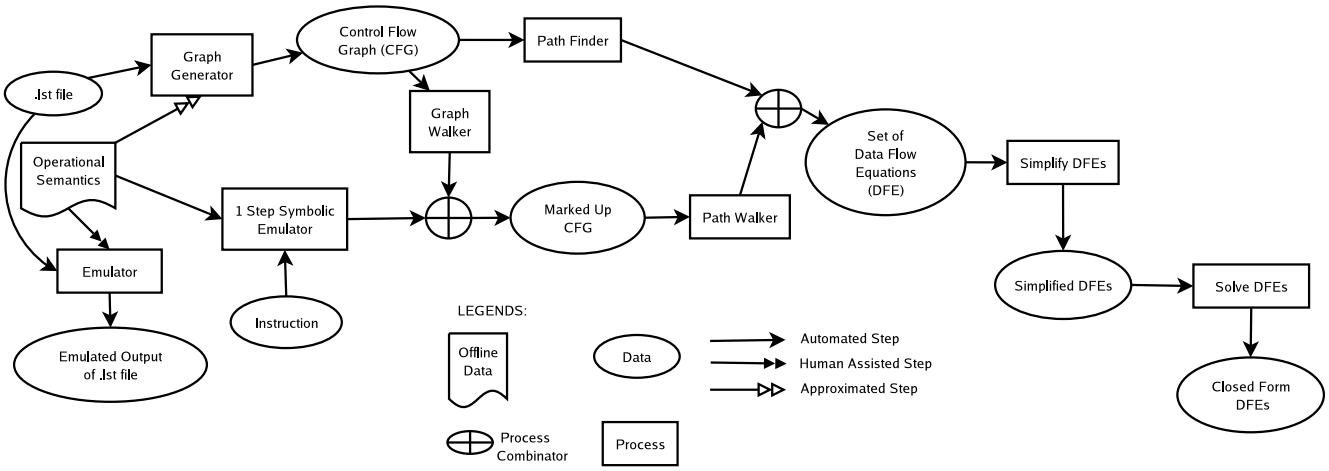

Figure 1. The steps for symbolic interpretation

- Solve those simplified DFEs to find the closed form representations and accompanying preconditions.

In the next several sections, we describe each of these steps in more detail, but first we describe the IBM-1800.

4 The IBM-1800

The IBM-1800 Data Acquisition and Control System [1] was developed to handle a wide variety of real time applications such as process control and data acquisition. The following section is a (slightly modernized) extract from [1].

The registers of the IBM-1800 are

- **Accumulator (`A`)**: Stores one factor of an arithmetic operation; the (internal) D register contains the other factor. It contains the result of an arithmetic operation and can be shifted right or left.
- **Accumulator Extension Register (`Q`)**: An extenstion of the low order end of the accumulator; 16 bits. It stores the 16 least significant bits of a multiplication operation and the remainder of a division operation.
- **Instruction Address Register (`I`)**: A 16 bit register, maintains the address of the next instruction.
- **Index Registers (`XR`)**: Three Index Registers (`XR1, XR2, XR3`), mainly used for address modification.

For the IBM-1800, core storage array are available in two sizes, 4096 (4K) words and 8192 (8K) words. Each word consists of 18 bits (of which 16 are visible to the program). Some instructions can store 32 bits in `A` and `Q` simultaneously, which is then called `AQ`.

Two basic instruction formats are used: a single-word instruction and a two-word instruction.

```
Single-word:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   OP   |D|F| T |    DISP        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Two-word:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   OP   |D|F| T |I|B|    COND    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          ADDRESS                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
OP = OpCode of the instruction
D = 5th Bit
F = Format (0 = One-Word, 1 = Two-Word)
T = Tag Value
I = Indirect Addressing (0 = Direct,1 = Indirect)
B = Branch Out (0 = BSC, 1 = BOSC)
COND = Condition flags checked on a BSC or BSI instruction
DISP = 8 bit displacement address
ADDRESS = Address of the core storage location
```

A two-word instruction contains the full core storage address in the 16 bits of the low order word. A single-word instruction is used when it is not necessary to furnish the full core-storage address, but only to modify (displace) a base address already existing in a designated 16-bit register. The displacement bits, 8 through 15, can be used to address a range of core storage locations from 127 adresseses above the base address to 128 below the base address. The address portion of a two-word instruction can also be modified by adding to the contents of a designated 16-bit index register. The core storage address is called the effective address.

For example, for instruction `0xC382`, the displacement value is `0x82` and the tag value is `0x3`. The opcode `0xC` is the Load instruction, with effective address `XR3 + 82`, where `XR3` stands for the content of Index Register 3. So this instruction transfers the content of core storage location specified by the effective address into the accumulator (`A`).

