Function Extraction: Automated Behavior Computation for Aerospace Software Verification and Certification

Redge Bartholomew
Software Design Support
Engineering and Technology
Rockwell Collins, Inc.
Cedar Rapids, IA
rgbartho@rockwellcollins.com

Luanne Burns, Tim Daly, Rick Linger, and Stacy Prowell
*CERT STAR*Lab*
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA
lburns, daly, rlinger, sprowell@cert.org

[Abstract] The complex aerospace systems of the future will challenge the capabilities of present-day software engineering, which is reaching cost and complexity limits of development technologies evolved in the first fifty years of computing. A new science for the next fifty years is required to transform software engineering into a computational discipline capable of fast and dependable software development. This paper describes verification and certification challenges for avionics software, in particular, the need to verify behavior in all circumstances of use. The emerging technology of function extraction (FX) for automated computation of software behavior is discussed as a new technology for avionics software certification. An FX demonstration system is employed to illustrate the role of behavior computation in the avionics certification process.

I. Verification and Certification of Avionics Software

The purpose of software verification in the certification context is to eliminate errors introduced during development, typically in compliance with industry or government guidance. In the case of software for commercial aviation, the FAA recognizes[1] the guidance provided by RTCA DO-178B[2], and aviation software developers use it as a compliance document. Among other things it advises the development team to make sure system requirements that were allocated to software have been developed into high-level software requirements, and that those have been decomposed into a software architecture and successively lower-level requirements in a hierarchical structure. Subsequent reviews, testing, and analyses then confirm that the architecture and low-level requirements have been correctly and completely decomposed into source code. The team creates deliverable verification artifacts to demonstrate that the reviews, tests, and analyses have been performed correctly and with required thoroughness. The team must also demonstrate traceability between the requirements, their implementation, and their verification. Ultimately, the goal is to identify what the software does, verify that it does what it was meant to do (and no more), and verify that it does it correctly. That is, the verification process is intended to provide evidence of correct behavior of the software in all possible circumstances of use.

The purpose of reviewing high-level and low-level requirements is to find errors introduced during the process of defining, deriving, and allocating requirements – e.g., requirements that were incorrectly or incompletely defined or derived, or requirements that were not allocated. This includes confirming that system requirements allocated to software are accurately and completely defined and derived and that none of them conflict with the others, or conflict with the requirements of the target hardware or infrastructural software (e.g., RTOS). This also includes determining that defined and derived requirements can be verified (e.g., can be quantified). In particular, the developer must demonstrate traceability from system requirements to defined and derived software requirements. As requirements are decomposed into lower levels of detail, clear traceability must be maintained through the successive layers to demonstrate that all requirements are implemented and that no requirements have been derived and allocated that cannot be traced back to a system requirement.

This level of comprehension and traceability also applies to the development of the software architecture. Reviews and analysis verify that the software architecture does not conflict with requirements (e.g., required partitioning), that the data and control flow among architectural components is correct, that the architecture does not conflict with the hardware or the infrastructural software (e.g., interrupts, initialization, synchronization, etc.), and that all requirements are allocated to architectural components and that those components contain only features and capabilities that trace back to system requirements.

Review and analysis of the source code is meant to determine the correctness of the code, but again, also to confirm that the developer has implemented in code all requirements from the system level to the lowest level and that nothing has been implemented that cannot be traced back to a system level requirement. The review also verifies that the code is compatible with the software architecture and with the hardware and infrastructural software. This includes demonstrating that the source code implements data and control flow that are compliant and compatible with those defined by the software architecture, that it contains nothing (e.g., capabilities, data structures) that cannot be verified, and that verification does not require the code to be modified. Further, the developer must demonstrate that the code complies with complexity minimization requirements (e.g., minimum coupling among components, minimum nesting, minimum complexity of logical and arithmetic expressions), that it is correct (free of coding errors – e.g., stack overflow, uninitialized stack variable), and that each line of code traces back to a low level requirement and all requirements allocated to that component have been implemented. As above, compliance must be demonstrated and captured in deliverable artifacts.

Subsequently, test cases and test procedures must trace back to code and all code must trace forward to test cases and procedures. Test results demonstrate correctness and completion of the implementation and complete the traceability chain. Software must be tested across normal ranges of inputs, conditions, and outputs, but also across abnormal and error conditions. Stress tests are required to confirm that system behavior cannot be driven to violate requirements (e.g., cannot be driven into infinite loops, or into system states that have no exit conditions).

Given the rigor of the development methodology and its necessary cost and schedule, avionics development projects look for ways of reducing the human time and effort as much as possible. Tools used for this purpose must be qualified. Software verification tools – for example, a tool that would help demonstrate the correct and complete implementation of a low level requirement into executable code – are qualified by demonstrating that they comply with their defined operational requirements under normal operational conditions. This requires development of a requirements document capturing functions and features, installation guides and user manuals, and a description of the tool's operating environment. A tool qualification plan identifies the verification activities that are to be automated, eliminated, or reduced, and describes the tool's architecture, the tests to be performed, and the qualification data to be captured and delivered to the certification authority.

