The Impact of Function Extraction Technology on Next-Generation Software Engineering

Alan R. Hevner, Software Engineering Institute and University of South Florida
Richard C. Linger, Software Engineering Institute
Rosann W. Collins, University of South Florida
Mark G. Pleszkoch, Software Engineering Institute
Stacy J. Prowell, Software Engineering Institute
Gwendolyn H. Walton, Software Engineering Institute

July 2005

TECHNICAL REPORT
CMU/SEI-2005-TR-015
ESC-TR-2005-015

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

The Impact of Function Extraction Technology on Next-Generation Software Engineering

CMU/SEI-2005-TR-015
ESC-TR-2005-015

Alan R. Hevner, Software Engineering Institute and University of South Florida
Richard C. Linger, Software Engineering Institute
Rosann W. Collins, University of South Florida
Mark G. Pleszkoch, Software Engineering Institute
Stacy J. Prowell, Software Engineering Institute
Gwendolyn H. Walton, Software Engineering Institute

July 2005

Networked Systems Survivability Program

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Acknowledgments

We gratefully acknowledge the contributions of Casey K. Fung to this report. We also appreciate the support for this study provided by the Software Engineering Institute (SEI) Independent Research and Development (IRAD) program.

Executive Summary

The task of understanding program behavior today is an error-prone, resource-intensive process carried out in human time scale, primarily through program reading and analysis. Yet fast and precise understanding of program behavior is essential, not only for discovering errors and vulnerabilities, but also for improving software specification, architecture, design, implementation, and maintenance artifacts and the development processes that produce them. Large and complex software systems are hard to understand because they contain an immense number of execution paths, any of which may contain errors or security exposures. Faced with massive sets of executions, developers often achieve no more than a general understanding of specified and unspecified (malicious or simply unintended) system behaviors.

This technology gap in program understanding lies at the heart of many persistent problems in software and systems engineering, and it is a major cause of security exposures and failures.

Although this situation has seemed inevitable in the past, it should not be so in the future. Function-theoretic mathematical foundations of software illuminate a challenging but feasible strategy to develop automated tools to address the problem of understanding program behavior. The objective of this Function Extraction (FX) technology is to help move from an uncertain understanding of program behavior derived in human time scale (days) to a precise understanding automatically computed in computer time scale (seconds).

The SEI CERT® organization is conducting fundamental research and development on FX technology. The first FX application is the development of the Function Extraction for Malicious Code (FX/MC) system. FX/MC will compute the behavior of malicious code expressed in Assembler Language to permit analysts to develop effective countermeasures. FX technology, however, can be applied to virtually any language environment, and it has the potential to impact many aspects of the software development life cycle as well as the entire field of software engineering. The goal of this study is to understand these impacts and chart a course for maximizing the value of FX technology for SEI sponsors.

This report summarizes FX research and development and investigates the impact of FX on software engineering. Data collected from active software professionals through a survey instrument provided objective and informed guidance on high-leverage paths for future FX initiatives. The report concludes with seven key survey findings for the future direction of FX research and development:

® CERT is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

1. Complete development of the FX prototype for Assembler Language programs.
2. Create FX automation for correctness verification of programs.
3. Create FX automation for high-level programming environments starting with Java.
4. Perform research on semantics of software specification and architecture for FX automation.
5. Perform research on human/computer interfaces for FX automation.
6. Perform experimentation with FX technology to evaluate its impact.
7. Perform research on the semantics of software quality attributes for FX automation.

Abstract

Currently, software engineers lack practical means to determine the full functional behavior of complex programs. This gap in intellectual control is the source of many long-standing and intractable problems in security, software, and systems engineering. Function Extraction (FX) technology is directed to automated computation of full program behavior. FX is based on function-theoretic mathematical foundations of software that illuminate algorithmic methods for behavior computation. FX holds promise to replace resource-intensive, error-prone analysis of program behavior in human time scale with fast and correct analysis in computer time scale. The CERT® organization of the Software Engineering Institute is conducting research and development in FX technology and is developing a Function Extraction for Malicious Code system to rapidly determine the behavior of malicious code expressed in Assembler Language. FX technology has the potential for transformational impact across the software engineering life cycle, from specification and design to implementation, testing, and evolution. This study investigates these impacts and, based on a survey of software professionals, defines a strategy for FX evolution that addresses high-leverage opportunities first. FX is an initial step in developing next-generation software engineering as a computational discipline.

1 Next-Generation Software Engineering

Traditional engineering disciplines depend on rigorous methods to evaluate the expressions (equations, for example) that represent and manipulate their subject matter. Yet current-generation 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 unlimited resources, no programmer or analyst can say for sure what the behavior of a sizable program is in all circumstances of use. This sobering reality lies at the heart of many problems in software. The result of this technology gap is deployment of systems containing unknown errors, vulnerabilities, and malicious code. Systems at the heart of the nation's infrastructure and defense are especially vulnerable. The risks are substantial for acquisition organizations that lack means to validate the full behavior of delivered systems, and offshore development of software further compounds the problem for homeland security. Even when software systems are developed onshore, they remain dependent on system and application libraries that have often been developed offshore.

1.1 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, no one was able to produce the required sums despite the obvious interest in the results. The census had been recorded in Roman numerals and no one knew how to add up so many numbers in that notational system. No amount of trying harder and being careful would suffice; the best minds of the day were overwhelmed by the complexity of the task. Yet if the census had been recorded in decimal arithmetic and place notation, a few school children could have produced the required sums in short order.

There is a lesson here for the problems of present-day software engineering; technology can either add complexity to block human capabilities or avoid complexity to augment human capabilities for achieving extraordinary results. In this case, Roman arithmetic adds complexity because it does not scale to large problems. Decimal arithmetic avoids complexity because it is scale-free; large problems simply require more of the same operations used to solve 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.

1.2 A New Science for Computational Software Engineering

The future of software system development faces two seemingly intractable 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 the intricate relationships among them that quickly exceed human understanding. No other engineering discipline requires its practitioners to remember and reason about so many details. As noted, developers today have no effective means to determine the full functional and non-functional behavior of programs written by themselves or others. Furthermore, no testing process, no matter how thorough, can validate the full behavior of programs in all circumstances of use. The inevitable result is that software systems are often fielded with unforeseen errors and vulnerabilities that no amount of trying harder can prevent. The problem of cost is closely related to complexity. Battalions of developers and programmers require a great deal of time to develop today's large software systems because individuals are complexity-limited in present technologies and coordination-limited in present organizations.

Evidence suggests that software engineering is reaching the limits of technologies developed in the first 50 years of computing. New technologies are required for the next 50 years that will enable more efficient and effective development of software systems than is possible with current-generation technologies. Manual methods of software engineering must be replaced by computational automation that will transform software engineering into a true computational engineering discipline. Other engineering disciplines have made this transformation to their everlasting benefit. Computational models of subject matter dominate mature engineering disciplines, such as electrical and aeronautical engineering. Analogous computational models for software system analysis and development are now emerging. While much of the focus of the first 50 years of computing was on correct syntax-directed computation of details for computer execution, the focus of the next 50 years can shift to semantics-directed computation of correct abstractions for human understanding and manipulation. An opportunity exists for a research and development program to exploit a new generation of scale-free computational models for fast and reliable manipulation of software artifacts, based not only on processing their syntactic expressions but also on processing their semantic meanings.

1.3 Understanding Software Behavior with Function Extraction

A necessary first step in building a foundation for next-generation software engineering is to investigate the theory and technology for understanding program behavior. Full knowledge of software behavior is essential for fast development of correct programs. Lacking better technology, behavior discovery today is a haphazard and imprecise drain on resources carried out by program reading and analysis with inevitable human fallibility. We believe that a key enabling capability for next-generation software engineering is the transformation of program behavior discovery into a precise, automated calculation. An emerging technology termed

Function Extraction (FX) holds promise to make this next-generation capability a reality. The objective of FX technology is routine, automated calculation of the full functional behavior of programs. The semantics of program behavior revealed by FX methods directly address the Department of Defense (DoD) challenges of *determining expected properties of software systems before they are built and confirming their as-built properties*, and *dramatically decreasing the amount of effort required for implementing new software-intensive systems*.

The SEI CERT organization is performing fundamental research and development on FX theory and technology. An automated Function Extraction for Malicious Code (FX/MC) system is being developed to compute the behavior of malicious code expressed in an Assembler Language environment in order to develop effective countermeasures. FX technology, however, can be applied to virtually any language environment and has the potential to impact many aspects of the software development life cycle as well as the entire field of software engineering. This report summarizes ongoing research and development on FX technologies and investigates the impact of FX on software development projects and the software engineering profession. Survey data collected from active software professionals provides informed guidance on high-leverage paths for future FX initiatives. Specific areas of fundamental research are highlighted as essential elements for FX evolution.

2 Function Extraction Theory and Technology

The study of Function Extraction at the SEI began in 2004 in the CERT organization, resulting in the publication of a paper detailing the technology and its potential [Pleszkoch 04] and the development of a proof-of-concept prototype. This work led to sponsorship of a project to develop the Function Extraction for Malicious Code (FX/MC) system, which is currently underway.

2.1 The Idea of Function Extraction

Function Extraction deals with the semantics of software behavior. All levels of abstraction in the development of software systems deal with behavioral semantics, from low-level machine language operations to high-level system capabilities. As software systems are developed and evolve over time, semantic content is continuously created, intentionally or unintentionally, correct or incorrect. Effective development and evolution of a system depends on how well its behavior is understood by its developers. The complexity and quantity of semantic information can overwhelm developers, leading to loss of intellectual control. This loss of semantic understanding occurs for many reasons at all levels of a system. Table 1 illustrates examples of the creation and inevitable loss of behavioral semantics information, from individual chips to entire information systems.

Table 1: Creation and Loss of Semantic Information in Software Development

Level	Creation and Loss of Semantic Knowledge
Processors	**Creation:** Engineers create the behavioral semantics of chip operations by combining circuits. **Loss:** errors and ambiguities in processor manuals
Languages	**Creation:** Designers create the behavioral semantics of language instructions by combining chip operations. **Loss:** errors and ambiguities in language manuals; compilers define semantics
Components	**Creation:** Programmers create the behavioral semantics of components by combining language instructions. **Loss:** full functional behavior of components not documented
Applications	**Creation:** Programmers create the behavioral semantics of applications by combining components. **Loss:** “Bob knows the application, but he’s retiring.”
Systems	**Creation:** Engineers create the behavioral semantics of systems by combining applications. **Loss:** Systems ‘go natural’ from accumulated knowledge loss.

The ultimate goal of Function Extraction is to calculate full semantic behavior at all levels of system abstraction, from specification to design to implementation. This goal can be achieved by automating the computation and composition of behaviors in the languages employed to express such artifacts. These languages, whatever their level of abstraction, embody definitions of the behavioral semantics of their language structures and rules of structure combination. These semantics can be captured and employed for Function Extraction as shown in Figure 1.