Another instruction is `0x7500 8532`, an MDX instruction with the address value `0x8532`. As the tag value is `1`, this instruction adds the address to Index Register 1 (`XR1`) and stores the modified word back into `XR1`. Then it

will skip the next instruction if the modified word reaches zero or changes sign.

5 From the operational semantics

As a first step, it is clear that every single instruction (for this and most other processors) has a complete operational description. By complete, we mean that every instruction has a premise-free description. Furthermore this operational description straightforwardly induces a denotational semantics, as a pure state transformer, where our state includes the whole memory as well as all registers. Lastly both of these semantics are (by definition) compositional.

More precisely, we want to model the effect of executing an instruction as a total function on states $[\![\ ]\!]$ to be a total function on states:

$$[\![\texttt{Instruction}]\!] : (\texttt{State} \to \texttt{State})$$

We define the State to be the (partial) function which contains the full Memory, the Instruction Register (I), Accumulator (A), Accumulator Extension Register (Q), all Index Registers (XR1, XR2, XR3) and the Overflow and Carry bits in its domain, and the range is either a 16-bit value (most cases) or a one-bit value (for Overflow and Carry).

We need some notation to represent the operational semantics of the instructions of IBM-1800 assembly language [1].

Inst(I)	Contents of core storage at the location specified by I (Instruction Register)
DB	D (5th) bit of the instruction.
FB	Format bit of the instruction.
displ	Displacement associated to the instruction
$^\wedge y$	Contents of state component y
$\delta_y(f)(x)$	Short for $y \leftarrow f(^\wedge y, x)$ – "update"
$S_y(x)$	Short for $y \leftarrow x$ – "set"
O_{6-8}(Inst)	checks bits 6–8 of the opcode, then according to bits 7&8, return the contents of I, XR1, XR2, XR3 if bit 6 is 0, otherwise return 0, XR1, XR2, XR3.
X	$O_{6-8}(\texttt{Inst}(I))$
loc(X)	If Inst is indirect then $^\wedge(X + I)$ else $X + I$

where DB, FB, and displ are implicitly functions of Inst, f ranges over a few builtin operations (arithmetic and logical), y can be any of the components of the domain of State, and 0 denotes an abstract location with constant value 0.

We can then straightforwardly provide a translation of the operational semantics from [1] using the above primitives. Using $\odot$ to represent (non-commutative) composition of state transformers, $\mathbb{I}$ as the identity transformer, and $\_?\_:\_$ as the notation for if-then-else, this can be written as

$$\begin{aligned}[\![Inst]\!]s = &\delta_I(+)(1+\texttt{FB})\odot \\ &S_A(\texttt{FB} = 1\ ?\ ^\wedge(X + \texttt{displ})\ :\ ^\wedge\texttt{loc}(X)) \\ &\odot (\texttt{DB} = 1\ ?\ S_Q(\texttt{FB} = 1\ ? \\ &^\wedge(X + \texttt{displ} + 1)\ :\ ^\wedge(\texttt{loc}(X) + 1)))\ :\ \mathbb{I})\end{aligned}$$

One can then clearly see that in all cases both I and A are modified, and in some cases so is Q.

5.1 Emulator

If we have a complete State, then by using a complete translation of the operational semantics of the IBM-1800, we can create a complete emulator. We "load up" a complete state via reading in a .lst file, and creating a representation of the state, including the (initial) value of all of memory as an array with 2^{16} entries.

We have compared the output of this emulator and the output of an independently written C emulator for the IBM-1800, and obtained the same results. This certainly increases our confidence that our implementation is correct.

5.2 One Step Symbolic Emulator

By the symbolic analysis of a program, we mean executing the program with the value of (parts of) the state as arbitrary. We represent these arbitrary values by *symbols*, and execute as much of the program as is possible. Since a program may branch on the value of one of these symbols, we are forced to represent the results by using piecewise functions of the symbols. We also "lift" all the basic arithmetic and logical operations to function symbols representing them – as is routinely done in mathematics, but also in Computer Algebra Systems (CAS).

As the result of such a symbolic analysis reflects variables' values and a programs' behaviour, symbolic analysis can be seen as a compiler that translates a program into a different language. As a target language we employ (symbolic) arithmetic and logical expressions, (symbolic representations of) piecewise functions and (symbolic representations of) linear recurrence equations.

Since we are dealing with the symbolic interpretation of assembly programs which do not have predefined control structures in the program syntax, control must be inferred in some other way. As we would like to obtain large scale semantics, this cannot be done via (straightforward) emulation. See the next section for how we achieve this.

