FXplorer: Exploration of Software Behavior
A New Approach to Code Understanding and Verification

Luanne Burns
Timothy Daly
Richard Linger
*CERT STAR*Lab*
Software Engineering Institute
Carnegie Mellon University
lburns@cert.org
tdaly@cert.org
rlinger@sei.cmu.edu

Abstract

The craft of software understanding and verification can benefit from technologies that enable evolution toward a true engineering discipline. In current practice, software developers lack practical means to determine the full functional behavior of programs under development, and even the most thorough testing can provide only partial knowledge of behaviors. Thus, an effective technology for revealing software behaviors could have a positive impact on software understanding. This paper describes the emerging technology of function extraction (FX) for computing the full behavior of programs and how the knowledge of program behavior can be used in user-directed program exploration for code understanding and verification. We explore how the use of FX technologies can transform methods for functional verification of software. Several examples are presented illustrating the FXplorer interface and its use in exploring the behavior of programs, a capability that, without function extraction technology, has not been possible until now.

1. Transforming Software Understanding

FXplorer an example of a value-added software understanding application that uses function extraction technology to provide capabilities current tools cannot match.

The objective of function extraction technology is to compute the behavior of software to the maximum extent possible with mathematical precision.

Computed behavior defines what a program does in all possible circumstances of use. It is the as-built specification of the code.

Routine availability of computed software behavior will permit development of many value-added applications with capabilities beyond what is possible today.

These applications will enable knowledge generation and informed decisions all the way from the bedrock level of IT infrastructure right on up to the executive suite.

FXplorer provides a unique and different view into program behavior and how that behavior accumulates as a program executes. These views provide new approaches and strategies in software understanding and verification. In section 2 we discussion the concepts of Function Extraction and the function-theoretic view of software as the mathematical foundation for the computation of behavior. Section 3 describes the FX system that implements such a system. Section 4 illustrates the of concepts underlying FXplorer as a value-added software application made possible through the use of function extraction technology and section 5 describes the FXplorer interface. Section 6 gives a brief discussion of FXplorer's underlying algorithm for computing all pathways through a program using its computed behavior. Finally, section 6 discusses FXplorer impact and future direction.

2. Function Extraction Concepts

CERT STAR*Lab of the Software Engineering Institute at Carnegie Mellon University is conducting research and development in the emerging technology of function extraction. The objective is to compute the behavior of software to the maximum extent possible with mathematical precision. FX presents an opportunity to reduce dependencies on slow and costly testing processes to assess software functionality by moving to fast and cheap computation of functionality at machine speeds.

The goal of behavior computation is to compose and record the semantic information in programs in order to augment human capabilities for analysis, design, and verification. In the current paper we limit the discussion of function extraction to the domain of sequential logic, postponing concurrent and recursive topics. Computing the behavior of programs is a difficult problem, and our intent is to say the first words on the subject, not the last words.

The well-known function-theoretic view of software provides mathematical foundations for computation of behavior [Linger et al. 1979, Pleszkoch et al. 1990, Prowell et al. 1999]. In this perspective, programs are treated as rules for mathematical functions or relations, that is, mappings from inputs (domains) to outputs (ranges), regardless of subject matter addressed or implementation languages employed.

The key to the function-theoretic approach is the recognition that, while programs may contain far too many execution paths for humans to understand 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), with variants and extensions permitted but not necessary. 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 function extraction and composition processes are assured by the finite number of control structures present in any program [Linger and Pleszkoch 2004].

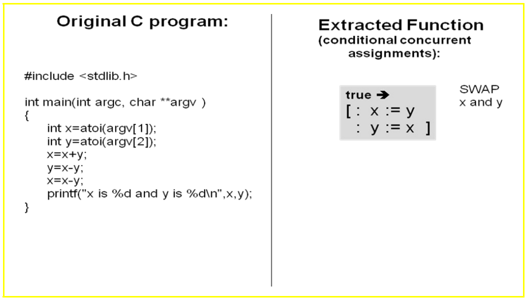
The first step in behavior extraction is to transform any spaghetti logic in the input program into structured form, to create a hierarchy of nested and sequenced control structures. The behavior of leaf node control structures is then computed with net effects propagated to the next level while local details of processing and data are left behind. These computations reveal new leaf nodes and the process repeats until all behavior has been computed.

Behavior computation for sequence and alternation structures involves composition and case analysis. Because no comprehensive theory for loop behavior computation can exist, mathematical foundations and engineering implementations short of a general theory but sufficient for practical use has been developed for use in FX [Mili et al. 2007].

The general form of the expressions produced by function extraction is a set of conditional concurrent assignments (CCA) organized into behavior databases 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 behavior databases 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. The left side of Figure 1 shows a program that swaps two variables, *x* and *y*; the right side shows the behavior of the program as conditional concurrent assignments. Note that there are many algorithm alternatives that one might choose for doing the swap but all would result in the same extraction.

Behavior databases, thus, are the central repository for the actual behaviors contained in a software system. The behavior databases 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 databases have many uses ranging from basic human understanding of code, to program correctness verification, to analysis of security and other attributes, to component composition, and so on [Hevner et al. 2005].