II. Function Extraction Technology for Computing Software Behavior

Given the stringent requirements for correctness and certification of avionics software, it is important for developers to have complete knowledge of the functional behavior of the software in all possible circumstances of use. Traditional code reviews and inspections are helpful but resource intensive and subject to human fallibility. Testing is likewise resource intensive and no process of testing, no matter how well conceived, can reveal full functional behavior. Modern methods such as model checking are an important addition to the verification process, but cannot reveal full behavior. What is needed is an "all cases of behavior" view of software that is difficult to achieve with current methods. This problem motivates a closer look at the possibilities for automated computation of software behavior with mathematical precision at machine speeds.

The objective of behavior calculation is to reveal the net functional effect of software as a means to augment human capabilities for analysis and design. We discuss these possibilities below in the context of sequential logic, in the full knowledge that concurrent and recursive logic must be dealt with as well. Our goal is to say first words on the subject, not last words.

The function-theoretic view of software[5,7,8] provides mathematical foundations for computation of behavior. In this view, programs are treated as rules for mathematical functions or relations, that is, mappings from inputs (domains) to outputs (ranges), regardless of the subject matter addressed or the implementation languages employed. The key to the function-theoretic approach is that, while programs may contain far too many execution paths for humans to comprehend or computers to analyze, every program (and thus every system of programs) can be described as a composition of a finite number of control structures, each of which implements a mathematical function or relation in the transformation of its inputs into outputs. In particular, the sequential logic of programs can be expressed as a finite number of single-entry, single-exit control structures: sequence (composition), alternation (ifthenelse), and iteration (whiledo). The behavior of every control structure in a program can be extracted and composed with others in a stepwise process based on an algebra of functions that traverses the control structure hierarchy. Termination of the extraction and composition processes is assured by the finite number of control structures present in any program. While no comprehensive theory for computing the behavior of iteration structures is possible, satisfactory engineering solutions are possible.

In more detail, function-theoretic foundations prescribe procedure-free equations that represent net effects on data of control structures and provide a starting point for behavior extraction. These equations are expressed in terms of function composition, case analysis, and, for iteration structures, a recursive expression based on an equivalence of iteration and alternation structures. Representative equations are given below for control structures labeled P, data operations g and h, predicate q, and program function f.

The program function of a sequence control structure (P: g; h) is defined by

$$f = [P] = [g; h] = [h] \text{ o } [g]$$

where square brackets enclose the program function and "o" is the composition operator. That is, the program function of a sequence can be calculated by simple composition of its constituent operations.

The program function of an alternation control structure (P: if q then g else h endif) is defined by:

$$f = [P] = [\text{if } q \text{ then } g \text{ else } h \text{ endif}] = ([q] = \text{true} \rightarrow [g] \mid [q] = \text{false} \rightarrow [h])$$

where "|" is the or operator. That is, the program function is a case analysis of the true and false branches, with the possibility of combining them into a more general function such as max, min, etc.

The program function of a terminating iteration control structure (P: while q do g enddo) is defined by

$$f = [P] = [\text{while } q \text{ do } g \text{ enddo}] = [\text{if } q \text{ then } g; \text{ while } q \text{ do } g \text{ enddo endif}] = [\text{if } q \text{ then } g; f \text{ endif}]$$

and f must satisfy the following recursive equation:

$$f = ([q] = \text{true} \rightarrow [f] \text{ o } [g] \mid [q]= \text{false} \rightarrow I)$$

Engineering methods must be developed to transform this recursive equation into a more understandable recursion-free form for human comprehension. Because no general theory for loop behavior computation can exist, it may be the case that a small percentage of loops will always require human insight to be combined with automated analysis.

These equations define an algebra of functions that can be applied bottom up to the control structure hierarchy of a program in a stepwise function extraction process, whereby each control structure is in turn transformed to procedure-free functional form. This process propagates and preserves the net effects of control structures through successive levels of abstraction while leaving behind complexities of local computations and data not required for expressing behavior at higher levels.

In notional illustration of the behavior computation process, consider the following miniature program fragment composed of five control structures that operate on integer variables (machine precision aside):

```
if x > y
then
  t := x;
  x := y;
  y := t
else
  x := x + y;
  y := x – y;
  x := x – y
endif;
t := x;
x := y;
y := t
```

The composition computations for the thenpart and elsepart sequence structures are given by accumulating trace tables,

Operation	**t**	**x**	**y**
t := x	t = x	unchanged	unchanged
x := y	unchanged	x = y	unchanged
y := t	unchanged	unchanged	y = x

which produces the sequence-free concurrent assignment

t, x, y := x, y, x

where ":=" is the assignment operator, and

Operation	**x**	**y**
x := x + y	x = x + y	unchanged
y := x – y	unchanged	y = x + y – y = x
x := x – y	x = x + y – x = y	unchanged

which produces the sequence-free concurrent assignment

x, y := y, x.

These functions can be combined in the ifthenelse alternation analysis as a conditional concurrent assignment:

x > y → (t, x, y := x, y, x) | x <= y → (x, y := y, x)

Treating t as a local variable, both cases of the alternation compute x, y := y, x, that is, the values of x and y are exchanged no matter how the iftest evaluates. Thus, the net computed behavior for the alternation is

x, y := y, x.