We start the symbolic analysis of IBM-1800 assembly language programs by designing a one-step symbolic emulator. This will interpret each instruction in a program execution path and find the symbolic representation of each instructions. For all statements of the program, our symbolic

analysis uses exactly the same description of the semantics as in the previous section, but allows the values of any *non-instruction* part of the state to be symbolic. Of course, instead of returning a value, this step produces a representation (as an abstract data-structure) of the state-transformer which corresponds to the current instruction.

The one-step symbolic emulator produces the symbolic representation of the state transformer induced by each of the instructions. After execution of each instruction, it will output a state transformer representation that contains expressions of the variables that are being changed by the execution of this instruction, and a symbolic path condition representation that reflects possible branching behaviour of the instruction.

Here we give some sample output of the one-step symbolic emulator. For the instruction `LD 1 41` (opcode `C129`), a textual representation of the output of the one-step symbolic interpreter will be:

```
(True, [A := C(XR1+41)])
```

We can take a look at one of the MDX instructions (opcode `7500 000D`). The output will be:

```
(XR1+13(Reaches Zero or Changes Sign)==True,
                  [I += 3, XR1 += 13])
(XR1+13(Reaches Zero or Changes Sign)==False,
                  [I += 2, XR1 += 13])
```

As can be seen for the above, the results are a set of pairs, where the first component is a guard predicate, and the second component is a list of representations of changes of individual state components. It is guaranteed that the guard predicates will be both complete and independent. See section 6 for a more precise definition of this output.

5.3 Control Flow Graph (CFG)

We use a tool which, given the same operational semantics as used by the previous components, produces an *approximate* control flow graph for a program. This program proceeds by looking at only looking at updates to `I` and selected memory locations (as used by branching instructions) to approximate the control flow. In general, this approximation is very good, but in a few cases where the code is self-modifying, this automated step fails. This is not necessarily a problem, as we can also provide hand-written or hand-corrected CFGs as input to the next step.

For each instruction to be executed, we obtain a node in the graph. By traversing the list of instructions, all the possible next addresses from each instruction are found, and edges to those nodes from the current node. A node contains various information such as its address, the stored opcode, any available textual labels, and the path-condition (to be defined later) of the corresponding instruction. See [3] for more details.

Input and output of the resulting control flow graph is done via GXL (Graph eXchange Language) [16], so that standard tools may be re-used to manipulate and display these graphs. Note that our internal representation of graphs does not use GXL as that would be quite inconvenient, but a graph format which contains only the necessary information for symbolic interpretation of the code.

6 Marked-up Control Flow Graph

Given a control flow graph, we want to use the results of the one-step symbolic interpreter to mark-up the edges of the graph with the symbolic representation of the state-transformer corresponding to that edge. The main difficulty is that while the complete interpretation of an instruction can be done symbolically, this cannot be done for even medium sized programs because the resulting output would be so large as to be useless.

Another aspect to consider is that we are only really interested in the semantics of larger chunks of programs, which hopefully correspond to natural functions. These larger chunks invariably contain conditionals, and it makes no sense to interpret the meaning of one branch of the conditional in a context which does not include the reason why this particular branch was chosen, in other words the truth-value of the boolean condition that caused the program to choose that particular branch.

These two aspects have a common remedy. Inspired by [4] where similar techniques are used for (very) high-level programs, we define a program context to be $[p, s]$ where p is a path condition and s is a state.

The path condition p describes the condition under which control flow reaches a given program statement from a given starting point. Every instruction (see subsection 5.2) induces a condition under which each outgoing edge in the control flow graph is followed. For sequential instructions, this condition is just TRUE, and for branch instructions this condition is a logical formula that encodes the condition expressed by the operational semantics. The path condition at a particular node is the disjunction of the path conditions of all the path conditions of the ingoing edges of that node. The path condition along an outgoing edge is the conjunction of the path condition at the source node and the path condition given by the one-step symbolic emulator.

We use a combinator which weaves a Graph Walker (in our case a depth-first graph traversal) with the one-step symbolic emulator to produce a new function which, given a CFG, will return a marked-up CFG with the edges labelled by a program context.

