Introducing Function Extraction into Software Testing

Mark G. Pleszkoch
Carnegie Mellon University

Richard C. Linger
Carnegie Mellon University

Alan R. Hevner
University of South Florida

Acknowledgments

We wish to acknowledge the efforts of the Function Extraction research team within CERT. Alan Hevner has been partially supported by the U.S. National Science Foundation (NSF) during an assignment at NSF.

Abstract

Software testing can benefit from technologies that enable evolution toward increased 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, effective scientific principles and engineering technology for revealing software behavior should have a positive impact on software testing. This paper describes the emerging technology of function extraction (FX) for computing the behavior of programs to the maximum extent possible with mathematical precision. We explore how the use of FX technologies can transform methods for functional verification of software. An example illustrates the value of full behavior knowledge for complete and confident assessment of software function and fitness for use. We conclude by describing a transition strategy for introducing FX technology into the development and operation of software systems.

ACM Categories:

Keywords: Function Extraction, Program Behavior, Software Systems, Software Testing

Transforming Software Testing

The activities of testing and verifying software-intensive systems are difficult and exhausting work, both mentally and physically (Whittaker, 2000). It is well recognized that the cost and schedule for testing activities can account for a significant percentage of a software system’s project budget. The state-of-the-practice for software testing consists of techniques for black-box and white box testing, often supported by automated tools (Jorgensen, 2002). Overall, software testing, as practiced today, remains a costly craft with results highly dependent upon the skills of the testers.

Research activities in recent years have focused on finding scientific foundations for the theory and practice of software testing, for example, in the use of statistical, usage-based testing with its predictive power to certify software fitness for use in operational environments (e.g., Poore et al., 1993; Sayre and Poore, 2007). However, such formal testing methods have not significantly transformed practice in the field.

Software testing can benefit from technologies that help move toward a true engineering discipline. We believe that function extraction (FX) can have substantial impact on the theory and practice of software testing. Current technologies do not provide software developers with practical means to determine the full functional behavior of programs in all circumstances of use. Testing, together with inspections and reviews, is a principal means for behavior discovery today, but even extensive testing can provide only partial knowledge of software behavior. Thus, research in technologies that can reveal full behavior has potential for transformative impact on testing processes and results. The goal of this paper is to explore the impact of FX technology on software testing.

Function Extraction Concepts

The objective of function extraction is to compute the behavior of software to the maximum extent possible with mathematical precision. CERT STAR*Lab of the Software Engineering Institute at Carnegie Mellon University is conducting research and development in this emerging technology. FX presents an opportunity to reduce dependencies on slow and costly testing processes to assess software functionality by moving too fast and inexpensive computation of functionality at machine speeds. Because a principal objective of testing is to validate functionality, automated computation of functional behavior can be expected to streamline testing processes and permit increased testing focus on system-level issues of component interaction and dynamic system properties, such as security, reliability, and performance.

The goals of behavior calculation are to compose and record the semantic information in programs as a means to augment human capabilities for analysis, design, and verification. We discuss function extraction below in the context of sequential logic, in full knowledge that concurrent and recursive logic must be addressed as well. Computing the behavior of programs is a difficult problem, and our intent in this paper is to begin a discussion on how to move software testing into future technologies; to say 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 (if-then-else), and iteration (while-do), 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 behaviors of leaf node control structures are 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 behaviors have 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, are under development. This work permits the effect of theoretical limitations on loop behavior computation to be made arbitrarily small.

The general form of the expressions produced by function extraction is a set of conditional concurrent assignments (CCA) organized into 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 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.

Behavior databases, thus, are the central repository for the actual behaviors contained in a software system. They 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, as seen in Figure 1, 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). Later in the paper, we will focus on the specific impacts of the FX technology on software testing activities.

