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

Luanne Burns
Timothy Daly
Software Engineering Institute
Carnegie Mellon University
lburns@cert.org
daly@cert.org

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 functional behavior of programs and how the knowledge of program behavior can be used in user-directed program exploration for 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 is an example of a value-added software understanding application that capitalizes on the availability of 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 and can be described as the "as-built" specification of the code. Routine availability of computed software behavior permits the development of many value-added applications with capabilities beyond what is possible today. For example, FXplorer provides a unique and different view into program behavior and how that behavior accumulates as a program executes. This view provides new approaches and strategies in software understanding and verification. In section 2, we discuss FX in the context of cyber security. Section 3 discusses the concepts of Function Extraction and the function-theoretic view of software as the mathematical foundation for the computation of behavior. Section 4 describes the FX system that implements such a system. Section 5 illustrates the of concepts underlying FXplorer as a value-added software application made possible through the use of function extraction technology and section 6 describes the FXplorer interface. Section 7 gives a brief discussion of FXplorer's underlying algorithm for computing all pathways through a program using its computed behavior. Finally, section 8 discusses FXplorer impact and future direction.

2. Background and Cyber Security

Gallagher and Lyle [2,7] use the idea of slicing a program along a single variable in order to isolate the effect of the variable on the rest of the program. Interactions might arise due to side-effects that are not directly related to the variable, such as changing the value of one of the processor flags or side-effecting aliased memory locations. Since FX computes the "ground truth" of the processor, it should be possible to track the result of these side-effects. We have not considered tracing a single user visible variable but this might be a very valuable addition to FX in a software maintenance role.

Walton et al. [7,10] ask, "What can be computed with respect to security attributes?" The 9 attributes identified were (1) a trusted mechanism, (2) trusted data transmission, (3) authentication, (4) authorization, (5) non-repudiation, (6) privacy, (7) confidentiality, (8) integrity, and (9) availability. These attributes would be specified in a behavior catalog giving, for example, the required authentication behavior of a login program. The program would be restricted to use

only trusted data sources and be required to acquire the needed privilege level only during certain operations.

Using FX with a behavior catalog which specifies the program behavior, the FXplorer program can be used to calculate the current program behavior. This can then be compared with the required security behavior which has been specified in terms of data and transformations on data.

3. 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 [1,3,4,5,8,9]. 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 inexpensive 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 [4]. 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 a program [5].

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 [8].

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 a 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 [3].

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 computa-

tion as a means to understanding and verifying programs.

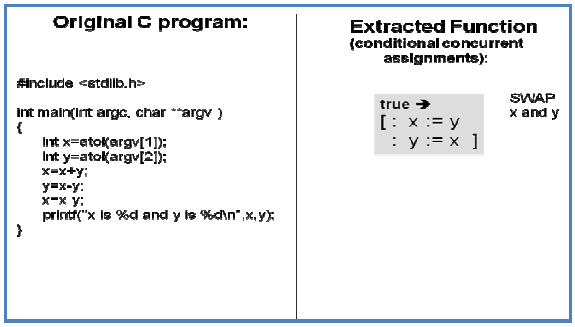

Figure 1: Swap program and extraction

4. 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.

```
 1 //begin sequence
 2 top:
 3 :      // sequence function
 4 :      push eax
 5 :      push ebx
 6 :      add esp, 0x00000004
 7 :      (jmp 0x0000001E)
 8 :      pop eax
 9 :      (jmp $-0x14)
10 :      sub eax, ebx
11 :      add ebx, eax
12 :      push ecx
13 :      sub ecx, ecx
14 :      sub ecx, eax
15 :      add ecx, ebx
16 :      sub eax, eax
17 :      add eax, ecx
18 :      clc
19 :      pop ecx
20 :      (jmp $-0x12)
21 :      (ret)
22 :      label = exit
```

Figure 2: Assembler source

Figure 3 shows the FX Code/Behavior display. The left side of the split pane shows the original assembly language after transformation by the system into structured form. It is revealed to be a simple sequence structure. 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 are conditional concurrent assignments; the left hand sides represent final values at program exit while the right hand sides are initial values at program entry; all equations are assigned concurrently, not sequentially.

