Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior

Mark G. Pleszkoch

IBM Global Services and CERT Research Center Software Engineering Institute Carnegie Mellon University, Pittsburgh, PA markp@us.ibm.com

Abstract

Malicious attacks on systems are a threat to business, government, and defense. Many attacks exploit system behavior unknown to the developers who created it. In today's state of art, software engineers have no practical means to determine how a sizable program will behave in all circumstances of use. This sobering reality lies at the heart of many problems in security and survivability. If full behavior is unknown, so too are embedded errors, vulnerabilities, and malicious code. This paper describes function-theoretic foundations for automated calculation of full program behavior. These foundations treat program control structures as mathematical functions or relations. The function, or behavior, of control structures can be abstracted in a stepwise process into procedurefree expressions that specify their net functional effects. Problems of computability and complexities of language semantics appear to have engineering solutions. Automated behavior calculation will add rigor to security and survivability engineering.

1. Understanding Program Behavior

Traditional engineering disciplines depend on rigorous methods to evaluate the expressions (equations, for example) that represent and manipulate their subject matter. Yet the discipline of software engineering has no practical means to fully evaluate the expressions it produces. In this case, the expressions are computer programs, and evaluation means understanding their full behavior, right or wrong, intended or malicious. Short of substantial time and effort, no software engineer can say for sure what a sizable program does in all circumstances of use. Yet modern society is dependent on the correct functioning of countless large-scale systems composed of programs whose full behavior and security properties are Richard C. Linger CERT Research Center Software Engineering Institute Carnegie Mellon University, Pittsburgh, PA rlinger@sei.cmu.edu

not reliably known. Many of these systems control key infrastructures in communication, energy, finance, and transportation. The existence and malicious exploitation of unknown functionality is the Achilles heel of software. It is little wonder that systems experience an endless flood of errors, vulnerabilities, and malicious code with frequently serious consequences.

The task of understanding program behavior is a haphazard, error-prone, resource-intensive process carried out by programmers and analysts in human time scale. Yet reliable understanding is essential to discover vulnerabilities and malicious code. And because attackers can make deleterious modifications to programs at any time, the task of behavior discovery never ends. The problem clearly exceeds manual capabilities and must be addressed through automation.

Sizable programs are hard to understand because they contain a huge number of execution paths, any of which may contain security exposures. Faced with massive sets of possible executions, programmers can often do no more that achieve a general understanding of mainline program behavior. There is simply no way to understand and remember it all in today's state of practice. The situation is illuminated by an argument of the open source software movement, that more people looking at code will find more security flaws. It is interesting to note there is no open source arithmetic movement, seeking more people to determine if sums are flawed. Society knows how to make sums correct and has automated the process. It turns out the same may be true of software. Function-theoretic mathematical foundations of software illuminate a feasible strategy to develop innovative automation to address security exposures. An opportunity exists to move from an uncertain understanding of program security properties laboriously derived in human time scale to a precise understanding automatically computed in CPU time scale.

The function-theoretic model of software [4, 5, 6, 7, 10, 11 14] treats programs as rules for mathematical functions, that is, mappings from domains (inputs, stimuli)

to ranges (outputs, responses), no matter what subject matter programs deal with. The key to the functiontheoretic approach is the recognition that, while programs may contain an enormous number of execution paths, they are at the same time composed 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 single-entry, single-exit sequence (composition), alternation (ifthenelse), and iteration (whiledo) control structures, plus variants and extensions [7, 15]. This finite property of program logic viewed through the lens of function theory opens the possibility of automated calculation of program behavior. Every control structure in a program has a behavior signature, which can be extracted and composed with others in a stepwise process based on an algebra of functions that traverses the control structure hierarchy. The behavior signature of a program represents the specifications or business rules that it implements. These concepts are the basis for function extraction (FX) technology.

Section 2 of this paper introduces concepts of behavior computation, and section 3 defines the equations that map control structures into behavioral representations. Section 4 describes an algebra of functions for stepwise behavior extraction, and section 5 elaborates on research areas that must addressed. Section 6 describes a malicious code example, and section 7 discusses the architecture of a function extractor. Section 8 connects function extraction to related research.