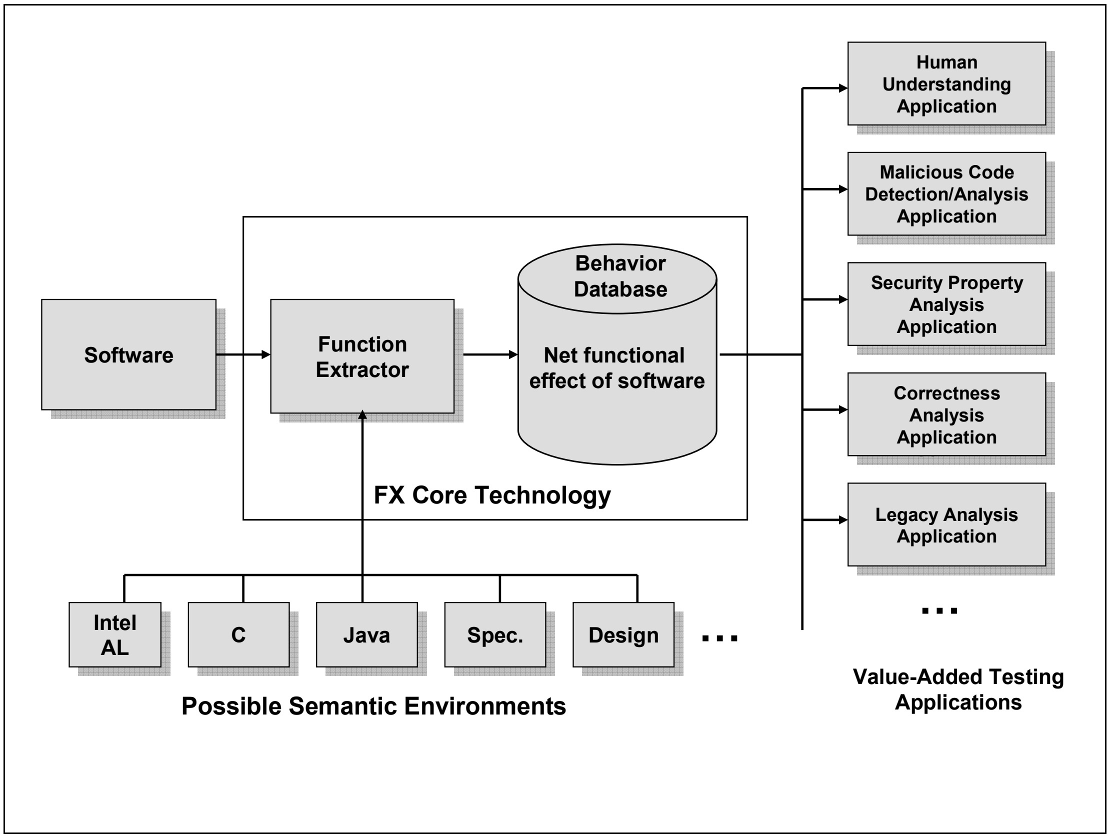

Human Understanding Application
Malicious Code Detection/Analysis Application
Security Property Analysis Application
Correctness Analysis Application
Legacy Analysis Application
...
Behavior Database
Software
Function Extractor
Net functional effect of software
FX Core Technology
Intel AL
C
Java
Spec.
Design
...
Possible Semantic Environments
Value-Added Testing Applications

Figure 1. Function Extraction Technology

The first application of FX technology at CERT is a function extraction system that computes behavior for programs written in, or compiled into, Intel assembly language, to support analysts in malicious code detection and understanding of malware behavior. Sample outputs from the evolving FX system are employed in the next section to illustrate the role of behavior computation as a means to satisfy testing objectives.

A Behavior Computation Example

In miniature illustration of the role of FX in a testing environment, consider the problem of developing a program that returns 1 if three given integers form the sides of a proper triangle, otherwise the program returns 0. The C language program depicted below appears to satisfy this requirement by first checking that each side is a positive number, and then checking each permutation of the triangle inequality:

```
int test_triangle (int a, int b, int c)
{
  int answer = 1;
  if (a <= 0) {
   answer = 0;
  }
  if (b <= 0) {
   answer = 0;
  }
  if (c <= 0) {
   answer = 0;
  }
  if (a + b <= c) {
   answer = 0;
  }
  if (a + c <= b) {
   answer = 0;
  }
  if (b + c <= a) {
   answer = 0;
  }
  return answer;
}
```

Since the FX system computes behavior for compiled machine code, the first step is to disassemble the object code to produce the listing shown below. The IDA Pro disassembler is used for this purpose:

```
; test_triangle(int, int, int)
                          public __Z13test_triangleiii
```

```
__Z13test_triangleiii proc near

var_4          = dword ptr -4
arg_0          = dword ptr 8
arg_4          = dword ptr 0Ch
arg_8          = dword ptr 10h

               push    ebp
               mov     ebp, esp
               sub     esp, 4
               mov     [ebp+var_4], 1
               cmp     [ebp+arg_0], 0
               jg      short loc_1A
               mov     [ebp+var_4], 0

loc_1A:        cmp     [ebp+arg_4], 0
               jg      short loc_27
               mov     [ebp+var_4], 0

loc_27:        cmp     [ebp+arg_8], 0
               jg      short loc_34
               mov     [ebp+var_4], 0

loc_34:        mov     eax, [ebp+arg_4]
               add     eax, [ebp+arg_0]
               cmp     eax, [ebp+arg_8]
               jg      short loc_46
               mov     [ebp+var_4], 0

loc_46:        mov     eax, [ebp+arg_8]
               add     eax, [ebp+arg_0]
               cmp     eax, [ebp+arg_4]
               jg      short loc_58
               mov     [ebp+var_4], 0

loc_58:        mov     eax, [ebp+arg_8]
               add     eax, [ebp+arg_4]
               cmp     eax, [ebp+arg_0]
               jg      short loc_6A
               mov     [ebp+var_4], 0

loc_6A:        mov     eax, [ebp+var_4]
               leave
               retn

__Z13test_triangleiii endp
```