Note here that EAX register has been assigned the initial value of the EBX register and EBX has been assigned the initial value of the EAX register. This code thus performs a swap of the two registers. This program involves only a simple sequence control structure. The FX system computes the functional behavior of every control structure in a program, thereby populating its behavior database with comprehensive information about its behavior from low level structures up to the entire program.

5. Behavior Exploration with FX Technology

The Code/Behavior display 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 verifying 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

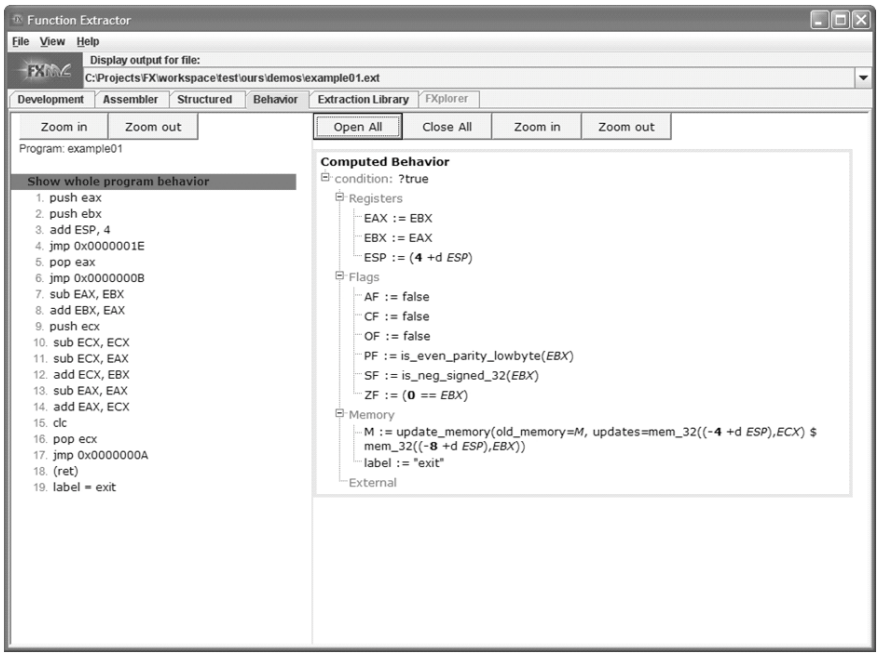

Figure 3: FX Code/Behavior View

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 accumulating and 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 accumulating 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 does this result come from?" (BehaviorCase), "What happens if this path is executed?" (BehaviorPath), and "How does this program 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.

6. Exploring the User Interface

Glance ahead to Figure 5 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. FX also displays external items such as file system values but those will not be discussed in this context. 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 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 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 computed by 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 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 database. Using "Come Here" in FXplorer allows a user to explore the program from the statement level.

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.

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 seletion, the user selects the "Path Selection" checkbox at the top of the tabular view. A new column appears 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