The first application of FX technology is to programs written in, or compiled into, Intel assembly language to support analysts in malicious code detection and understanding of malware behaviors. Sample outputs from the evolving FX system are employed later in the paper to illustrate the role of behavior computation as a means to debugging programs.



```
#include <stdlib.h>

int main(int argc, char **argv )
{
     int x=atoi(argv[1]);
     int y=atoi(argv[2]);
     x=x+y;
     y=x-y;
     x=x-y;
     printf("x is %d and y is %d\n",x,y);
}
```

```
true ➔
[ :  x := y
  :  y := x  ]
```

Figure 1: Swap program and extraction

3. Function Extraction Examples

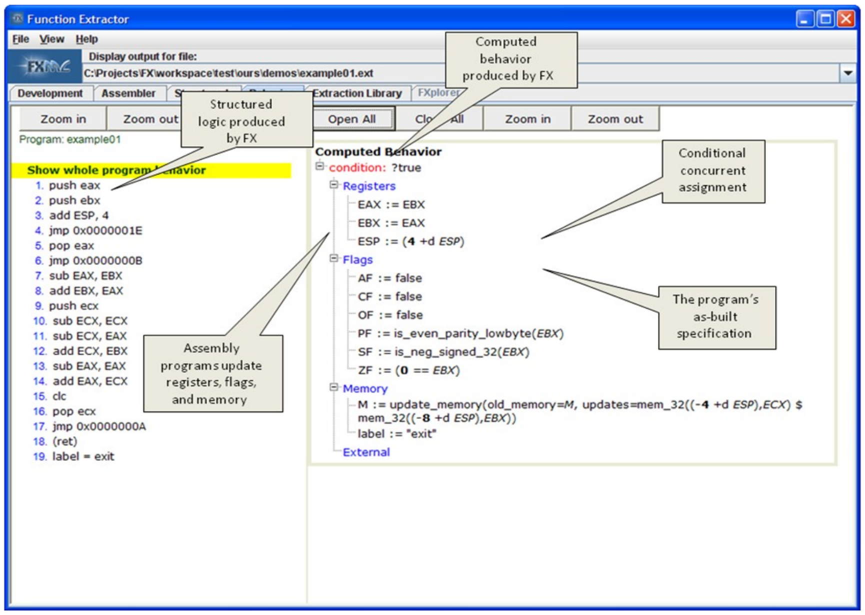
In the example shown in Figure 2, the assembler source code for a short program is shown. It contains statements such as push and pop, add, subtract, and jump. The code contains spaghetti logic with various jumps throughout. A code analyst would need to manually trace through the statements to determine the program behavior. Using the FX system, however, the analyst is able to determine the behavior of this sequence with the push of a button.

```
//begin sequence
top:
:       // sequence function
:       push eax
:       push ebx
:       add esp, 0x00000004
:       (jmp 0x0000001E)
:       pop eax
:       (jmp $-0x14)
:       sub eax, ebx
:       add ebx, eax
:       push ecx
:       sub ecx, ecx
:       sub ecx, eax
:       add ecx, ebx
:       sub eax, eax
:       add eax, ecx
:       clc
:       pop ecx
:       (jmp $-0x12)
:       (ret)
:       label = exit
```

Figure 2: Assembler source

Figure 3 shows the FX Code/Behavior tab. The left side of the split pane shows the original assembly language sequence. The jumps are shown in parentheses for traceability but they have been untangled and removed from the sequence. The right side of the split pane shows the automatically generated behavior database for the code. The equations in the gray box are conditional concurrent assignments; the left hand sides represent final values while the right hand sides are initial values; all equations are assigned concurrently, not sequentially.

Note here that EAX register has been assigned the initial value of the EBX register and EBX. This code represents a swap of the two registers.


Function Extractor
File View Help
Display output for file:
C:\Projects\FX\workspace\test\ours\demos\example01.ext
Development
Assembler
Extraction Library
FXplorer
Zoom in
Zoom out
Open All
Zoom in
Zoom out
Structured logic produced by FX
Computed behavior produced by FX
Conditional concurrent assignment
The program's as-built specification
Assembly programs update registers, flags, and memory

Program: example01

Show whole program behavior

```
push eax
push ebx
add ESP, 4
jmp 0x0000001E
pop eax
jmp 0x0000000B
sub EAX, EBX
add EBX, EAX
push ecx
sub ECX, ECX
sub ECX, EAX
add ECX, EBX
sub EAX, EAX
add EAX, ECX
clc
pop ecx
jmp 0x0000000A
(ret)
label = exit
```

```
Computed Behavior
⊟ condition: ?true
  ⊟ Registers
      EAX := EBX
      EBX := EAX
      ESP := (4 +d ESP)
  ⊟ Flags
      AF := false
      CF := false
      OF := false
      PF := is_even_parity_lowbyte(EBX)
      SF := is_neg_signed_32(EBX)
      ZF := (0 == EBX)
  ⊟ Memory
      M := update_memory(old_memory=M, updates=mem_32((-4 +d ESP),ECX) $
      mem_32((-8 +d ESP),EBX))
      label := "exit"
    External
```

Figure 3: FX Code/Behavior View

4. Behavior Exploration with FX Technology

As a starting point, the Code/Behavior tab in FX allows the user to see the resulting behavior database for the whole program, as well as the behavior for each individual control structure or statement. These behaviors are defined in terms of conditional concurrent assignments. FXplorer allows the user to see behavior across statements, that is, the composition, or net effect, of accumulating behavior from one statement or structure to the next. This greatly assists a programmer in debugging and following the execution of his programs.In essence, FXplorer allows user-controlled behavior exploration. The knowledge of program behavior gives new exploratory power to programmers in the debugging phase of their development.

The three FXplorer capabilities are called:

- **BehaviorCase** or *Path Quest*
- **BehaviorPath** or *Connect the Dots*
- **BehaviorHere** or *Come Here*

By default, FX displays the whole program behavior database. Using this display, the user might decide that one or more of the behaviors looks suspicious or erroneous. He might want to know which code statements and their accumulating behaviors contribute to the case in question.

BehaviorCase, FXplorer's "PathQuest" function, starts with a user-selected case in the behavior database of a program. It determines and displays the compositions of all the accumulating behavior along all the code paths that produce that case. All other code and behavior is eliminated. Thus, the programmer can find out what part of the original program is responsible for a given result.

BehaviorPath, or "Connect the Dots," starts with a user-selected code path through the program. It determines and displays all the compositions of the accumulating behavior along that path. Thus, the programmer can examine a particular path through the program to see the final behavior it causes.

BehaviorHere, or FXplorer's "Come Here" function, starts with a user-selected statement in the program. It determines and displays the compositions of all the accumulating behaviors along all possible code paths to that statement. Thus, the programmer can find a particular point in a program and see all the paths and behaviors leading to that point.