Next, the FX system is executed from an IDA Pro plug-in, with a screen shot of the resulting output depicted below in Figure 2. On the left side of the screen, the spaghetti-logic of the disassembled C program has been transformed into structured form. On the right side, the computed behavior is presented in terms of the net effect of the program on registers, flags, and memory.

There are two cases (conditions) in the computed behavior, each defined as a conditional concurrent assignment (CCA). The registers section of the first case shows that the EAX register is set to 0, resulting in returning 0 to the calling program. Additionally, the EBP register is used but is finally set back to its original value, and the ESP register is incremented by 4 (by the RET instruction which pops the return address off the stack). The memory section shows the final value of the local variable “answer” in the original C program, as well as other data that were saved on the stack. The second case is similar, except that the EAX register is set to 1, thereby returning 1 to the calling program. Thus, provided that the conditions themselves are correct, there is no need to execute any test cases on this program to determine its functional behavior.

The condition for the second case is as follows:

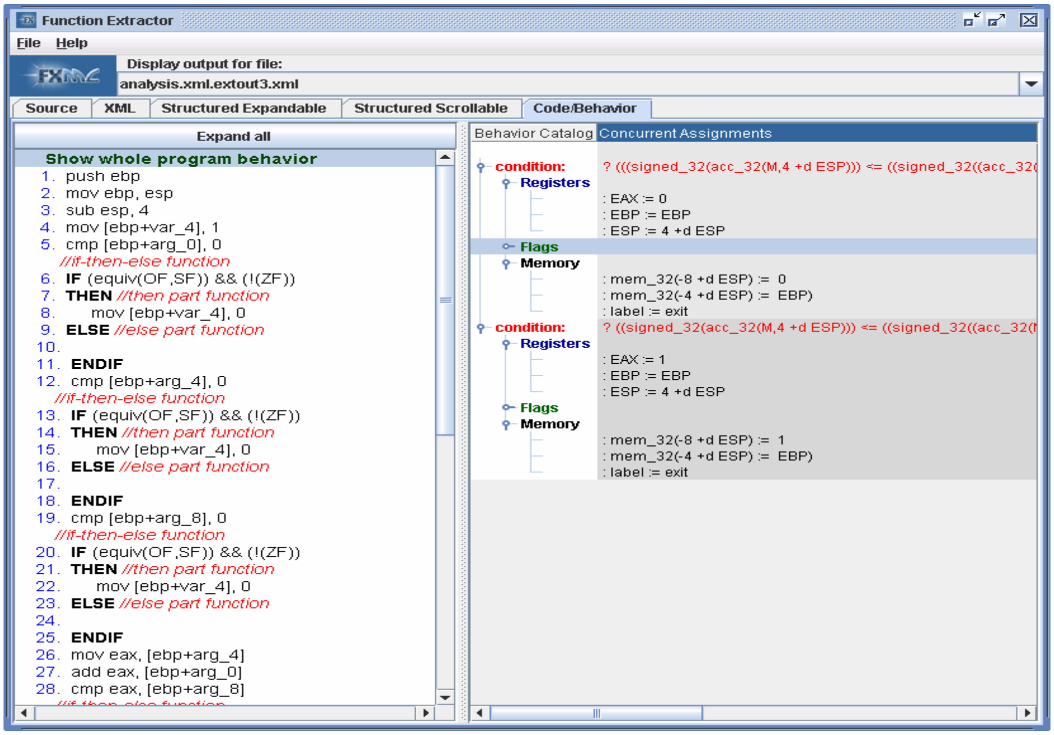
```
    (parm_a <= (signed_32(parm_b +d parm_c) + -1))
 && (parm_b <= (signed_32(parm_a +d parm_c) + -1))
 && (parm_c <= (signed_32(parm_a +d parm_b) + -1))
 && (1 <= parm_a)
 && (1 <= parm_b)
 && (1 <= parm_c)
```

where

```
  parm_a :== signed_32(acc_32(M,4 +d ESP))
  parm_b :== signed_32(acc_32(M,8 +d ESP))
  parm_c :== signed_32(acc_32(M,12 +d ESP))
```

For readability, the output of the FX system has been manually formatted to use symbolic names for the function parameters instead of the low-level memory access expressions shown in the definitions of those parameters. As the system evolves, this formatting will be incorporated.