The trace table for the final sequence of three operations is identical to the first one above, giving the sequence-free concurrent assignment

x, y := y, x

and thus the entire program fragment reduces to a sequence of two functions:

```
x, y := y, x;
x, y := y, x
```

The trace table computation for the sequence

Operation	x	y
x, y := y, x	y	x
x, y := y, x	x	y

reveals the overall behavior to be

```
x, y := x, y
```

or simply the identity function, and the possibility arises of a specification or programming error, as the program has no functional effect.

The general form of the expressions produced by function extraction is a set of conditional concurrent assignments (CCAs) organized into catalogs that define program behavior in all circumstances of use. The CCAs are disjoint and thus partition behavior on the input domain of a program. The catalogs define behavior in non-procedural form, and represent the as-built specification of a program. Each CCA is composed of a predicate on the input domain, which, if true, results in simultaneous assignment of all right-hand side domain values in the concurrent assignments to their left-hand side range variables. Catalogs can be queried, for example, for particular behavior cases of interest, or to determine if any cases satisfy, or violate, specified conditions or constraints. Behavior catalogs have many uses, ranging from correctness verification to analysis of specific properties to component composition.

To explore the potential of FX technology, CERT STAR*Lab developed a proof-of-concept prototype that calculates the behavior of programs written in a small subset of Java and presents it to users in the form of behavior catalogs. A controlled experiment was conducted to compare use of traditional methods of program reading and inspection with FX-based methods[3]. Experienced programmers were divided into a control group using traditional techniques and an experimental group using the FX prototype. Each group answered questions dealing with comprehension and verification of three Java programs of increasing difficulty.

Results of the experiment showed that the group using the FX prototype reduced the time required to derive the functional behavior of the most difficult program by about three orders of magnitude over the control group, was about four times better at providing correct answers to the program comprehension and verification questions, and required about one-fourth of the time to do so. The FX group achieved these results with 45 minutes of instruction on use of the prototype, compared to years of education and experience for the control group[3].

III. Function Extraction Application to Avionics Software

The objective of function extraction technology is to compute the behavior of software to the maximum extent possible with mathematical precision[4,6,7,9]. CERT STAR*Lab is currently engaged in developing an operational function extraction system that computes the behavior of programs expressed in or compiled into Intel assembly language. This system expresses an input program in terms of the functional semantics of its assembly language instructions, transforms the program to structured form, and computes the behavior of each of its constituent control structures in a stepwise abstraction process. Machine precision is taken into account. The resulting behavior catalog can be inspected for conformance to requirements, as well as queried and analyzed by automated processes to verify specific properties and behaviors of interest.

In notional illustration of how a function extraction system could be employed to compute the behavior of avionics software components, consider the following contrived example that fits within the space constraints of the paper. Imagine investigating the behavior of the program depicted in Figure 1. The program requirement is to set register EAX to an angle value, based on certain conditions, that is used by its invoking program as the denominator in a tangent calculation. It is important to ensure that the angle is never set to 90 degrees, which would cause the calculation to fail. For this illustration, the requirement has been implemented in an intentionally obscure manner to simulate the difficulty of understanding larger programs that exhibit more complexity. For example, the program contains substantial spaghetti logic expressed as jumps in the code. The point is, here is a program that would require substantial resource-intensive human analysis, with full human fallibility, to determine a key property. In what follows, we will show the alternative process of behavior computation with full precision at machine speeds.

```
    ; pi radians = 180
    ; eax is number of degrees

        push eax
        push ebx
        jl skip
        jmp elseO
 skip:
        push eax
        jge elseT
        push ebx
        pop eax
        pop ebx
        pop eax
        pop ebx
        sub ebx, ebx
        sub eax, ebx
        add ebx, eax
        jmp target4

target2:
         xor eax, 0x10
         add eax,10
         jmp endT

target3:
        add eax, ecx
        xor eax, 0x40
        pop ecx
        jmp target2

target4:
        push ecx
```