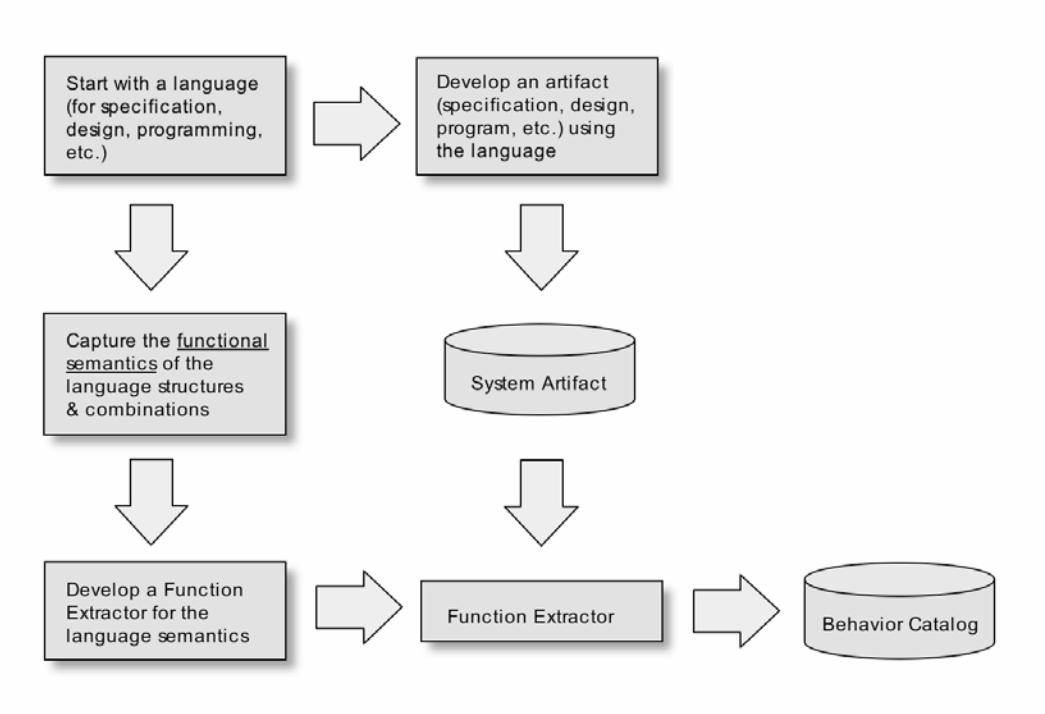

Figure 1: The Basic Concept of Function Extraction

The Function Extraction process at any system level begins with a well-defined language whose semantics can be captured in terms of the functions of language structures and the rules that govern their combination. An automated Function Extractor can then be developed for the language. Any system artifact written in that language can then be submitted to the Function Extractor and a behavior catalog produced containing all the behavior defined by the artifact.

The foundations of Function Extraction have been developed through a process of research and development that will continue into the future. The following sections summarize this research stream.

2.2 Fundamentals of Program Behavior Calculation

The function-theoretic model of software [Hausler 90, Hevner 02, Hoffman 01, Linger 79, McCarthy 63, Mills 86, Mills 02, Pleszkoch 90, Prowell 99] treats programs as rules for

mathematical functions or relations. The purpose of automated behavior calculation is to extract the full functional behavior of programs, that is, how programs transform inputs into outputs in every circumstance and present the behaviors to users as precise as-built specifications in procedure-free form for analysis. In today's technology, the totality of program behavior is difficult to understand because it is distributed across an enormous number of possible execution paths. Testing selects paths from this set and so cannot reveal full behavior.

The fundamental insight in Function Extraction technology is the realization that, while sizable programs contain a virtually infinite number of execution paths, they are constructed of a finite number of nested and sequenced control structures, each of which makes a finite contribution to overall behavior. These structures correspond to mathematical functions or relations, that is, mappings from inputs to outputs. The functional mappings can be automatically extracted in a stepwise process that traverses the finite control structure hierarchy. At each step, details of local code and data are abstracted out, while their net effects are preserved and propagated in the extracted behavior. While no general theory for loop abstraction can exist, use of recursive expressions and patterns for loops provides an engineering solution. The mathematical foundations for function-theoretic behavior calculation are currently being applied to the specialized problem malicious code analysis in the Function Extraction for Malicious Code project. An opportunity now exists to explore the full effect that this technology can have on the broader software engineering life cycle.

In more detail, function-theoretic foundations prescribe procedure-free equations that represent net effects on data of common 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) can be given by the following equation:

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

where square brackets denote the program function 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.

The program function of an alternation control structure (P: if q then g else h endif) can be given by the following equation:

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

where | is the "or" symbol. That is, the program function is given by a case analysis of the true and false branches, with the possibility of abstracting them to a single case.

The program function of a terminating iteration control structure (P: while q do g enddo) can be expressed as

> f = [P] = [while q do g enddo] = [if q then g; while q do g enddo endif] = [if q then g; f endif]

and f must therefore satisfy

> f = ([q] = true → [f] o [g] | [q]= false → I)

Because no general theory for iteration abstraction can exist, engineering solutions must be developed to recognize patterns of iteration behavior.

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. This process propagates and preserves the net effect 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.

2.3 A Function Extraction Example

In notional illustration of the stepwise behavior extraction process, consider the miniature program on the left in Figure 2 and the question of what it does. The program takes as input and produces as output a queue of integers named Q, and it defines local queues of integers named odds and evens and a local integer variable named x (<> stands for not equal, || for concatenation). The stepwise behavior calculation process is depicted in the series of displays progressing to the right. 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 assignments. Next, the three whiledo structures, now leaf nodes in the remaining hierarchy, can likewise be abstracted to conditional rules and assignments. 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. 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.

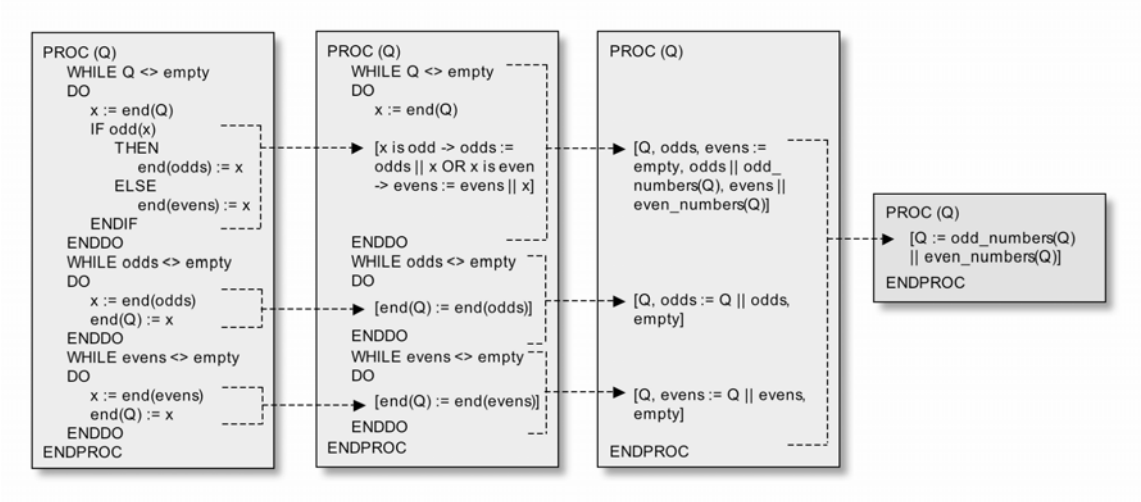

Figure 2: A Function Extraction Example

This 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 definition eliminates intermediate expressions at successive levels of abstraction. As noted above, programs can exhibit an enormous number of execution paths but are composed 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, the extracted behavior on the right side of the figure 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 precise 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, limits complexity in behavior extraction of large programs.

2.4 Automating Function Extraction

The mathematical foundations of FX theory provide an opportunity for development of automated tools to support human understanding of program behavior. Figure 3 depicts the general architecture of an automated Function Extractor. Functional semantics are defined for the control and data structures of the target language and their rules of combination, as well as for the forms of the behavior expressions 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 abstracted. The extractor itself employs abstraction and simplification rules to the stepwise extraction of program functions of the input program's control structures. The behavior calculations are provided to a graphical interface to create presentation formats with appropriate human factors. Users need never be exposed to the underlying mathematics, but they can have confidence in the abstracted behavior based on the knowledge that it was derived with sound mathematical methods.

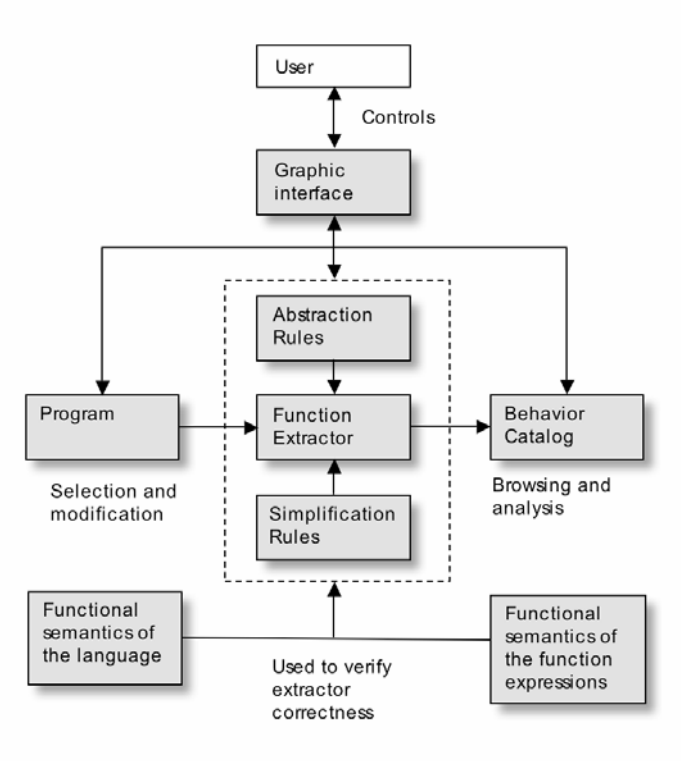

Figure 3: Function Extractor Architecture

The behavior catalogs that are produced by Function Extractors exhibit useful properties for human understanding. Consider the example Java program of Figure 4, which advances loans in $100 increments to bank accounts with negative balances, and its behavior catalog depicted in Figure 5, which was derived through manual application of extraction algorithms.

```
public class AccountRecord {
   public int acct_num;
   public double balance;
   public int loan_out;
   public int loan_max;
} // end of AccountRecord

public class AdjustRecord
extends AccountRecord {
   public bool default;
} // end of AdjustRecord

public static AdjustRecord classify_account
(AccountRecord acctRec) {
   AdjustRecord adjustRec = new AdjustRecord();
   adjustRec.acct_num = acctRec.acct_num;
   adjustRec.balance = acctRec.balance;
   adjustRec.loan_out = acctRec.loan_out;
   adjustRec.loan_max = acctRec.loan_max;
      while ((adjustRec.balance < 0.00) &&
         (adjustRec.loan_out + 100) <= adjustRec.loan_max))
   {
      adjustRec.loan_out = adjustRec.loan_out + 100;
      adjustRec.balance = adjustRec.balance + 100.00;
   }
  adjustRec.default = (adjRec.balance < 0.00);
  return adjustRec;
}
```

Figure 4: An Example Java Program

The behavior catalog shows three cases of behavior that can be applied to each record examined. The cases are uniformly defined in terms of the fundamental structure of behavior expression, namely, conditional current assignment statements. Each case is expressed as a condition, which, if true, results in concurrent assignment of the values on the right of the assignment statements to the data items on the left. That is, these assignments are procedure free and occur all at once; they represent the net functional effect of the program from input to output for each case. These cases are disjoint; only one case is applied to each record. Expressing behavior in disjoint form is extremely important in localizing human reasoning; each case can be understood in isolation with the knowledge that no side effects are present.