The condition for the program to return 1 is the logical “and” of six checks, corresponding to the six if-then-elses in the program. The checks for positive parameters are easily seen to be correct. However, an examination of the other checks shows that the overflow case has not been considered. (Note that “+d” in the condition represents addition modulo 2^32, and the “signed_32” operation interprets the result as a two’s complement signed number.) As a result, the program gives the wrong answer whenever the addition of two of the sides causes an arithmetic overflow. For example, as presently implemented, the program would incorrectly indicate that 2^30, 2^30, and 10 do not form the sides of a triangle. Note that this error would likely not have been found by any of the common test coverage strategies, including statement coverage, branch coverage, and path coverage.



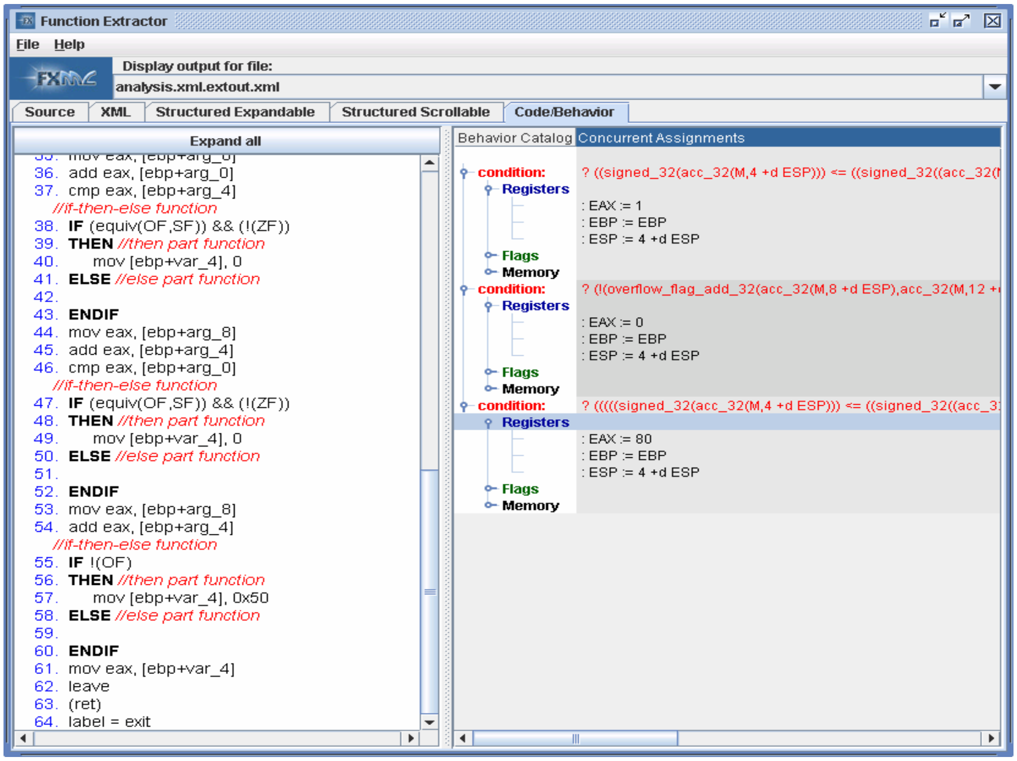
Display output for file: analysis.xml.extout3.xml

Show whole program behavior

```
push ebp
mov ebp, esp
sub esp, 4
mov [ebp+var_4], 1
cmp [ebp+arg_0], 0
 //if-then-else function
IF (equiv(OF,SF)) && (!(ZF))
THEN //then part function
    mov [ebp+var_4], 0
ELSE //else part function

ENDIF
cmp [ebp+arg_4], 0
 //if-then-else function
IF (equiv(OF,SF)) && (!(ZF))
THEN //then part function
    mov [ebp+var_4], 0
ELSE //else part function

ENDIF
cmp [ebp+arg_8], 0
 //if-then-else function
IF (equiv(OF,SF)) && (!(ZF))
THEN //then part function
    mov [ebp+var_4], 0
ELSE //else part function

ENDIF
mov eax, [ebp+arg_4]
add eax, [ebp+arg_0]
cmp eax, [ebp+arg_8]
 //if-then-else function
```

```
condition:   ? (((signed_32(acc_32(M,4 +d ESP))) <= ((signed_32((acc_32(
  Registers
               : EAX := 0
               : EBP := EBP
               : ESP := 4 +d ESP
  Flags
  Memory
               : mem_32(-8 +d ESP) :=  0
               : mem_32(-4 +d ESP) :=  EBP)
               : label := exit
condition:   ? ((signed_32(acc_32(M,4 +d ESP))) <= ((signed_32((acc_32(M
  Registers
               : EAX := 1
               : EBP := EBP
               : ESP := 4 +d ESP
  Flags
  Memory
               : mem_32(-8 +d ESP) :=  1
               : mem_32(-4 +d ESP) :=  EBP)
               : label := exit
```