These three functions provide a unique way of understanding a program. It allows direct answers to common programmer questions like: "Where did this result come from?" (BehaviorCase), "What happens if we chose this path?" (BehaviorPath), and "How can I get here?" (BehaviorHere). The ability to answer these questions in full without doing a line-by-line analysis greatly improves the programmer's ability to understand program behavior, to verify that the results are correct, and to validate the results against a specification.

5. Exploring the User Interface

Glance ahead to Figure 7 for a moment to see the basic structure of the FXplorer interface; FXplorer provides a tabular view of program behavior where the rows are program statements and the columns are registers, flags, and memory values. Think of the tabular view as a checkbook register. Each statement is an entry in the register; the dark shaded lines labeled "after composition" are the "balance", if you will, resulting after each statement. The light shaded line, statement 8, is the current statement. The first two rows of this table represent the behavior database, or all behaviors, for the program as a whole.

Now let's look at the accumulation of behavior.
For statement 1, note that 1 is added to the value of EAX. The darker blue line below statement 1 shows the behavior afterwards.

Statement 2 then adds 20 to EBX and the following line shows the accumulated behavior after both statements 1 and 2.

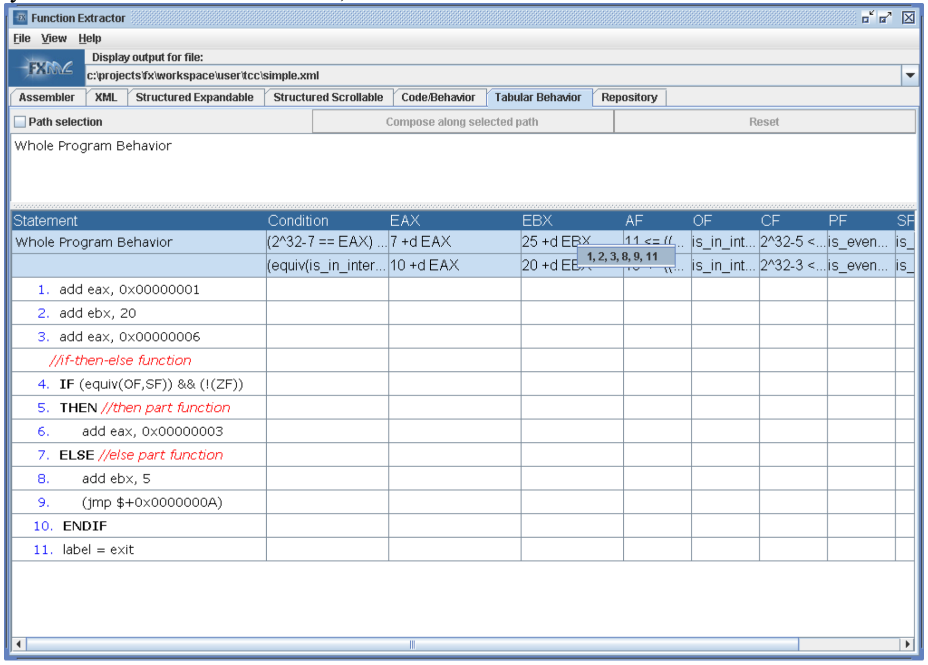
Finally, Statement 3 adds 6 to EAX and the following composition line shows the accumulated behavior after statements 1, 2, and 3.

The tabular view allows the user to resize and move columns as desired. Resizing is done by dragging the title area of the column header. Moving a column is done by simply dragging the column to a new location in the table. This allows the user to place columns of interest in proximity to each other. If the content of a cell in the table is too long to be displayed without increasing the column width to an unreasonable size, the cell can be clicked on and the contents will be displayed in the top area of the table.

Using this example, let's suppose a user starts by viewing the whole program behavior database in FX as shown in Figure 4 and finds Case 1 to be of particular interest; that is, when EAX has been incremented by 7 and EBX by 25.

Using FXplorer, when the user right clicks anywhere on the row with that case, a drop down menu is displayed showing all pathways through the code that will result in this behavior. The sequence of statement numbers is displayed on the menu. In this case, there is only one pathway that results in this behavior. This is illustrated in Figure 4.

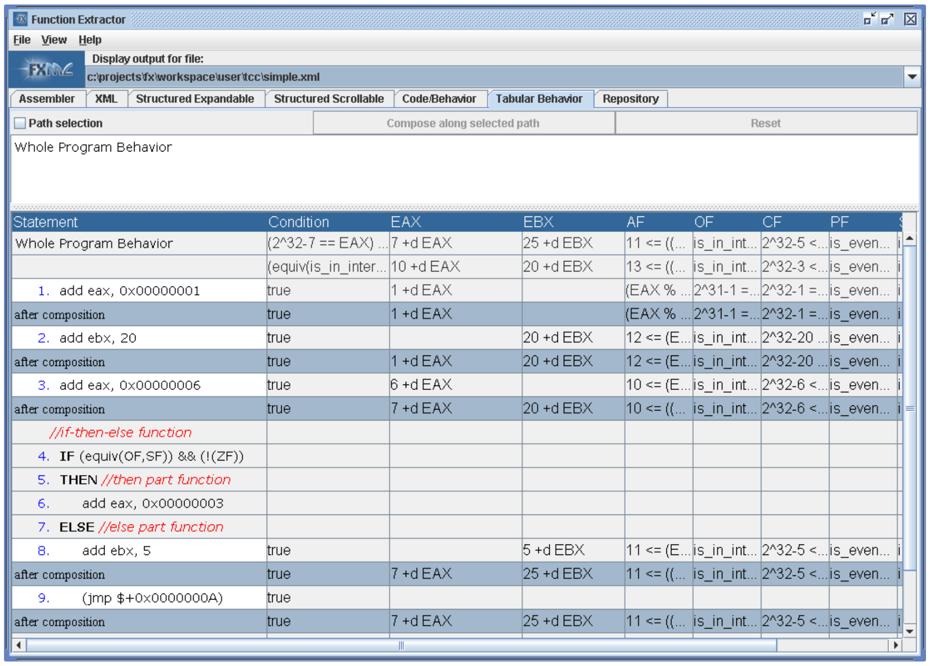
By clicking on the path dropdown menu, FXplorer "Path Quest" displays the statement sequence and the