2. Program Behavior Signatures

The behavior signature of a program control structure defines its net functional effect in terms of how it transforms input data values into output data values [3, 13]. Behavior signatures are inherently procedure-free, that is, they define behavior with all sequence, branching and looping logic and local data items abstracted out to facilitate human understanding. What remains is a precise specification of the overall transformation carried out by the structure. In informal illustration, Table 1 depicts behavior signatures of representative control structures operating on small non-negative integers.

How are behavior signatures computed? Consider the sequence control structure below constructed of assignments that operate on small integers x and y (matters of machine precision are left aside for the moment). The behavior extraction question asks: What does this program do, that is, what function does it compute? The answer is not obvious at first glance.

Table 1. Example Behavior Signatures

Control	Behavior
Structure	Signature
do	
$\mathbf{a} := \mathbf{b} + \mathbf{c}$	set a to 2 and b to
b := a + 2	b + c + 2
a := b - a	
enddo	
if $x > y$	set z to the
then $z := x$	maximum of x and
else $z := y$	y and leave x and
endif	y unchanged
while $x > 1$	
do	set odd x to 1,
x := x - 2	even x to 0
enddo	

do

x := x - yy := y + xx := y - xenddo

Function extraction requires deriving a procedure-free expression of what this structure does from beginning to end for all values of x and y. For a sequence structure, this requires composing the statements to determine their net, sequence-free effect. A simple trace table as depicted in Table 2 can be used for this purpose, with a row for every assignment and a column for every data variable assigned. Cells in the table record the effect of the row assignments on the variables. Subscripts are attached to variables to index effects from row to row, with 0 denoting initial values.

Table 2. Trace Table Construction

Operation	Effect on x	Effect on y
$\mathbf{x} := \mathbf{x} - \mathbf{y}$	x1 = x0 - y0	y1 = y0
$\mathbf{y} := \mathbf{y} + \mathbf{x}$	$x^2 = x^1$	$y_2 = y_1 + x_1$
$\mathbf{x} := \mathbf{y} - \mathbf{x}$	x3 = y2 - x2	y3 = y2

The derivations below express final values in the table in terms of initial values through algebraic substitution:

$$x3 = y2 - x2$$

= y1 + x1 - x1
= y1
= y0

$$y3 = y2$$

= y1 + x1
= y0 + x0 - y0
= x0

Thus, the computed final values reveal that the control structure exchanges the initial values of x and y. This is its behavior signature, and can be written as a concurrent assignment in which initial values on the right are simultaneously assigned in order to final values on the left, and attached to the control structure in square brackets as a comment to document its function:

[x, y := y, x]do x := x + yy := x - yx := x - yenddo

The control structure in this example is a simple sequence whose behavior was derived in a trace table composition: alternation and iteration structures require trace tables that incorporate columns for conditions (predicates) as well, and derivation of their final values in terms of initial values in similar fashion.

This behavior function defines the net effect of the sequence with matters of machine precision aside. If necessary, however, the finite nature of machine precision can be integrated into the analysis. For example, properties of overflow and underflow can be dealt with in several referentially transparent ways, with the best approaches ultimately determined through experience with function extraction technology and user preferences, as discussed next.

The first approach is to ignore overflow and underflow in the extraction process. This method corresponds to performing referentially transparent extraction on a program and machine model that have infinite precision. In this case, the calculated behavior precisely defines the net effect of the sequence with machine precision not accounted for, and is sufficient for many analytical purposes. It is the obvious choice where machine precision has no effect on particular operations. An advantage of this approach is that the calculated behavior signature is not complicated by details of finite precision; however, any behavior resulting from finite precision is lost. For example, if, in the sequence structure above, two's complement arithmetic is preformed and overflow and underflow do not cause machine traps, then the exchange behavior is always correct, even when overflow does occur, because the result must be correct modulo the word size of the executing machine. If necessary, however, the finite nature of machine precision can be incorporated through a second or third approach as follows.