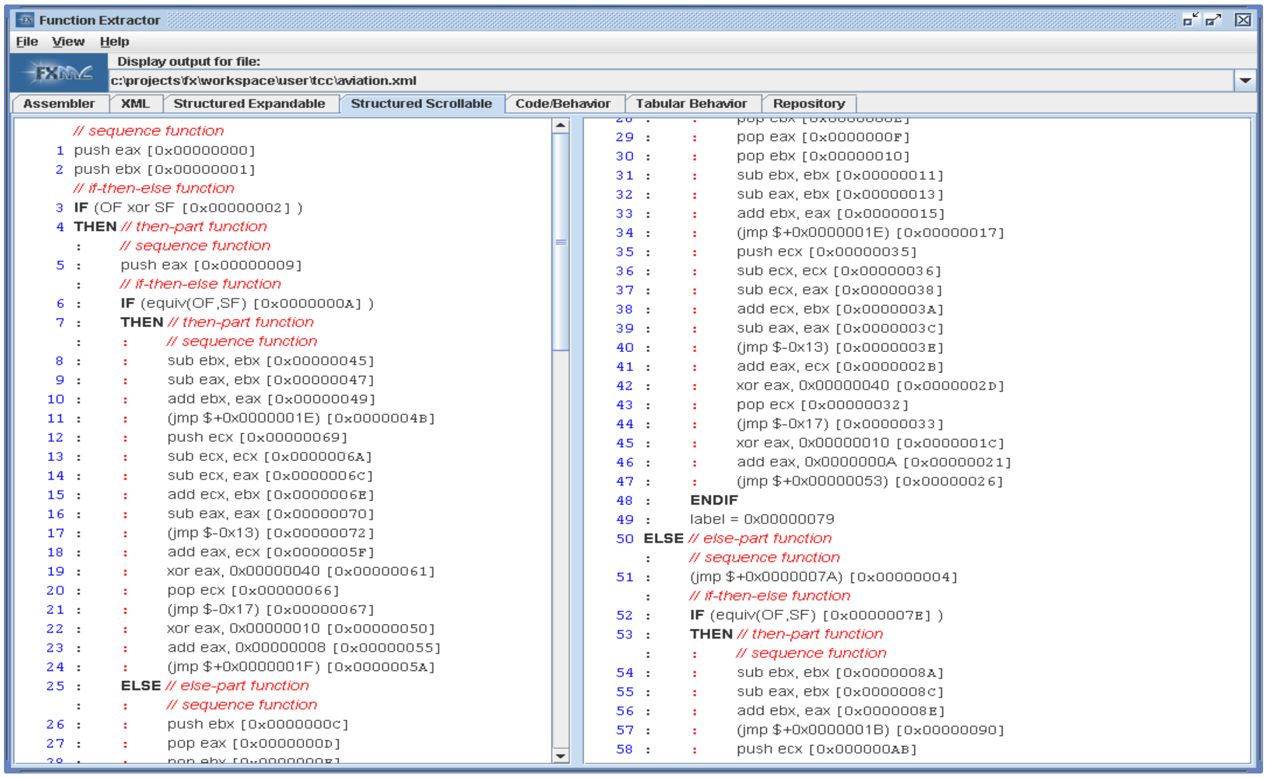
```
        sub ecx, ecx
        sub ecx, eax
        add ecx, ebx
        sub eax, eax
        jmp target3
        jmp endT
elseT:
        sub ebx, ebx
        sub eax, ebx
        add ebx, eax
        jmp target41

target21:
        xor eax, 0x10
        add eax,8
        jmp endT

target31:
        add eax, ecx
        xor eax, 0x40
        pop ecx
        jmp target21

target41:
        push ecx
        sub ecx, ecx
        sub ecx, eax
        add ecx, ebx
        sub eax, eax
        jmp target31
        jmp endT
endT:
        jmp endO
elseO:
```

```
        jge elseE
        add eax,7
        jmp endE
elseE:
        sub ebx, ebx
        sub eax, ebx
        add ebx, eax
        jmp target43

target23:
        xor eax, 0x10
        add eax,16
        jmp endT

target33:
        add eax, ecx
        xor eax, 0x40
        pop ecx
        jmp target23

target43:
        push ecx
        sub ecx, ecx
        sub ecx, eax
        add ecx, ebx
        sub eax, eax
        jmp target33
        jmp endT


endE:
endO:
```

Figure 1. The Assembly Language Program Input to the FX System.

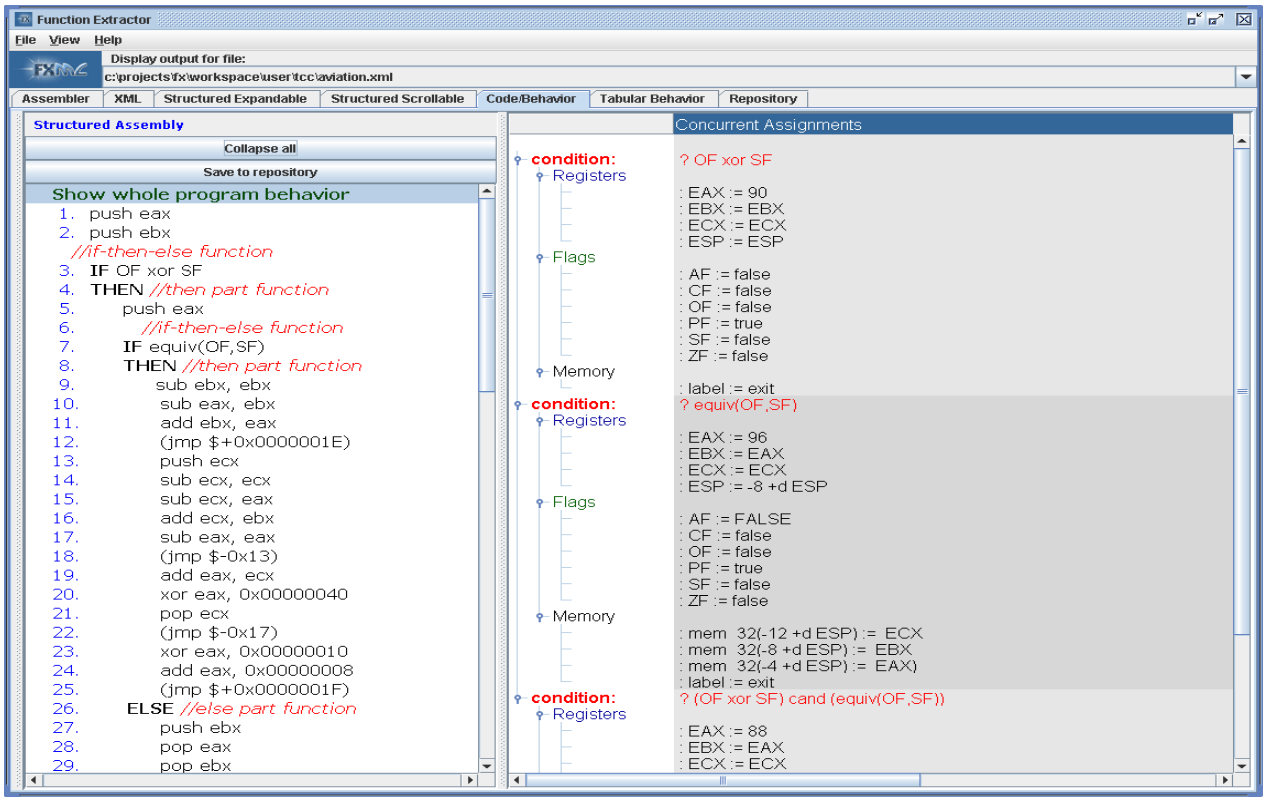
Figure 2 shows output from the function extraction system, a partial view of the function-equivalent structured version of the input program expressed in terms of standard control structures with all arbitrary branching logic eliminated. Jump instructions are retained in the text as comments for traceability, but have no functional effect.