Figure 2. Triangle Program in FX System



Display output for file: analysis.xml.extout.xml

```
mov eax, [ebp+arg_0]
add eax, [ebp+arg_0]
cmp eax, [ebp+arg_4]
 //if-then-else function
IF (equiv(OF,SF)) && (!(ZF))
THEN //then part function
    mov [ebp+var_4], 0
ELSE //else part function

ENDIF
mov eax, [ebp+arg_8]
add eax, [ebp+arg_4]
cmp eax, [ebp+arg_0]
 //if-then-else function
IF (equiv(OF,SF)) && (!(ZF))
THEN //then part function
    mov [ebp+var_4], 0
ELSE //else part function

ENDIF
mov eax, [ebp+arg_8]
add eax, [ebp+arg_4]
 //if-then-else function
IF !(OF)
THEN //then part function
    mov [ebp+var_4], 0x50
ELSE //else part function

ENDIF
mov eax, [ebp+var_4]
leave
(ret)
label = exit
```

```
condition:   ? ((signed_32(acc_32(M,4 +d ESP))) <= ((signed_32((acc_32(M
  Registers
               : EAX := 1
               : EBP := EBP
               : ESP := 4 +d ESP
  Flags
  Memory
condition:   ? (!(overflow_flag_add_32(acc_32(M,8 +d ESP),acc_32(M,12 +
  Registers
               : EAX := 0
               : EBP := EBP
               : ESP := 4 +d ESP
  Flags
  Memory
condition:   ? (((((signed_32(acc_32(M,4 +d ESP))) <= ((signed_32((acc_3
  Registers
               : EAX := 80
               : EBP := EBP
               : ESP := 4 +d ESP
  Flags
  Memory
```