In the second approach, the domain of each potential overflow or underflow can be explicitly incorporated into the conditions of the conditional assignment statement. The finite nature of integer representations on a given machine introduces the possibility of underflow and overflow into the functional effect of the sequence, and the possibility of producing other than the intended result. This corresponds to performing referentially transparent extraction on a program and machine model with finite precision to a behavior model with infinite precision. This possibility can be accounted for by partitioning the domain and range of each assignment in the sequence into equivalence class regions, based in this example on subsets of initial values of x and y, within each of which the same functional results will be obtained. Some classes will produce the program function calculated above, others will not. Incorporation of the operational semantics of machines is important for analysis of programs for vulnerabilities and malicious code intended to exploit, for example, finite properties and overflow characteristics of number representations or data structures such as buffers or registers. When the behavior calculations are augmented by operational semantics, such problems become obvious, with no additional analysis on the part of the user required. For example, consider the following program function for a single assignment:

 $[((x + y) \ge 2^{3}) \rightarrow \text{overflow occurs} \\ |((x + y) < -2^{3}) \rightarrow \text{negative overflow occurs} \\ | \text{true} \qquad \rightarrow z := x + y] \\ \text{do} \\ z := x + y \\ \text{enddo} \end{cases}$

An advantage of this approach is that the complete behavior of the program is captured in the behavior specification, however, the overflow and underflow conditions may obscure the primary logic of the program. This disadvantage can be mitigated by introducing variable bounds as preconditions and treating the behavior outside those preconditions as undefined, as the following example illustrates:

```
[ (abs(x) < 10^8) and (abs(y) < 10^8) \rightarrow z := x + y
| true \rightarrow undefined]
do
z := x + y
enddo
```

The third approach is to incorporate the operational semantics of the executing machine into the behavior

calculation and simplification process as part of the trace table analysis. This method corresponds to performing referentially transparent extraction where the program and machine model, and the behavior model, are finite precision. That is, arithmetic operations in the behavior specification are subject to the same overflow and underflow as in the program. For example, consider the following treatment of the exchange program:

$$[x, y := (x + y) - ((x + y) - y), (x + y) - y]$$

do
$$x := x + y$$

$$y := x - y$$

$$x := x - y$$

enddo

In this method, "((x + y) - y)" cannot always be simplified to "x," because the original expression can exhibit overflow, while the simplified expression cannot. A disadvantage of this approach is that overflow and underflow semantics are embedded in calculated behavior just as deeply as in the program statements. Behaviors that do not require simplification, however, will more clearly reflect the primary logic of the program.

These examples illustrate the power of functiontheoretic methods to deal with any behavioral and operational semantics appropriate to the problem at hand. As work on function extraction progresses, suitable vocabulary, definitions, reduction and simplification rules, and flexible user interfaces will emerge to support human preferences and reasoning methods.

In any case, it is important to recognize that the process is capable of extracting the true and complete behavior of any program or program part, the very behavior that exposes unforeseen errors, vulnerabilities, and embedded malicious code. These behaviors are generated in the programmed functional logic and in its interaction with executing machines, and function-theoretic behavior calculation can deal completely and correctly with both.

3. Function-theoretic Foundations for Behavior Calculation

The canonical forms of behavior signatures of the basic control structures can be expressed through function composition and case analysis as follows (for control structure labeled P, operations on data labeled g and h, predicate labeled p, and program function labeled f). These function equations are independent of language syntax and program subject matter, and define the mathematical basis for behavior calculation: Sequence Control Structure: The program function of a sequence

can be given by

$$f = [P] = [g; h] = [h] o [g]$$

where the square brackets denote the behavior signature of the enclosed program and "o" denotes the composition operator. That is, the program function of a sequence can be calculated by ordinary function composition of its constituent parts as illustrated in the example above.

Alternation Control Structure: The behavior signature of an alternation control structure

P: if p then g else h endif

can be given by

$$f = [P] = [if p then g else h endif]$$

= ([p] = true \rightarrow [g] | [p] = false \rightarrow [h])

where | is the "or" symbol. That is, the program function of an alternation is given by a case analysis of the true and false branches, and the opportunity to combine them into a single abstraction as in the maximum operation of Table 1.

Iteration Control Structure: For iteration control structures, the program function is given by a mathematical analysis of the potentially infinite number of loop iterations using quantification over the natural numbers:

P: while p do g enddo

can be reexpressed as

 $f = [P] = \{(s,t) : Exists finite n \ge 0 \text{ such that } (s,t) \in ([p] = false) \text{ o } ([g] \text{ o } ([p] = true))^n \}$

Notice that this results in the function [P] being undefined on all initial states that cause the loop to fail to terminate. Fortunately, there is an alternative analysis that is very useful in practice, using function composition and case analysis in a recursive equation based on the equivalence of an iteration control structure and an iteration-free control structure (an ifthen structure):

f = [P] = [while p do g enddo]

= [if p then g; while p do g enddo endif]

= [if p then g; f endif]

Function f must therefore satisfy

$$f = ([p] = true \rightarrow [f] \circ [g] | [p] = false \rightarrow I)$$

where I is the identity function. Note that the converse is not true; there may be other functions that satisfy this recursive equation that are only subsets of f. However, even these subsets can prove useful in behavior calculation.