Figure 3 depicts a portion of the structured logic on the left, and the computed behavior on the right. Assembly language programs can set registers (EAX, EBX, etc.), flags (OF – overflow flag, CF – carry flag, SF – sign flag, etc.) and memory locations. As described above, the behavior is organized into disjoint cases expressed as conditional concurrent assignments. Each case defines a predicate condition which, if true on entry, results in the defined concurrent assignments of expressions of initial values on entry to final values on exit. Thus, for the first case, if the exclusive or of the overflow flag and the carry flag is true, register EAX is indeed set to the undesired value of 90 degrees. The other two cases set EAX to the acceptable values of 96 and 88 degrees, respectively.



```
    // sequence function
    push eax [0x00000000]
    push ebx [0x00000001]
    // if-then-else function
    IF (OF xor SF [0x00000002] )
    THEN // then-part function
  :      // sequence function
  :      push eax [0x00000009]
  :      // if-then-else function
  :      IF (equiv(OF,SF) [0x0000000A] )
  :      THEN // then-part function
  :      :      // sequence function
  :      :      sub ebx, ebx [0x00000045]
  :      :      sub eax, ebx [0x00000047]
  :      :      add ebx, eax [0x00000049]
  :      :      (jmp $+0x0000001E) [0x0000004B]
  :      :      push ecx [0x00000069]
  :      :      sub ecx, ecx [0x0000006A]
  :      :      sub ecx, eax [0x0000006C]
  :      :      add ecx, ebx [0x0000006E]
  :      :      sub eax, eax [0x00000070]
  :      :      (jmp $-0x13) [0x00000072]
  :      :      add eax, ecx [0x0000005F]
  :      :      xor eax, 0x00000040 [0x00000061]
  :      :      pop ecx [0x00000066]
  :      :      (jmp $-0x17) [0x00000067]
  :      :      xor eax, 0x00000010 [0x00000050]
  :      :      add eax, 0x00000008 [0x00000055]
  :      :      (jmp $+0x0000001F) [0x0000005A]
  :      ELSE // else-part function
  :      :      // sequence function
  :      :      push ebx [0x0000000C]
  :      :      pop eax [0x0000000D]
  :      :      pop ebx [0x0000000E]
  :      :      pop eax [0x0000000F]
  :      :      pop ebx [0x00000010]
  :      :      sub ebx, ebx [0x00000011]
  :      :      sub eax, ebx [0x00000013]
  :      :      add ebx, eax [0x00000015]
  :      :      (jmp $+0x0000001E) [0x00000017]
  :      :      push ecx [0x00000035]
  :      :      sub ecx, ecx [0x00000036]
  :      :      sub ecx, eax [0x00000038]
  :      :      add ecx, ebx [0x0000003A]
  :      :      sub eax, eax [0x0000003C]
  :      :      (jmp $-0x13) [0x0000003E]
  :      :      add eax, ecx [0x0000002B]
  :      :      xor eax, 0x00000040 [0x0000002D]
  :      :      pop ecx [0x00000032]
  :      :      (jmp $-0x17) [0x00000033]
  :      :      xor eax, 0x00000010 [0x0000001C]
  :      :      add eax, 0x0000000A [0x00000021]
  :      :      (jmp $+0x00000053) [0x00000026]
  :      ENDIF
  :      label = 0x00000079
    ELSE // else-part function
  :      // sequence function
  :      (jmp $+0x0000007A) [0x00000004]
  :      // if-then-else function
  :      IF (equiv(OF,SF) [0x0000007E] )
  :      THEN // then-part function
  :      :      // sequence function
  :      :      sub ebx, ebx [0x0000008A]
  :      :      sub eax, ebx [0x0000008C]
  :      :      add ebx, eax [0x0000008E]
  :      :      (jmp $+0x0000001B) [0x00000090]
  :      :      push ecx [0x000000AB]
```

