Next-Generation Software Engineering: Function Extraction for Computation of Software Behavior

Richard C. Linger,
Mark G. Pleszkoch, and
Luanne Burns
*CERT STAR*Lab*
Software Engineering Institute
Carnegie Mellon University
rlinger@sei.cmu.edu
lburns, mpleszko@cert.org

Alan R. Hevner
College of Business Administration
University of South Florida,
National Science Foundation,
and
*CERT STAR*Lab*
Software Engineering Institute
Carnegie Mellon University
ahevner@coba.usf.edu

Gwendolyn H. Walton
Dept. of Mathematics & Computer Science
Florida Southern College,
and
*CERT STAR*Lab*
Software Engineering Institute
Carnegie Mellon University
gwalton@flsouthern.edu

Abstract

*The ultra-large-scale systems of the future require the transformation of software engineering into a computational discipline capable of fast and dependable software development. This paper discusses an emerging next-generation software engineering research area: function extraction (FX) technology for automated computation to the maximum extent possible of the behavior, correctness, and quality attributes of software components and their compositions into systems. An introduction to the mathematical foundations for computation of software behavior is provided, followed by an overview description of a rigorously designed experiment to quantify the potential for FX technology, and a discussion of a CERT STAR*Lab first application of FX technology to compute the behavior of code expressed in the Intel assembly language instruction set.*

1. A History Lesson in Complexity

When the Normans conquered England in the 11th century, a census was ordered to catalog what had been won. But after the data were collected, the required summations were not produced despite the obvious interest in the results. The census had been recorded in Roman numerals, and the complexity of adding up so many numbers in that system was overwhelming. Yet if the census had been recorded in decimal arithmetic with place notation, the required sums could have been produced in short order.

There is a lesson here for the problems of present-day computing. It is that technology can either add complexity that thwarts human intellectual control, or it can reduce complexity to enable what seem like superhuman capabilities. In this case, Roman arithmetic adds complexity because it does not scale to large problems. Decimal arithmetic reduces complexity because it is scale-free; large problems simply require more of the same operations used for small problems. And the correctness of the operations themselves, whatever human fallibility may be present in their application, is guaranteed by the theoretical foundations of arithmetic.

2. Next-Generation Software Engineering

Present-day software engineering faces two seemingly intractable and interrelated problems: complexity and cost. Complexity is an ever-present barrier in system development and evolution. Its principal manifestation is the massive accumulation of low-level details and intricate relationships among them that quickly exceeds human understanding. No other engineering discipline requires its practitioners to remember and reason about so many details. It is a sobering reality that programmers today have no practical means to determine the full functional behavior of programs written by themselves or others. Furthermore, no testing process, no matter how thorough, can validate the full functional behavior of programs in all circumstances of use. The result is systems fielded with unforeseen errors and vulnerabilities that no amount of being careful and trying harder has been able to prevent.

The problem of cost is closely related to complexity. Battalions of programmers and lots of time are required to create large systems because programmers

are complexity-limited in present development technologies and coordination-limited in present development organizations. It is increasingly clear that software engineering is reaching the limits of development technologies evolved in the first fifty years of computing. Science foundations for next-generation software engineering (NGSE) are required for the next fifty years that will enable fast and correct development of the ultra-large-scale systems of the future.

We believe that an overarching requirement for NGSE will be new forms of computational automation that will augment human intelligence in analyzing and developing software engineering artifacts ranging from specifications and architectures to designs and implementations. This automation should analyze artifacts in terms of complete behavior and properties, because partial analysis can produce only partial knowledge. It must reduce cost by substituting computational analysis for human effort, and limit complexity by producing analytical results that preserve intellectual control.

A principal objective of NGSE is thus to make software engineering computational, in the sense that, say, present-day aeronautical and electrical engineering are computational. These disciplines depend on routine capabilities for computational evaluation of the expressions that define and manipulate their subject matter. They have encountered the problems of large-scale systems, and have developed mathematical foundations and engineering automation to cut costs and control complexity in their development. For example, in electrical engineering, automation-supported design of processor chips containing millions of circuits is a common practice, a feat well beyond human analytical capabilities and totally dependent on automated computational methods. We believe the same must be made true for software engineering.