In addition to these function equations, key theorems of function-theoretic mathematics provide important guidance for the behavior calculation process [7, 8]:

Logic Structure Theorem: This theorem guarantees the sufficiency of sequence, alternation, and iteration control structures to represent any sequential logic. (Extensions and variants of these structures are included as well.) Thus, program logic can be expressed in nested and sequenced single-entry, single-exit structures, each with a common underlying mathematical model, namely, the function mappings defined above.

Abstraction/Refinement Theorem: This theorem and an associated Axiom of Replacement define conditions for substitution of behavior signatures and their control structure refinements, thereby enabling behavior extraction in a stepwise, algebraic process.

Flow Verification Theorem: This theorem defines conditions for correctness of control structures with respect to their behavior signatures. As noted above, even though programs can contain an enormous number of paths, they are expressed in a finite number of control structures, each of which can be verified in from one to three reasoning steps as defined by the theorem. Verification is thus reduced to a finite process.

4. Stepwise Extraction of Behavior in an Algebra of Functions

A behavior signature defines behavior identical to that of the control structure from which it was extracted, that is, the signature and control structure are functionequivalent mappings of inputs into outputs. Thus, signatures can be freely substituted for corresponding control structures [14]. Such substitution defines an algebra of functions that permits stepwise extraction of program behavior by traversing control structure hierarchies from bottom to top. At each step, net effects of control structures are composed and propagated while details are left behind. In illustration, consider the miniature program on the left of Figure 1 and the question of what it does. The program takes as input and produces as output a queue of integers named Q, and uses local queues of integers named odds and evens and a local integer variable named x (declarations not shown). The

symbol > stands for not equal, || for concatenation. The stepwise behavior calculation process is depicted in Figures 1 through 3.



Figure 1. Stepwise Extraction: First Step



Figure 2. Stepwise Extraction: Second Step



Figure 3. Stepwise Extraction: Final Step

The control structures of the program form a natural hierarchy with a number of leaf nodes. To begin the stepwise extraction process, the lowest level, leaf-node ifthenelse and sequence control structures are abstracted into behavior signatures expressed as conditional rules and assignments as depicted on the right of Figure 1. Next, the three whiledo structures, now leaf nodes in the remaining hierarchy, can likewise be abstracted to concurrent assignments, as shown on the right of Figure 2. Finally, the sequence of three behavior signatures can be composed into a single assignment expressing the overall behavior signature of the program as shown on the right of Figure 3. This assignment defines what the program does in functional terms. It is the as-built behavior specification, that is, the calculated behavior of the entire program. The extraction process reveals that the program creates a new version of queue Q, now containing its original odd numbers followed by its original even numbers. Note in this process that intermediate control structures and data uses drop out to simplify scale-up by subsuming their functional effects into higher-level abstractions. The principal behavior calculation process is function composition through value substitution, which by eliminates intermediate expressions at definition successive levels of abstraction. As noted above, programs can exhibit an enormous number of execution paths, but are comprised of a finite number of control structures, so the behavior calculation process is itself finite and guaranteed to terminate. Furthermore, behavior is recorded at each step, to produce functional documentation for human understanding at all levels.

This miniature example illustrates in informal terms a stepwise behavior extraction process that is invariant with respect to scale—the same mathematics and operations are employed at all levels of extraction, no matter the size of the program. Were this program embedded in a larger system, it is the extracted behavior of Figure 3 that would participate in further extraction, and not the program itself. In this way, local details are left behind at each step

with no loss of information while abstractions propagate to higher levels. Abstraction does not mean vagueness; the extracted behavior embodies the precise net effect of implementation details. This process, combined with other techniques, can limit complexity in behavior extraction of large programs. Additional mathematical methods for unification and reduction must be applied to simplify intermediate expressions. These methods address the question of scale up to a practical industrial process, and are key elements of the required work to automate function extraction. Note that such methods were applied in the example, where auxiliary functions odd numbers and even numbers were created to express the overall program behavior. Automated function extractors to carry out such behavior calculation will permit help eliminate error-prone human analysis in understanding both intended and unintended functionality of programs.