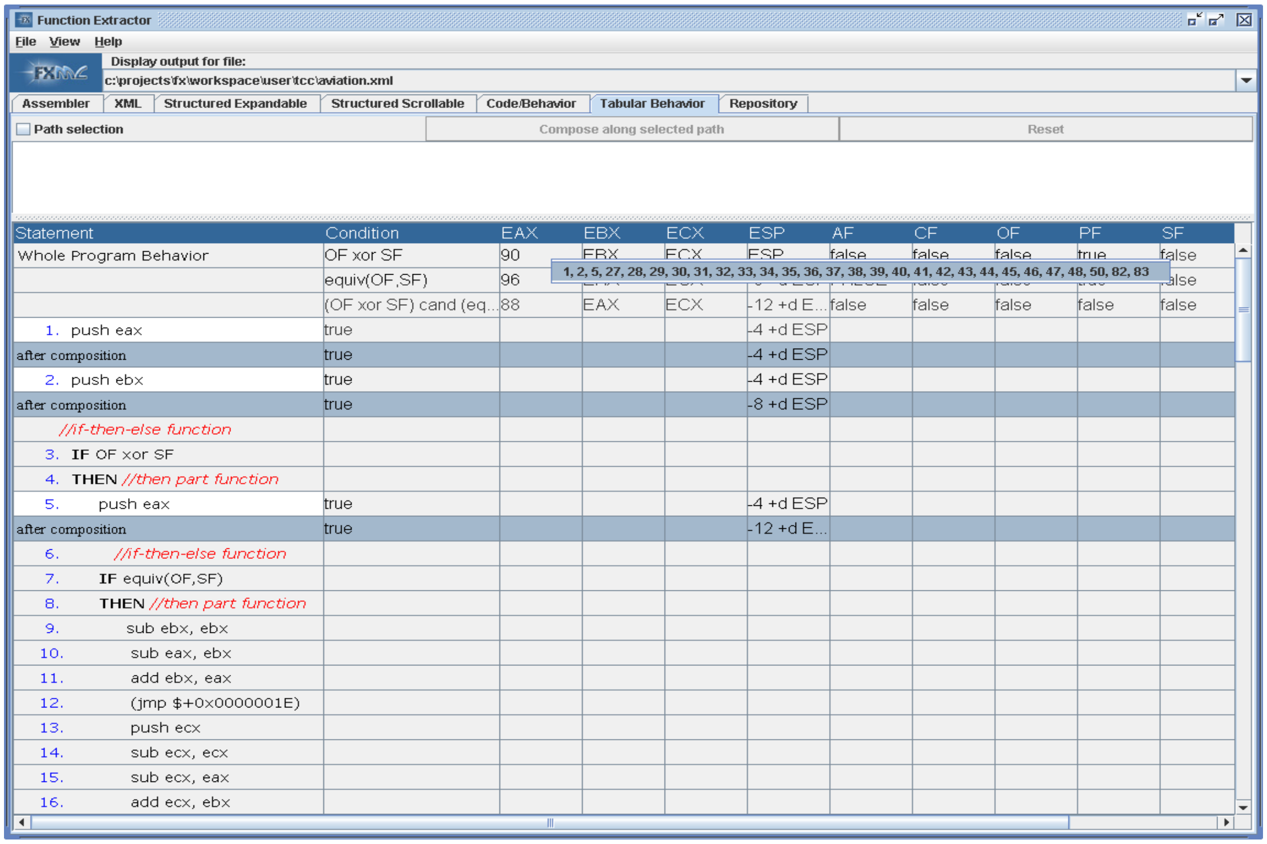
Figure 2. FX-Generated Function-Equivalent Structured Version of the Input Program (partial view).



```
push eax
push ebx
//if-then-else function
IF OF xor SF
THEN //then part function
     push eax
     //if-then-else function
     IF equiv(OF,SF)
     THEN //then part function
          sub ebx, ebx
          sub eax, ebx
          add ebx, eax
          (jmp $+0x0000001E)
          push ecx
          sub ecx, ecx
          sub ecx, eax
          add ecx, ebx
          sub eax, eax
          (jmp $-0x13)
          add eax, ecx
          xor eax, 0x00000040
          pop ecx
          (jmp $-0x17)
          xor eax, 0x00000010
          add eax, 0x00000008
          (jmp $+0x0000001F)
     ELSE //else part function
          push ebx
          pop eax
          pop ebx
```

```
condition:      ? OF xor SF
  Registers
                : EAX := 90
                : EBX := EBX
                : ECX := ECX
                : ESP := ESP
  Flags
                : AF := false
                : CF := false
                : OF := false
                : PF := true
                : SF := false
                : ZF := false
  Memory
                : label := exit
condition:      ? equiv(OF,SF)
  Registers
                : EAX := 96
                : EBX := EAX
                : ECX := ECX
                : ESP := -8 +d ESP
  Flags
                : AF := FALSE
                : CF := false
                : OF := false
                : PF := true
                : SF := false
                : ZF := false
  Memory
                : mem 32(-12 +d ESP) := ECX
                : mem 32(-8 +d ESP) := EBX
                : mem 32(-4 +d ESP) := EAX)
                : label := exit
condition:      ? (OF xor SF) cand (equiv(OF,SF))
  Registers
                : EAX := 88
                : EBX := EAX
                : ECX := ECX
```

Figure 3. FX-Generated Computed Behavior of the Input Program Expressed in Three Cases.