Using the code first shown in section2, here is a textual representation of the marked-up control flow graph

```
<nodeID ="test35b6",OpCode = 49449>
   <EdgeFrom ="test35b6",
    Annotation:(True, [A := C(XR1 + 41)]),
    EdgeTo = "test35b7">
<nodeID ="test35b7",OpCode = 41258>
   <EdgeFrom ="test35b7",
```

```
    Annotation:(True, [A  = A*C(XR1 + 42)]),
    EdgeTo = "test35b8">
<nodeID ="test35b8",OpCode = 4226>
   <EdgeFrom ="test35b8",
    Annotation:(True, [A <<:= 2]),
    EdgeTo = "test35b9">
<nodeID ="test35b9",OpCode = 37163>
   <EdgeFrom ="test35b9",
    Annotation:(True,[A := A-C(XR1 + 43)]),
    EdgeTo = "test35ba">
<nodeID ="test35ba",OpCode = 41260>
   <EdgeFrom ="test35ba",
    Annotation:(True, [AQ  = A*C(XR1 + 44)]),
    EdgeTo = "test35bb">
<nodeID ="test35bb",OpCode = 4239>
   <EdgeFrom ="test35bb",
    Annotation:(True, [AQ <<:= 15]),
    EdgeTo = "test35bc">
<nodeID ="test35bc",OpCode = 43309>
   <EdgeFrom ="test35bc",
    Annotation:[(True, [A := AQ/C(XR1 + 45),
                        Q := AQ%C(XR1 + 45)]),
    EdgeTo = "test35bd">
<nodeID ="test35bd",OpCode = 53550>
   <EdgeFrom ="test35bd",
    Annotation:(True, [C(XR1 + 46) := A]),
    EdgeTo = "test35be">
```

7 Finding Paths in the Control Flow Graph

There can be many different types of control flow in assembler code. Given a (connected) subgraph S of a complete CFG C, we say that

- S is *single-entry* if all edges from $C \setminus S$ to S go to a single node of S; this node is called the *entry point* of S. We also require that all nodes of S be reachable from the entry point.
- S is *single-exit* if all edges from S to $C \setminus S$ go from a single node of S, and this node is called the *exit point* of S. We also require that the dual of a single-exit graph be a single-entry graph.
- E is an *execution path* of S if E is a single-entry, single-exit connected subgraph of S where all nodes except the entry point have in-degree 1 and all nodes except the exit point have out-degree 1.
- a *loop* L is a single-entry, single-exit connected subgraph of C where all nodes have in-degree 1 and out-degree 1 except for one node which has out-degree 2. Note that we include the "exit point" in the loop.

For the purposes of this paper, we will only treat single-entry single-exit subgraphs. Given this restriction, we divide control flow graphs into three broad categories (see Figure 2 for a pictorial representation):

- **Straight-Line Code (SC):** In other words, an execution path.
- **Generalized Straight-Line Code (GSC):** May contain a branch or jump, but that branch or jump will have all the outgoing edges to different nodes inside that code segment. In other words, globally this code is single-entry single-exit, but may contain multiple execution paths.
- **Looping code (LC):** An execution path which ends in a loop; the exit point of the loop may be extended by a (possibly empty) exectution path.

We want to represent a (single-entry single-exit) program as a system of dataflow equations. But to be able to do this, we need to be able to identify the set of all execution paths through a particular program. Note that we only consider paths that go from the starting point to the exit point. Our implementation takes a CFG as input, and returns a complete set of execution paths.

8 Finding Data Flow Equations

In a modern high level language, non sequential control flow is encapsulated in a small number of statements that implement variations on the control flow patterns of iteration and alternation. In an assembler program there are no restrictions on the control flow patterns that may be used by the programmer. We therefore need a more general mechanism to describe control flow. As we described in an earlier section, here we use the set of execution paths to model the control flow of a program.

For each path, we gather all the annotations of the edges in an execution path. We use a stack for this purpose, to accumulate an "environment" in which we can look up the values of any parts of the state at any point of the execution path. As we are interested in finding dataflow equations at this point, the representation used in the output of the one-step symbolic emulator is not the best. We need to see the assignments as the top-level operation, and thus need to "push in" the conditionals. This is a straightforward operation.

We also need to define the inputs and outputs along an execution path. The inputs are the state components which are read along an execution path, and the outputs are those state components which are modified along the same execution path. For specificity, our implementation (in Haskell) reads

```
data TypeCast = Upper16 | Lower16
data Exp =  Constant Word8
     | MemoryConstant MemRef
     | Variable StateComp
     | VariableX Tag
     | UnaryOperation (TypeCast,Exp)
     | BinaryOperation (Exp,Operator,Exp)
     | ConditionalValue ((CondFunc,Exp), (CondFunc,Exp))
          deriving (Eq)
data StateRef = SC StateComp | SCX Tag
                  | Mem MemRef deriving (Eq)
data Stmt
  =  Assign (StateRef,Exp) deriving (Eq)
```