5. Function Extraction Research Areas

Research areas important to developing FX capabilities can be addressed through the following technical approaches:

Loop abstraction: No general mathematical theory for loop abstraction can exist, however an engineering solution for behavior extraction from loops is possible. Because even a single while loop can compute an arbitrary partial recursive function, many results from computability theory stand in the way. For example, the undecidability of the Halting Problem [2] means that there will be some terminating loops that automated function extractors will not be able to detect as terminating. The undecidability of program function equivalence implies that a function extractor must employ multiple representations of the same program function. The research approach here includes use of recursive expressions as discussed above to represent loop operations as a starting point for behavior extraction, and development and application of canonical patterns and behavior templates for loops. Initial analysis suggests a surprisingly small number of patterns can cover a variety of loop structures. Undecidability results from computability theory will be used to guide research choices along feasible directions. Potential limitations at the mathematical level can often be dealt with effectively at the engineering level to produce satisfactory solutions. This appears to be the case with respect to loop abstraction.

Indirect data references: Popular programming languages permit indirect references to data structures through pointers and pointer manipulation, with attendant complexities in extracting and understanding behavior. The solution strategy involves representing pointerreferenced data in canonical storage maps to systematize data references in a common framework. The underlying storage maps will permit uniform treatment of all data operations. Pointer references to data items, either explicit or implicit, introduce a level of indirection that must be accommodated in the semantics of behavior calculation. The approach here includes using data references expressed in canonical reference frameworks as a starting point for analysis. In particular, all objects are allocated on the heap in Java, so the problem of aliasing, that is, different variable names referring to the same storage location, is substantial. This is an area where automatic analysis can bring significant benefits to understanding implications of the data layout of an unknown program.

Scale-up: Extracted behavior expressions must be limited in complexity for rapid human understanding and analysis. It is thus important to control complexity in expressions as they propagate to higher levels. As discussed above, much complexity reduction is intrinsic to behavior extraction. function-theoretic Additional simplification can be achieved through mathematical methods for unification and elimination of cases in behavior expressions, as well as through human factors engineering for effective display and analysis of behavior catalogs. A key strategy in this complexity reduction is introduction of definitions to represent units of behavior that recur frequently throughout a program. Such use of definitions has long been applied in mathematics to render theorems and proofs more understandable. For example, although it is possible to do set theory using only "epsilon" (set membership) as the sole non-logical symbol, in practice it is impossible to express even the axioms of Zermelo-Fraenkel set theory in this manner without a complete loss of understandability. Facilities for specifying and integrating definitions into the process will be an important capability for automated function extraction.

Insights in these research areas will permit function extractor development as a key enabler for the secure systems of the future. In fact, it is difficult to imagine how security goals for critical systems can be achieved without knowing what programs do in all circumstances of use. In the current state of the art, this knowledge is sporadically and imperfectly accumulated from specifications, designs, code, and test results, all potentially incomplete and incorrect. Dynamic program modifications and just-intime compositions in modern network-centric systems severely limit the value and relevance of even this hard won but static and suspect knowledge. But programs are mathematical artifacts subject to mathematical analysis. Human fallibility still exists in interpreting the analytical results, but there can be little doubt that routine availability of calculated behavior would substantially reduce vulnerabilities and malicious code in software and make intrusion and compromise more difficult and detectable. Furthermore, broader questions about system security capabilities for authentication, encryption, filtering, etc., are in large part questions about the behavior of the programs that implement these functions. And because programs are subject to dynamic change and adaptation, only automated analysis can maintain the currency and relevance of such behavior knowledge at acceptable cost and quality.

6. A Malicious Code Example

Large programs are capable of extensive behavior in mapping their inputs into outputs. The calculated behavior of these programs is often extensive, and can be usefully organized into behavior catalogs. These catalogs are repositories of program behavior expressed in graphic structures that organize behavior expressions represented in conditional concurrent assignment statements, essentially if-then rules, themselves indexed according to the predicates involved. All behavior expressions are defined in identical syntax and semantics at all levels of abstraction. Behavior catalogs thus embody a uniform, hierarchical structure that can be searched, browsed, and analyzed according to users' objectives in investigating what a program does. In miniature illustration, consider the problem of understanding the behavior of the Java financial program depicted in Figure 4, perhaps as delivered by a software vendor. The behavior catalog documented in Figure 5 was derived through manual application of a behavior calculation algorithm. It defines non-trivial behavior that would require substantial effort to derive through program reading.