Because the program behavior is coalesced and aggregated into these compact forms, it is straightforward, for example, for an analyst to quickly determine whether or not the three cases correctly implement the bank's business rules that define policies for advancing loans to accounts with negative balances.

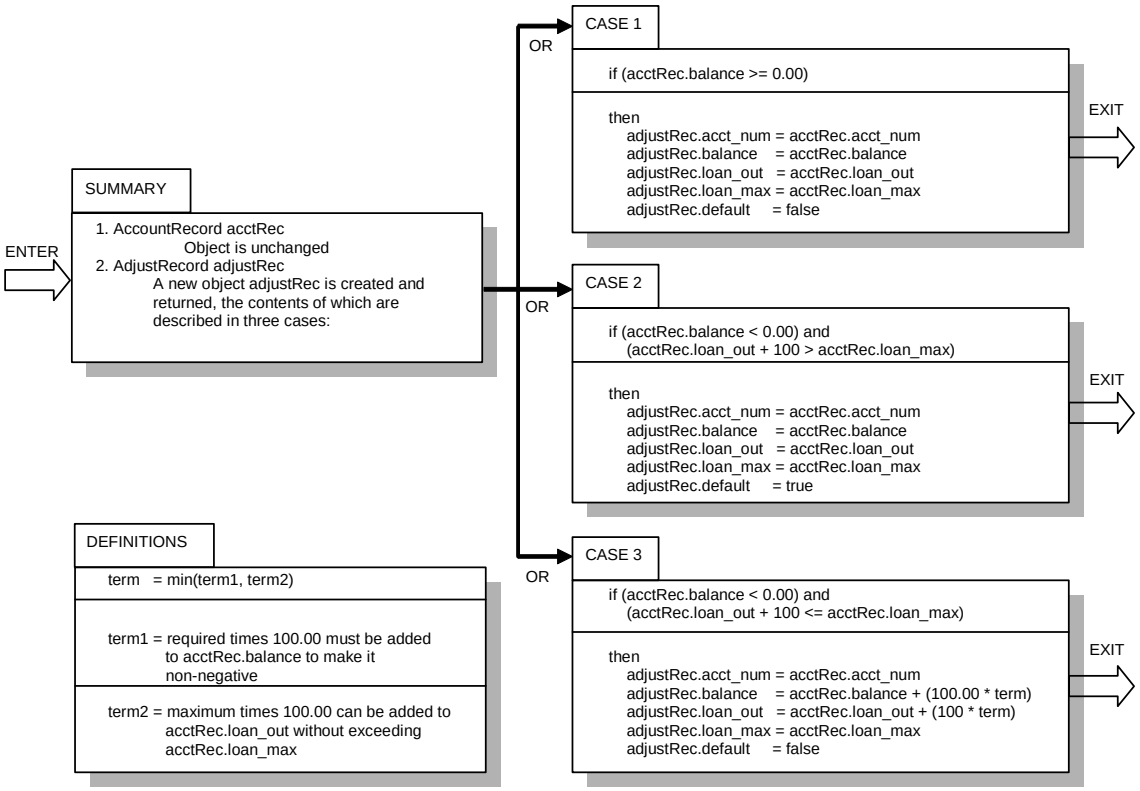

Figure 5: The Behavior Catalog of the Example Program

2.5 Function Extraction and Correctness Verification

Function Extraction and correctness verification are closely related. The function-theoretic verification process requires determining the actual functional behavior of a program and comparing it to the intended function for equivalence (or not). The intended function can be furnished by the programmer, and the actual function can be derived by Function Extraction. All that remains is to compare the actual and intended functions.

In illustration of this process, consider the miniature program of Figure 6, a sequence of three assignments operating on small integer variables x and y. The programmer has attached a comment (in square brackets) to the do keyword that defines the intended function of the program as a concurrent assignment, namely x, y := y, x - y. A Function Extraction is performed on the program, in this case carried out by a simple composition of the effects of each assignment and a derivation of final values for x and y in terms of initial values. This process reveals that the program exchanges the values of x and y, that is, x3, y3 := y0, x0, or simply x, y := y, x. It is then a simple matter to compare this actual behavior to the intended behavior and determine that the program is indeed incorrect.

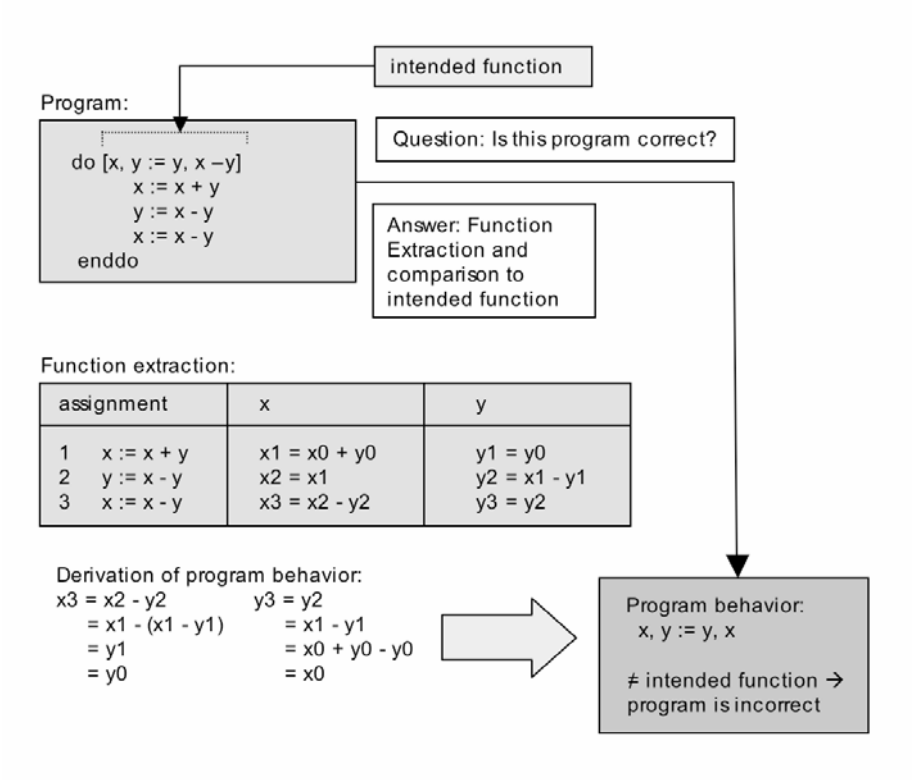

Figure 6: Correctness Verification Through Function Extraction

3 FX Impacts

To better understand the potential impact of FX technology, we begin by assessing possible applications for FX within the software development life cycle over a future 10-year time frame. We then extend the discussion to FX impacts on various programming language environments, software technologies (e.g., agile methods, Web services) and the broader field of software engineering (e.g., education, workforce).

3.1 FX Impacts on Software Development Life-Cycle Activities

Automated behavior calculation has potential for widespread use and transformational impact on the software development life cycle. Improvements are possible in developer productivity and in the quality of the software development artifacts produced in each of the life-cycle activities. However, it is important to evaluate where FX will have the greatest impact in the software life cycle in order to focus research efforts and resources for maximum leverage. Here we briefly discuss the anticipated impacts of FX in the following activities.

3.1.1 System Specification

Software system requirements are typically represented in natural language (e.g., structured English) to support the essential dialog between system stakeholders (e.g., owners, users, managers) and developers as a system is being defined. The inherent difficulties of capturing the semantics of natural languages make FX technologies ill-suited for initial elicitation and analysis of system requirements. However, translation of requirements to more precise specification languages will allow FX to support the activities of system requirements engineering effectively. The semantics of a specification language can be mapped into a Specification Behavior Extractor and used to extract and analyze the behavior defined by system specifications in a Specification Behavior Catalog as shown in Figure 7. Such analysis can be augmented and extended by interactive Behavior Catalog Analyzers that parse, organize, search, and produce subsets of behavior definitions to assist human understanding.

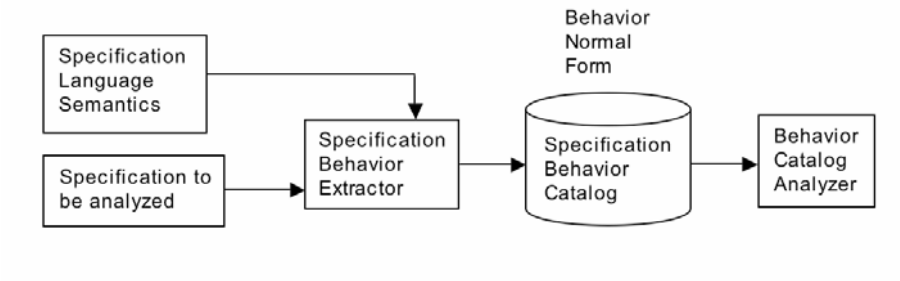

Figure 7: FX Use for System Specification

In this context, a related application of FX technology involves reengineering existing software components to capture their as-built specifications for use in new or revitalized system specifications.

3.1.2 System Architecture

The future of software system development will be increasingly architecture-centric. The growing importance of component-based development and service-oriented architectures points to system architectures as the key blueprints for implementation, integration, and testing. Research on representation languages for system architectures is active; however, no single architecture language is in wide use. System architectures are largely represented by informal diagrams and pictures. Thus, the impact of FX technology on system architecture will remain limited until the semantics of system architecture becomes better understood and represented. Figure 8 demonstrates how FX can support the understanding of architecture behavior. In this case, an opportunity exists to develop Behavior Catalog Analyzers that compare specified behavior to the behavior of architectural representations for conformance.