Figure 3. Triangle Program in FX with Malicious Code Condition

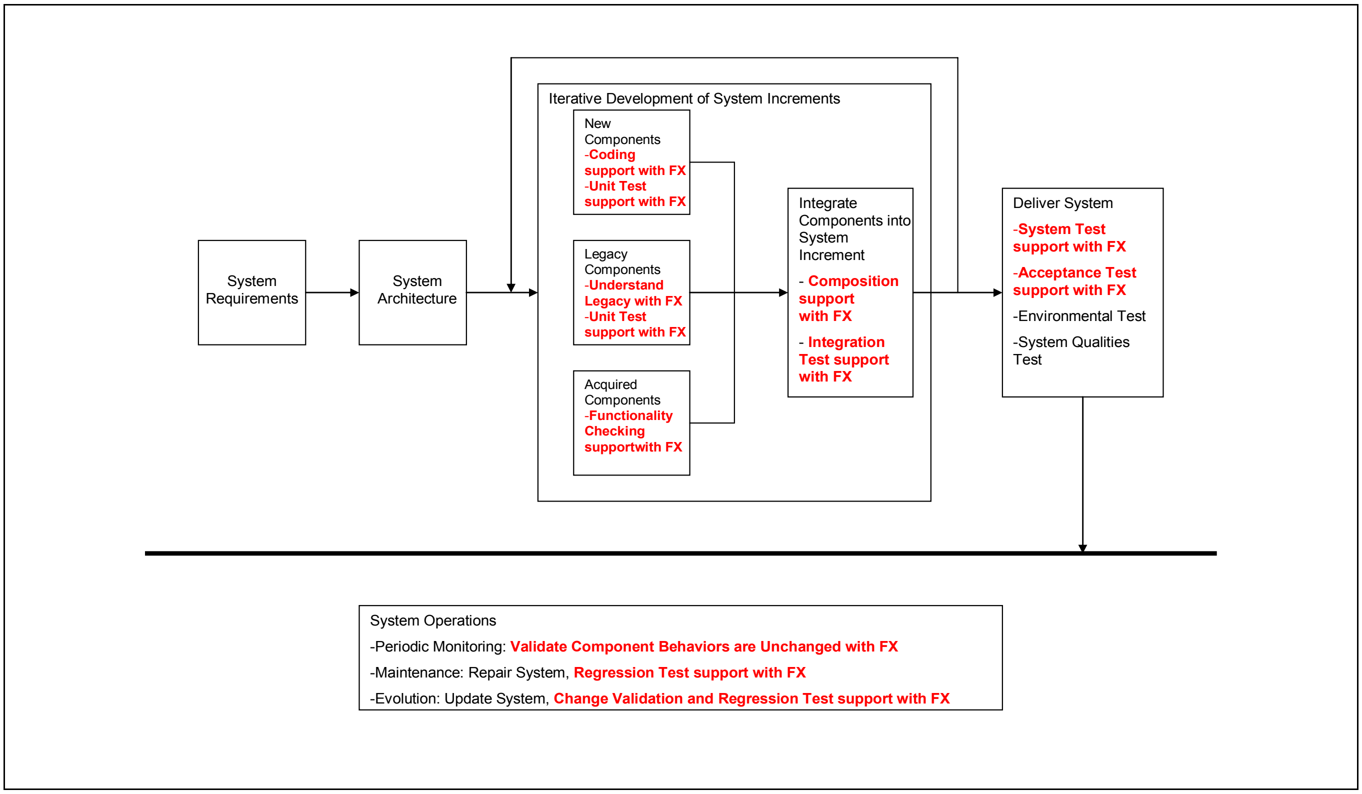

System Requirements
System Architecture
Iterative Development of System Increments
New Components
-Coding support with FX
-Unit Test support with FX
Legacy Components
-Understand Legacy with FX
-Unit Test support with FX
Acquired Components
-Functionality Checking supportwith FX
Integrate Components into System Increment
- Composition support with FX
- Integration Test support with FX
Deliver System
-System Test support with FX
-Acceptance Test support with FX
-Environmental Test
-System Qualities Test
System Operations
-Periodic Monitoring: Validate Component Behaviors are Unchanged with FX
-Maintenance: Repair System, Regression Test support with FX
-Evolution: Update System, Change Validation and Regression Test support with FX

Figure 4. Transition of FX Testing into System Development and Operations

Consider next the problem of determining whether malicious content has been added to this program, in particular, whether the EAX register is being maliciously employed under conditions defined by an intruder.

The FX screenshot in Figure 3 shows computed behavior for such an instance. A third case of behavior appears, where the malicious intent is invoked only for inputs that cause the addition of sides b and c to overflow. The assignment of a value of 80 to EAX could cause a buffer overflow or some other problem in the calling program, which is expecting a return value of at most 1. This example illustrates the discovery and exploitation of a program vulnerability by an intruder. This malicious execution would be just as difficult to find through testing as the overflow error in the original program. However, the FX system immediately computes a third case of behavior with a return value of 80, revealing the malicious functionality for all to see.

Software Testing with FX Technology: A Transition Strategy

Based on our research and development activities with FX technologies, we have identified a wide range of FX impacts across many software development activities (Hevner et al., 2005). For this paper, we focus on how FX can transform testing practices during software-intensive system development and operation. In particular, we address the question of how to effectively transition FX ideas and tools into current testing activities. The beneficial impacts of the new FX technologies are discussed.

Figure 4 shows a typical development process that builds a software system via an iterative cycle of incremental development. The opportunities for application of FX technologies are highlighted throughout the process. The process begins with activities of identifying system requirements and specifying a software architecture for building the system. Future research directions of FX are aimed at understanding the behaviors defined by architectural models and languages. Additional research on the development of semantically precise representations of architectures and designs is needed to enable FX ideas to be effective in these upstream development activities.

Component Development

Each system increment is built from individual software components that are integrated into the architecture for that increment. Components are provisioned in one of three ways – original development, modified legacy development, or acquired as pre-developed components. Each of these approaches provides opportunities for the introduction of FX technology.

Original Component Development. When a decision is made to develop a required software component from scratch, FX automation can play an important role during the evolving implementation. As each set of required functions is developed, a software developer can work interactively with an FX system to determine if the evolving implementation indeed provides the set of functions intended. As new code is introduced into an evolving component, the FX system can report on the corresponding additional behaviors, as well as any changes to prior behaviors. Errors of commission or omission can thus be identified during the implementation process, and extraneous behavior isolated and removed.

Significant time and effort are often allocated during software development to verify the correctness and quality of software designs and implementations. Reviews, inspections, and unit testing are resource-intensive activities used to evaluate components against their specifications. At its core, FX technology is closely related to correctness verification. Programmers can add intended functions (expressed in a standard language form as comments) to the control structures of implementations to permit FX automation to compare the extracted behavior of each control structure to the corresponding intended function to determine whether or not it is correct. Alternately, if programmers do not wish to add intended functions to their programs, calculated behavior can be easily inspected to verify that a program indeed does what is desired, and that no unforeseen cases of behavior are present.