The behavior catalog is expressed as a tree structure entered on the left and exited on the right. The summary box defines key properties of the program, namely, that Account Record is unchanged and AdjustRecord is updated by one of four possible cases. The definition box gives helpful intermediate terms introduced by the extractor to simplify and systematize the behavior expressions. The four possible behaviors in the caseboxes on the right are disjoint; case selection is based on evaluation of predicates in the top section of each casebox. A predicate that evaluates true results in the functional mapping defined in the bottom of its casebox. Although the operations that carry out the functional mappings are displayed in a sequential format for readability, they are concurrent, that is, all initial values of data elements on the right are simultaneously assigned as the corresponding final values of data elements on the left. Behavior expressions are procedure-free.



Figure 4. A Java Financial Program Containing Malicious Code

A scan of the cases reveals that the program is dealing with account balances that may be negative, and is advancing loans in 100.00 increments as necessary and permissible under the business rules of the bank. Inspection of case 1 shows that under the condition that acctRec.balance is not negative the program copies Account Rec into Adjust Rec and sets in default to false. Case 3 handles the situation where increments of 100.00 can be added to the account to create a positive balance without exceeding the loan maximum, and in default is likewise set to false. A programmer or financial analyst could quickly verify that cases 1 and 3 correctly carry out the banking functions desired. But cases 2 and 4 where in default is set to true reveal suspicious behavior. In case 2 where acctRec.balance is negative and adding 100.00 exceeds acctRec.loan_max, the balance field is skimmed by a penny, which is added to spec.balance. The skimming also occurs in case 4, which is executed when

the number of 100.00 advances that can be made without exceeding the maximum is insufficient to create a positive balance. The behavior signature of this malicious code is an unavoidable product of the behavior calculation. No matter how carefully concealed in program code by an attacker, malicious code will be aggregated and coalesced into defined cases of program behavior. It is also important to note that behavior calculation is independent of data naming. Malicious code cannot be disguised by changing variable names because the function extraction process will inevitably arrive at the same behavior signature, which is name independent. In addition, traceability to the malicious code is built into the extraction process.

7. The Architecture of a Function Extractor

Figure 6 depicts a notional architecture of a program function extractor. Functional semantics are defined for the control and data structures of the target language, and possibly the machine, as well as for the behavior expression forms that will represent the extracted behavior. These semantics are stored in data repositories and employed to verify the correctness of the extractor, to ensure that the calculated behavior indeed corresponds to the behavior of the program being analyzed. The extractor itself employs abstraction and simplification rules to the stepwise extraction of program functions of the control structures of the input program. The behavior calculations are provided to a graphical interface with appropriate human factors. Users need never be exposed to the underlying mathematics, but can have confidence in the extracted behavior in the knowledge that it was derived using sound mathematical methods.

8. Related Research and Future Steps

The objectives of behavior computation are to reveal the behavior signatures of vulnerabilities and malicious code as well as other security properties, and to provide a catalog of all program behavior for assessment of security properties and risks. Research efforts based on syntactic scanning of programs deal with surface features, and cannot get at the underlying semantics that reveal the function and intent of malicious code attacks. Because the functionality of a particular malicious code attack can be programmed and disbursed in any number of syntactic forms, syntactic detection can be difficult. However, all such forms will result in the same behavior signature at the semantic level, thereby reducing the deceptive power of syntactic variations.



Figure 5. Extracted Behavior Catalog Containing Malicious Code Signature

While research efforts based on code slicing permit semantic analysis bounded by particular slices, they cannot address full functional behavior that encompasses all parts of a malicious code attack that has been disbursed throughout a program. Nevertheless, important research results have emerged from syntactic and slicing analysis, as well as from decompilation, data structure discovery, reverse engineering, and predicate abstraction. These results can be applied in FX development.