Function Extractor

File View Help

Display output for file: c:\projects\fx\workspace\user\tcc\aviation.xml

Assembler | XML | Structured Expandable | Structured Scrollable | Code/Behavior | Tabular Behavior | Repository

Path selection | Compose along selected path | Reset

Statement	Condition	EAX	EBX	ECX	ESP	AF	CF	OF	PF	SF
Whole Program Behavior	OF xor SF	90	EBX	ECX	ESP	false	false	false	true	false
	equiv(OF,SF)	96	EAX	ECX	[illegible] +d ESP	FALSE	false	false	true	false
	(OF xor SF) cand (eq...	88	EAX	ECX	-12 +d E...	false	false	false	false	false
1. push eax	true				-4 +d ESP					
after composition	true				-4 +d ESP					
2. push ebx	true				-4 +d ESP					
after composition	true				-8 +d ESP					
//if-then-else function										
3. IF OF xor SF										
4. THEN //then part function										
5. push eax	true				-4 +d ESP					
after composition	true				-12 +d E...					
6. //if-then-else function										
7. IF equiv(OF,SF)										
8. THEN //then part function										
9. sub ebx, ebx										
10. sub eax, ebx										
11. add ebx, eax										
12. (jmp $+0x0000001E)										
13. push ecx										
14. sub ecx, ecx										
15. sub ecx, eax										
16. add ecx, ebx										

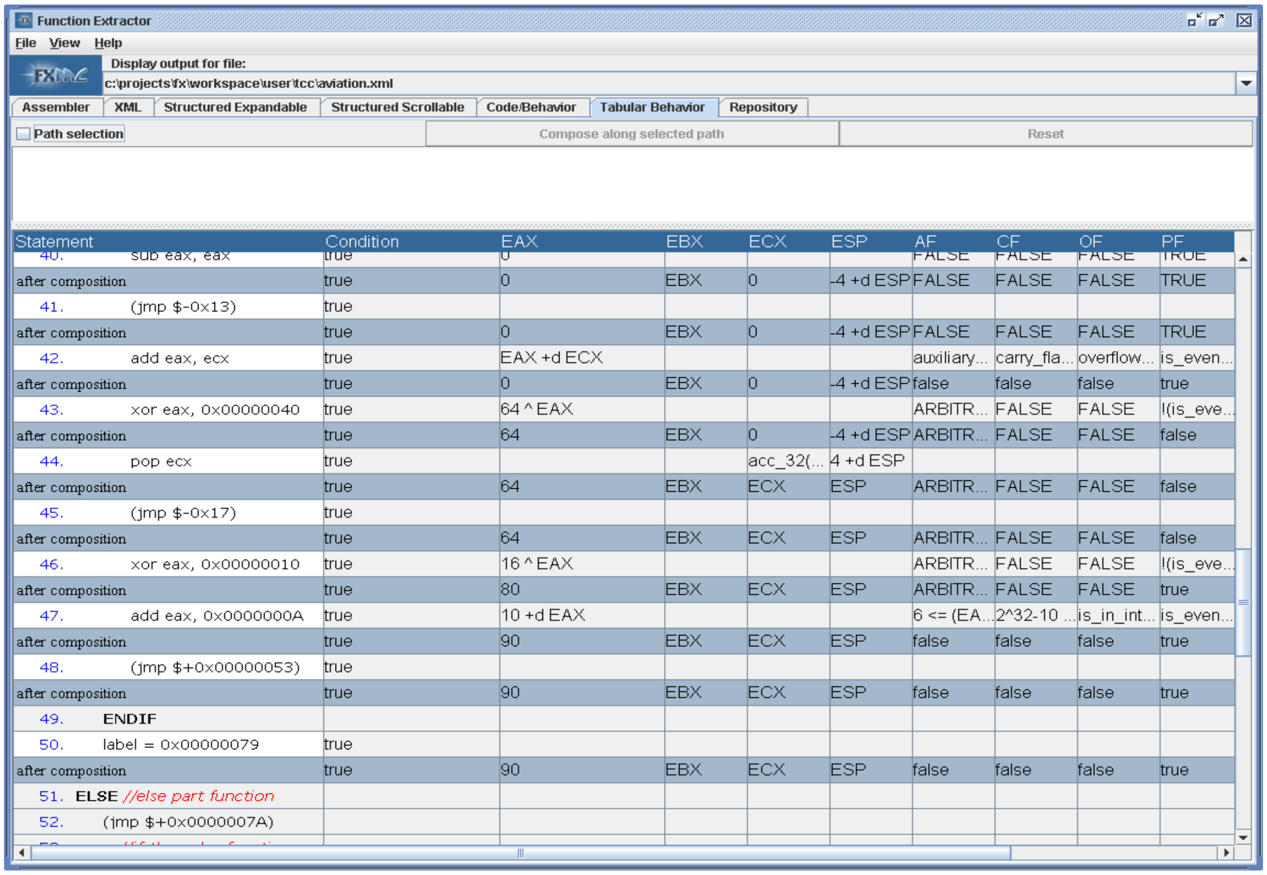
1, 2, 5, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 50, 82, 83

Figure 4. FX-Generated Trace of the Pathway to the Incorrect Behavior.