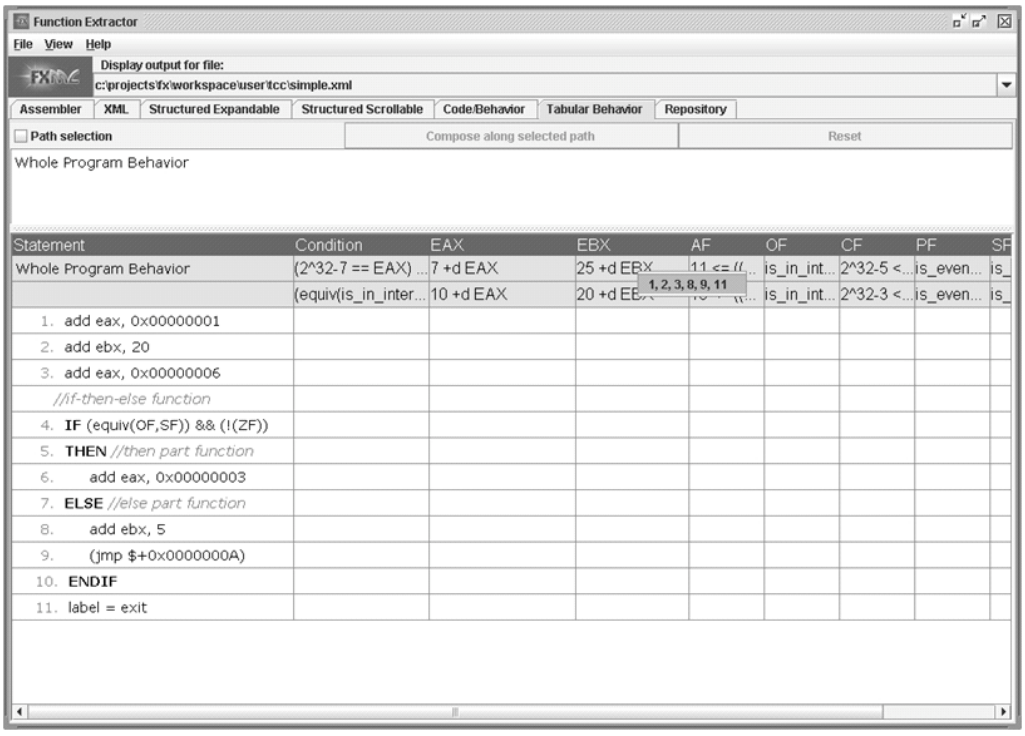

Figure 4: PathQuest Drop-down menu showing statement sequence

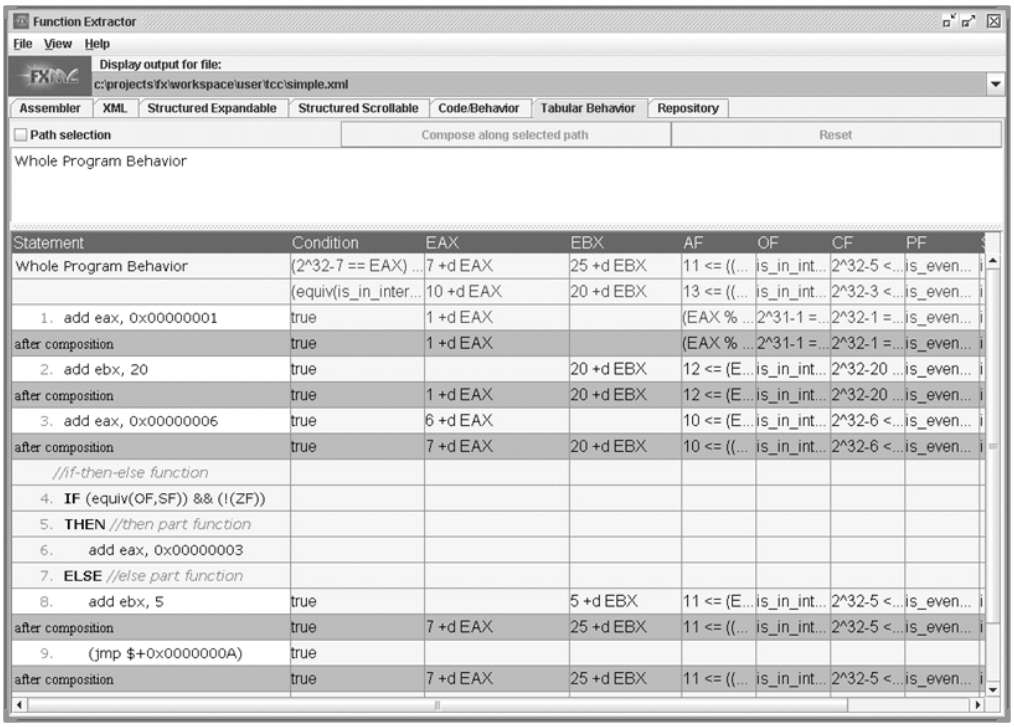

Figure 5: Highlighted Accumulated Behavior Sequence

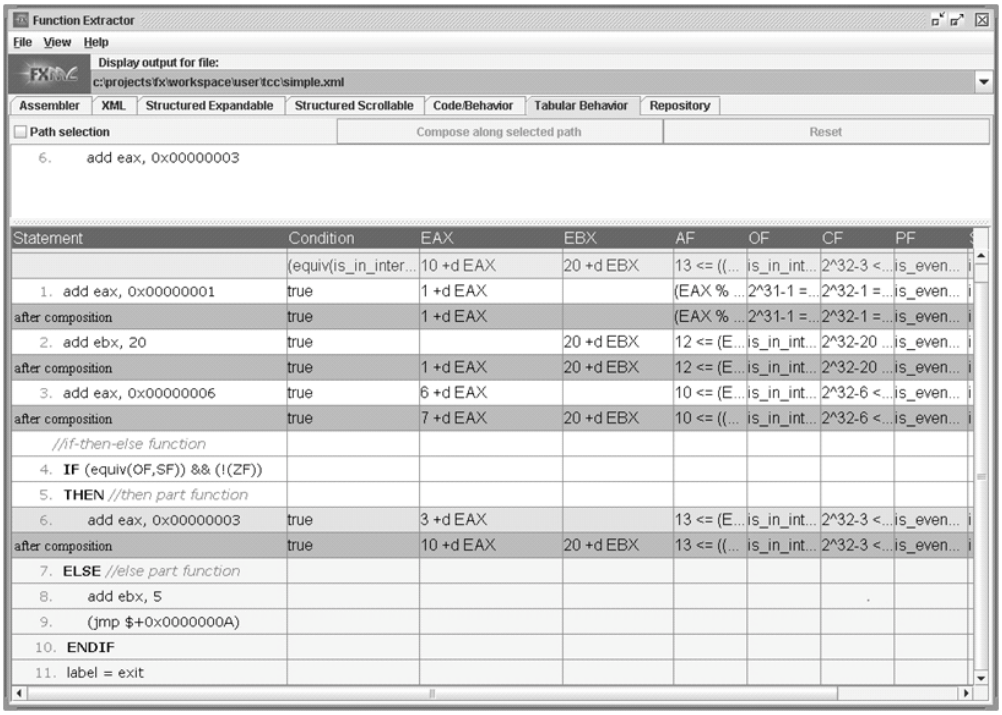

Figure 6: Come-here on statement 6

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

pathway through the program. Now, selecting 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 computed 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 an extensive 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.