Many threads of research and much effort will be required to develop theoretical foundations and engineering automation for NGSE. Substantial work in this area has been underway for years, for example, in automation for the sequence-based specification [1] and statistical certification [2] technologies of Cleanroom software engineering [3, 4], as well as in model checking [5] and proof-carrying code [6], to name just a few. This paper discusses an emerging NGSE research area: function extraction (FX) technology for automated computation to the maximum extent possible of the behavior, correctness, and quality attributes of software components and their compositions into systems. FX is based on a view of software components as implementations of mathematical functions or relations. Foundations of automated behavior computation are emerging in the form of computational semantics defined by a function-theoretic view of software.

3. An NGSE Example: Function Extraction

A key to intellectual control in system development and evolution is capturing and understanding the full functional behavior of software with mathematical precision. Such complete understanding is very difficult in today's state of practice. The behavioral semantics of software building blocks ranging from processor instruction sets to programming languages to operating, database, and middleware systems may be well or poorly documented, and well or poorly understood. Engineers create the behavioral semantics of processor operations by combining circuits; programming language designers create the behavioral semantics of language instructions by combining chip operations; programmers create the behavioral semantics of components by combining language instructions; and integrators create the behavioral semantics of systems by combining components. At each step, ever more semantic content accumulates, some of which may be unavoidably lost or misunderstood as the sheer quantity and complexity of programmed behavior exceeds human analytical capabilities. And, of course, whatever semantic structures may have been created, computers simply fetch the next instruction and execute it, to produce the behavioral effects that have been programmed, right or wrong, understood or not. This inevitable loss of semantic information has been the source of many long-standing problems in software engineering, and motivates a closer look at possibilities for automated calculation of software behavior.

The objective of behavior calculation is to reveal and preserve the semantic information in programs and other software artifacts as a means to augment human capabilities for analysis and design. We discuss these possibilities below in the context of sequential logic, in the full knowledge that other problems exist, for example, in concurrent and recursive logic. Our goal is to say first words on the subject, not last words.

The well-known function-theoretic view of software [3, 4, 7] provides bedrock mathematical foundations for computation of behavior. In this perspective, programs are treated as rules for mathematical functions or relations, that is, mappings from inputs (domains) to outputs (ranges), regardless of the subject matter addressed or the implementation languages employed. The key to the function-theoretic approach is 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 composed of 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. While no comprehensive theory for computing the behavior of iteration structures can exist, satisfactory engineering solutions are possible.

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

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

$$f = [P] = [g; h] = [h] \circ [g]$$

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

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

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

That is, the program function is a case analysis of the true and false branches, with the possibility of combining them into a more general function, for example, max, min, etc.

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

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

and f must satisfy

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

As noted above, no general theory for iteration abstraction can exist, so engineering solutions must be developed for semantic analysis and recognition. In any event, it is indeed fortunate that the behavior of sequence and alternation structures, which comprise the bulk of program logic, can be computed by such straightforward means.

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

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

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

The composition computations for the thenpart and elsepart sequence structures are easily carried out through accumulating trace tables,

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

which gives the sequence-free concurrent assignment

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

and

Operation	x	Y
x := x + y	x = x + y	Unchanged
y := x – y	unchanged	y = x + y – y = x
x := x – y	x = x + y – x = y	Unchanged

which gives the sequence-free concurrent assignment

x, y := y, x.

These functions combine in the alternation analysis as a conditional concurrent assignment:

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

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

x, y := y, x.

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

x, y := y, x

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

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

whose trace table

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

reveals the overall behavior to be

x, y := x, y

or simply the identity function, and the possibility arises of either a programming error in the fragment or an intentional obfuscation of its logic.

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

4. Quantifying the FX Business Case

To explore the potential of FX technology, CERT STAR*Lab developed a proof-of-concept prototype that operates on a small subset of the Java programming language. The prototype takes in a Java program written in the language subset and automatically calculates and displays its functional behavior. The behavior is expressed in conditional, concurrent assignments that provide the complete, correct function for the program. That is, the prototype outputs a non-procedural, user-readable format in terms of how the outputs of the program are produced from its inputs in all possible uses, in effect producing the as-coded specification of the program in the form of a behavior catalog. A rigorously designed experiment was conducted to quantify the impact of FX technology on the ability of programmers to comprehend and verify programs. The methodology and results of this experiment are summarized here. Complete details are available in reference [8].