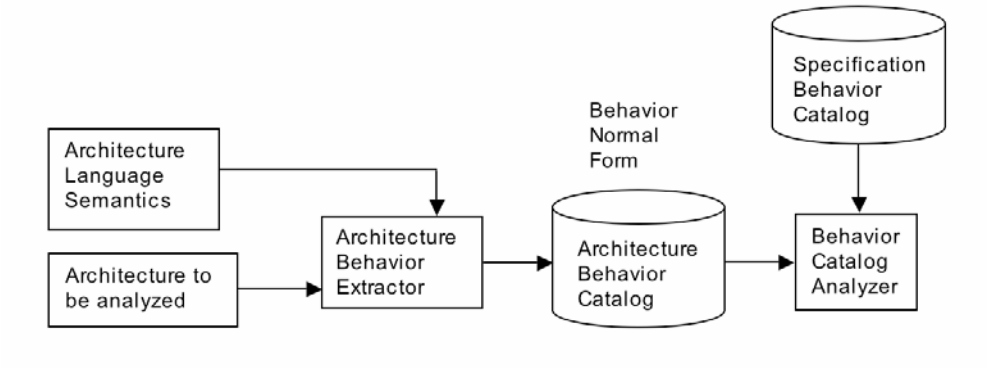

Figure 8: FX Use for System Architecture

3.1.3 Component Design

System architectures typically define software components as units of functionality. The functionality defined by components can be provisioned in several ways—by purchase of

commercial components (e.g., commercial off-the-shelf [COTS] products), use of online services, or in-house implementation. In all cases, the development organization must have full understanding of components to ensure that their behaviors are necessary and sufficient and that no malicious or undesirable behaviors are present.

Formal design languages, including both text and graphic languages, can provide well-defined semantics for use in FX automation. As depicted in Figure 9, the behavior of components expressed in structured form can be extracted from design representations and analyzed for correctness and completeness based on a comparison with system specifications and architectures. While rigorous design languages are not widely used throughout the software development industry, movements to require formal design of safety-critical systems are evident. FX technology will help developers demonstrate compliance to such requirements in the future. In addition, the evolution of popular modeling languages such as Unified Modeling Language (UML) may eventually result in sufficient semantic precision to permit FX automation to be developed.

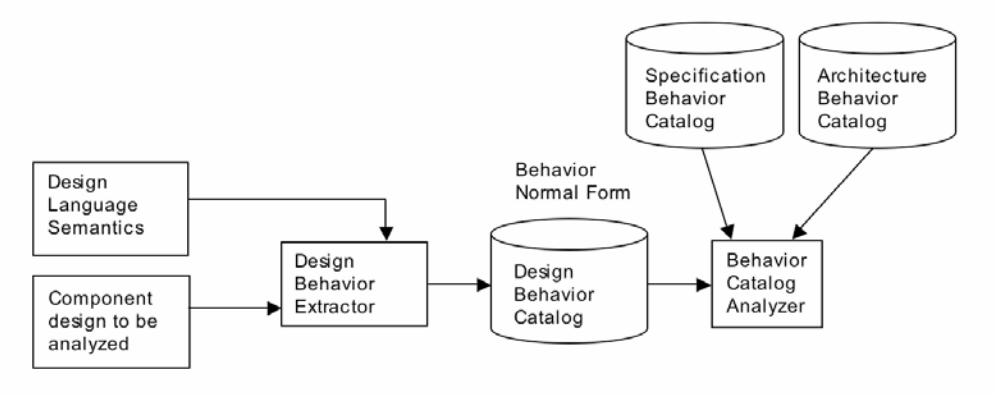

Figure 9: FX Use for Component Design

3.1.4 Component Evaluation and Selection

Components that are acquired from external vendors or even from internal reuse repositories present major challenges to developers who must understand their behavior. FX automation can provide a solution. As shown in Figure 10, a Function Extractor based on the semantics of the component's programming language can accept an unknown component and produce a complete behavior catalog. Note that this analysis is preceded by a Structure Transformer that maps the input component logic into structured form to permit a stepwise Function Extraction process. Automated program structuring is defined by the constructive proof of the Structure Theorem [Prowell 99]. The resulting behavior catalog can then be analyzed and compared to its component design catalog, as well as system specification and architecture catalogs, if available. By evaluating several components in this manner, developers can create a basis for the best selection to meet requirements.

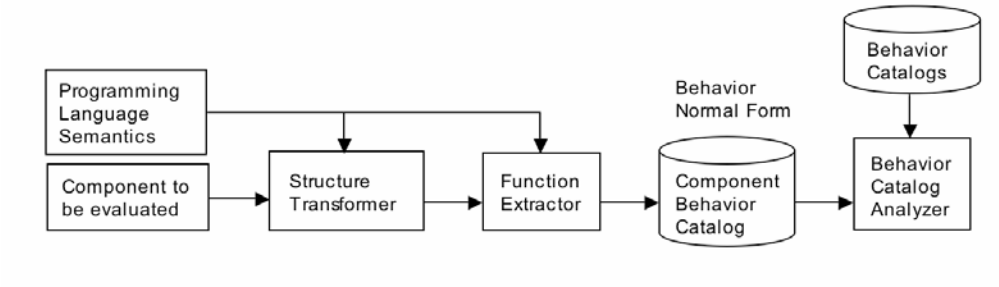

Figure 10: FX Use for Component Evaluation and Selection

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

- *legacy programs* – A developer submits a legacy program to an FX system to understand its behavior in order to integrate it with newly developed components.
- *COTS products* – A systems engineer requests a product behavior catalog from a COTS vendor to evaluate for planned use in a new system.
- *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 the full set of service behaviors. Note that the provider need not expose any proprietary code to the service user, only the service behaviors.

3.1.5 Component Implementation

When a decision is made to develop a required component from scratch, FX automation can play an important role during the evolving implementation. As each set of required functions is developed, a software engineer can work interactively with an FX system to determine if the evolving implementation indeed provides the set of functions intended. In this role, an FX system is employed as shown in Figure 10. 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.

3.1.6 Component Correctness Verification

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. As noted above, 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 extract the behavior of each control structure and compare it to the corresponding intended function to

determine whether or not it is correct. This process is depicted in Figure 11, where the FX capability is embedded within a Correctness Verifier.

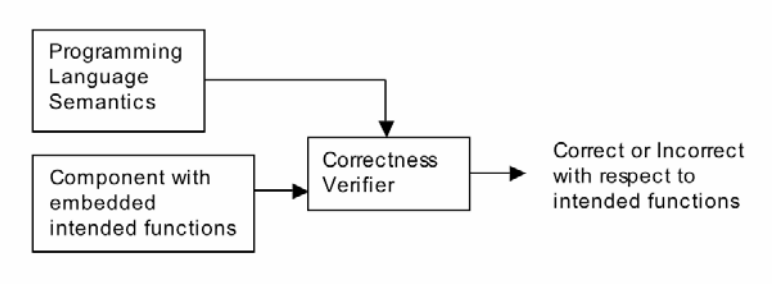

Figure 11: FX Adaptation for Correctness Verification

3.1.7 System Integration

Function Extractors are essentially generalized composition engines, and they can also play a role in the integration of software components as guided by a system architecture. Based on the behavior catalog 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 catalog. The architecture specifies intended and allowable usage patterns (i.e., control flows and data flows) among the integrated components. Figure 12 illustrates this process, where FX technology is embedded within a Component Composition Generator.

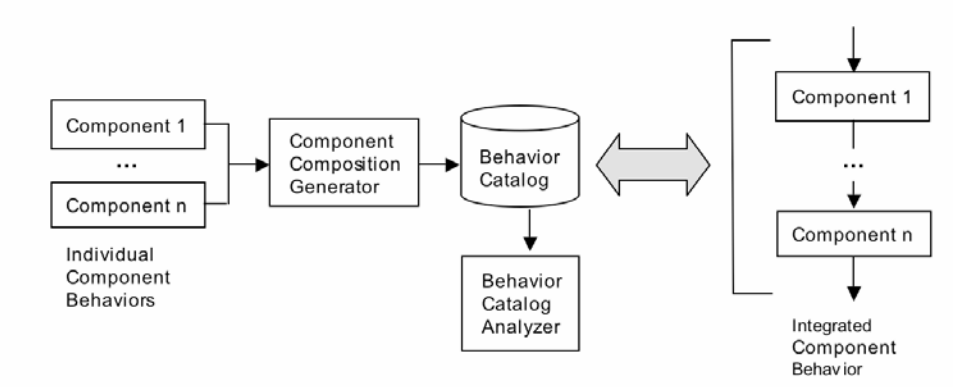

Figure 12: FX Use for System Integration

3.1.8 System Testing

With the advent of FX technology, an opportunity exists for subsystem, system, and customer acceptance testing to shift from defect detection to certification of fitness for use. Subsystems and entire systems can be processed by FX automation, and resulting behavior catalogs compared with specifications and analyzed by stakeholders. A reduced set of test scenarios can 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, testing of assump-

tions regarding environmental conditions, hardware platforms, and usage patterns must be carried out as well.

Additional testing effort can be devoted to validating the level of quality attributes provided by the system. For example, system testing for the qualities of performance, security, privacy, reliability, survivability, and maintainability, to mention a few, will become a greater focus of system testing.

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.

3.1.9 System Maintenance and Evolution

It is generally accepted that approximately 80% 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 can support maintenance and evolution activities while providing opportunities for cost savings and quality improvements. Figure 13 illustrates the use of FX automation for system maintenance and evolution.

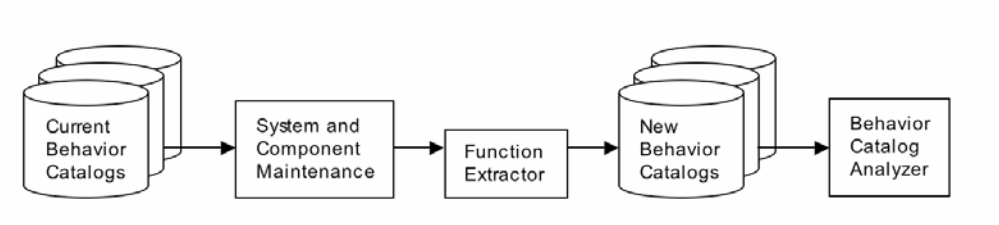

Figure 13: FX Use in System Maintenance and Evolution

The key to system maintenance with FX technology is keeping behavior catalogs 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 catalog. 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 catalogs provide a formal baseline against which all changes can be compared. New or modified behaviors can be specified initially in specification behavior catalogs and traced through component design and implementation catalogs. Thus, developers can determine where and how to make required changes in system specifications, component designs, and code. Once code changes are made, FX automation can 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 help 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 breaches have occurred since the previous FX analysis.

3.2 FX Impacts on Programming Language Environments

FX automation requires a considerable one-time effort to build a Function Extractor for each language (see Figures 1 and 3). Thus, it will be important to focus resources on developing Function Extractors for programming languages and environments that will have the greatest impact for industrial software development.

3.2.1 Assembler Languages

CERT has selected an Assembler Language environment as the initial target for development of a Function Extractor. The goal of the Function Extraction for Malicious Code system is to generate behavior catalogs for programs written in Intel Assembler Language. Fast and precise analysis of the malicious code's function will enable analysts to develop countermeasures quickly. A prototype of the FX/MC system is under development. This effort is providing knowledge and experience that will guide development of Function Extractors for other language environments.