Whole Program Behavior

Statement	Condition	EAX	EBX	AF	OF	CF	PF	SF
Whole Program Behavior	(2^32-7 == EAX) ...	7 +d EAX	25 +d EBX	11 <= ((...	is_in_int...	2^32-5 <...	is_even...	is_
	(equiv(is_in_inter...	10 +d EAX	20 +d EBX	[illegible]	is_in_int...	2^32-3 <...	is_even...	is_
1. add eax, 0x00000001								
2. add ebx, 20								
3. add eax, 0x00000006								
//if-then-else function								
4. **IF** (equiv(OF,SF)) && (!(ZF))								
5. **THEN** //then part function								
6. add eax, 0x00000003								
7. **ELSE** //else part function								
8. add ebx, 5								
9. (jmp $+0x0000000A)								
10. **ENDIF**								
11. label = exit								

Figure 4: PathQuest Drop-down menu showing statement sequence



Whole Program Behavior

Statement	Condition	EAX	EBX	AF	OF	CF	PF	
Whole Program Behavior	(2^32-7 == EAX) ...	7 +d EAX	25 +d EBX	11 <= ((...	is_in_int...	2^32-5 <...	is_even...	i
	(equiv(is_in_inter...	10 +d EAX	20 +d EBX	13 <= ((...	is_in_int...	2^32-3 <...	is_even...	i
1. add eax, 0x00000001	true	1 +d EAX		(EAX % ...	2^31-1 =...	2^32-1 =...	is_even...	i
after composition	true	1 +d EAX		(EAX % ...	2^31-1 =...	2^32-1 =...	is_even...	i
2. add ebx, 20	true		20 +d EBX	12 <= (E...	is_in_int...	2^32-20 ...	is_even...	i
after composition	true	1 +d EAX	20 +d EBX	12 <= (E...	is_in_int...	2^32-20 ...	is_even...	i
3. add eax, 0x00000006	true	6 +d EAX		10 <= (E...	is_in_int...	2^32-6 <...	is_even...	i
after composition	true	7 +d EAX	20 +d EBX	10 <= ((...	is_in_int...	2^32-6 <...	is_even...	i
//if-then-else function								
4. **IF** (equiv(OF,SF)) && (!(ZF))								
5. **THEN** //then part function								
6. add eax, 0x00000003								
7. **ELSE** //else part function								
8. add ebx, 5	true		5 +d EBX	11 <= (E...	is_in_int...	2^32-5 <...	is_even...	i
after composition	true	7 +d EAX	25 +d EBX	11 <= ((...	is_in_int...	2^32-5 <...	is_even...	i
9. (jmp $+0x0000000A)	true							
after composition	true	7 +d EAX	25 +d EBX	11 <= ((...	is_in_int...	2^32-5 <...	is_even...	i

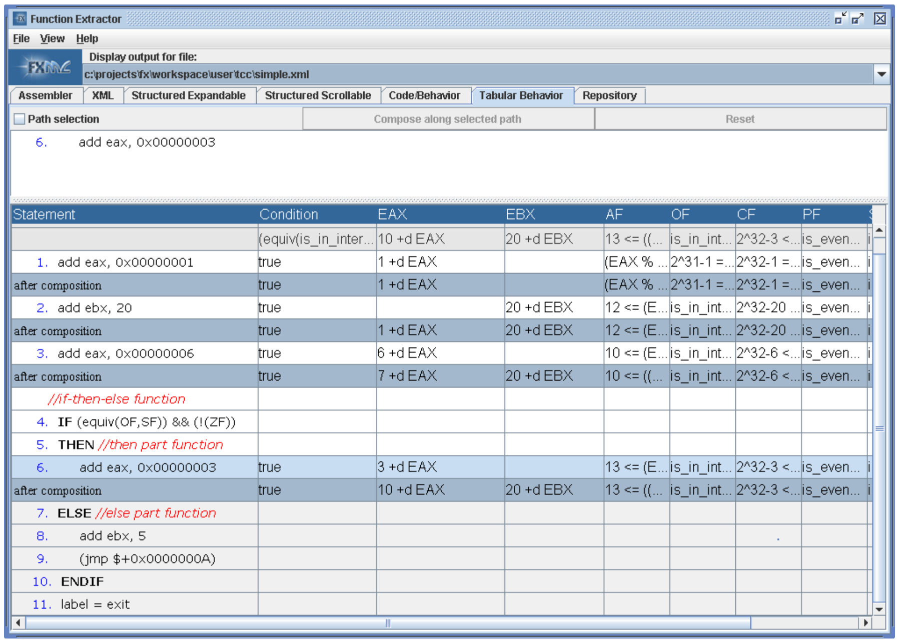
Figure 5: Highlighted Accumulated Behavior Sequence

composition of accumulated behavior that produces the behavior defined in the selected case. The accumulated behavior sequence is illustrated in Figure 5.