The experimental subjects were 26 Carnegie Mellon University (CMU) graduate students, each of whom had substantial computer science education and software development experience. The experiment was approved by the CMU Institutional Review Board, calibrated in two pilot tests, and conducted according to rigorous experimental protocols. The 26 subjects received 45 minutes of classroom instruction on the purpose of the experiment and the process of program comprehension. They were then randomly divided into two groups: the control group (manual manipulation) and the experimental group (automated FX manipulation).

The control group subjects were given three Java programs of varying size and difficulty, together with functional requirements for the programs and a set of questions to answer. They applied traditional methods of code reading and inspection to understand the behavior of each program and then answered the questions.

The experimental group subjects installed the Function Extraction prototype on their personal laptop computers. These subjects, who had no previous exposure to the prototype, received an additional 45 minutes of classroom instruction on its use and were then given the same three Java programs, requirements, and questions. The experimental group ran each program through the FX prototype to derive and display the program's functional behavior, and then answered the questions.

For both the control group and the experimental group, all activities were self-timed by the subjects, and a post-hoc questionnaire was completed. Both the time to perform the experimental tests and the correctness of the answers were measured and analyzed. The correctness of the answers provided a measure of the ability of both groups to understand the program behaviors and to verify the behavior against requirements.

The validity of the experimental design and procedures was accessed in four ways:

- The participants answered a manipulation check question appropriately.
- The participants' rankings of the difficulty of the three programs were consistent with the experimental design. The shortest program, named *Algorithm*, was ranked as the least difficult of the three programs, and was rated by the participants as less difficult than the participants' usual program comprehension tasks. The *Ordering* program was longer than the *Algorithm* program. *Ordering* was ranked by the participants as more difficult to comprehend than *Algorithm* and was rated at about the same level of difficulty as the participants' usual program comprehension tasks. The longest program, named *Bonus Points*, was ranked as the most difficult and was rated as more difficult than the participants' usual comprehension tasks.
- The participants agreed that the training they received was sufficient to complete the study tasks.
- The participants were experienced with program comprehension tasks and were randomly assigned.

Statistical analysis of the participants' education and experience was performed using Levene's test for equality of variance and the t-test for equality of means. The analysis demonstrated that there was no significant difference between the control and experimental groups in their experience in program comprehension; paid, non-classroom experience; or number of programming classes taken.

Descriptive data (mean and range for time on task and percentage correct) was calculated for the performance of each group (control and experimental) on each of the three programs. The subjects in the experimental group also evaluated the FX prototype on several standard criteria and their measures. The evaluation criteria, the scale reliability, and results are provided in reference [8]. Performance on the three experimental tasks was measured by three dependent variables: accuracy of program comprehension, total time on task, and time required to derive program behavior. Study data were analyzed using analysis of variance and Welch/Brown-Forsythe Robust Test of Equality of Means.

Results of the experiment showed that the group using the FX prototype reduced the time required to derive the functional behavior of the most difficult program by about three orders of magnitude over the control group, and was about four times better at providing correct answers to the program comprehension and verification questions in about one-fourth of the time. Statistical analysis of the experimental results showed that the probability that the results could be attributed to chance was close to zero. In addition, the experimental group expressed a high level of satisfaction in the use of the FX prototype. [8]

5. Development of a Function Extractor

The need for fast and reliable understanding of code is nowhere more important that in analyzing malicious code to quickly develop countermeasures. CERT STAR*Lab at the CMU Software Engineering Institute has initiated a first application of function extraction technology to compute the behavior of code expressed in the Intel assembly language instruction set to the maximum extent possible [9, 10]. The Function Extraction for Malicious Code (FX/MC) system operates as a plug-in to the Ida Pro disassembler. It applies the function equations and many other mathematical techniques to compute the behavior of assembly language programs that have been identified as malicious. The development of FX/MC is a substantial effort that is currently underway. In what follows, we describe the structure and capabilities of the system.