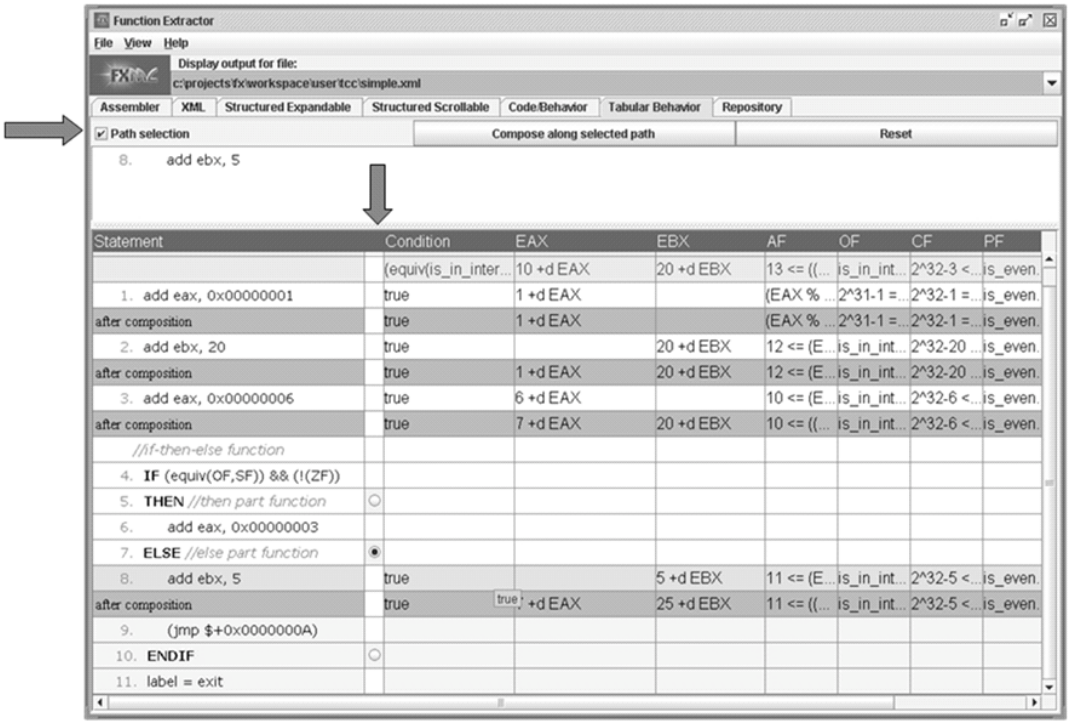

Figure 8: Connect-the-Dots User-controlled exploration

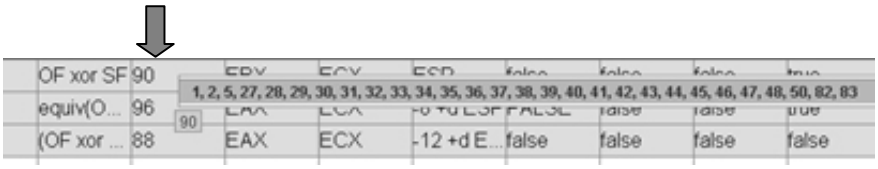

Figure 9: FXplorer on Aviation Program Extraction

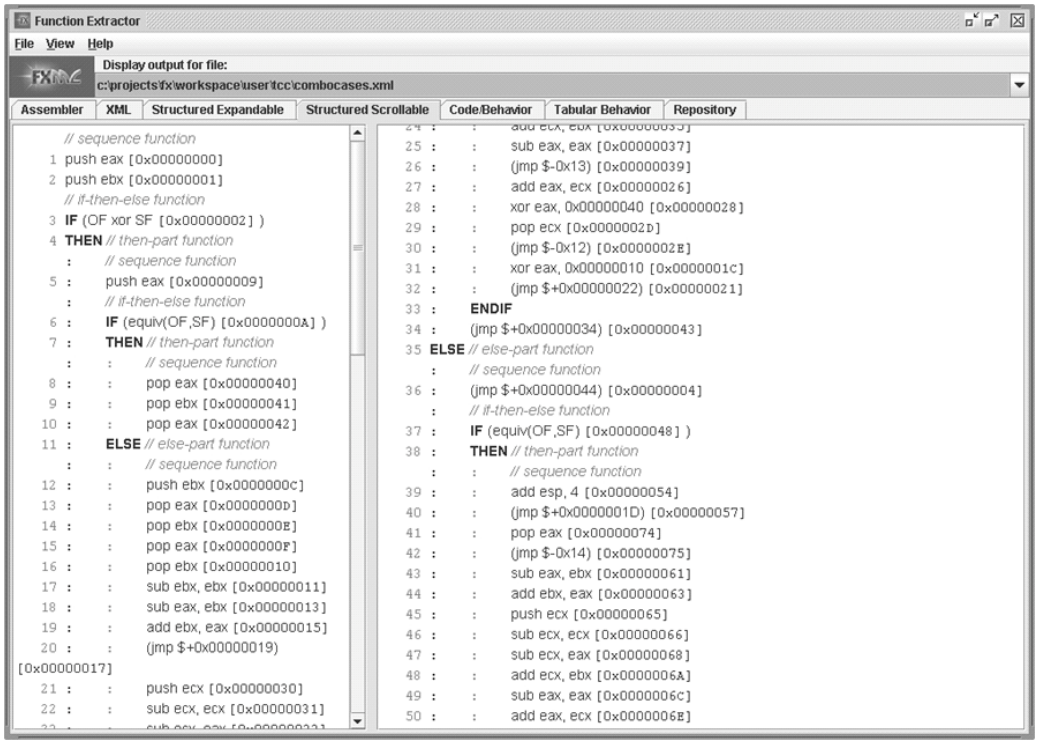

Figure 10: Aviation Structured Source Code

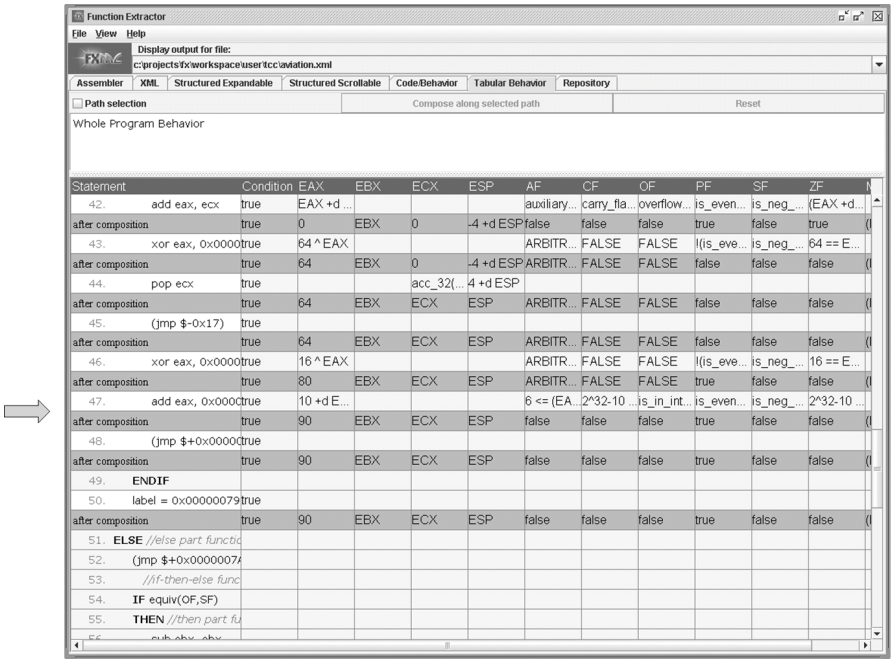

Figure 11: Accumulated Behavior

7. 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.

8. 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 has been overlooked. FXplorer is an example of a new generation of software engineering automation that can capitalize on computed behavior to amplify human capabilities for program understanding and verification against specifications. As an emerging discipline, FX technology holds promise for engineering correct programs, and FXplorer provides a powerful illustration of how computed behavior can be leveraged to create new engineering capabilities.

9. References

[1]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.

[2]Gallagher, J.R. Lyle, *Using Program Slicing in Software Maintenance,* IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 751-761, Aug. 1991

[3]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.

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

[5]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.

[6]Linger, R., Pleszkoch, M., Burns, L., Hevner, A., and Walton, G., "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.

[7]Lyle, J.R., Gallagher, K.B., "A program decomposition scheme with applications to software modification and testing," *Proceedings of the 22th Annual Hawaii International Conference on System Sciences (HICSS22)*, Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 1989.

[8]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.

[9]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.

[10]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.