3.2.2 Imperative Languages: COBOL and C

Many legacy systems and components have been programmed in imperative languages such as COBOL, FORTRAN, and C. FX automation for these languages would assist in maintenance and evolution of legacy code, reengineering of components, and development of new systems. FX development can deal with the lack of complete and rigorous semantics for many imperative language environments by augmenting semantic definitions as necessary based on the semantics of compiler implementations.

3.2.3 Object-Oriented Languages: C++ and Java

The past several decades have seen the growing use of object-oriented (OO) programming languages (e.g., C++ and Java) for system development. While FX automation for OO languages would be similar to imperative languages, the resulting behavior catalogs might be represented quite differently, for example, as interactions among objects rather than as operations for updating state data. Again, any uncertainties in formal semantics for OO languages can be managed in developing FX automation.

3.2.4 Rapid Development Languages: Visual Basic

Rapid development languages such as Visual Basic are in widespread use, often to create event-driven systems with extensive user interface capabilities. FX automation for these languages will require semantic definitions for input and output events and their associated functions.

3.3 FX Impact on Software Engineering Issues

The following discussion illustrates the potential impact of Function Extraction on aspects of software engineering and software development projects.

3.3.1 Component-Based Development

Mature software development organizations employ the best practices of component-based systems development. Rigorous system specifications and architectures provide the framework for provisioning referentially transparent components from a wide variety of sources. Software components that meet a required set of functional behaviors in the system architecture may be purchased commercially (COTS products), leased on a per-use basis as a service, modified from a internal legacy component, reused from an internal reuse repository, or built from scratch either internally or externally through vendor contracts.

FX automation could be used in component-based development to define the functional behaviors of all provisioned components and to match these component behaviors with the desired behaviors in the system specification. The design, evaluation, selection, implementation, verification, and composition of components would make effective use of FX as described in life-cycle activities discussed in Section 3.1.

3.3.2 Reengineering of Legacy Systems

Effective reengineering and revitalization of legacy systems requires first and foremost a complete understanding of their functional behavior. Capturing the behavior of an existing system (i.e., reverse engineering) through FX automation allows a software engineer to make modifications, additions, and deletions to the system at the fundamental level of behavior specification. The changes can then be reengineered into the legacy system through design, implementation, and testing activities (i.e., forward engineering).

3.3.3 Quality Assurance of Software

Software system quality is judged primarily by a system's conformance to its behavior specifications. Component verification through reviews and inspections can be guided by the application of FX automation to evolving designs and implementations to ensure correct behavior throughout the development process. Quality metrics could include the percentage of

specified behaviors currently verified to be available in the component and eventually in the software system.

The quality of a software system extends beyond functional correctness and conformance to specifications. Additional quality attributes to be validated include system performance, reliability, usability, survivability, and many other relevant qualities. Research is required to represent and manipulate these quality attributes in computational formats that are amenable to analysis through FX automation.

3.3.4 Component Reuse

Creation of an effective component reuse strategy for an organization requires a focus on both the production and consumption of reusable components. FX technology can support both of these aspects of reuse. Producing behavior catalogs for components can enhance their potential reuse by providing an efficient way to search for desired behaviors. For example, an automated search engine in the form of a Behavior Catalog Analyzer could apply a desired set of behaviors to create a list of available components in a reuse repository whose behavior catalogs provide full or partial matches. The search engine could also support composition analysis of multiple components and suggest the most effective set of reuse components to satisfy the greatest percentage of required behaviors.

3.3.5 Automated CASE Tools

The mathematical foundations of FX make it readily adaptable to automated computer assisted software engineering (CASE) tools. Development of automation will enable the advantages of FX to be integrated into all system development life-cycle activities that produce semantically rich artifacts. Research is required to develop effective human/computer interfaces to support human decision making. We envision integration of FX technology into existing CASE environments to provide a full set of essential software engineering capabilities.

3.3.6 Web Services

Service-oriented architectures (SOAs) focus on integration of internal and external (e.g., Web-based) services with other software components in an application system. The use of a Web service in a critical application carries risks to performance, availability, reliability, privacy, and security. The full behavior of a Web service is often unknown to its users. However, with FX automation, a user could request a service behavior catalog as a prerequisite for integrating the service into a SOA application. In this way, the propriety design and implementation of the service can remain confidential while its functional behavior can be safely exposed to users.

3.3.7 Agile Methods of Software Development

Many of the techniques of agile methods can be supported by FX automation. For example, in pair programming, the team can use behavior catalogs of system specifications and designs to develop test cases before building the system code. The pair would receive rapid feedback from a Function Extractor on whether the evolving code meets the required functional behavior. Also, the agile method of design refactoring meshes well with the FX concept of continually evolving a software system to meet a full set of required behaviors. In general, the agile philosophy of producing software artifacts and evolving them instead of following a disciplined development process from beginning to end requires a clear statement of specifications, such as is found in a behavior catalog, to be successful.

3.3.8 Distributed Computing

Distributed computing architectures, including applications of grid computing, place challenging demands on the development and dynamic execution of software systems. The distribution of application components and data to different platforms involves complex tradeoffs of performance, availability, security, and other system quality attributes. Research on extending FX technology to real-time system monitoring is needed to understand the dynamic behaviors of components and their real-time compositions in flow structure architectures.

3.3.9 System Documentation

FX automation can capture the behavior of legacy systems and document it for system users. For new system development, FX can be employed to provide behavioral documentation for the evolving system. Documentation writers can use system behavior catalogs as the basis for producing both system and user manuals.

3.4 FX Impacts on the Software Engineering Field

FX theory and technology can have an impact on a number of important areas of the software engineering field.

3.4.1 Software Engineering Education

If FX becomes an integral technology throughout the software development life cycle, education and training on FX theory and practice will be necessary. Software engineering education could be transformed at the undergraduate and graduate levels, just as computational methods have transformed education in other engineering disciplines. FX could be incorporated in beginning programming courses to show students the actual behaviors of the code they write as compared to intended behaviors. Courses within the software engineering curricula, along with accompanying textbooks and training materials, would require modification to include FX concepts and applications. In addition, new industrial skills and positions may emerge. For example, there will be a need for individuals who specialize in subject-

matter semantics definition for analysis and design of systems in specific application domains.

3.4.2 Software System Acquisition

The procurement and acquisition of software systems will likely be influenced by knowledge gained from FX systems. For example, organizational and mission objectives can be represented in specified behaviors, and candidate software systems analyzed with FX automation to determine which systems best meet behavioral objectives. Acquisition decisions can thus be based on quantifiable parameters of the number of behaviors that are satisfied or not. Industry-wide compliance with FX standard technologies will enable vendors to demonstrate the range and quality of behaviors in their systems, while providing customers with a common platform for evaluating and comparing competitors' systems.

3.4.3 Management of Software Development

FX automation can give software project managers a new set of capabilities to improve control and reduce risk. In place of costly and time-consuming reviews and inspections, managers would have the ability to submit specification, design, and implementation artifacts to FX systems to judge correctness and assess progress. Tracking the number of intended behaviors that have been successfully designed, coded, and integrated into a system provides a clear way to measure project status. The use of FX technology across projects could increase predictability of project schedules, staffing, and required resources in the development organization.

3.4.4 Software Development Teams

The structure of software development teams may be affected by the introduction of FX technology. A decision will be required on whether to train all team members on FX techniques or to designate specific individuals as FX experts. Since FX can be applied to many activities in the development life cycle, all project team roles will require some FX training. Experience in using FX on actual projects should help determine how team roles may change. Another consideration is the increased span of intellectual control afforded by FX automation, which may mean that fewer people are required to develop complex systems.

3.4.5 Software Development Organization

The software development organization and information technology (IT) staffing may change to reflect the use of FX technologies. For example, a distinct FX technology group could be formed to maintain and enhance the scope of languages and application domains supported by FX automation. As described in the section on FX theory, significant effort is required to capture the semantics of various specification, design, and programming languages. In addition, the adaptation of FX technology to different application domains (e.g., finance, tele-

communication, aerospace, defense) is an area of ongoing research. A commitment to support FX technology may require rethinking the structure of software organizations.

3.4.6 Software Development Industry

If FX becomes a way of life in software development, the industry may experience significant changes. For example, FX standards could transform component-based software development. Component and service vendors could produce documentation in FX-compatible formats to support evaluation, selection, and acquisition of components and services in an open marketplace. Outsourcing of key development activities could become more attractive with improved capabilities to communicate specifications and verify the function and quality of outsourced deliverables. Organizations could focus their resources on key competencies while outsourcing software development with confidence that FX methods will allow them to ensure compliance with quality standards. We may also see the creation of software vendors who specialize in FX products and services to support software development organizations.

3.4.7 Software Engineering Workforce

The use of FX technologies could enhance the professionalism and diversity of the software engineering workforce. Movement of FX theory into engineering practices and automation will be a clear sign of increasing maturity of the field. FX technology can support a diverse business structure of system component development, selection, and integration that cuts across all cultures and regions, and provides benefits to developers of the best competitive solutions, regardless of their origins and locations. Facility with FX-based technologies represents an advanced software engineering capability that could be retained onshore while implementation tasks migrate to offshore organizations.

3.4.8 Software Engineering Economics

Current software engineering technologies and practices incur considerable costs in terms of staffing, budget, schedule, and economic resources. Current methods could gradually be replaced as software development organizations gravitate to the predictability, quality, reduced development time, and lower cost of FX technologies. A tipping point could occur when conventionally developed software becomes comparatively uneconomical to create and use.

4 The FX Research Study

In order to inform and guide the direction of FX research and development, there is an immediate need to better understand how enterprises will employ FX technologies in their software engineering environments. Which of the impacts discussed in Section 3 matter most to software development groups?

Table 2 shows a portion of the many opportunities we face in deciding what direction to take. It lists software development activities in the rows and potential language environments in the columns. The highlighted cell is our starting point—the current effort to develop a Function Extractor for Assembler Language. To help determine the next steps of FX development, we gathered and analyzed data from potential users of FX technologies. With this information, we are better able to recommend high-payoff areas for FX research and development.

Table 2: FX Impacts – Where to Next?