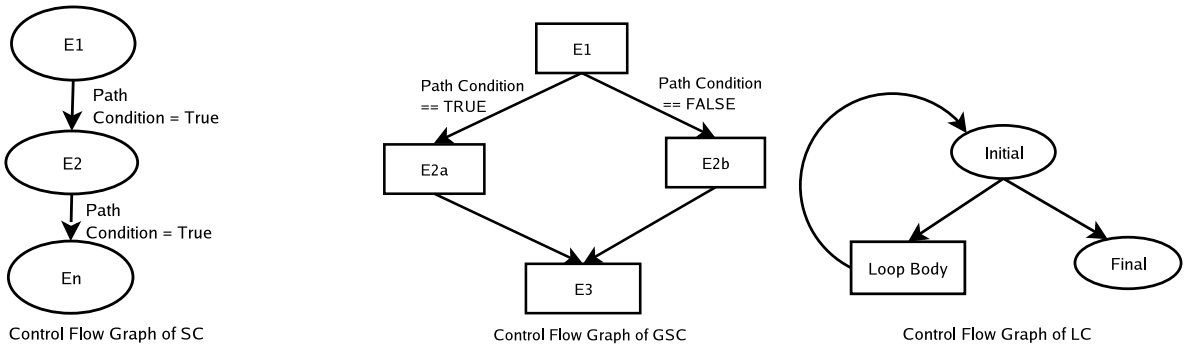

Figure 2. Pictorial representation of code categories

```
type Input = [StateRef]
type Output = [StateRef]
```

For each type of control flow graph noted above, we use a different strategy to find the corresponding dataflow equations.

Straight-Line Code (SC): We gather all the annotations of the edges of the single execution path. We use sequential composition $E1; E2$ to represent this.

Modelling the control flow and finding the semantic context of straight-line code is simple. In particular, there are no new path conditions that are imposed. we just need to find the state transformer corresponding to each statement, and using a stack to keep track of the current environment, we sequentially compose all the expressions representing these state transformers. Since each state transformer obtained from the previous stage is always of the form $V' = f(S)$ where V' is a single state component, and S is a finite set of state components, we get a very nice representation of a set of the simplest kind of dataflow equations.

Using the same example as before, we would get something like the following *ordered* system of equations:

```
A := C(XR1 + 41)]
AQ := A*C(XR1 + 42)
AQ <<= 2
A -= C(XR1 + 43)
AQ := A*C(XR1 + 44)
AQ <<= 15
A := AQ/C(XR1 + 45)
Q := AQ%C(XR1 + 45)
C(XR1 + 46) := A
```

Such a "system" is naturally trivial to "solve", as it can simply be unwound. Also, remark that such a system also corresponds to an (obvious) state transformer.

Generalized Straight-Line Code (GSC): For GSC (refer to middle graph in Figure 2), we proceed as follows:

- find the nodes in the code that correspond to a split (a branch instruction) and a join (the meeting point of two different paths which started at a split). This divides the CFG as several SCs, which we label E_1, E_{2a} , E_{2b} , E_3.
- for each of the SCs, we generate the system of dataflow equations. We use E_1 to also denote the resulting system (context will ensure no confusion will occur).
- Write the system of dataflow equations for the whole code as $E_1; (g_t \rightarrow E_{2a} || \neg g_t \rightarrow E_{2b}); E_3$ where g_t denotes the *guard* which corresponds to the choice for branch t and $||$ denotes parallel composition.

Note that sequential and parallel composition commute [6]. these equations can be rewritten as

$$[(E_1; g_t \rightarrow E_{2a})||(E_1; \neg g_t \rightarrow E_{2b})]; E_3.$$

Guards do not generally commute with sequential composition, however since the guard t is in terms of state components, from solving E_1 we can derive another expression t' such that $E_1(t') = t$. Using this relation, in two more steps the above equation can be rewritten as

$$[g_{t'} \rightarrow (E_1; E_{2a}; E_3)||\neg g_{t'} \rightarrow (E_1; E_{2b}; E_3)].$$

The inner systems are now reduced to be straight-line code (SC). Both of these forms of the equations will be used when we try to solve such systems.

Looping Code (LC): One significant challenge in modelling any program, symbolically or otherwise, is to correctly model loops. Since in general it is uncomputable to automatically derive the necessary invariants for expressing the Hoare semantics of a loop [10], it is best to accept right at the outset that some implicit representation of the semantics is necessary. This is what drove us, amongst other reasons, to choose a "systems of equations" approach to modelling semantics. For loops, we will use (symbolic) recurrence equations as a model [4]. If we are lucky, these recurrences will be solveable in closed form, but even if they are not, we can continue with this implicit representation. Frequently, properties of the solution of recurrence equations

can be derived from the recurrence itself without needing the closed-form solution.