The next task is to determine how register EAX is set to 90 degrees in the code in order to fix the problem. Figure 4 depicts another FX system capability that can be used for this purpose. The first three lines of this tabular view of behavior show the same three cases of behavior, one row per case, with the condition and associated concurrent assignments in the columns. Clicking on row one, the incorrect behavior, produces a list of the line numbers of the instructions whose accumulating behavior produces the 90 degree value. Clicking on this list produces the display of Figure 5, which shows part of the corresponding progression of calculations. Each line labeled "after composition" gives the net effect of the computations to that point, resulting in setting register EAX to 90 at line 47. Armed with this tracing information, the developer can make the necessary changes in the program and rerun the extraction process to verify that the modified program is indeed correct.

IV. Implications for Avionics Software Certification

Function extraction is an emerging technology that has the potential to substantially impact verification and certification of avionics software. FX can provide precise computation of functional behavior at low cost compared to current resource-intensive methods. Most importantly, the final calculated behavior represents repeatable and documented assurance evidence required for certification of the software. In the long run, availability of automated behavior computation may help reduce cost and cycle time for avionics software development.


Function Extractor
File View Help
Display output for file:
c:\projects\fx\workspace\user\tcc\aviation.xml
Assembler XML Structured Expandable Structured Scrollable Code/Behavior Tabular Behavior Repository
Path selection
Compose along selected path
Reset

Statement	Condition	EAX	EBX	ECX	ESP	AF	CF	OF	PF
40. sub eax, eax	true	0				FALSE	FALSE	FALSE	TRUE
after composition	true	0	EBX	0	-4 +d ESP	FALSE	FALSE	FALSE	TRUE
41. (jmp $-0x13)	true								
after composition	true	0	EBX	0	-4 +d ESP	FALSE	FALSE	FALSE	TRUE
42. add eax, ecx	true	EAX +d ECX				auxiliary...	carry_fla...	overflow...	is_even...
after composition	true	0	EBX	0	-4 +d ESP	false	false	false	true
43. xor eax, 0x00000040	true	64 ^ EAX				ARBITR...	FALSE	FALSE	!(is_eve...
after composition	true	64	EBX	0	-4 +d ESP	ARBITR...	FALSE	FALSE	false
44. pop ecx	true			acc_32(...	4 +d ESP				
after composition	true	64	EBX	ECX	ESP	ARBITR...	FALSE	FALSE	false
45. (jmp $-0x17)	true								
after composition	true	64	EBX	ECX	ESP	ARBITR...	FALSE	FALSE	false
46. xor eax, 0x00000010	true	16 ^ EAX				ARBITR...	FALSE	FALSE	!(is_eve...
after composition	true	80	EBX	ECX	ESP	ARBITR...	FALSE	FALSE	true
47. add eax, 0x0000000A	true	10 +d EAX				6 <= (EA...	2^32-10 ...	is_in_int...	is_even...
after composition	true	90	EBX	ECX	ESP	false	false	false	true
48. (jmp $+0x00000053)	true								
after composition	true	90	EBX	ECX	ESP	false	false	false	true
49. ENDIF									
50. label = 0x00000079	true								
after composition	true	90	EBX	ECX	ESP	false	false	false	true
51. ELSE //else part function									
52. (jmp $+0x0000007A)									

Figure 5. FX Behavior Accumulation Across Instructions That Contribute to the Incorrect Behavior.

References

[1]Federal Aviation Administration, *Advisory Circular 20-115B: RTCA, Inc., Document RTCA/DO-178B*, 11 January 1993.

[2]RTCA SC-167, *RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, RTCA Inc., 1992.

[3]Collins, R., Walton, G., Hevner, A., and Linger, R. *The CERT Function Extraction Experiment: Quantifying FX Impact on Software Comprehension and Verification*, Technical Note CMU/SEI-2005-TN-047, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.

[4]Hevner, A., Linger, R., Collins, R., Pleszkoch, M., Prowell, S., and Walton, G. *The Impact of Function Extraction Technology on Next-Generation Software Engineering*, Technical Report CMU/SEI-2005-TR-015, Software Engineering Institute, Carnegie Mellon University, July 2005.

[5]R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Inc., 1979.

[6]Linger, R. and Pleszkoch, M. "Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior," *Proceedings of the 37th Annual Hawaii International Conference on System Science (HICSS35)*, Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 2004.

[7]R. Linger, M. Pleszkoch, L. Burns, A. Hevner, and G. Walton, "Next-Generation Software Engineering: Function Extraction for Computation of Software Behavior," *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS40)*, Hawaii, IEEE Computer Society Press, Los Alamitos, CA, Jan. 2007.

[8]Prowell, S., Trammell, C., Linger, R., and Poore, J. *Cleanroom Software Engineering: Technology and Practice*, Addison Wesley, Reading, MA, 1999.

[9]Walton, G., Longstaff, T, and Linger, R., *Technology Foundations for Computational Evaluation of Security Attributes*, Technical Report CMU/SEI-2006-TR-021, Software Engineering Institute, Carnegie Mellon University, December 2006.