Reuse of Legacy Components. Significant time and effort are saved by effective reuse of legacy components. First, however, it is important to fully understand the behaviors embodied by a component to be reused. FX technology provides value by automatically calculating all behaviors in a legacy component. Modifying a component for reuse then consists of removing any undesired behaviors, improving the existing desired behaviors, and adding any new behaviors. The verification of a modified component via inspections and unit testing is also supported by FX technology as described above.

Component Acquisition. Components and services that are acquired from external vendors or even from internal corporate repositories present challenges to developers who must understand their behaviors. FX automation can provide a solution. A function extractor based on the semantics of the component's programming language can accept an unknown component and produce a complete behavior database. The resulting behavior can then be analyzed and compared to its contractual design specification. By evaluating several components in this manner, developers can create a basis for the best selection to meet acquisition requirements.

As examples of the application of FX technology for component evaluation and selection, consider the following situations:

- *COTS products* - A systems engineer requests a set of product behaviors from a COTS vendor to evaluate its planned use in a new system. FX will validate whether the behaviors are actually delivered and whether additional undesired behaviors are also present.
- *Service integration* - Before signing an agreement to include an online service in a critical supply chain application, a systems integrator requires the service provider to run the service through an FX system in order to analyze its full set of service behaviors. Note that the provider need not expose any proprietary code to the service user, only the computed behaviors.

Integration Testing

Function extractors are essentially generalized composition engines and, thus, they can also play a role in the integration of software components as determined by a system architecture. Based on the behavior database of each component, FX technology, guided by mathematical rules of component composition, can be adapted to integrate uses of the components into an assembled subsystem with a new, composite behavior database. The architecture specifies intended and allowable usage patterns (i.e., control flows and data flows) among the integrated components. The goals of integration testing can thus be supported by FX automation.

Systems Testing

With the advent of FX technology, an opportunity exists for systems testing and customer acceptance testing to shift from defect detection to certification of fitness for use. As the technology evolves and more automation becomes available, subsystems and entire systems could eventually be processed by FX automation, and resulting behavior databases compared with specifications and analyzed by stakeholders. A reduced set of test scenarios could be developed to demonstrate correct execution, because only one test per disjoint case of behavior is sufficient to validate all the behavior defined by that case.

Of course, the testing of system behaviors is only a portion of the full range of systems testing goals. The time and effort saved by use of FX automation can be devoted to more thorough testing and evaluation of environmental conditions (e.g., hardware platforms, external interfaces) and dynamic system qualities. For example, system testing for the qualities of performance, security, privacy, reliability, survivability, and maintainability, to mention a few, can and should become a greater focus of system testing (Walton et al., 2006).

Another important consideration is that eventual industry standards for FX technology could support outsourcing of system testing to independent groups that specialize in certifying the correctness and quality of software systems. As in more mature engineering fields, independent certification of quality standards for software systems with an industry-wide stamp of approval will help provide greater levels of trust in critical systems.

System Operations

The lower portion of Figure 4 shows the system in operation. It is generally accepted that a significant majority of the cost of a software system occurs after it is deployed, in the form of maintenance and upgrades to meet evolving customer requirements. FX technology could eventually support maintenance and evolution activities while providing opportunities for cost savings and quality improvements.

The key to system maintenance with FX technology is keeping behavior databases up to date automatically. As maintenance is performed on an operational system (for example, to improve performance or enhance security), the resulting system must still produce the same intended behaviors for unaffected functions as found in the database. As in system testing, a reduced set of regression test scenarios can provide a level of confidence that unaffected behaviors have remained unchanged.

In terms of system evolution, behavior databases provide a formal baseline against which all changes could be compared. New or modified behaviors could be specified and traced through component design and implementation behavior databases. Thus, developers could determine where and how to make required changes in system specifications, component designs, and code. Once code changes are made, FX automation could help ensure they have desired effects, while checking the integrity of behaviors that must remain unmodified.

Even when an operational system is not subject to maintenance and evolution activities, it may be wise to periodically perform function extraction to monitor and ensure that no malicious or inadvertent modifications have been introduced. Frequent application of the FX technology can help provide users with a level of confidence that no security compromises have occurred since the previous FX analysis.

Discussion of FX Impacts on Software Testing

It is not surprising that the software testing process can benefit from a thorough analysis of the code being tested. Test coverage metrics, such as statement or branch coverage, are one example of this. There are also situations where examination of the code and its structure can be used to reduce the number of test cases needed without the possibility of missing an error in the program. For example, if both the code and the specification are linear functions (in the mathematical sense) of N numeric input values, then any N+1 linearly independent test cases suffice to demonstrate correctness. For functions of a single numeric input value, if the code and the specification are polynomials of degree M, then any M+1 distinct test cases suffice.

However, if all that is known about the code and the specification is that they are total recursive functions, then every possible input value must be tested.

<u>Observation 1:</u> *If nothing is known about the internals of a sequential program (a black box as far as testing is concerned), then in order to guarantee correctness, it is necessary to execute every possible test case where the program specification makes a non-trivial requirement about the program output.*

The advent of function extraction technology provides the ability, outside the strict confines of linear or polynomial functions, to reduce the number of test cases without the possibility of missing a program error.

<u>Observation 2:</u> *For a sequential program, if the functional behavior extracted from the program implementation satisfies the program specification, then the program is functionally correct, and no testing for functional behavior is required.*

In scaling up to computation and comparison of the behavior of large programs to their specifications, a divide and conquer strategy forms the basis for a stepwise process that operates on components of manageable size. Specifications can be written in a form amenable to automated comparison, but even if specifications do not exist, inspection of behavior databases through human and/or automated means can validate desired behavior and reveal any unwanted functionality. The idea is to replace labor-intensive manual verification with automation wherever possible, in the knowledge that some level of human analysis will always be required. In any event, we believe that routine availability of computed behavior can have significant impact on the cost and quality of software system development, by substituting cheap computing power for resource-intensive, human-based activities.

These observations identify a potential paradigm shift supported by FX capabilities for software testing. Test coverage metrics are no longer the sole basis for determining whether sufficient testing has been completed. All behaviors in the program code are identified and can be inspected and evaluated for correctness against the program specification, whether the specification is documented or exists as a mental model of desired behavior.

Function extraction could also make it easier to determine if malware or corrupted functionality is present in operational programs. Behavior databases can be generated on a periodic basis and compared with baseline databases to help detect any malicious content.

As noted, many other testing objectives must be satisfied, including evaluation of the performance and interaction of programs in complex computational environments. Function extraction has the potential to free testing resources to focus on these objectives with the knowledge that the functional behavior of constituent programs is known and validated. This resource shift can significantly impact the economics of software engineering, resulting in faster and cheaper development of higher quality systems (Collins et al., 2008).

References

Collins, R., Hevner, A., Walton, G., and Linger, R. (2008). "The Impacts of Function Extraction Technology on Program Comprehension: A Controlled Experiment," *Information and Software Technology*, doi: 10.1016/j.infsof.2008.04.001.

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

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

Linger, R., Mills, H., and Witt, B. (1979). *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA.

Linger, R. and Pleszkoch, M. (2004). "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.

Linger, R., Pleszkoch, M., Burns, L., Hevner, A., and Walton, G. (2007). "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.

Pleszkoch, M., Hausler, P., Hevner, A., and Linger, R. (1990). "Function-Theoretic Principles of Program Understanding," *Proceedings of the 23rd Annual Hawaii International Conference on System Science (HICSS35),* Hawaii*,* IEEE Computer Society Press, Los Alamitos, CA.

Poore, J., Mills, H., and Mutchler, D. (1993). "Planning and Certifying Software System Reliability," *IEEE Software*, Vol. 10, No. 1, pp. 88-99.

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

Sayre, K. and Poore, J. (2007). "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.

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

Whittaker, J. (2000). "What Is Software Testing? And Why Is It So Hard?" *IEEE Software*, Vol. 17, No. 1, pp. 70-79.

About the Authors

Mark Pleszkoch, Senior Technical Staff Member at the Software Engineering Institute (SEI), researches automated proof checking and its application to formal verification of programs. Dr. Pleszkoch earned his Ph.D. in Computer Science from the University of Maryland, and an MS in mathematics from the University of Virginia where he was also a Putnam Fellow of the Mathematics Association of America.

Richard Linger manages the Carnegie Mellon University Software Engineering Institute's CERT STAR*Lab, and the Survivable Systems Engineering group. Previously at IBM, he co-developed Cleanroom software engineering specification, design, verification, and certification technologies for creating high-reliability software. He has taught software and security courses at Carnegie Mellon University, and has published three software engineering textbooks and a number of book chapters and technical papers.

Alan Hevner, Eminent Scholar and Professor in the Information Systems and Decision Sciences Department at the University of South Florida, also holds the Citigroup/Hidden River Chair of Distributed Technology. Dr. Hevner's research includes information systems development, software engineering, distributed database systems, health care information systems and telecommunications. Dr. Hevner earned his Ph.D. in Computer Science from Purdue University.