Function extractor front-ends and back-ends are programming language specific as seen in Figure 1. Front-end processing must accommodate the syntax of the target language and assign complete and correct functional semantics to every instruction in an input program. Thus, a necessary precursor to extractor development is creation and verification of semantic

definitions on an instruction-by-instruction basis. This is a significant task in itself: the Intel processor supports over 1000 opcodes. Back-end processing must likewise accommodate the syntax of the language in displaying the output program and its behavior catalog.

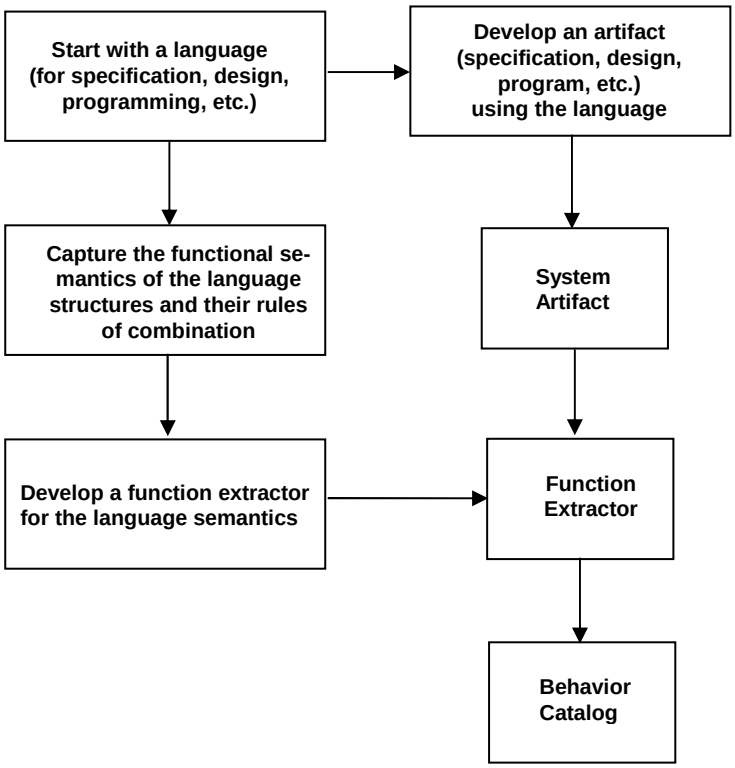

Figure 1: Function Extractor Development

The internal logic of function extractors is more general and is organized along the following lines. The first step, *instruction semantics analysis*, expresses instructions in an input program in terms of their functional semantics for subsequent processing. The second step, *control deobfuscation*, determines the true control flow of the input program. For some languages, most notably assembly languages, this can be a difficult task in the presence of computed jumps and other dynamic control flow operations. The next step, *control structuring*, transforms the true control flow, often in confusing, spaghetti-logic form, into a function-equivalent structured program composed entirely of fundamental control structures in a nested and sequenced hierarchy with no arbitrary jumps present. This algebraic structure is the starting point for behavior computation. The next step, *stepwise extraction*, computes the behavior of leaf-node control structures, thereby revealing new leaf nodes, continuing in this manner until the behavior of the entire program has been extracted and recorded in a behavior catalog as a set of disjoint conditional concurrent assignments. In this step, a simplification process, *expression normalization*, reduces intermediate behavior definitions to their simplest forms prior to propagation to higher levels. Local details of control and data operations are left behind while net effects are preserved. Normalization requires substantial mathematical machinery, which extends to computer algebras and theorem provers, to maximize the value of the simplification process. This entire processing takes place under user control; significant human factors research and development is required to optimize the value of function extraction in augmenting human capabilities.