To make the discussion more concrete, we will use the following simple example:

```
0ADDR REL OBJ. ST.N.  LABEL  OPCD FT OPRNDS
 35CE  0  1001   0708         SLA      1
 35CF  0  72FF   0709         MDX  2  -1
 35D0  0  70FD   0710         MDX    *-3
```

In this simple loop, the accumulator A value is shifted left by one and XR2 is decreased by one at each loop execution. We can express this change in terms of recurrences: $A_{n+1} = 2 * A_n$ and $\texttt{XR2}_{n+1} = \texttt{XR2}_n - 1$, which expresses that the value of the accumulator and XR2at time $n+1$ are a function of their values at time n, where $n \geq 0$. As on loop entry, both A and XR2 have a value, this gives us the necessary initial conditions for this first-order recurrence. We will use $\overline{A}$ and $\overline{\texttt{XR2}}$ to denote these initial values. We can represent the symbolic meaning of the loop using these recurrence equations and the initial conditions.

To determine the value of A after the loop, we need to know if and when the loop will stop. We define a stopping criterion ϕ : State $\rightarrow \mathbb{B}$ which will symbolically determine the number of iterations for the loop. This stopping criterion naturally corresponds with the loop condition – which for our simple loop is $\phi = \texttt{XR2} > 0$. The recurrence equation, initial condition and stopping criterion are sufficient to completely describe all the loop information symbolically.

For each component of the state v which is modified in a loop, we represent the corresponding information as a function $\mu(v, s, c)$, from the variable, state and a program context c (see Section 6). The stopping criteria is given by the path condition of the program context c, and the initial condition is determined from s. The result of μ is a representation of the recurrence equation for that state component.

The next section introduces some techniques for solving dataflow equations, include recurrences, as well as "solving" the equations given by the stopping criterion.

9 Solving Data Flow Equations

Straight-line code induces dataflow equations which is essentially already solved - one just needs to "unroll" the equations sequentially. More precisely, consider the ordered equations $A = F(X, Y); B = G(A, X, Z)$. Initially, the empty code sequence has neither input nor output. By proceeding inductively, the first equation tells us that X, Y are part of the "input", and A the output. The second equation has A, X, Z as input, but since A is already known to be an output of the system, it can be eliminated. More precisely, the inputs of equation n are the free variables of the right-hand side of equation n, minus the outputs from stage $n-1$, union the inputs from stage $n-1$. The output variables at stage n is the output of stage $n-1$ union the variable on the left-hand side of stage n. Working this through, the above has X, Y, Z as inputs and A, B as outputs. To work out the actual equations, pure sequential substitution of the equations from stage $n-1$ into the right-hand side of stage n is sufficient.

By solving the above equations for the straight-line code previously shown, as well as (automatically) using the symbolic names associated with the memory locations that are found to be inputs and outputs, we get

```
Input:    C(XR1 + 41) : ERR
          C(XR1 + 42) : 1.067
          C(XR1 + 43) : PREV_ERR
          C(XR1 + 44) : K
          C(XR1 + 45) : 5861
Output: C(XR1 + 46) : DELX
System Of Equations:
DELX = (((Lower16((ERR*1.067)<<2)-PREV_ERR)*K)<<15)
        /5861
Q = (((Lower16((ERR*1.067)<<2)-PREV_ERR)*K)<<15)
    %5861
A = (((Lower16((ERR*1.067)<<2)-PREV_ERR)*K)<<15)
      /5861
```

These could be further simplified by appropriately using "let" statements to factor out common sub-expressions.

For programs with simples branches (GSC), by applying the appropriate commutation relations, the problem is reduced to pure dataflow equations. It is also possible to solve the parallel composition equations "directly". More precisely, given $[g_1 \rightarrow E_1 || \neg g_1 \rightarrow E_2]$, for any variable v which is modified by *either* E_1 or E_2, the result can be given as $v = g_1 \ ? \ E_1(v) \mid E_2(v)$.

For loops, we try to solve the recurrence equations from the "body" of the loop. First, we need to solve for the stopping criterion, to show that the loop terminates. This translates to finding the least n such that $\phi^{-1}(\texttt{XR2}_n) = \min\{n \mid \texttt{XR2}_n > 0\}$. However we know that $\texttt{XR2}_{n+1} = \texttt{XR2}_n - 1$ and $\texttt{XR2}_0 = \overline{\texttt{XR2}}$, so that $\texttt{XR2}_n = \overline{\texttt{XR2}} - n$. In other words, $\phi^{-1}(\texttt{XR2}_n) = \overline{\texttt{XR2}} < \infty$. It is important to remark that we do not need to know anything about A_n to derive this. We can also solve the recurrence for A_n, giving $A_n = A * 2^n$. As we know that at loop end $n = \overline{\texttt{XR2}}$, this means that at loop end $\texttt{XR2} = 0$ and $A = \overline{A} * 2^{\texttt{XR2}}$.

Generally, if we consider Figure 2 then we can express the dataflow equations for loops as $I; \mu(L)|_{n=\phi^{-1}(v)}; F$ where $\phi^{-1}(v) = min\{n \mid \phi(v_n)\}$ and $\mu(L)$ represents the recurrence equations (and initial conditions) for each of the state variables modified by L. The closed form of these dataflow equations can be given by the solution of μ i.e. $v_n = \mu(L)$. As long we we can solve for ϕ^{-1}, that is sufficient to continue with (implicit) representations of the results of a loop.

10 Finding Preconditions

Often we are interested about the partial correctness of programs. A program can be defined as partially correct, with respect to a given precondition and a postcondition, if

the initial state satisfies the precondition and if the program terminates, the final state satisfies the postconditions. Now if we are given postconditions for the program, we can try to use the dataflow equations to "push backwards" (as in backwards state transformers) these predicates to find preconditions.

Additionally, as no exceptions (in other words carry, overflow etc.) are handled by the programs we reviewed, we can add these as post-conditions as well, and also propagate them backwards through the dataflow equations. Let us consider the control flow graph in Figure 3. Suppose that a, b, c are all state variables in this context. At node Q, the initial state is $I = \{a \mapsto a_0, b \mapsto b_0, c \mapsto c_0\}$, and at node P, assume that we have $c = a + b$. As we know that c must be a valid value ($< 2^{16}$ on the IBM-1800), we can conclude that $a + b < 2^{16}$ is true at P. We can push this backwards to (potentially) derive additional necessary conditions at Q, as well as pushing it forward to (potentially) find a more precise description of the state at R, by potentially removing some infeasible paths.

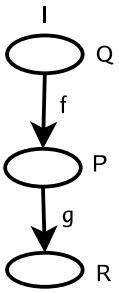

Figure 3. Finding Preconditions

11 Related Work

Morris and Filman [9] developed a system called MANDRAKE to translate IBM-370 code into a high-level language. As IBM-370 code contains many GOTOs, Morris and Filman mainly focus on removing these GOTOs, and then work on introducing high-level procedure-like structures for that code. In their tool they converted the IBM-370 programs into a higher level program for better understanding. In our approach, we instead found the (mathematical) system of equations statisfied by the state of the machine.

Feldman and Friedman [5] also described a system (BOGART) for interpreting IBM 370 assembly. Like MANDRAKE, they adopted a flow analysis technique (much as we do). Explicit control flow graph is not a part of their input values. BOGART needs extensive manual modification of the code before it can be applied. The BOGART system gives less emphasis on detailed steps of 370 assembler. This is due to the fact that for both BOGART and MANDRAKE, the assembly they analyse is higher-level and better structured than the IBM-1800 code we look at. BOGART system produces code for direct execution whereas MANDRAKE produces code for human consumption (i.e. better understanding). In our approach we mainly focus on producing mathematical descriptions of the semantics of the programs, so that the results can be used for higher-level understand as well as for automated property proving and verification.

Ward [14] described a case study of an automated plus manual process to convert an IBM-370 assembler program to high level specifications using the FermaT transformation system. Via a "Wide Spectrum Language" (WSL) which simultaneously incorporates low-level coding features and high-level specification features, they (automatically) translated assembler programs to WSL before introducing specification-level abstractions in subsequent passes using the human-assisted FermaT transformation system. During the FermaT transformation process, the WSL code is further refined using some predefined constructs for control structures. At this point the code becomes more understandable and high level specifications can be generated in the same WSL language. This method explicitly makes many approximations, some of which rule out analyzing IBM-1800 code (like ignoring self-modifying code and register-indexed branching).

Watson and Fidge [15] have described a technique for assembler semantics that is based on advanced compiler theory and technology, as well as programming language semantics. Their work is close to ours, except that while they are describing a theoretical framework, we have a working reverse-engineering tool. As we do, they used execution paths to find the semantics of assembly language programs. They divided programs into basic blocks (our straight-line code) and a path is described as a sequence of basic blocks. Path semantics is the association of the semantics of all the basic blocks in that path. In their paper, they illustrated an approach to generate weakest liberal precondition (WLP) semantics of some abstract code and then demonstrated the correctness of compilation of high level code fragments to assembler code, using path semantics and WLP. They did not address the issue of reverse engineering.