	Life-Cycle Activity	Specification Automation	Architecture Automation	Assembler Automation	C Automation	C++ Automation	Java Automation	Other Languages
1	**Specification Development**	Specification Behavior Extractor Behavior Catalog Analyzer	*Not applicable*	*Not applicable*	*Not applicable*	*Not applicable*	*Not applicable*	*Not applicable*
2	**Architecture Development**	*Not applicable*	Architecture Behavior Extractor Behavior Catalog Analyzer	*Not applicable*	*Not applicable*	*Not applicable*	*Not applicable*	*Not applicable*
3	**Component Development: Evaluation & Selection and Design & Implementation**	*Not applicable*	*Not applicable*	Structure Transformer Function Extractor Behavior Catalog Analyzer	Structure Transformer Function Extractor Behavior Catalog Analyzer	Structure Transformer Function Extractor Behavior Catalog Analyzer	Structure Transformer Function Extractor Behavior Catalog Analyzer	Structure Transformer Function Extractor Behavior Catalog Analyzer
4	**Correctness Verification**	*Not applicable*	*Not applicable*	Correctness Verifier	Correctness Verifier	Correctness Verifier	Correctness Verifier	Correctness Verifier
5	**System Integration**	*Not applicable*	*Not applicable*	Component Composition Generator Behavior Catalog Analyzer	Component Composition Generator Behavior Catalog Analyzer	Component Composition Generator Behavior Catalog Analyzer	Component Composition Generator Behavior Catalog Analyzer	Component Composition Generator Behavior Catalog Analyzer
6	**System Testing**	*Not applicable*	*Not applicable*	Behavior Catalog Analyzer	Behavior Catalog Analyzer	Behavior Catalog Analyzer	Behavior Catalog Analyzer	Behavior Catalog Analyzer
7	**System Maintenance and Evolution**	*Not applicable*	*Not applicable*	Behavior Catalog Analyzer	Behavior Catalog Analyzer	Behavior Catalog Analyzer	Behavior Catalog Analyzer	Behavior Catalog Analyzer

4.1 Research Questions

To structure the research study, we posed two sets of three questions each. It is generally understood that program comprehension is a critical aspect of all software development and maintenance activities [Rajlich 02]. Prior research has found that both program and task characteristics interact to impact the nature of program comprehension [Storey 99], so it is important to develop tools with specific software engineering activities in mind. Therefore the first three research questions for this study focused on understanding the current approaches to, cost of, and impacts of program comprehension, with particular attention to how these vary by type of activity:

Research Question 1: What techniques are in current practice to understand and document program behavior?

Research Question 2: What are the typical costs of program comprehension and documentation to development?

Research Question 3: What is the relationship of program comprehension and system quality?

The second set of research questions centered on the views about the potential of FX technology from developers:

Research Question 4: In which system development activities and environments does FX technology have the potential for greatest impact?

Research Question 5: What are the potential impacts of FX technology on other software engineering technologies and issues?

Research Question 6: What are the challenges to adoption of FX technology?

4.2 Study Design

To answer these research questions, an empirical study was performed. Study participants were experienced system developers. The study questionnaire was developed by the FX research team and was pilot tested with an academic audience (professors and doctoral students in information systems at a large research university). Based on the pilot test data and open-ended feedback, the questionnaire was significantly revised. The final version of the questionnaire is provided in Appendix A. The initial questionnaire items focused on the potential of FX technology (Research Questions 4-6). When the questionnaire was revised, items to investigate the current status of program comprehension in industry (Research Questions 1-3) were added.

The questionnaire uses a combination of direct-answer questions, Likert scale ratings, rankings, and open-ended questions to solicit the data desired to answer the research questions.

Table 3 shows the relationship of research questions and goals of the study to the specific items on the questionnaire. Since we are particularly concerned with understanding the potential impact of FX on specific software engineering activities, there are three items of different question types (rating on a Likert scale, ranking, and open-ended) included to improve the reliability of answers to these questions.

Table 3: Questionnaire Items and Types

Research Questions/Goals of Study	**Questionnaire Items and Types**
Demographic data on participants	1 (direct, enter age) 2 (select from list)
Acceptance challenges for FX	11 (open-ended)
Guidance for FX research program	12 (open-ended)
Research Questions 1 – 3: Existing Software Development Environment	
Methods and tools currently used in program comprehension	3a (open-ended)
Cost in time of program comprehension (both overall and for specific software engineering [SE] activities)	3b and 4 (enter amount of time)
Impact of program comprehension	3c (enter amount of understanding) 3d (open-ended)
Research Questions 4 – 6: Potential of FX Technology	
Potential impact on SE activities	5 (Likert scale rating) 6 (ranking) 7 (open-ended)
Important programming environments for FX	8 (ranking)
Potential impact on SE technologies	9 (Likert scale rating)
Potential impact on SE areas	10 (Likert scale rating)

The FX study for this report was performed at a major Fortune 100 company with a large and sophisticated group of software developers. The session began with a detailed presentation on FX technology. This training was provided by the researchers to a roomful of software developers and remotely located individuals on a Webcast. The remote group could see the presentation slides and had two-way audio. The training presented FX technology and detailed examples of how the technology could work in development. This presentation lasted approximately 90 minutes followed by an open question-and-answer session.

After the training session, participants were requested to complete the final questionnaire on potential impacts of FX technology. Software engineers from both on-site and remote locations provided usable questionnaire data. The following section discusses the results of this study.

4.3 Study Results

4.3.1 Demographic Data

The average number of years of experience for the software engineers in the study is 23, with a range between 8 and 40 years. Most respondents reported that their primary area of experience in software engineering has been in technical work in industry, but a few respondents reported most experience in managerial work in industry and one respondent reported most software engineering experience in research.

4.3.2 Existing Software Development Environment: Research Questions 1-3

Research Question 1: What techniques are in current practice to understand and document program behavior?

Industry respondents report a wide range of models and tools currently in use that support better understanding of system behaviors. The models and tools shown in Table 4 capture parts of system information that can be used manually to build a partial mental picture of overall system behaviors. None of the listed models or tools provides a complete behavioral specification of the software system. This was a direct-answer question on the questionnaire.

Table 4: Current Program Understanding Techniques

Technique for Understanding Programs	**Percentage of Responses**
Object-oriented system models (e.g., UML)	23%
Traditional system models (e.g., data flow diagrams, engineering review diagrams)	23%
Simulation tools	15%
Statistical comparison of test and quality analysis results	15%
Reading and analyzing code	8%
Rapid application development	8%
Primitive techniques (e.g., PRINT statements)	8%

Research Question 2: What are the typical costs of program comprehension and documentation to development?

The data indicate the high cost to developers of building comprehension of program behavior by reading system development artifacts. On average, the respondents believe that developers spend over a quarter of their time (28%) reading and understanding the behaviors of system development artifacts written by themselves or others. This overall average goes up slightly

when asked about the percentage of time spent on such reading during specific software development activities. This was a direct-answer question. The data are presented in Table 5.

It should be noted that individual respondents varied widely in their assessments of the percentage of time spent on these activities, with estimates ranging between a low of 1 or 5 to a high of 80 or 90. Some of this variation in range can be attributed to one individual reporting consistently low estimates of the percentage of time spent (e.g., one person's estimates were between 1 and 10 for all activities), but most respondents clearly differentiated between activities in their estimates.

Table 5: Percentage of Developer Time Spent Understanding Behaviors

System Development Activity	**Percentage of Time Spent Understanding Behaviors**
System specification	27%
System architecture	30%
Component design	32%
Component evaluation and selection	40%
Component implementation	29%
Component correctness verification	38%
System integration	31%
System testing	38%
System maintenance and evolution	40%

Research Question 3: What is the relationship of program comprehension and system quality?

By the end of a typical software development project, the respondents reported that most, but not all, of the behaviors of the system are completely understood. The average estimate was that 84% of the system behaviors are understood upon project completion. As a consequence of this incomplete understanding of system behaviors, respondents identified several negative impacts on the quality of the system:

- reduced system performance
- incomplete and/or incorrect specifications, that lead to data errors (because invalid values are not caught or because imprecise business rules were implemented) and logic errors (because needed options were not specified)
- extensive rework when problems are discovered late in testing or in operations

As one subject expressed it, incomplete understanding of system behavior results in "surprises in expectations, surprises in interactions, but generally undesired behavior."

4.3.3 The Potential of FX Technology: Research Questions 4-6

Research Question 4: In which systems development activities and environments does FX technology have the potential for greatest impact?

The study subjects were asked to evaluate the potential positive impacts of FX in comparison to other technologies they have used to support software engineering work. They found clear positive impacts of FX in nearly all of the software development life-cycle activities. The questionnaire gathered data on FX impacts in two forms. First, a Likert-style rating produced a value from 1 to 7 where 1 is "no impact" and 7 is "very strong impact." Second, the respondent was asked to rank the nine software development activities from 1 (greatest impact) to 9 (least impact). Two ways of obtaining the same information are often used in surveys to validate the consistency of data. The average data values for the Likert-style ratings and the rankings are presented in Table 6.

Table 6: FX Impacts on System Development Activities

System Development Activity	Average Likert Impact Rating (1-Low to 7-High)	Average Impact Ranking (1-High to 9-Low)
System specification	2.3	6.2
System architecture	2.6	6.1
Component design	3.3	5.4
Component evaluation and selection	4.2	4.3
Component implementation	4.3	4.9
Component correctness verification	5.8	3.1
System integration	4.5	4.4
System testing	4.8	3.8
System maintenance and evolution	4.5	4.3

Both sets of values are presented in Figure 14. The Likert rating scale is shown at the top of the graph from lowest impact (1) to highest impact (7), and the ranking scale is shown at the bottom of the graph from lowest rank (9) to highest rank (1). This figure shows that the respondents were very consistent between their ratings and rankings. Component correctness verification was the activity for which there is the strongest potential impact (average rating of 5.8 and average ranking of 3.1). As one respondent stated, "FX's strength is in discovery of complex system behavior, which is good for correctness verification and 'debugging' production applications."

FX technologies were rated as having above-average impact on the system development activities of component evaluation and selection (average rating of 4.2 and average ranking of

4.3), component implementation (average rating of 4.3 and average ranking of 4.9), system integration (average rating of 4.5 and average ranking of 4.4), system testing (average rating of 4.8 and average ranking of 3.8), and system maintenance and evolution (average rating of 4.5 and average ranking of 4.3) activities. One respondent noted, “The tool would make testers salivate.”

System specification, system architecture, and component design were rated as activities for which there is potentially below-average potential impact. The lower ratings are reflected in this comment by a respondent:”Since specifications will probably not be documented in a rigorous semantic language representation for the foreseeable future, FX will have limited impact in those areas, growing slightly as the specification becomes more implementation specific.” However one developer commented, “The technology should be applied at the earlier stages of development life cycle. Specification level would be the best. *SEI should team up with other organizations to develop formal and syntactically precise software specification languages for this purpose* [italics added].” Thus, the lack of well-defined languages for the front-end of the system development life cycle will inhibit short-term use of FX technologies in this area. The respondents identified a major need for FX capabilities in the areas of specification, architecture, and design. However, more research is required on rigorous semantics in these activities before FX can be applied.