Operation of the current version of FX/MC is described next through a behavior computation example. Consider the following miniature assembly language fragment and the question of what it does.

```
Loc_8000000:
   push eax
   pop ebx
   push ecx
   jmp short loc_8000015
loc_8000005:
   sub esp, 4
   pop ebx
   jmp short loc_8000019
loc_800000B:
   push ecx
   pop eax
   pop eax
   jmp short loc_8000005
loc_8000010:
   push eax
   pop ebx
   push ecx
   jmp short loc_8000005
loc_8000015:
   pop edx
   push ebx
   jmp short loc_800000B
loc_8000019:
```

The function-equivalent structured output produced by FX/MC is shown below. It turns out that the fragment is in fact a simple sequence control structure despite the presence of jump instructions in the original that may have represented an intentional obfuscation of the logic. Malicious code writers often obfuscate control flow to make it more difficult for analysts to understand functional intent. The transformation to structured form substantially reduces the effectiveness of control obfuscation as a weapon for intruders. Note that the structuring process identified and removed the dead code at loc_8000010:

```
    do
 1.    push eax
 2.    pop ebx
 3.    push ecx
 4.    pop edx
 5.    push ebx
 6.    push ecx
 7.    pop eax
 8.    pop eax
 9.    sub esp, 4
10.    pop ebx
    enddo
```

The extracted function for the fragment computed by FX/MC is depicted below, where concurrent assignments are made to registers EAX, EBX, and EDX, to stack pointer ESP, and to two memory locations denoted by expressions of the form M_dword(location) := value. Flags ZF (zero flag), SF (sign flag), PF (parity flag), CF (carry flag), OF (overflow flag) and AF (auxiliary carry flag) are also concurrently assigned values. The +d symbol stands for doubleword addition.

```
[ EAX := EAX
: EBX := EAX
: EDX := ECX
: ESP := ESP

: M_dword(ESP +d -4):= EAX
: M_dword(ESP +d -8):= ECX

: ZF := (4 == ESP)
: SF := is_neg_signed_dword(ESP +d -4)
: PF := is_even_parity_lowbyte(ESP +d -4)
: CF := (ESP <= 3)
: OF := is_in_interval(ESP,2^31,(2^31)+3)
: AF := false
]
```

Note that this extracted behavior is procedure free. The assignments occur all at once – they are concurrent updates of final values on the left hand side in terms of initial values on the right hand side. The extracted function defines the complete, as-coded behavior of the fragment.

What does the function tell us? Some of the computed behavior does not appear to be obvious from examining the program. The first concurrent assignment, EAX := EAX, shows that the final value of the EAX register is the same as its initial value. This can be validated by observing that the EAX register is set on line 7 and again on line 8, both times by means of the pop instruction. The value assigned to EAX on line 7 comes from the push on line 6, which is the value of ECX. But this value is overwritten on line 8 by the value of EBX that was pushed on line 5. However, the value held in the EBX register at that point is the one that was assigned to EBX on line 2, which in turn comes from the value of EAX pushed on line 1. Thus, the original value of EAX is actually restored to the EAX register on line 8.

The second concurrent assignment, EBX := EAX, can be similarly validated by tracing the program values back from the pop into EBX on line 10. The third concurrent assignment, EDX := ECX, comes from the assignment to EDX on line 4. The fourth concurrent assignment, ESP := ESP, declares that the stack use is balanced, that is, there are an equal number of pushes and pops. On the surface, this appears to be untrue, as there are five pops but only four pushes in the program. However, note that the operation on line 9 (sub ESP, 4) has the same effect as a push on the stack pointer, except that no value is written to the stack memory, so that the pop on line 10 actually returns the "dirty" memory value that was already popped on line 8. The fifth and sixth concurrent assignments document the side effects to memory caused by the stack use. The concurrent assignments after that document the final values of the processor flags, which are set by the subtract instruction on line 9, and can be similarly validated.

The informal validation outlined here confirms the behavior computation carried out by FX/MC. However, the objective of the system is to make such checks unnecessary, to permit software engineers to compute program behavior at CPU speeds and make immediate use of the results.

6. Behavior Computation and NGSE

As a computational discipline, we believe that next-generation software engineering will require extensive capabilities for automated analysis of engineering artifacts, from specifications to designs to implementations, at all levels of abstraction. Function extraction can contribute to this automation in a number of areas, as in the following illustrations.