Sometimes, however, a user might like to start exploration from a point other than the behavior For example, in Figure 6, clicking on statement 6 shows the accumulated behavior to that point in the program, namely that EAX is incremented by 10 and EBX by 20. In this case, only one path reaches statement 6. Contrast this with the result of a "Come Here" on statement 8, in Figure 7, where EAX is increment by 7 and EBX by 25.

database. Using "Come Here" in FXplorer allows a user to explore the program from the statement level. Sfd

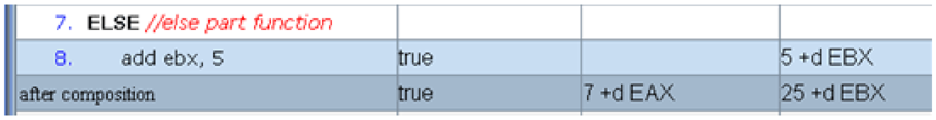
While "Come Here" allows a user to explore behavior to a given statement and "Path Quest" allows a user to explore starting from the behavior database, "Connect the Dots" gives a user control of exploration along a path of his choosing. To activate path selection, the user selects the "Path Selection" checkbox at the top of the tabular view. A new column appears



6. add eax, 0x00000003

Statement	Condition	EAX	EBX	AF	OF	CF	PF
	(equiv(is_in_inter...	10 +d EAX	20 +d EBX	13 <= ((...	is_in_int...	2^32-3 <...	is_even...
1. add eax, 0x00000001	true	1 +d EAX		(EAX % ...	2^31-1 =...	2^32-1 =...	is_even...
after composition	true	1 +d EAX		(EAX % ...	2^31-1 =...	2^32-1 =...	is_even...
2. add ebx, 20	true		20 +d EBX	12 <= (E...	is_in_int...	2^32-20 ...	is_even...
after composition	true	1 +d EAX	20 +d EBX	12 <= (E...	is_in_int...	2^32-20 ...	is_even...
3. add eax, 0x00000006	true	6 +d EAX		10 <= (E...	is_in_int...	2^32-6 <...	is_even...
after composition	true	7 +d EAX	20 +d EBX	10 <= ((...	is_in_int...	2^32-6 <...	is_even...
//if-then-else function							
4. IF (equiv(OF,SF)) && (!(ZF))							
5. THEN //then part function							
6. add eax, 0x00000003	true	3 +d EAX		13 <= (E...	is_in_int...	2^32-3 <...	is_even...
after composition	true	10 +d EAX	20 +d EBX	13 <= ((...	is_in_int...	2^32-3 <...	is_even...
7. ELSE //else part function							
8. add ebx, 5						.	
9. (jmp $+0x0000000A)							
10. ENDIF							
11. label = exit							

Figure 6: Come-here on statement 6



7. ELSE //else part function			
8. add ebx, 5	true		5 +d EBX
after composition	true	7 +d EAX	25 +d EBX

Figure 7: Come-here on statement 8

with radio buttons for every Then, Else, and Endif, as shown in Figure 8. By default the Endif options are selected. From there the user can select the desired pathway through the program. Now, selecting a statement in column 1 will show the result along the specified "Connect the Dots" path up to the selected statement. Clicking the Compose along selected path button at the top will compose the accumulating behavior using the connected path.

Figure 9 shows a more substantial program example using FXplorer. Imagine a program in an embedded avionics system that sets register EAX to the value of an angle for use in a tangent computation by the invoking program. It is important that the angle not be 90 degrees, since the tangent of 90 is infinite. This example is intentionally programmed in an obscure manner to simulate the difficulty of understanding a much larger program.

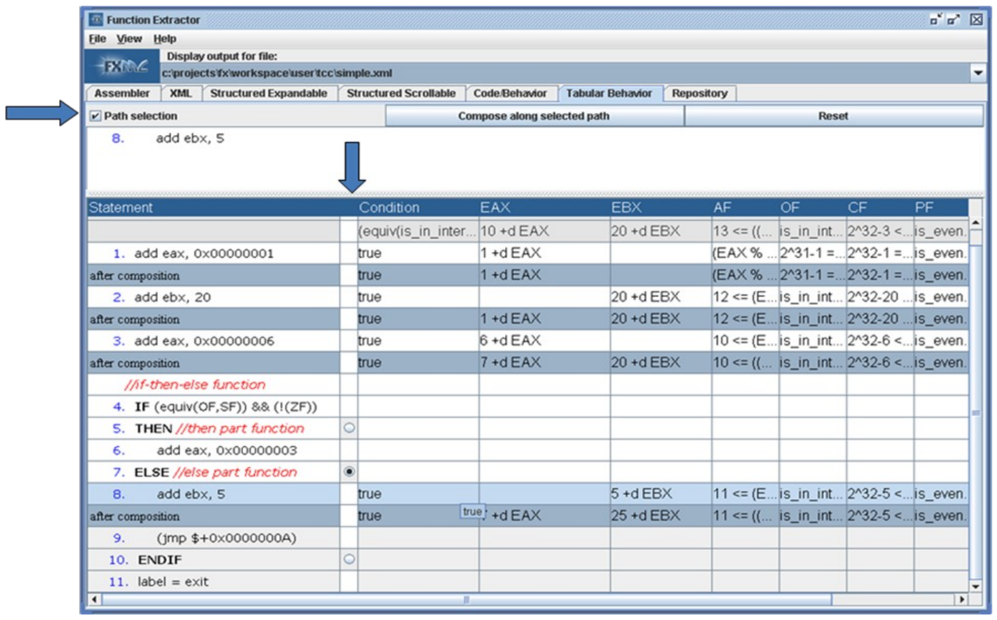
Note that the behavior shows three cases that set EAX to 90, 96, and 88 degrees, respectively. The 90-degree case is of immediate interest because of the problem it creates for the tangent computation. Nowhere in the initial code is it apparent that is EAX explicitly set to 90, or to any other value for that matter (Figure 10 contains source listing).

If the user wanted to explore the pathway that results in this behavior, he can right click on the row in the behavior database showing the behavior of interest, that is, the 90 degree case, and see the resulting sequence of statement numbers in the path. Selecting the sequence will then display the composition of accumulating behavior along that path. Figure 11 shows the resulting accumulated behavior. Note that we can now see exactly the point at which EAX becomes 90, that is, at statement 47.

In looking at these examples, it is important to note that no current software engineering tool can provide these capabilities, because no current tool has computed behavior available to it.