Static analysis of programs has a long and successful history of application in computer science. It has been used to help determine the run-time types of variables in dynamically typed languages, as well as to determine many other useful run-time properties of programs. The theory of static analysis is rich and deep, and has been extensively generalized using mathematical lattice theory [1]. Recent application of static analysis to the problem of identifying data aliasing [12] has proven to be a powerful tool in program understanding. However, it is important to note that the objective of function extraction goes well beyond the scope of static analysis. FX considers the complete behavior of a program, whereas static analysis focuses only on certain aspects of a program being analyzed. For example, static analysis typically assumes that either path of an alternation control structure can be executed, and presents only that information that holds for both the then-part and the else-part. Function extraction considers the then-part and the else-part separately, and determines which path will be applicable using conditional trace table analysis. Nevertheless, static analysis may be useful as a pre-processing step, particularly for languages with dynamic typing, including inheritance in object oriented languages.

Function extraction is very close in spirit to symbolic evaluation. In particular, substitution of symbolic results through composition of program statements is at the foundation of trace table analysis. Like McCarthy [9], we share a deep appreciation for the application of recursiontheoretic results to the problem of expressing program behavior symbolically. Yet in addition to this respect for recursion theory, behavior extraction also draws significant benefit from the experience of decades of human application of functional verification, especially within the context of Cleanroom Software Engineering [14]. By combining recursion theory with correctness verification and the increasing effectiveness of automation, behavior extraction has potential to impact how programming will be done in the future.



Figure 6. Function Extractor Architecture

The FX extraction language is similar in appearance to functional programming. For example, the recursive expression used to represent unsimplifiable loop behavior is similar to the "let" statement in ML. The algebra of functions at the heart of FX is also a fundamental feature of Backus' FP language; however in this instance the algebra of functions in program verification preceded FP. The important difference between the extraction language and functional programming languages is that the latter are constrained to be executable. The extraction language includes full first order logic, and thus cannot always be executed. By removing the executability constraint, the extraction language can focus on representing behavior in an easily understood manner.

The CERT Research Center is initiating a project to develop FX technology and an engineering prototype.

9. Acknowledgements

It is a pleasure to acknowledge valuable feedback and suggestions on function extraction technology and

engineering provided by Robin Abello, Alan Hevner, Tom Longstaff, and John McHugh, as well as from the referees. Special appreciation is due to Robin Abello for his work on an FX engineering pre-prototype.

10. References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison Wesley, Reading, MA, 1986.

[2] M. Davis, *Computability and Unsolvability*, Dover Publications, New York, 1982.

[3] P. Hausler, M. Pleszkoch, R. Linger, and A. Hevner, "Using Function Abstraction to Understand Program Behavior." *IEEE Software*, 7, 1, IEEE Computer Society Press, Los Alimitos CA, January, 1990.

[4] A. Hevner, R. Linger, A. Sobel, and G. Walton, "Specifying Large-Scale, Adaptive Systems with Flow-Service-Quality (FSQ) Objects," *Proceedings of the 10th OOPSLA Workshop on Behavioral Semantics*, Tampa, FL, October, 2001, ACM Press, New York, 2001.

[5] A. Hevner, R. Linger, A. Sobel, and G. Walton, "The Flow-Service-Quality Framework: Unified Engineering for Large-Scale, Adaptive Systems," *Proceedings of the 35th Annual Hawaii International Conference on System Sciences,* Hawaii, January 7-10, 2002, IEEE Computer Society Press, Los Alamitos, CA, 2002.

[6] D. Hoffman and D. Weiss, (eds.), *Software Fundamentals: Collected Papers by David L. Parnas*, Addison Wesley, Upper Saddle River, NJ, 2001.

[7] R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice*, Addison Wesley, Reading, MA, 1979.

[8] R. Linger, M. Pleszkoch, G. Walton, and A. Hevner, *Flow-Service-Quality Engineering: Foundations for Network System Analysis and Development, CMU/SEI-2002-TN-01, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002.*

[9] J. McCarthy, "A Basis for a Mathematical Theory of Computation," *Computer Programming and Formal Systems*, (P. Braffort and D. Hirschberg, eds.), North-Holland, Amsterdam, 1963.

[10] H. Mills, R. Linger, and A. Hevner, *Principles of Information System Analysis and Design*, Academic Press, San Diego, CA, 1986.

[11] H. Mills and R. Linger, "Cleanroom Software Engineering," *Encyclopedia of Software Engineering, 2nd ed.*, (J. Marciniak, ed.), John Wiley & Sons, New York, 2002.

[12] R. O'Callahan, *Generalized Aliasing as a Basis for Program Analysis Tools*, (Ph.D. Dissertation CMU-CS-01-124), Carnegie Mellon University, Pittsburgh, PA, Nov. 2000.

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

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