Correctness Verification Computation. NGSE can benefit from increasing the granularity of dependable software engineering artifacts by orders of magnitude. For example, a billion-line system becomes a million-unit system if the dependable unit of construction is thousand-line components, which in turn build to reliable ten- and hundred-thousand-line subsystems. Evaluation of correctness will require that the full functional behavior of components be known. Computed behavior at the component level will permit direct correctness verification by human and machine-based processes, thereby scaling up the reliable unit of construction for software systems.

Quality Attribute Computation. NGSE can also benefit from treating quality attributes, for example, performance and security, as time-varying functions for computational evaluation, rather than as static and a priori estimates that have limited value in system operation. Quality attributes defined in functional terms, combined with availability of computed behavior, will permit computational analysis and prediction of attribute values in system development, as well as real-time evaluation of attribute dynamics in system operation as usage and threat environments change over time. CERT STAR*Lab has initiated research on use of the behavior catalogs produced by the function extraction process for computational analysis of security attributes of programs [11].

Truth-Preservation Computation. NGSE can benefit from real-time self-description of system structure and function for use in computer and human analysis. Automated truth maintenance capabilities could initially derive and subsequently preserve the correctness and completeness of definitions of software function as systems change and evolve. The full effect of modifications on system behavior and attributes could be computed, and violations of behavioral invariance under system refactoring and reorganization could be detected.

Component Composition Computation. NGSE will require that the composite behavior of components architected into systems be calculated for fast and reliable development. Such a capability is important for creating dynamic systems-of-systems, and for achieving confidence in distributed, component-based and service-oriented architectures. Component composition computation will help scale up the dependable unit of construction for complex software systems. Research will be required in this area to combine behavior computation with interface ontologies and domain-specific semantics.

Reduced Abstraction Set Engineering. Languages employed for specification, architecture, design, and implementation could be configured with special properties that facilitate computational analysis and human reasoning. In particular, simplifying and unifying additions and constraints could be applied to existing languages and drive development of new ones with no loss of expressiveness to reduce complexity and support automated computation of behaviors, quality attributes, and compositions. That is, languages could be designed to conform to the behavior computation process and thereby reduce the quantity and variety of behavior abstractions.

7. Future Research

The promise of next-generation software engineering will require research and development in a number of critical areas. Foremost, human developers need the capability to fully understand the behavior of system artifacts, such as programs, in minutes rather than the days and months currently required. In this paper, we report our initial research and development directions on function extraction as a means to capture the behavior of programs.

CERT STAR*Lab intends to investigate the application of function extraction to correctness verification, security attribute analysis, truth preservation in system definition, and component composition. In addition, the technology can be extended to other languages, for example, Java and C, and even to FORTRAN and COBOL for analysis of legacy systems [12]. Efforts are also underway to develop stand-alone structuring engines to improve human understanding of program structure and control flow.

Controlled experiments are planned to better understand the human factors involved in use of computed behavior, in particular, for team operations where reliable knowledge sharing is a key to efficiency and effectiveness in software development.

8. References

[1] Prowell, S. and Poore, J. “Foundations of Sequence-Based Software Specification,” *IEEE Transactions on Software Engineering*, Vol. 29, 2003.

[2] Poore, J., Mills, H., and Mutchler, D. “Planning and Certifying Software System Reliability,” *IEEE Software*, Vol. 10, 1993.

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

[4] Mills, H. and Linger, R. "Cleanroom Software Engineering," *Wiley Encyclopedia of Software Engineering: Second Edition*, Wiley, New York, 2002.

[5] Clarke, E., Grumberg, O., and Peled, D. *Model Checking*, MIT Press, 1999.

[6] Necula, G. and Lee, P. “Safe Kernel Extensions Without Run-Time Checking,” *OSDI '96: Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, ACM Press, New York NY, 1996.

[7] 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 (HICSS35), Hawaii*, IEEE Computer Society Press, Los Alamitos, CA, January 1990.

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

[9] Hausler, P., Pleszkoch, M., Linger, R., and Hevner, A. "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, Vol. 7, No. 1, January 1990, pp. 55-63.

[10] 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.

[11] *CERT Research 2005 Annual Report*, Software Engineering Institute, Carnegie Mellon University, 2005, www.cert.org/research/.

[12] 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.