	FX IMPACT RATING											
	Low Impact		Minimal Impact		Below Avg. Impact		Average Impact		Above Avg. Impact		Strong Impact	Very Strong Impact
SYSTEM DEVELOPMENT ACTIVITY	1		2		3		4		5		6	7
System Specification			**2.3**									
					(6.2)							
System Architecture				**2.6**								
					(6.1)							
Component Design						**3.3**						
						(5.4)						
Component Evaluation & Selection							**4.2**					
							(4.3)					
Component Implementation							**4.3**					
							(4.9)					
Component Correctness Verification										**5.8**		
									(3.1)			
System Integration							**4.5**					
							(4.4)					
System Testing								**4.8**				
									(3.8)			
System Maintenance & Evolution							**4.5**					
							(4.3)					
	(9)	(8)		(7)		(6)	(5)	(4)		(3)		(2) (1)
	Lowest	-----	----------	---------	-----	----------	----------	-----	----------	----------	-----	--------- Highest
						FX IMPACT RANKING						

Figure 14: FX Impacts on Software Development Activities (Data from Table 6)

Assembler Language was the programming language environment rated as most important for the application of FX technology. One software developer noted that FX would be especially helpful in the Assembler Language environment in identifying malicious code and de-

vice driver verification. The Java language environment was the second most highly rated for FX application. C and C++ were also highly ranked as important environments, although one person was concerned that C and C++ might not be in wide use if it takes a long time to develop FX. Table 7 shows the average rankings for various programming language environments. The environments were ranked from 1 to 5 with 1 indicating the most important language environment for FX application.

Table 7: Programming Language Environments for FX Application

Programming Language Environment	**Average Ranking (1-High to 5-Low)**
Assembler Languages	2.0
C Language	2.8
C++ Language	3.2
Java Language	2.4
Visual Basic (VB)	4.6
COBOL Language	3.8
Other Languages (Python, UML-MDA, .Net, C#, VB.Net, SysMI)	3.0

Research Question 5: What are the potential impacts of FX technology on other software engineering technologies and issues?

This research question helps us better understand the relationships and synergies that may exist between FX technology and other software engineering technologies and issues. The data collected from the respondents are contained in Tables 8 and 9. In both cases, the tables show the average impact of FX technology on the designated technology or issue based on a Likert scale from 1 (no impact) to 7 (very strong impact). The respondents were asked to judge the impact in comparison with other technologies that they had used in their software engineering work.

Table 8: Impact of FX Technology on Software Engineering Technologies

Software Engineering Technology	**Average Rating (1-Low to 7-High)**
Component-based development	4.0
Reengineering of legacy systems	4.7
Quality assurance of software	5.9
Component reuse	5.0
Automated CASE tools	4.0
Web services	3.4

Table 8: Impact of FX Technology on Software Engineering Technologies (cont.)

Software Engineering Technology	Average Rating (1-Low to 7-High)
Agile methods of software development	3.3
Distributed computing	3.6
System documentation	4.8

As seen in Table 8, respondents rated FX technology as having the greatest potential impact on quality assurance of software (average rating of 5.9), component reuse (average rating of 5.0), system documentation (average rating of 4.8), and reengineering of legacy systems (average rating of 4.7). They rated FX technology as having a moderate impact on component-based development, automated CASE tools, and distributed computing. Little potential FX technology impact was seen on Web services and agile methods.

Respondents identified five areas of software engineering that held above average potential for FX technology impact: software engineering economics (average rating of 4.6), the software development industry (average rating of 4.5), software development teams (average rating of 4.3), the software development workforce (average rating of 4.2), and the management of software development (average rating of 4.1). It is clear from these numbers in Table 9 that most respondents felt that the introduction of effective FX technologies will have a substantial impact in many areas of software engineering.

Table 9: Impact of FX Technology on Software Engineering Issues

Software Engineering Issue	Average Rating (1-Low to 7-High)
Software engineering education	3.8
Software system acquisition	3.7
Management of software development	4.1
Software development teams	4.3
Software development organization	3.3
Software development industry	4.5
Software development workforce	4.2
Software engineering ethics	3.3
Software engineering economics	4.6

Research Question 6: What are the challenges to adoption of FX technology?

In response to open-ended questions about FX technology challenges and research directions, the respondents identified many key issues. First, they were concerned about the length of

time required to perform the needed FX research and to develop usable FX technology. One respondent noted, “This looks like a long-term research project.” Another stated, “The planning horizon is farther out than two years for a production capability—it is in the research stage.” Still others said, “This technology seems to be a long way off,” and “It appears that FX is decades out, not just years out.”

Other respondents recognized the barriers to acceptance of anything new: “The biggest problem we encounter is basic resistance to change. Once past that, a clear understanding of how FX would be used is necessary.” Reiterating that thought is this quote: “[FX] must overcome high skepticism: formal approaches and languages are regarded as academic by industry; even formal specialists claim that the halting problem is unsolvable, yet FX claims that it will know?” In addition, they are concerned about potential high learning curves and cost of the tool.

Key characteristics of FX technology that would be critical to its acceptance are listed below:

- FX technology would have to be a mature, complete tool before being accepted by industry.
  (Illustrative quote: “It would have to be a well-developed product before it was accepted. A lot of engineers don't trust what they can't “see” … there can be a lot of heel dragging when it comes to automation for design. This includes software design.”
- FX should demonstrate a positive return on investment (ROI).
- FX should be a user-friendly tool that is integrated within development environments. Thus, a high priority must be placed on human/computer interfaces in an FX tool.
- FX should be able to demonstrate benefits of alternative functions as revealed by the behavior catalog.
- FX should not be designed to a specific type of coding or test environment.

The respondents offered some specific concerns about the FX technology approach that reflected skepticism. For example, one respondent stated, “All the other software engineering technologies are also trying to make a program more understandable by modularizing it early on. Trying to modularize it based on code is kind of late and difficult.” Other comments were more specific about potential problems, such as standard data types and the impact of reuse on commonality. Other respondents were concerned about how FX technology was currently positioned, and cautioned, “It should advertise itself as a technology assist to senior software engineering practitioners rather than a technology automation solution that can function in CPU [central processing unit] time scales instead of human time scales. FX has the classic risk of overselling its capability and disillusioning its audience.”

In terms of how to proceed with the development of FX technology, it was suggested that the project should “focus on one or two areas that you can bring to market with a success story instead of selling to multiple constituencies.”

5 Recommendations of the FX Study

The objective of this section is to identify the next steps of the FX research and development program based on the industry survey data seen above. The data clearly indicate the need for the following six project goals. In addition, a seventh goal not discussed in the survey instrument is recommended by the IRAD study authors. Thus, we strongly recommend that FX technology directions be focused on well-defined milestones toward the achievement of the following goals.

5.1 Goal 1: Complete Development of the FX Prototype for Assembler Language Programs

It is important that the FX project continue with development and deployment of the FX/MC prototype. The Assembler Language environment was rated as the most important for showing FX impacts by the surveyed software engineers. A success story in the initial development of FX technology for understanding malicious code will be a key advantage for demonstrating its potential to industry.

5.2 Goal 2: Create FX Automation for Correctness Verification of Programs

The software engineers identified the activity of correctness verification as having the greatest potential for FX impacts. Software developers are demanding improved methods for understanding the behaviors of programs and verifying the correctness of these behaviors with respect to specifications and designs. This information tells us that a short-term goal of the FX project must be to demonstrate automation of program correctness verification using FX technology.

5.3 Goal 3: Create FX Automation for High-Level Programming Environments Starting with Java

The software engineers rated the programming languages of Java, C, and C++ as very important for the application of FX technology. It is clear that the software development industry has great need for support in understanding the behaviors of programs written in these high-level languages. Thus, another important short-term goal of the FX project will be to develop a Function Extractor prototype for one or more of the most popular programming languages. The engineers recommended Java as their first choice.

5.4 Goal 4: Perform Research on Semantics of System Specification and Architecture for FX Automation

The software engineers in the survey demonstrated concern and even skepticism that the promise of FX theory can be successfully transitioned into effective engineering practices for the front-end activities of system specification and architecture development. A major issue is the inability to define and represent the semantics of software specifications, architectures, and high-level designs with state-of-the-art methods. In fact, FX technology is seen as having little potential impact in these front-end software development activities due largely to the lack of clear semantics in those areas. An initiative to perform research on the semantics of software system specification and architecture is required for FX technology to be applied in these areas. The central thesis of this research is that the ultra-large-scale systems of the future can be successfully developed only by exploiting a new generation of semantics-based computational models at very high levels of abstraction. These models will provide the foundation for a science of software design and prescribe engineering automation to define, predict, control, and optimize the behaviors of complex systems regardless of size.

5.5 Goal 5: Perform Research on Human/Computer Interfaces for FX Automation

The effective use of innovative technologies such as FX depends on adaptable and user-friendly human/computer interfaces. It is important that research on user interfaces for FX be performed in parallel with development of the automation itself. Computed behavior has not been available to software engineers in the past, and new reasoning and analysis patterns are sure to emerge. Research is required to understand the dynamics of this new augmentation of human intelligence for optimal design of its user interfaces.

5.6 Goal 6: Perform Experimentation with FX Technology to Evaluate Its Impact

Scientific research requires rigorous experimentation to evaluate the quality and effectiveness of results. The design artifacts of the FX research and development activities are the theories, practices, and automated tools that are produced [Hevner 04]. Evaluation of these artifacts will provide the evidence required by eventual users of the technology to accept and adopt FX into their software development processes and activities. Any new technology faces initial resistance because it requires a learning curve and changes in entrenched software development practices. Rigorous experimentation with FX technologies resulting in clear evidence that they improve development productivity and system quality will ease their acceptance [Green 04, Green 05].

5.7 Goal 7: Perform Research on the Semantics of Software Quality Attributes for FX Automation

In the current state of the art, analysis of software quality attributes such as performance and security is often carried out through subjective evaluations of little value in the dynamics of system operation where attribute values can change quickly. A capability to compute attribute values with mathematical precision would permit both rigorous assessment and improvement of the security attributes of software during development and the real-time evaluation of system performance during operation. Research is required to define computational models for quality attributes that can be evaluated by FX automation. That is, quality attributes must be treated as functions to be computed as dynamic properties of systems.

6 Conclusions

This report summarizes FX research and development and investigates the impact of FX on software engineering. Data collected from active software professionals through a survey instrument provides objective and informed guidance on high-leverage paths for future FX initiatives. The report concludes with seven key recommendations for the future direction of FX research and development:

1. Complete development of the FX prototype for Assembler Language programs.
2. Create FX automation for correctness verification of programs.
3. Create FX automation for high-level programming environments starting with Java.
4. Perform research on semantics of software specification and architecture for FX automation.
5. Perform research on human/computer interfaces for FX automation.
6. Perform experimentation with FX technology to evaluate its impact.
7. Perform research on the semantics of software quality attributes for FX automation.

These goals prescribe a challenging strategy for FX evolution that can result in substantial progress toward next-generation software engineering as a computational discipline.

Appendix Function Extraction Technology Impacts Questionnaire

The objective of this Function Extraction (FX) Technology Impacts questionnaire is to assess the impacts of FX technology across the software engineering life cycle, other software engineering technologies, and the software engineering field. Based on your understanding of the FX theory and applications, please respond to the following questions. All responses are anonymous and the results will be aggregated for analysis.