Lake and Blanchard [7] discussed a model-based approach to transform assembler program into a high-level language. They make some simplifying assumptions which do not hold in our case, and use another programming language instead of a mathematical language for expressing their results. While the Portable Reverse Engineering Environment (PARE) for assembler programs of Roberts, Piazza and Katz [13] seems interesting, they too assume more structure from their assembler programs than we do.

Apart from the application of standard techniques from

(high-level) programming language semantics like operational and denotational semantics, the work that most influenced ours is that of Fahringer and Scholz[4]. They developed an approach to find the symbolic interpretation of imperative programs written in a high-level language. We applied some of their constructs (path conditions, recurrence equations) in our approach to interpret assembler programs. As assembler programs do not have any predefined control structures, we used an explicit control flow graph to find execution paths, and used them to guide our symbolic interpretation.

12 Contributions

We see our main contributions as adapting the tools from symbolic computation, compiler construction (dataflow and control flow graphs) and denotational semantics to the situation of understanding and reverse-engineering the semantics of legacy assembler programs. An important contribution of our approach is that we have made no simplifying assumptions, and yet managed to derive mathematical descriptions of subprograms.

13 Future Work

The solution presented in this paper is still preliminary. As we have indicated earlier, only some special kinds of control flow is currently analysed. We are working on analysing more complex control flow – quite a bit of weird control flow can be recast in our setting by a simple duplication of some of the code, and we are actively pursuing this. As well, we are working on simplifying the output expressions, and developing tools to push boolean conditions through systems of (more complex) dataflow equations.

Another important aspect is that any time we can find that some paths are infeasible, through whatever methods, this simplifies the analysis tremendously. So our current analysis should be seen as one step in a fixed-point analysis, where each analysis pass provides a better approximation to the complete semantics.

We are also exploring the use of redundant descriptions of the semantics; for example a left-shift instruction can be described either as a left-shift or as a multiplication by 2. The "best" description usually depends on how that value is used in other contexts. If it is used in a context of arithmetic operations, then multiplication by 2 is likely more descriptive, whereas if used in a context of bit operations, a shift is likely better. The work of [6] is relevant here.

References

[1] *IBM Field Engineering Theory of Operation, 1800 Data Acquisition and Control System, Processor-Controller*. IBM Systems Development Division, Product Publications, Department G24, San Jose, California 95114, 1970.

[2] D. L. Clutterbuck and B. A. Carre. The verification of low-level code. *Software Engineering Journal*, 3(3):97–111, May 1988.

[3] K. Everets. Assembly language representation and graph generation in a pure functional programming language. Master's thesis, Dept. of Computing and Software, McMaster University, December 2004.

[4] T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization*, volume 2628 of *Lecture Notes in Computer Science*. Springer, 2003.

[5] Y. Feldman and D. A. Friedman. Portability by automatic translation; a large scale case study. In *Proc. 10th Knowledge-Based Software Engineering Conference*, 1995.

[6] W. Kahl, C. K. Anand, and J. Carette. Choices in data flow for declarative assembly. In I. Düntsch, W. MacCaull, and M. Winter, editors, *8th International Conference on Relational Methods in Computer Science, RelMiCS 8*, 2004. (participants' proceedings, to appear).

[7] T. Lake and T. Blanchard. Reverse engineering of assembler programs: A model-based approach and its logical basis. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*. IEEE Computer Society, 1996.

[8] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 7 Programming Guide*. Waterloo Maple Inc., 2001.

[9] P. Morris and R. Filman. Mandrake: A tool for reverse-engineering ibm assembly code. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*, pages 58–65, November 1996.

[10] H. R. Nielson and F. Nielson. *Semantics With Applications: A Formal Introduction*. John Wiley and Sons, July 1999.

[11] I. M. O'Neill, D. L. Clutterbuck, P. F. Farrow, P. G. Summers, and W. C. Dolman. The formal verification of safety-critical assembly code. In W. D. Ehrenberger, editor, *Safety of Computer Control Systems 1988*, pages 115–120. International Federation of Automatic Control, Pergamon Press, November 1988.

[12] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide, Language Reference and Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[13] S. N. Roberts, R. L. Piazza, and D. G. Katz. A portable assembler reverse engineering environment (PARE). In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*. IEEE Computer Society, 1996.

[14] M. Ward. Reverse engineering from assembler to formal specifications via program transformations. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, Nov. 2000.

[15] G. Watson and C. Fidge. Modelling assembler programs with an application to compilation. Technical Report 03-GW-1, Software Verification Research Centre, The University of Queensland, July 2003.

[16] A. Winter. Exchanging graphs with GXL. Technical Report 9-2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1,Koblenz, 2001. D-56075.