FXplorer provides powerful understanding and debugging information for software development, acquisition, testing, and verification. But Fxplorer is only the first of a virtually limitless suite of value-added applications that can be built around calculated software behavior.

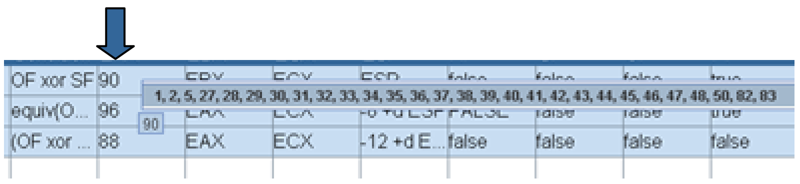
The next section briefly discusses the algorithm for calculating all the pathways through a given program.


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

```
8.      add ebx, 5
```

Statement		Condition	EAX	EBX	AF	OF	CF	PF
		(equiv(is_in_inter...	10 +d EAX	20 +d EBX	13 <= ((...	is_in_int...	2^32-3 <...	is_even.
1. add eax, 0x00000001		true	1 +d EAX		(EAX % ...	2^31-1 =...	2^32-1 =...	is_even.
after composition		true	1 +d EAX		(EAX % ...	2^31-1 =...	2^32-1 =...	is_even.
2. add ebx, 20		true		20 +d EBX	12 <= (E...	is_in_int...	2^32-20 ...	is_even.
after composition		true	1 +d EAX	20 +d EBX	12 <= (E...	is_in_int...	2^32-20 ...	is_even.
3. add eax, 0x00000006		true	6 +d EAX		10 <= (E...	is_in_int...	2^32-6 <...	is_even.
after composition		true	7 +d EAX	20 +d EBX	10 <= ((...	is_in_int...	2^32-6 <...	is_even.
//if-then-else function								
4. IF (equiv(OF,SF)) && (!(ZF))								
5. THEN //then part function	○							
6. add eax, 0x00000003								
7. ELSE //else part function	●							
8. add ebx, 5		true		5 +d EBX	11 <= (E...	is_in_int...	2^32-5 <...	is_even.
after composition		true	true +d EAX	25 +d EBX	11 <= ((...	is_in_int...	2^32-5 <...	is_even.
9. (jmp $+0x0000000A)								
10. ENDIF	○							
11. label = exit								

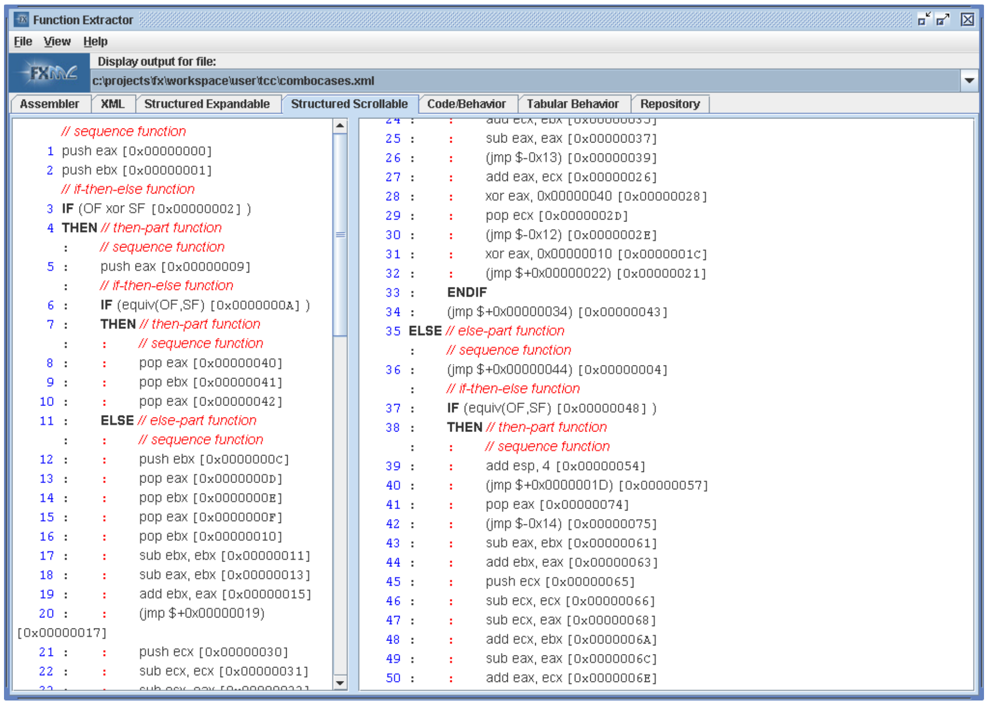
Figure 8: Connect-the-Dots User-controlled exploration



OF xor SF	90	EBX	ECX	ESP	false	false	false	true
equiv(O...	96	EAX	ECX	-6 +d ESP	FALSE	false	false	true
(OF xor ...	88	EAX	ECX	-12 +d E...	false	false	false	false

```
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
```

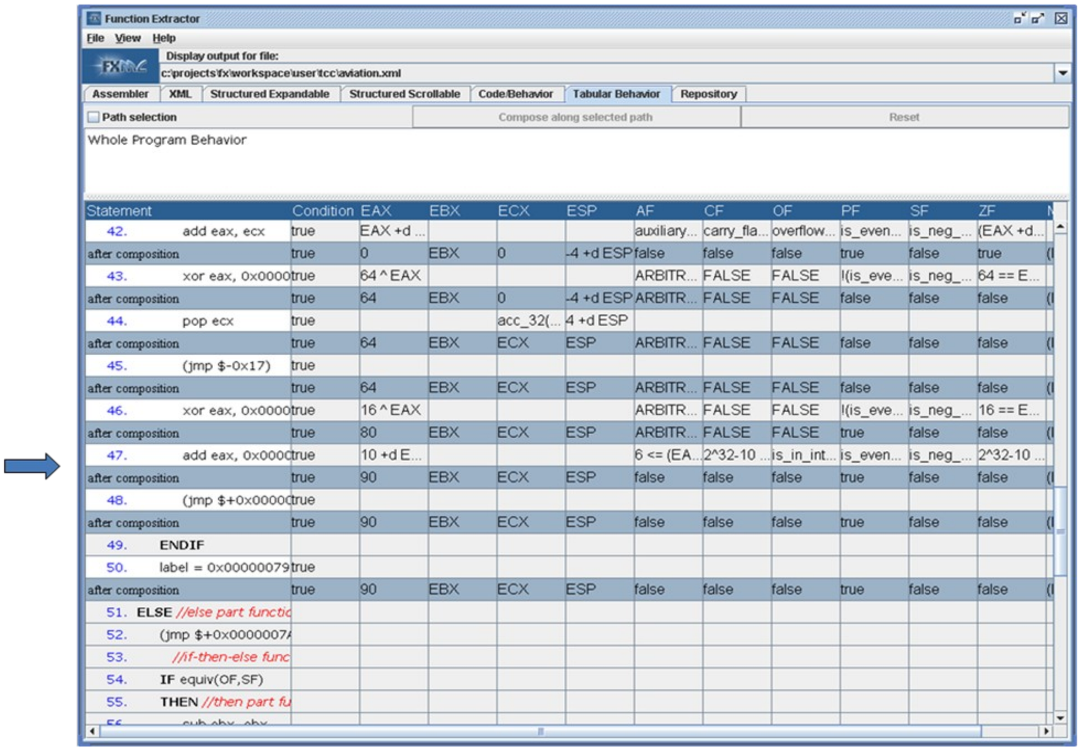
Figure 9: FXplorer on Aviation Program Extraction



```
    // sequence function
 1  push eax [0x00000000]
 2  push ebx [0x00000001]
    // if-then-else function
 3  IF (OF xor SF [0x00000002] )
 4  THEN // then-part function
    :    // sequence function
 5  :    push eax [0x00000009]
    :    // if-then-else function
 6  :    IF (equiv(OF,SF) [0x0000000A] )
 7  :    THEN // then-part function
    :    :    // sequence function
 8  :    :    pop eax [0x00000040]
 9  :    :    pop ebx [0x00000041]
10  :    :    pop eax [0x00000042]
11  :    ELSE // else-part function
    :    :    // sequence function
12  :    :    push ebx [0x0000000C]
13  :    :    pop eax [0x0000000D]
14  :    :    pop ebx [0x0000000E]
15  :    :    pop eax [0x0000000F]
16  :    :    pop ebx [0x00000010]
17  :    :    sub ebx, ebx [0x00000011]
18  :    :    sub eax, ebx [0x00000013]
19  :    :    add ebx, eax [0x00000015]
20  :    :    (jmp $+0x00000019)
[0x00000017]
21  :    :    push ecx [0x00000030]
22  :    :    sub ecx, ecx [0x00000031]
```

```
25 :    :    sub eax, eax [0x00000037]
26 :    :    (jmp $-0x13) [0x00000039]
27 :    :    add eax, ecx [0x00000026]
28 :    :    xor eax, 0x00000040 [0x00000028]
29 :    :    pop ecx [0x0000002D]
30 :    :    (jmp $-0x12) [0x0000002E]
31 :    :    xor eax, 0x00000010 [0x0000001C]
32 :    :    (jmp $+0x00000022) [0x00000021]
33 :    ENDIF
34 :    (jmp $+0x00000034) [0x00000043]
35 ELSE // else-part function
   :    // sequence function
36 :    (jmp $+0x00000044) [0x00000004]
   :    // if-then-else function
37 :    IF (equiv(OF,SF) [0x00000048] )
38 :    THEN // then-part function
   :    :    // sequence function
39 :    :    add esp, 4 [0x00000054]
40 :    :    (jmp $+0x0000001D) [0x00000057]
41 :    :    pop eax [0x00000074]
42 :    :    (jmp $-0x14) [0x00000075]
43 :    :    sub eax, ebx [0x00000061]
44 :    :    add ebx, eax [0x00000063]
45 :    :    push ecx [0x00000065]
46 :    :    sub ecx, ecx [0x00000066]
47 :    :    sub ecx, eax [0x00000068]
48 :    :    add ecx, ebx [0x0000006A]
49 :    :    sub eax, eax [0x0000006C]
50 :    :    add eax, ecx [0x0000006E]
```

Figure 10: Aviation Structured Source Code



Whole Program Behavior

Statement	Condition	EAX	EBX	ECX	ESP	AF	CF	OF	PF	SF	ZF
42. add eax, ecx	true	EAX +d ...				auxiliary...	carry_fla...	overflow...	is_even...	is_neg_...	(EAX +d...
after composition	true	0	EBX	0	-4 +d ESP	false	false	false	true	false	true
43. xor eax, 0x0000	true	64 ^ EAX				ARBITR...	FALSE	FALSE	!(is_eve...	is_neg_...	64 == E...
after composition	true	64	EBX	0	-4 +d ESP	ARBITR...	FALSE	FALSE	false	false	false
44. pop ecx	true			acc_32(...	4 +d ESP						
after composition	true	64	EBX	ECX	ESP	ARBITR...	FALSE	FALSE	false	false	false
45. (jmp $-0x17)	true										
after composition	true	64	EBX	ECX	ESP	ARBITR...	FALSE	FALSE	false	false	false
46. xor eax, 0x0000	true	16 ^ EAX				ARBITR...	FALSE	FALSE	!(is_eve...	is_neg_...	16 == E...
after composition	true	80	EBX	ECX	ESP	ARBITR...	FALSE	FALSE	true	false	false
47. add eax, 0x000C	true	10 +d E...				6 <= (EA...	2^32-10 ...	is_in_int...	is_even...	is_neg_...	2^32-10 ...
after composition	true	90	EBX	ECX	ESP	false	false	false	true	false	false
48. (jmp $+0x0000C	true										
after composition	true	90	EBX	ECX	ESP	false	false	false	true	false	false
49. ENDIF											
50. label = 0x00000079	true										
after composition	true	90	EBX	ECX	ESP	false	false	false	true	false	false
51. ELSE //else part functio											
52. (jmp $+0x0000007A											
53. //if-then-else func											
54. IF equiv(OF,SF)											
55. THEN //then part fu											

Figure 11: Accumulated Behavior

6. FXplorer All Paths Algorithm

We can consider the various blocks of code as black boxes for the purpose of finding all the possible paths through the code. The only case of interest for this algorithm involves handling IF statements. Thus, for a simple sequence of code:

```
GIVEN: (a b c)
(a b c)
```

When there is an IF statement we need to return the code sequence for the true case and the code sequence for the false case:

```
GIVEN: (a b c (if d e))
(a b c d)
(a b c e)
```

The IF statement can occur anywhere, including the first statement of the code sequence:

```
GIVEN: ((if a b) c)
(a c)
(b c)
```

It can even be the only statement in the code:

```
GIVEN: ((if a b))
(a)
(b)
```

Or, in general, the IF statements can be nested within other if statements:

```
GIVEN: (a b (if d e) (if (if i j) (if m n)) q)
(a b d i q)
(a b d j q)
(a b d m q)
(a b d n q)
(a b e i q)
(a b e j q)
(a b e m q)
(a b e n q)
```

So we need a recursive algorithm that walks the code blocks looking for IF statements. When one is found we construct two sequences, one sequence containing the TRUE case and one sequence containing the FALSE case.Within each case we need to recursively search for further IF statements.

The top level function "handle" looks for the outermost IF statement and recursively calls itself on the true branch, printing the result and then calls itself on the false branch, printing the result. Because it is recursive, this walks the whole program tree, printing each path once.

The other interesting function, "changelastif" is used to "flatten" the if statement into one of the two possible cases, depending on the boolean argument b.

```
public class AllPaths {
 public static void handle(String arg)
 { if (hasif(arg) == true)
   { handle(changelastif(arg,true));
     handle(changelastif(arg,false)); }
   } else { System.out.println(arg); }

 public static boolean hasif(String arg)
 { if (arg.indexOf("if") > 0) return(true);
   return(false); }

 public static String changelastif(String arg, boolean b)
 { String result = arg;
   int i = arg.indexOf("(if");
   if (i > 0)
   { int j = findmatchingparen(arg,i);
     if (b == true) { result =
      arg.substring(0,i)+truepart(arg,i)+arg.substring(j+1);
    } else { result =
      arg.substring(0,i)+falsepart(arg,i)+arg.substring(j+1);}}
   return(result); }

 public static int findmatchingparen(String arg, int i)
 { for(int j=i+1; j<arg.length() ; j++)
   { if (arg.charAt(j) == ')') return(j);
     if (arg.charAt(j) == '(') j=findmatchingparen(arg,j);}
   return(i);}

 public static String truepart(String arg, int i)
 { String result=arg.charAt(i+4)+"";
   if (arg.charAt(i+4) == '(')
    result=arg.substring(i+4,findmatchingparen(arg,i+4)+1);
   return(result);}

 public static String falsepart(String arg, int i)
 { String result=arg.charAt(i+6)+"";
   if (arg.charAt(i+4) == '(') {
    int k=findmatchingparen(arg,i+4);
    result=arg.substring(k+2,findmatchingparen(arg,k+2)+1);}
   return(result);}
}
```

7. Impact and Direction

FX gives software developers a practical means to determine the full functional behavior of programs. FXplorer adds three radically new abilities built on the FX knowledge of program behavior.

BehaviorCase or *Path Quest* answers the question "What parts of the program are responsible for this part of the final program behavior?"

BehaviorPath or *Connect the Dots* answers the question "What is the result of following this path?"

BehaviorHere or *Come Here* answers the question "What parts of the program are involved in reaching this point?"

Since FX covers ALL of the behavior of a program we need not worry that some special case was overlooked.

Because FXplorer can use FX to dynamically compute the composition of statement behavior we can now construct useful tools to verify that the program works and validate the program against specifications.

The FX technology opens the way to engineer correct programs from specifications. FXplorer leverages this power to make it possible for programmers to engineer programs meeting specifications.

8. References

[Collins et al. 2005] 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.

[Hevner et al. 2005] 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.

[Jorgensen 2002] P. Jorgensen, *Software Testing: A Craftsman's Approach*, 2nd Edition, CRC Press, Inc., Boca Raton, 2002.

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

[Linger and Pleszkoch 2004] 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.

[Linger et al. 2007] 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, January 2007.

[Mili et al. 2007] Mili, A., Daly, T., Pleszkoch, M., and Prowell, S., "Next-Generation Software Engineering: A Semantic Recognizer Infrastructure for Computing Loop Behavior," *Proceedings of Hawaii International Conference on System Sciences (HICSS41)*, Hawaii, IEEE Computer Society Press, Los Alamitos, CA, 2007.

[Pleszkoch et al. 1990] Pleszkoch, M., Hausler, P., Hevner, A., and Linger, R. "Function-Theoretic Principles of Program Understanding," *Proceedings of the 23rd Annual Hawaii International Conference on System Science (HICSS23),* Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 1990.

[Poore et al. 1993] Poore, J., Mills, H., and Mutchler, D. "Planning and Certifying Software System Reliability," *IEEE Software*, Vol. 10, 1993.

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

[Sayre and Poore 2007] Sayre, K. and Poore, J., "Automated Testing of Generic Computational Science Libraries," *Proceedings of the 40th Annual Hawaii International Conference on System Science (HICSS40),* Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 2007.

[Walton et al. 2006] 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.

[Whittaker 2000] J. Whittaker, "What Is Software Testing? And Why Is It So Hard?" *IEEE Software*, Vol. 17, No. 1, January/February 2000.