1. The years of work experience (including research) you have in software engineering activities: _____ years

2. If you have some software engineering experience, what is the primary area of the experience?
   a. Industry – technical
   b. Industry – management
   c. Research (academic or industry)
   d. Government
   e. Other, please specify _____

3. In your organization (or in a software development organization that you have been associated with):

a. What methods or tools are used to understand system behaviors?

b. What percentage of time do developers spend reading and understanding the behaviors of system development artifacts (e.g., specifications, architectures, designs, code, test cases) written by themselves or others?

_____ %

c. By the end of a typical development project, how complete is the developers' understanding of all behaviors in the system?

_____% (0-100%), where 100% represents complete understanding of the behaviors in the system

d. When system behaviors are not completely understood, what have you found to be the biggest impact(s) on system quality?

4. Based on your observations and experiences, estimate what percentage of total time is typically spent on reading and understanding artifacts during each of the following software development activities:

   a. _____ Of the 100% of time in *System Specification* development, the percentage spent reading and understanding *specifications*?

   b. _____ Of the 100% of time in *System Architecture* development, the percentage spend reading and understanding *architectures*?

   c. _____ Of the 100% of time in *Component Design*, the percentage spent reading and understanding behavior of the *designs*?

   d. _____ Of the 100% of time in *Component Evaluation and Selection*, the percentage spent reading and understanding behavior of the *vendor components*?

   e. _____ Of the 100% of time in *Component Implementation*, the percentage spent reading and understanding behavior of *code*?

   f. _____ Of the 100% of time in *Component Correctness Verification*, the percentage spent reading and understanding behavior of the *code*?

   g. _____ Of the 100% of time in *System Integration*, the percentage spent reading and understanding behavior of *components to be integrated*?

   h. _____ Of the 100% of time in *System Testing*, the percentage spent reading and understanding behaviors of *specifications* and *test cases*?

   i. _____ Of the 100% of time in *System Maintenance and Evolution*, the percentage spent reading and understanding the behaviors of all *system artifacts*?

5. Using a 7 point Likert scale, rate the potential impact of FX technologies on each of the following software development life-cycle activities. Consider Average Impact in comparison with other technologies that you have used in your software engineering work.

	No Impact	Minimal Impact	Below Average Impact	Average Impact	Above Average Impact	Strong Impact	Very Strong Impact
a. System Specification	1	2	3	4	5	6	7
b. System Architecture	1	2	3	4	5	6	7
c. Component Design	1	2	3	4	5	6	7
d. Component Evaluation and Selection	1	2	3	4	5	6	7
e. Component Implementation	1	2	3	4	5	6	7
f. Component Correctness Verification	1	2	3	4	5	6	7
g. System Integration	1	2	3	4	5	6	7
h. System Testing	1	2	3	4	5	6	7
i. System Maintenance and Evolution	1	2	3	4	5	6	7

6. Rank the software development life-cycle activities from 1 to 9 based on the level of impact you believe that FX technologies will have on that activity. (1 = Greatest Impact to 9 = Least Impact)

a. System Specification _____

b. System Architecture _____

c. Component Design _____

d. Component Evaluation and Selection _____

e. Component Implementation _____

f. Component Correctness Verification _____

g. System Integration _____

h. System Testing _____

i. System Maintenance and Evolution _____

7. Please provide the rationale for your assessments of the impacts of FX technologies on the software development life-cycle activities.

8. Rank the five most important programming language environments (in your opinion) from 1 to 5 based on the level of importance for application of FX technologies. (Just rank your top five language environments – 1 = Most Important Language for FX Application to 5 = Fifth Most Important Language for FX Application.)

a. Assembler Languages _____

b. C Language _____

c. C++ Language _____

d. Java Language _____

e. Visual Basic (VB) _____

f. COBOL _____

g. Other _____ _____

h. Other _____ _____

i. Other _____ _____

9. Using a 7 point Likert scale, rate the potential impact of FX technologies on each of the following software engineering technologies. Consider Average Impact in comparison with other technologies that you have used in your software engineering work.

	No Impact	Minimal Impact	Below Average Impact	Average Impact	Above Average Impact	Strong Impact	Very Strong Impact
a. Component-Based Development	1	2	3	4	5	6	7
b. Reengineering of Legacy Systems	1	2	3	4	5	6	7
c. Quality Assurance of Software	1	2	3	4	5	6	7
d. Component Reuse	1	2	3	4	5	6	7
e. Automated CASE Tools	1	2	3	4	5	6	7
f. Web Services	1	2	3	4	5	6	7
g. Agile Methods of Software Dev.	1	2	3	4	5	6	7
h. Distributed Computing	1	2	3	4	5	6	7
i. System Documentation	1	2	3	4	5	6	7

10. Using a 7 point Likert scale, rate the potential impact of FX technologies on each of the following areas of software engineering. Consider Average Impact in comparison with other technologies that you have used in your software engineering work.

	No Impact	Minimal Impact	Below Average Impact	Average Impact	Above Average Impact	Strong Impact	Very Strong Impact
a. Software Engineering Education	1	2	3	4	5	6	7
b. Software System Acquisition	1	2	3	4	5	6	7
c. Management of SW Development	1	2	3	4	5	6	7
d. Software Development Teams	1	2	3	4	5	6	7
e. Software Development Organization	1	2	3	4	5	6	7
f. Software Development Industry	1	2	3	4	5	6	7
g. Software Development Workforce	1	2	3	4	5	6	7
h. Software Engineering Ethics	1	2	3	4	5	6	7
i. Software Engineering Economics	1	2	3	4	5	6	7

11. Based on your understanding of FX technology, briefly describe any issues or problems that an organization, project team, or individual might encounter in the acceptance and application of FX technology. What would you need in your current work environment to adopt and effectively use FX? What would keep you from adopting and using FX?

12. Use the remainder of the questionnaire to provide the FX technology team with your guidance for the future structure of this research program.

References

URLs are valid as of the publication date of this document.

[Green 04] Green, G.; Collins, R.; & Hevner, A. "Perceived Control and the Diffusion of Software Development Innovations." *Journal of High Technology Management Research* 15, 1 (February 2004): 123-144.

[Green 05] Green, G.; Hevner, A.; & Collins, R. "The Impacts of Quality and Productivity Perceptions on the Use of Software Process Improvement Innovations." *Information and Software Technology 47,* 8 (June 2005): 543-553.

[Hausler 90] Hausler, P.; Pleszkoch, M.; Linger, R.; & Hevner, A. "Using Function Abstraction to Understand Program Behavior." *IEEE Software 7*, 1 (January 1990): 55-63.

[Hevner 02] Hevner, A.; Linger, R.; Sobel, A.; & Walton, G.. "The Flow-Service-Quality Framework: Unified Engineering for Large-Scale, Adaptive Systems," 4006-4015. *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. Big Island, Hawaii, January 7-10, 2002. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[Hevner 04] Hevner, A.; March, S.; Park, J.; & Ram, S. "Design Science Research in Information Systems." *Management Information Systems Quarterly 28*, 1 (March 2004): 75-105.

[Hoffman 01] Hoffman, D. & Weiss, D., eds. *Software Fundamentals: Collected Papers by David L. Parnas*. Upper Saddle River, NJ: Addison Wesley, 2001.

[Linger 79] Linger, R.; Mills, H.; & Witt, B. *Structured Programming: Theory and Practice*. Reading, MA: Addison Wesley, 1979.

[Linger 02] Linger, R.; Pleszkoch, M.; Walton, G.; & Hevner, A. *Flow-Service-Quality Engineering: Foundations for Network System Analysis and Development* (CMU/SEI-2002-TN-001, ADA339792). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports/02tn001.html

[McCarthy 63] McCarthy, J. "A Basis for a Mathematical Theory of Computation," 33-70. *Computer Programming and Formal Systems* (P. Braffort and D. Hirschberg, eds.). Amsterdam: North-Holland Publishing Co., 1963.

[Mills 86] Mills H.; Linger, R.; & Hevner, A. *Principles of Information System Analysis and Design*. San Diego, CA: Academic Press, 1986.

[Mills 02] Mills, H. & Linger, R. "Cleanroom Software Engineering: Developing Software Under Statistical Quality Control," 143-149. *Encyclopedia of Software Engineering, 2nd edition.* (J. Marciniak, ed.). New York,, NY: John Wiley & Sons, 2002.

[Pleszkoch 90] Pleszkoch, M.; Hausler, P.; Hevner, A.; & Linger, R. "Function-Theoretic Principles of Program Understanding," 74-81. *Proceedings of the 23rd Annual Hawaii International Conference on System Science*. Kailua-Kona, Hawaii, January 2-5, 1990. Los Alamitos, CA: IEEE Computer Society Press, 1990.

[Pleszkoch 04] Pleszkoch, M. & Linger, R. "Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior," 4789-4798. *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*. Big Island, Hawaii, January 5-8, 2004. Los Alamitos, CA: IEEE Computer Society Press, 2004.

[Prowell 99] Prowell, S.; Trammell, C.; Linger, R.; & Poore, J. *Cleanroom Software Engineering: Technology and Practice*. Reading, MA: Addison Wesley, 1999.

[Rajlich 02] Rajlich, V. & Wilde, N. "The Role of Concepts in Program Comprehension," 271-278. *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*, Paris, France, June 27-29, 2002. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[Storey 99] Storey, M. A. D.; Fracchia, F. D.; & Muller, H. A. "Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration." *The Journal of Systems and Software 44*, 3 (January 1999): 171-185.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
The Impact of Function Extraction Technology on Next-Generation Software Engineering	F19628-00-C-0003

6. AUTHOR(S)

Alan R. Hevner, Richard C. Linger, Rosann W. Collins, Mark G. Pleszkoch, Stacy J. Prowell, Gwendolyn H. Walton

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2005-TR-015

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2005-015

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	76

13. ABSTRACT (MAXIMUM 200 WORDS)

Currently, software engineers lack practical means to determine the full functional behavior of complex programs. This gap in intellectual control is the source of many long-standing and intractable problems in security, software, and systems engineering. Function Extraction (FX) technology is directed to automated computation of full program behavior. FX is based on function-theoretic mathematical foundations of software that illuminate algorithmic methods for behavior computation. FX holds promise to replace resource-intensive, error-prone analysis of program behavior in human time scale with fast and correct analysis in computer time scale. The CERT® organization of the Software Engineering Institute is conducting research and development in FX technology and is developing a Function Extraction for Malicious Code system to rapidly determine the behavior of malicious code expressed in Assembler Language. FX technology has the potential for transformational impact across the software engineering life cycle, from specification and design to implementation, testing, and evolution. This study investigates these impacts and, based on a survey of software professionals, defines a strategy for FX evolution that addresses high-leverage opportunities first. FX is an initial step in developing next-generation software engineering as a computational discipline.

14. SUBJECT TERMS	15. NUMBER OF PAGES
computational software engineering, function extraction, FX, function extraction for malicious code, FX/MC, independent research and development, next-generation software engineering	70

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL