Software Engineering Institute

Results of SEI Line-Funded Exploratory New Starts Projects

Bjorn Andersson, Stephany Bellomo, Lisa Brownsword, Yuanfang Cai, Sagar Chaki, William Claycomb, Cory Cohen, Julie Cohen, Peter Feiler, Robert Ferguson, Lori Flynn, David Gluch, Dennis Goldenson, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, John Hudak, Carly Huth, Wesley Jin, Rick Kazman, Mary Ann Lapham, Jim McCurley, John McGregor, David McIntire, Robert L. Nord, Ipek Ozkaya, Brittany Phillips, Robert Stoddard, Dave Zubrow

July 2013

TECHNICAL REPORT
CMU/SEI-2013-TR-004
ESC-TR-2013-004

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0000294

Table of Contents

List of Figures

List of Tables

Abstract

The Software Engineering Institute (SEI) annually undertakes several line-funded exploratory new starts (LENS) projects. These projects serve to (1) support feasibility studies investigating whether further work by the SEI would be of potential benefit and (2) support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. Projects are chosen based on their potential to mature and/or transition software engineering practices, develop information that will help in deciding whether further work is worth funding, and set new directions for SEI work. This report describes the LENS projects that were conducted during fiscal year 2012 (October 2011 through September 2012).

1 Introduction

1.1 Purpose of the SEI Line-Funded Exploratory New Starts

Software Engineering Institute (SEI) line-funded exploratory new starts (LENS) funds are used in two ways: (1) to support feasibility studies investigating whether further work by the SEI would be of potential benefit and (2) to support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. It is anticipated that each year there will be three or four feasibility studies and that one or two of these studies will be further funded to lay the foundation for the work possibly becoming an initiative.

Feasibility studies are evaluated against the following criteria:

- mission criticality: To what extent is there a potentially dramatic increase in maturing and/or transitioning software engineering practices if work on the proposed topic yields positive results? What will the impact be on the Department of Defense (DoD)?
- sufficiency of study results: To what extent will information developed by the study help in deciding whether further work is worth funding?
- new directions: To what extent does the work set new directions as contrasted with building on current work? Ideally, the SEI seeks a mix of studies that build on current work and studies that set new directions.

1.2 Overview of LENS Projects

The following research projects were undertaken in FY 2012:

- Aligning Acquisition Strategy and Software Architecture with Stakeholder Needs

These projects are summarized in this technical report.

2 Aligning Acquisition Strategy and Software Architecture with Stakeholder Needs

Lisa Brownsword

2.1 Purpose

Major acquisition programs now rely on software to provide substantial portions of system performance. Not surprisingly, therefore, software issues are driving system cost and schedule overruns. All too often, however, software is no more than a minor consideration when the early, most constraining, program decisions are made. This generally has negative consequences, most often in terms of misalignments between the software architecture and system acquisition strategies. Through analysis of troubled programs, we perceive that these misalignments lead to program restarts, cancellations, and failure to meet important mission or business goals. This research is therefore focused on enabling organizations to reduce their program failures by harmonizing their acquisition strategy with their software architecture.

2.2 Background

Complex programs have diverse sets of stakeholders. It is inevitable that some, perhaps many of their diverse goals and priorities are in conflict. Operational users, combat commanders, funding authorities, and acquisition team members may think they share the same priorities. But when interviewed, their answers often vary widely in term of the goals and features they see as the most important. In many cases, the solutions that are then created are based on goals of one set of stakeholders—goals that can conflict with those of other stakeholders. Ultimately, such conflicts in goals eventuate in the misalignment described above.

As an example, an organization we encountered showed how easily failures stemming from misalignment of strategy and architecture can occur. This organization was going to rebuild a major system to replace a critical capability. The program manager gave the requirements to the software architect, who returned with an architecture he believed to be well-suited to the requirements he received. The architect was baffled when the program manager cited the lack of a database as a problem. The architect had intentionally eliminated the database because it resulted, in his opinion, in a more elegant solution. This decision, however, was in conflict with an unstated goal of the program manager, who was in charge of an excellent database group that was dependent on work from this program. Though the presence of a database satisfied a legitimate business goal (at least in the program manager's mind), that goal was not captured in any of the requirements given to the architect.

2.3 Approach

The LENS project has two phases. The first phase, discussed in this report, was to identify and articulate the relationships between the key elements that are critical to alignment or misalignment of software architecture and acquisition strategy:

- the architectures themselves (both software and system)
- the planned acquisition strategy
- the quality attributes that drive those architectures and strategies
- the goals (both business and mission) of all of the stakeholders

By examining these elements, we sought to pinpoint the patterns of alignment or misalignment that tend either to keep the software architecture and acquisition strategy in harmony or to pull them apart. These patterns, particularly those that result in misalignment, form a core element of our research.

We will then use these results in the second project phase, where we hope to provide a method for organizations and project managers to avoid anti-patterns we have discovered. We will then validate the utility of these methods through pilot applications on projects and programs outside the SEI.

2.3.1 Phase One

Phase one activities started by using, as a basis, several independent technical assessments (ITAs) that have been performed by the Software Engineering Institute (SEI). Such assessments are commissioned by the Department of Defense (DoD) to provide third-party analyses of a program's health, quality of progress, and similar conditions. While the details of the programs in question must remain anonymous and confidential, we can describe some general attributes of the programs and systems. The domains of the systems extended from weapons systems to information systems. The majority of systems had a significant hardware component. All were large programs, with ambitious expectations; some dealt with unprecedented systems. But common to all was that, at some point in the program history, there were sufficient problems noted that one or more persons in authority had requested an assessment from the SEI, to provide an independent review of the program's execution. Thus, while the spectrum of success ranged from programs that had successfully fielded system, to those that had failed to field anything, all of the programs studied had evidence of at least some failing behavior.

We therefore felt that the findings of the ITAs would provide useful material for our investigation. We carried out confidential interviews with SEI personnel who had participated on these ITAs, and elicited from them data that we deemed relevant to the areas of alignment.[1]

[1] It is important to reiterate that we did not participate on the ITAs themselves, but rather interviewed the SEI personnel who had performed the ITAs.

In these interviews, we sought relevant information about each program's acquisition strategy, the software and system architectures, the quality attributes that those architectures manifest, and the different stakeholders' goals. The specific areas of the programs that we inquired about included the following:

- background of the program, including its scope and motivation
- details of the program, e.g., size, timeline, funding
- mission and business goals of the program
- the nature of the software element of the program
- extensive information about the system architecture, the software architecture, and the relation between the two
- acquisition details, particularly about the acquisition strategy, and its ongoing role as the program unfolded
- information about the relative success that the program achieved

We then analyzed this information looking for patterns of alignment or misalignment. Below we describe the patterns of misalignment, which we term "anti-patterns," that we observed.[2]

2.3.2 Phase Two

In phase two, we expect to describe in detail a method that will assist organizations in avoiding the pitfalls that result when acquisition strategy and architecture are misaligned. It will set forth practices, describe the artifacts that should be created, and describe the relationships that should be present between the key players in an acquisition program. It will also provide detail for an organization to validate that it has followed the method sufficiently that the misalignments we describe are not present. In defining this method, we plan to adapt and tailor existing methods where possible.

2.4 Collaborations

Collaborating on this effort were Lisa Brownsword, Charles Hammons, Cecilia Alberts, Patrick Place, and John Hudak of the SEI, David Carney, an independent contractor under contract to the SEI Acquisition Support Program, and Chris Gunderson of the Defense Technical Information Center. In addition, several members of the SEI who provided valuable information on ITAs were also collaborators, but their identity must remain confidential.

2.5 Evaluation Criteria

In evaluating whether actual anti-patterns were present, we used a number of different criteria:

- the degree to which the pattern of failure had a major negative effect on the program
- the degree to which other negative influences, which themselves may not qualify as anti-patterns (e.g., sudden severe budget cuts) were also present in the program

---

[2] The term "anti-pattern" is more fully described in section 2.6

- the relative scope of the system to be acquired (e.g., number of separate subcomponents, precedented vs. unprecedented)
- the relative scope of the program (e.g., number of contractors, size and diversity of stakeholder base)

2.6 Results

Analysis of these data yields several recurring patterns; because of their negative consequences, and following common usage throughout the software engineering community, [Brown 1998] we characterize these as *anti-patterns*.

Buschmann et al. [Buschmann 1996] provided a form for describing patterns, calling out the need to give patterns a name, define the context (environment), the problem, and then the solution. Thus, for our anti-pattern descriptions, we also call out the notion of consequence, as defined by Gamma et al. [Gamma 1994]. In lieu of calling the observed activity "Solution," we have titled that activity "Observed Response" (i.e., to the problem). Thus, each of our anti-patterns will have the following elements:

- pattern name (readily identifies some key aspect of the problem)
- context (situations where the pattern occurs)
- problem (the recurring issues and the forces that characterize the problem, such as requirements, constraints, and desirable properties, that the context creates)
- observed response (the manner in which people attempt, consciously or otherwise, to solve the problem and, where possible, how they balance the various forces)
- consequences (the results of applying the observed response to the problem in the given context)

2.6.1 Overview of Findings

In each of the programs studied, we observed several instances of activities and behaviors that qualify as anti-patterns. In some cases, the anti-patterns were of sufficient magnitude that they had severe negative effects on program success. While none were observed in all of the programs we studied, all were sufficiently pervasive that they were true patterns of behavior (i.e., none of the anti-patterns were seen in only one program).

Finally, the major source of our data was the analyses of interviews that we carried out. But these data are also supported by decades of experience, on the part of all of the authors of this report, in studying, assessing, and participating in actual programs. Virtually all of the conclusions that derived from our interviews were strongly supported by our aggregate experiences as active professionals, in both government and non-government roles, in the domain of DoD acquisition.

2.6.2 Observed Anti-Patterns

The set of anti-patterns we observed in these programs consists of the following:

1. Undocumented Business Goals

This anti-pattern stems from a lack of precise, well-defined, and well-documented business goals for a DoD acquisition, goals that would correspond to the precise, well-defined mission goals usually created for a program. It is a serious issue, since the DoD's business goals are the major driver for an acquisition strategy, and the software plays a major role in system functionality. But the actual role that the detailed business goals play in the software architecture is often minimal.

2. Unresolved Conflicting Goals

   This anti-pattern is often a direct consequence of the previous one, the distinction being that the first anti-pattern refers to the absence of well-documented business goals, while this one refers to the lack of an analysis and de-confliction of the known goals.

3. Failure to Adapt

   This anti-pattern often occurs when program duration is very long. The reasons for length can be inherent, e.g., when a system is unprecedented, and requires considerable time to solve massive engineering problems (for instance, creation of the Joint Strike Fighter). Or the reasons for highly extended program duration can be circumstantial, such as a protracted, complex protest to a contract award. This anti-pattern can also occur when a program evolves from providing limited capabilities to providing a much wider range of capabilities. This anti-pattern is very closely related to anti-pattern 4 (Turbulent Acquisition Environment), and though they are distinct, the boundaries between them are not clearly defined.

4. Turbulent Acquisition Environment

   This anti-pattern is closely related to anti-pattern 3 (Failure to Adapt). But in this case, the cause is not extended program evolution, but rather severe instability in multiple program elements. This instability is manifest by changes in goals, strategy, or architecture that are so frequent and contradictory, they require adaptation that, even under the best circumstances, the program is unable to accommodate. These changes can be political, strategic, technological, or fiscal.

5. Poor Consideration of Software

   This anti-pattern occurs when critical decisions are made (especially early in a program) that have strong negative implications on the system's software. There is a historical basis for this behavior: for decades, the DoD acquired systems that were primarily hardware. But while the role and importance of software has grown significantly in recent years, the traditions and habits of acquisition still reflect the earlier, hardware-centric posture.

6. Inappropriate Acquisition Strategies

   Time figures prominently in DoD acquisitions. Starting from the earliest moments of a program (i.e., the awareness of need for a new or updated system), one urgent yet often unstated imperative is to move quickly to avoid any eventualities that might delay or even prevent a program from achieving its desired milestones. This imperative is often exacerbated by the need to spend an amount of money that was established as many as two years previously, at a point where little was actually known about the realities that the program would face. It is further exacerbated by the lengthy review process before a new

contract can be awarded. These realities have led to acquisition strategies that are often poorly suited to an individual program's needs.

In addition, many failed acquisitions of large systems have produced a tendency within the DoD toward avoiding the risk of failure with a "big bang" acquisition of a large system; preference is often to acquire it in stages through multiple, independent programs.

7. Overlooking Quality Attributes

   In the earliest stages of a program's life, there may be no formal program office and only minimal accompanying funding to perform necessary work. Further, there is significant pressure to rapidly produce the acquisition strategy and initial architecture in order for the program to be funded. But there is no requirement to use software quality attributes (QAs) to define that architecture, and little incentive to do so.

   Further, in many cases, the detailed business goals are unwritten (see anti-pattern 1), or the importance of the software is ignored (see anti-pattern 5) and hence there is little opportunity to expose the software QAs that the system is expected to manifest.

2.6.3 Countering the Anti-Patterns

We believe that at least part of the solution to these undesirable behaviors lies in analyzing how the anti-patterns operate both at the micro and at the macro level. We now consider how they jointly can affect the major entities that mutually participate in the complex acquisition process.

We will first consider these entities, and how they are made manifest in specific artifacts. We then consider how these entities and artifacts are related. Our assertion is that the presence of each anti-pattern is an indication of weakness in either an artifact or in one or more of its relationships. One key novelty in our consideration is that business goals, like the mission goals, will have quality attributes that should be the main drivers for the acquisition strategy; we assert that these other quality attributes are at least as important as those derived from the mission goals. We will henceforth refer to these other QAs as "acquisition quality attributes."

2.6.4 Necessary Entities and Artifacts

The anti-patterns we observed related to a fairly small number of distinct entities, each of which has major importance for a program and the system it is building. We posit that there are several key entities of interest:

- mission goals
- the (mission) quality attributes implicit in those goals
- business goals
- the (acquisition) quality attributes implicit in those goals
- the acquisition strategy
- the software and system architectures, which are closely related, but separate
- the different sets of stakeholders who have expressed needs that are captured by the mission and business goals

While these entities represent a diverse set of things, including humans (stakeholders) and intangibles (goals), we also posit that all of these entities must, at least in some manner, be manifest in physical artifacts. Some such artifacts are immediately obvious: the mission goals will ultimately be reflected in a requirements specification; an acquisition strategy document is mandatory for any program. But other artifacts are less well-defined, and may not even be present in a given program. But we assert that they are just as necessary.

The stakeholders for a given acquisition, for instance, cannot simply be a vague collection of unknown persons, but must be defined with at least minimal specificity: "the human resources personnel who do data entry for the Air Force Logistics Command," "the Assistant Secretary of the Navy for XYZ," and so forth. The definition of the stakeholders may not have a formal document type associated with it, but it must be physical: there must be some way to determine who precisely has one or another goal, if only to assist in determining priorities and negotiating conflicts. Similarly, the business goals themselves may first be only general expressions found in a statement of need. But eventually, at some point, those must find some form of detailed notation in a physical document that is a real analog to a requirements specification.

Thus, we assert that each of the entities listed above has an associated artifact, which we can inspect and reason about, and compare with other artifacts of the acquisition process.

2.6.5 Necessary Relationships

These entities are related to each other by means of several different relationships. For each, we use the formula of: "Entity X — <relationship> — Entity Y." The following are the pertinent relationships:

- <have>
- <are embodied by>
- <are consistent with>
- <drive>
- <constrains>
- <informs>

We show all of these entities and relationships in Figure 1. Some relationships are unidirectional, and the arrow indicates which entity is the actor, e.g., for "X <drives> Y" the direction of the arrow is from X to Y. Some of these relationships are reflexive, e.g., "X <informs> Y" *and* "Y <informs> X." In these cases, the arrow is two-headed.

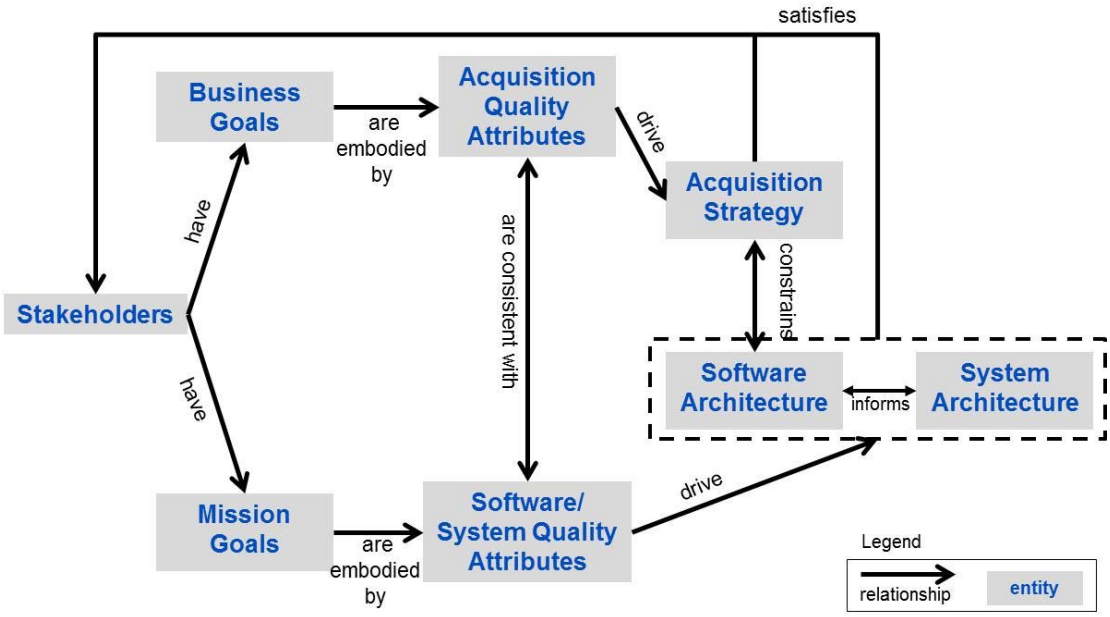

Figure 1: The Entities and Relationships that Pertain to Anti-Patterns

2.6.6 Avoiding the Anti-Patterns

In the preceding sections, we described the relationships between entities that should hold for an acquisition to be feasible. However, the anti-patterns are evidence that the relationships between these entities are either not present or are too weak. The following discussion connects weak or non-existent relationships to the anti-patterns. The stronger the relationship, the lower the chance that the anti-pattern will occur. Figure 2 shows the anti-patterns and the relationships they affect.

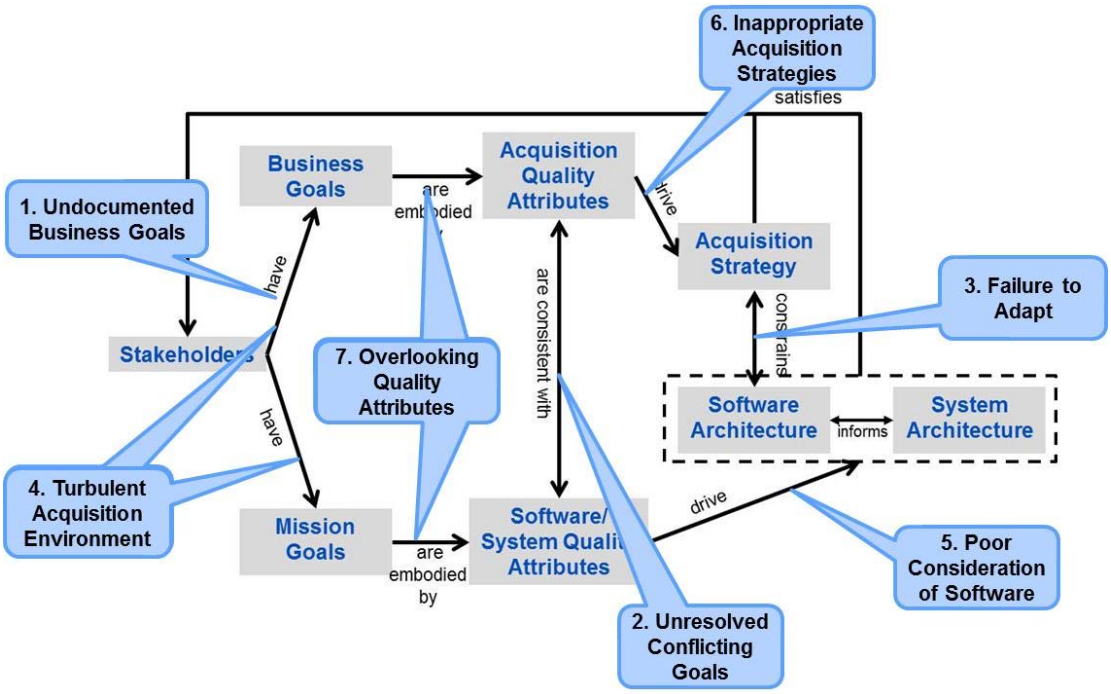

Figure 2: The Anti-Patterns that Affect Specific Relationships and Entities

Stakeholders *have* business goals. Business goals are often not expressed; the problem is exacerbated by the lack of a process for recording business goals. If they could be captured, so that the collection of business goals, stemming from all stakeholders, exists in a coherent document, then the anti-pattern 1 cannot occur.

Stakeholders *have* business goals and *have* mission goals. If a program office captures and records mission goals, then the office can reason about changes in the acquisition environment. It can determine whether the changes can be accommodated within the program's current scope or whether a reset of the acquisition strategy or the architectures will be required. If a program office can perform such reasoning then, though turbulence in the acquisition environment cannot be prevented, the office can have an appropriate response to the turbulence and prevent the occurrence of anti-pattern 4.

Business goals and mission goals *are embodied by* quality attributes. If the goals are analyzed and re-expressed in terms of acquisition quality attributes and software/system quality attributes respectively then it is clear that anti-pattern 7 cannot occur. Obviously, the completeness of these relationships is dependent on the expression of both business and mission goals.

Acquistion quality attributes *are consistent with* software and system quality attributes. While their representations may differ, we expect that it will still be possible to reason about all quality attributes, comparing and performing tradeoffs between the two types of attributes. If tradeoffs must be made, then some stakeholder expectations may not be met, but the program office can know which expectations will not be met, and negotiate with the stakeholders. If the quality attributes are consistent with each other, then conflicts among the goals will have been resolved and anti-pattern 2 will not occur.

Software and system quality attributes *drive* architecture. When quality attributes are used to create the architectures, then we can be certain that those qualities important to the program have been considered and the resultant architecture is consistent with the quality attributes. In such a case, anti-pattern 5 cannot occur.

Acquisition quality attributes *drive* the acquisition strategy. While there is no current practice for deriving the acquisition strategy from the acquisition quality attributes, the analogy to architecture and quality attributes is clear. If the acqusition strategy is derived from the qualities that have been developed from the business goals then the strategy will indeed reflect all the stakeholder expectations and cannot be considered to be inappropriate to the institutional goals of the organizations involved, thus preventing the occurrence of anti-pattern 6.

Acquisition strategy *constrains and is constrained by* architectures. As the system matures, new goals emerge that must be accommodated in the acquisition strategy or the architecture or both. Ensuring that the architectures continue to constrain the acquisition strategy and the acquisition strategy continues to constrain the architectures will increase the likelihood that the program is feasible. In such a way, anti-pattern 3 can be prevented from occurring.

2.6.7 Preliminary Thoughts on a Method

Discovering and documenting the anti-patterns is only the beginning of addressing the problems of misalignment. Characterizing the general shape of an acquisition model that would avoid (or at least minimize) these anti-patterns is a meaningful next step. To that end, the second phase of our project is to create a method that helps programs avoid the anti-patterns we have discovered and provides options that could help a program to better align its acquisition strategy and software architecture so stakeholders' mission and business goals are better satisfied.

Software-reliant systems are inherently social as well as technical endeavors. A key facet of our method, therefore, will be its ability to bring disparate actors together—often for the first time—to rationally identify and discuss issues of mutual concern and be able to make hard choices based on rational information.

We plan to adapt and tailor existing methods where possible. We are currently exploring methods in the following areas:

- *Identifying Salient Stakeholders*: There are many requirements elicitation and analysis methods. Unfortunately, most of these methods assume that it is possible to know which stakeholders will most affect or be most affected by the program. We are looking at Controlled Requirements Expression (CORE) [Mullery 1979], a method that assists developers in identifying stakeholders related to a given acquisition to help in developing a more complete list of stakeholders.
- *Defining Business and Mission Goals*: Pedigreed Attribute eLicitation Method (PALM) [Clements 2010] is a central element of our new method. PALM enables organizations to systematically identify the high-priority mission and business goals from the system stakeholders. The architectural implications of those goals are then captured in quality attribute requirements. We will extend PALM to investigate the acquisition strategy implications that a business or mission goal might have.
- *Analyzing Quality Attributes*: Quality Attribute Workshops (QAW) [Barbacci 2003] are a well-understood method for developing definitions of the quality attributes that form the basis for deriving the software architecture. We will look at using the same approach to derive attributes that should drive the acquisition strategy.
- *Trading off Architecture and Acquisition Strategy Options*: Methods such as Architecture Tradeoff Analysis Method (ATAM) [Clements 2001] or Cost Benefit Analysis Method (CBAM) [Kazman 2002] are used to ensure consistency of software and system quality attributes. We will analyze these methods to explore consistency between the acquisition strategy and its driving quality attributes.

We have also identified some areas where there is no obvious starting point. To fully represent the relationships shown in Figure 1 above, further research is needed in the following areas:

- We assert the existence of a set of acquisition quality attributes (AQAs)—attributes derived from the program's business goals that drive the quality of the acquisition strategy. Examples

of these acquisition quality attributes could be "supplier replaceability" or "contract manageability." We need to explore these AQAs in more detail so that we can

1. more clearly describe what they are and show their relationship to the acquisition strategy and, perhaps, the software, architecture
2. find a way to represent these AQAs in a way that allows the program office to reason about them, prioritize them, and de-conflict them with the QAs derived from the mission goals that drive the software architecture

- We need to define an approach to assess the extent to which the acquisition strategy and the software architecture are (or are not) aligned and characterize the risk this poses to program success.
- We need to tie all of these independent methods together in a way that supports very early program decisions. It must be able to operate on the data that is available before contract award; it cannot overwhelm limited program resources or stakeholders; and it must provide significant value to the program manager and those who oversee him or her.

2.7 Publications and Presentations

This work was introduced in a two-part series[3] of the SEI blog posts. An SEI technical report, titled "Isolating Patterns of Failure in Department of Defense Acquisition," capturing further details of our research results, including complete descriptions of the identified anti-patterns, is in its final approval process.

2.8 References

[Alexander 1977]
Alexander, Christopher, Ishikawa, Sara, & Silverstein, Murray. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. ISBN 0-19-501919-9

[Barbacci 2003]
Barbacci, Mario, Ellison, Robert J., Lattanze, Anthony J., Stafford, Judith A., Weinstock, Charles B., & Wood, William G. *Quality Attribute Workshops (QAWs), Third Edition* (CMU/SEI-2003-TR-016). Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/library/abstracts/reports/03tr016.cfm

[Bass 2003]
Bass, Len, Clements, Paul, & Kazman, Rick. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.

[Brown 1998]
Brown, William .J., Malveau, Raphael C., McCormick, Hays W. "Skip," & Mowbray, Thomas J. *Antipatterns: Refactoring Software, Architecture, and Projects in Crisis*. Wiley and Sons, 1998.

[3] The blog posts are available for part 1 at http://blog.sei.cmu.edu/post.cfm/reducing-project-failures-by-aligning-acquisition-strategy-and-software-architecture-with-stakeholder-needs and part 2 at http://blog.sei.cmu.edu/post.cfm/reducing-project-failures-by-aligning-acquisition-strategy-and-software-architecture-with-stakeholder-needs-1

[Buschmann 1996]
Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter, & Stal, Michael. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* Wiley and Sons, 1996.

[Clements 2001]
Clements, Paul, Kazman, Rick, & Klein, Mark. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001. ISBN-10: 0-201-70482-X; ISBN-13: 978-0-201-70482-2

[Clements 2010]
Clements, Paul & Bass, Len. *Relating Business Goals to Architecturally Significant Requirements for Software Systems* (CMU/SEI-2010-TN-018). Software Engineering Institute, Carnegie Mellon University, 2010. http://www.sei.cmu.edu/library/abstracts/reports/10tn018.cfm

[DAU 2011]
Defense Acquisition University. *Glossary of Defense Acquisition Acronyms and Terms*. 14th Edition, 2011.

[Gamma 1994]
Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2

[IEEE 1998]
IEEE. *Standard for Software Quality Metrics Methodology*, IEEE Std 1061-1998.

[Kazman 2002]
Kazman, Rick, Asundi, Jai, & Klein, Mark H. *Making Architecture Design Decisions: An Economic Approach* (CMU/SEI-2002-TR-035). Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/library/abstracts/reports/02tr035.cfm

[Klein 2010]
Klein, John & Gagliardi Michael J. *A Workshop on Analysis and Evaluation of Enterprise Architectures* (CMU/SEI-2010-TN-023). Software Engineering Institute, Carnegie Mellon University, 2010. http://www.sei.cmu.edu/library/abstracts/reports/10tn023.cfm

[Mullery 1979]
Mullery, G.P. *CORE—A Method for Controlled Requirement Specification*, CHI479-5/79/0000-0126500.75. IEEE 1979.
http://ss.hnu.cn/oylb/tsp/CORE-mullery.pdf

[SEI 2010]
Software Engineering Institute. *CMMI® for Development, Version 1.3.* (CMU/SEI-2010-TR-033). Software Engineering Institute, Carnegie Mellon University, 2010.
http://www.sei.cmu.edu

3 A Decision-Making Framework for Software Maintenance and Evolution

Rick Kazman
Yuanfang Cai

Estimating the duration, effort, cost, and complexity of software development activities is of vital importance for IT management, but notoriously challenging to do well. It is widely known that software projects frequently run over their budgets and schedules. Furthermore, it has been well established that more than half of the total development effort in software projects is spent on the maintenance phase. During the maintenance phase, software tends to age and the code base gets cluttered by an accumulation of changes, often referred to as *technical debt*.

If decision makers (e.g., project managers) do not have good insight into the benefits of refactoring, it is difficult to know *when* to refactor, or whether to refactor at all. A number of prediction models have been proposed to identify components that are error prone or to predict project development cost and effort. But existing work does not directly support a project's decision makers in answering the following question: *When is it worthwhile to refactor the software to reduce the complexity and make it better modularized*?

While the costs of modularization activities such as refactoring are significant and immediate, their benefits are largely invisible, intangible and long term. It has been known for decades that modularity decay can cause substantial problems in projects, such as reduced ability to provide new functionality and fix bugs, operational failures, and, in the extreme, canceled projects. But there is no established quantitative association between modularity variation and maintenance effort variation. That is to say, there is no way for a decision maker to know, with confidence, if a project's modularity gets worse (or better) how much more (or less) it will cost to maintain and extend. Without such a foundation, it is difficult to predict the costs of the technical debt incurred from a deterioration in a project's modularity. And it is equally difficult for decision makers to justify the potential cost savings from a proposed refactoring activity.

3.1 Purpose

This LENS project aims to provide an empirical foundation upon which sound refactoring decisions may be based, by relating variation in the complexity of code to variation in effort—and hence to cost. As a first step in that direction, we tested the following primary hypothesis:

> *There is a statistically significant correlation between software complexity variation over successive releases of software project files, and the variation of effort required to maintain those files.*

If this hypothesis is true, then it becomes possible to understand how the maintenance effort for a code unit, such as a file, varies with changes in one or more source file metrics that characterize

its complexity and modularization, which we can easily capture and track. That, in turn, makes it possible to predict the future maintenance costs of that file. This information can then be used to make economics-driven decisions about software maintenance, including refactoring. To test this hypothesis, we must measure (1) how structural relations in a file change over time, and (2) how maintenance effort spent on that file also changes.

Our first step in addressing this hypothesis was to select a suite of suitable file metrics. Numerous source code metrics have been proposed and studied, but not all of them have a proven correlation with maintenance costs. We conducted a literature survey and identified a set of metrics that have validated relationships to maintainability, with solid theoretical and empirical bases.

The second step was to select a suitable measurement of *effort* at the file level. Although effort at the project or task level is usually measured using person-months, there is no widely agreed-upon effort measurement at the file level. Maintenance effort spent on a file can be measured in multiple ways. Just measuring changes in the number of lines of code—the most obvious measure—may be misleading. For example, if the change is inherently difficult, even adding just a few lines of code may require a great deal of effort, in the form of intensive discussions within the project team, or multiple trial-and-error rounds leading to several revisions.

We thus employ a new, and we believe, more holistic approach to measuring effort, from three dimensions: the number of lines of code changed to resolve tasks (*churn*), the amount of discussion that tasks generated (*discussions*), and the number of atomic changes made to a file to resolve a change request (*actions*).

Employing our selected file metrics and new effort measures, we developed a new empirical approach to correlating file metrics variation to the variation of maintenance effort on a file. Using these techniques, we investigated the following research questions to test the primary hypothesis:

> Q1: Which file-level code metrics are significantly correlated to which types of maintenance effort?

The answer to this question will help us understand which code metrics best track the increase or decrease of maintenance effort. We examined the selected metrics and their correlation to the maintenance effort measures on a per-file basis, using statistical models.

> Q2: Does the correlation between file-level code metrics and maintenance effort differ between projects?

The answer to this question will help us understand which metrics to use to make predictions for a variety of projects, or if the types of maintenance effort thatbest correlate with file metrics are mostly project specific.

While testing these hypotheses is of interest to most software projects, few industrial projects are willing to contribute the data needed to answer the question with authority. We have therefore selected five Apache open source projects containing between 8 and 18 releases as our experimental subjects. For each file of each project, we calculated its structural properties using the selected file metrics, and computed the delta of each metric between subsequent releases. We

also extracted maintenance effort for each file, and their deltas between releases. After that, we analyzed the correlation between pairs of code metrics deltas and effort measure deltas. We are interested to know whether *changes* in code metrics are strongly correlated with changes in effort measures.

Our results demonstrated that there are strong and significant correlations between a small subset of the selected file metrics and two of the three maintenance effort measures: the number of actions and the number of code churns. This result is confirmed across all the five subject projects. Furthermore we observed that, when each project is considered separately, the most significantly correlated metrics-effort pairs differ, and in certain projects, the amount of discussion also shows strong and significant correlations. We also observed that the significance of the correlation between metrics and effort increases in later releases. These results provide a positive answer to the hypothesis, showing that it is possible to quantitatively predict maintenance effort changes from code-level file metrics, and that the best measures and metrics may be project-specific.

3.2 Background

As previously stated, our research aims to advance the way maintenance effort is measured, and the way source code metrics and maintenance effort are correlated. To situate our contribution we will first review prior research on code metrics and then examine the research on correlating these metrics to maintenance effort. Finally, we discuss how our approach differs from prior work.

3.2.1 Research on Metrics

Published source code metrics can be broadly divided into five categories, based on what they measure: size, complexity, coupling, cohesion, and inheritance. We will give a brief description of each category, along with some of the most influential publications on source code metrics.

3.2.1.1 Size

Size is the most straightforward metric for source code. The number of lines of code (LOC) is the most obvious and simplest way of measuring size. But it has its drawbacks. For example, it is always possible to write the same functionality with fewer (or more) lines of code, while maintaining similar complexity. To address this problem, several other metrics have been proposed.

3.2.1.2 Complexity

Measures of the complexity of a source file are postulated to affect modifiability and maintainability: lower complexity is better. Examples of complexity metrics are Halstead Volume—based on operator and operand counts and McCabe Complexity—based on the number of possible paths in program control graph [Halstead 1977; McCabe 1976].

3.2.1.3 Coupling

Coupling describes the number of connections a file or class has to other files or other classes. The assumption is that lower coupling is better. Briand and associates proposed a set of metrics

that measure different possible versions of class-to-class coupling [Briand 1997]. Another coupling metric is Propagation Cost, which MacCormack and associates first introduced in 2006 [MacCormack 2006].

3.2.1.4 Cohesion

Cohesion measures how strongly the responsibilities within a code unit are related. The rationale behind measuring cohesion is the belief that code units, such as source files or classes, should focus on only one thing, and that this single focus will improve maintainability.

3.2.1.5 Inheritance

Inheritance-based metrics only apply to object-oriented code. Less complex inheritance hierarchies are expected to be easier to understand and maintain.

Chidamber and Kemerer (henceforth C&K) developed the best known metrics suite aimed at measuring object-oriented source code, including metrics for coupling, cohesion and inheritance [Chidamber 1994].

3.2.2 Research on Metrics and Maintenance Effort

A number of papers have attempted to correlate source code metrics to maintenance effort. However, there is no generally agreed-upon method to predict maintenance effort at the level of a source code *file*. We now describe a number of the approaches that have been attempted.

3.2.2.1 Comparing against expert judgment

Welker and colleagues proposed a polynomial that uses complexity-based metrics to predict maintenance effort [Welker 1997]. The weights for each of these metrics are fitted automatically, so the polynomial matches data of expert judgment in eight systems. They presented this polynomial as the *Maintainability Index*.

3.2.2.2 Comparing against Maintainability Index

Misra and Zhou and Xu compared a list of complexity and inheritance metrics against the Maintainability Index at the system level [Misra 2005; Zhou 2008]. Both papers found significant correlations to both inheritance and complexity.

3.2.2.3 Comparing in controlled experiments

Harrison and associates compared metrics against both expert judgment and maintenance measurements obtained in a controlled experiment [Harrison 1998]. They found that both complexity and cohesion correlated with their maintenance measures, and that complexity correlated with the expert judgment of a system.

Arisholm looked at ten changes made to an industrial system and logged hours spent on each task. He found no correlation between source code metrics and effort, possibly due to the small size of the data set [Arisholm 2006].

3.2.2.4 Comparing against change

Li and Henry linked a set of metrics to the total volume of changes to classes in two different projects [Li 1993]. They found significant correlations for complexity, coupling, cohesion and inheritance metrics. Binkley and Schach looked at change volume of an industrial system [Binkley 1997]. They positively correlated this to coupling and complexity metrics and to one inheritance metric. Ware and associates looked at the number of changes and the number of lines changed for files in a commercial application [Ware 2007]. They found significant correlations for complexity and coupling measures.

In a slightly different vein, Anbalagan and Vouk found a significant correlation between the number of participants in a bug report and the time taken to complete it [Anbalagan 2009]. While this is not an effort measure, it is certainly related to the organizational dimension of a software project and the corresponding effort overhead.

3.2.2.5 Comparing over releases

Demeyer and Ducasse attempted to identify problem areas in the source code of a project and check if those problem areas were refactored in later releases [Demeyer 1999]. They did not find such a pattern, but they noted that the project they studied was in substantially good shape, so there might not have been a major need for refactoring. Alshayeb and Li attempted to correlate a polynomial, consisting of complexity, coupling, and inheritance metrics, to maintenance effort in iterative projects [Alshayeb 2003]. They measured lines of code added, deleted, and changed, first between releases of a project, then between changes within a release. They found that their constructed polynomial was reasonably good at predicting effort between changes, but not as good at predicting effort between releases.

3.2.3 Differences in Our Approach

Our research is different from earlier research as follows: First, instead of simply comparing file-based code metrics to maintenance effort, we compare an increase or decrease in file metrics to an increase or decrease in maintenance effort. This is intended to give us a more accurate insight into the effect of source code variations across the lifetime of a project. If there is a clear correlation between a change in a metric value and a change in a maintainability measure, a project manager will be able to use this knowledge to make informed decisions about maintenance and refactoring opportunities. Although D'Ambros and associates also employed variations of source code metrics, they used them to predict defects [D'Ambros 2010]. Our research is the first to correlate variations of file-level code metrics to maintenance effort.

Second, instead of measuring maintenance effort merely using the number of changes in lines of code (churn), we add two new measurements: the number of actions—atomic changes to resolve an issue—and the amount of discussion among developers for an issue. This gives us a more holistic, multi-dimensional view of maintenance effort, assuming that a complex change will require more discussion and more changes to files (since some of the initial changes will not be correct and will necessitate subsequent rounds of changes).

Third, only Alshayeb and Li have also looked at the relation between metrics and maintenance effort over multiple releases, but instead of comparing variations in metrics over different releases to variations in maintenance effort, they created a formula to predict maintenance effort from source code metrics, and tested the formula over various releases and changes on the project [Alshayeb 2003]. They examined only 13 changes in total, whereas we are looking at thousands of changes. While other researchers have leveraged code churn to predict project effort [Mockus 2003] and defect density [Nagappan 2005], we are the first to investigate the correlation between file metrics and code churn, which is a manifestation of effort at the file level.

3.3 Approach

In this section we describe the subject projects that we have studied, the set of measures we chose to collect from these projects, the rationale behind these choices, and our data collection methods.

3.3.1 The Subject Projects

The selection of projects is important to the quality of the data, and therefore to the validity of the research. Now we describe the criteria we used to select the projects for this research and the motivation behind those criteria.

1. Criteria: To ensure the generality of our research, we attempted to obtain a diverse set of projects. We specifically looked for heterogeneity along the following dimensions:

   Variation in domain of software: Uses of software can be categorized into various application domains. We tried to find projects from distinct domains to ensure that our research results would apply generically.

   Variation in source code sizes: Even though Dolado has shown that development productivity does not vary significantly across project sizes, maintaining a large-scale software project is still, in practice, different from maintaining a small scale software project [Dolado 2001].

   Variation in team size: There has been considerable research on the effects of team size on development speed. Brooks argues that smaller teams tend to have greater productivity per person, but many have argued that larger teams are better for productivity and quality in Open Source Software [Brooks 1975].

   Variation in project age: The age of a body of software can influence developer productivity in ways that may not be measurable by source code metrics. For example, the technology chosen (language of implementation, operating system, development libraries, etc.) can cease to be supported, and key developers can leave the project, resulting in a knowledge loss.

   We have, however, restricted our attention to projects written in Java, so that we could repeat the same metrics extraction process for our entire set of projects. Furthermore, we only selected projects that use a version control system and a bug tracking system, as our maintenance data is derived from these systems. The projects selected all contain source code

and maintenance data for at least eight releases and all have more than 300 resolved issues in their bug tracking systems.

2. Selected projects: We have summarized the project characteristics, the first and last release for which we have extracted data, and the number of resolved or closed issues that we were able to extract in Table 1. As we can see from the table, Derby[4], Lucene[5], PDFBox[6], Ivy[7], and FtpServer[8] are from different domains, with different team sizes and different project ages. The number of resolved issues ranges from 329 to 3058.

Table 1: Selected Projects

Project	# of Releases	Resolved Issues	# of Contributors	First Release	Last Release	Domain
Derby	18	3058	458	2005/08 (10.1.1.0)	2011/10 (10.8.2.2)	Relational Database
Lucene	18	2444	652	2006/03 (1.9.1)	2010/12 (3.0.3)	Distributed search
PDFBox	8	699	387	2010/02 (1.0.0)	2011/07 (1.6.0)	PDF document manipulation tool
Ivy	12	758	408	2006/11 (1.4.1)	2010/10 (2.2.0)	Transitive relation dependency manager
FtpServer	10	329	112	2007/02 (1.0.0-M1)	2011/07 (1.0.6)	Java-based FTP server

3.3.2 The Selected Metrics

In this section, we discuss the metrics that were calculated for each file of each project.

1. Criteria: As described in Section 3.2.1, numerous metrics (summarized by Xenos and colleagues and by Riaz and colleagues) have been proposed that are purported to predict software quality and maintenance effort [Xenos 2000; Riaz 2009]. Unfortunately, testing all these metrics for their power in predicting maintenance effort was infeasible. To select a smaller target set of metrics to analyze, we applied three criteria.

   The metric is widely applicable: Since we are restricting our research to projects written in Java, the metrics must be applicable at least to this language.

   The metric is defined at the file level: The unit of analysis in our research is the source file, so the metric must be interpretable at the file level. We do, however, employ metrics that are defined at the class level. To account for this discrepancy, we only consider files

[4] http://db.apache.org/derby/

[5] http://lucene.apache.org/core/

[6] http://pdfbox.apache.org/

[7] http://ant.apache.org/ivy/

[8] http://mina.apache.org/ftpserver/

that contain a single class. This constraint only eliminates around 7% of the total files from our candidate data set.

The metric has been consistently proven in previous research: To keep the scope of the research manageable, we chose only metrics that have been consistently shown to correlate with maintenance effort in previous studies.

2. Selected metrics: Given these criteria, we have selected the following metrics.

   Source Lines of Code (LOC): The total lines of code in the file. The idea behind this metric is that, all other things being equal, larger files are harder to maintain.

   Weighted Method Complexity (WMC): The sum of the complexities of the methods in a class.

   Response for a Class (RFC): Total number of methods that may be invoked as a result of invoking any method in a class.

   Coupling Between Objects (CBO): The number of other classes to which the class in this file connects.

   Lack of Cohesion of Methods (LCOM): The number of method pairs in the class in this file that do not share the usage of a single attribute of the class.

   Depth in Tree (DIT): The number of classes that are a superclass of the class in this file.

   Number of Children (NOC): The number of classes that have the class in this file as a superclass.

   Afferent Couplings (Ca): A measure of how many other classes use the specific class in this file.

   Number of Public Methods (NPM): The number of methods in a class that are declared as public.

WMC, RFC, CBO and LCOM, DIT and NOC were all described by C&K in 1994 [Chidamber 1994]. We have altered their definitions slightly to make them meaningful at the file level, as described above. The C&K suite has been studied heavily and its metrics have been validated in many other works.

In addition to the extended set of C&K metrics, we have selected one more metric—Propagation Cost, which aims to capture *architectural* complexity.

Propagation cost (PC): PC, first introduced by MacCormack and colleagues in 2006, is a coupling-based metric, and there has been some promising research on the predictive power of PC on maintenance effort [MacCormack 2006].

PC is based on a *visibility matrix*, which is a binary matrix where a project's files are the rows and columns, and dependencies between the files are the values. These dependency values are determined using a path length L, which allows a file A to be dependent on a file B through a

dependency chain of length L. For example, a path of length of 1 denotes traditional coupling, since only direct dependencies are represented in the matrix. The propagation cost is computed as the sum of all dependencies in the visibility matrix, divided by the total possible dependencies.

However, propagation cost calculated that way is defined at the project level. Instead, we compute *incoming propagation cost* at the file level, by taking the sum of the incoming dependencies for a file, divided by the total possible dependencies; concretely, this means that we take the sum of the column in the visibility matrix that represents the file, and divide this value by the length of the column. Analogously, to calculate *outgoing propagation cost* per file, we take the sum of the row in the visibility matrix that represents the file, and divide that by the length of the row.

We have also introduced a new variant of the propagation cost metric that employs a *decay rate*. With this decay rate, we reduce the strength of indirect dependencies by a factor for each additional step in the dependency chain. In the present study, we applied a decay rate factor of 0.1.

In this research, we have investigated both incoming and outgoing propagation costs with path lengths of 1, 3, 5, 10, and 20—with and without decay—to see which variant of the propagation cost metric has the most predictive power. When describing our results, we will use a naming convention; for example, PROP-OUT-10-N indicates outgoing propagation cost with path length 10 and without decay, whereas PROP-IN-5-D indicates incoming propagation cost with path length 5 and with decay.

Considering all combinations, we have a total of 18 propagation cost metrics (not 20, since when path length is 1 the decaying and non-decaying version of the metrics are the same).

The 18 propagation cost metrics, plus LOC and the 8 C&K metrics, give us a grand total of 27 metrics that we calculate for each file of each release of each project.

3.3.3 Data Collection Method

For each project, we study a number of releases, each of which has a set of files. Each project also has a list of issues, which are extracted from the project's bug- or issue-tracking software. Issues consist of both bug reports and change requests. Developers can submit patches to suggest a solution to an issue. Patches consist of a list of actions, which are changes to files that were made to resolve the issue. For each action, we measure code churn as the number of lines of code added and removed, where changed lines count as both added and removed. Each action corresponds to a change done in one file for one issue, but issues are often resolved using multiple actions. The patches that finally are accepted (i.e., that resolve the issue) are called *commits*.

Developers can also associate comments with issues. These comments are used for communication between developers.

1. Extracting the data: We populated our data model by extracting data from the bug tracking system and version control repositories for each project. Since our five projects were all maintained by the Apache foundation, the same technologies were used. The bug tracking system in use is Jira, which has a WSDL API that is usable for data extraction. Their version control system is Subversion. We constructed a number of tools that query

both systems for the data, format it, and insert it into our database based on our data model.

2. Compiling code metrics: We used three different programs to calculate file metrics. To calculate the source Lines of Code (LOC), we used the utility SLOCCount.[9] This gives us the number of lines of code in the file. For the extended set of C&K metrics, we used ckjm 1.9.[10] For the propagation cost-based metrics, we wrote our own tool, which first generates a Dependency Structure Matrix (DSM) [MacCormack 2006], from which we can count the number of incoming and outgoing dependencies per file over various path lengths.

For each release, we took a snapshot of the code base from the version control system of the project. After extracting the selected metrics per file per release from those snapshots, we stored them in our database for analysis.

3. Measuring maintenance effort: Since we are looking at open source projects, the developers did not log hours for their maintenance work. To approximate maintenance effort, we scrutinized our data set to see which proxy measures for maintenance effort we could find. We settled on and collected three file-based proxy measures.

   Discussion: This is the amount of discussion that occurred in resolving an issue. The assumption is that a more complex change is likely to generate more discussion. Concretely, we measure the number of comments that have been made in the bug tracking system for an issue, and we associate the numbers to the files modified to resolve that issue.

   Change in lines of code (Churn): Churn is the total number of lines of code that were changed in the file to resolve an issue. If a file is changed multiple times for the same issue, we see if the file changes overlap to make sure we don't count the same changes multiple times.

   Actions: We also measure the number of actions, as described in Section 3.3.3 that were performed to resolve an issue. Concretely, this counts the total number of patches and commits that were needed to resolve an issue affecting a file. The notion here is that the more complex the file, the more likely it is to require a large number of actions if something in it had to be changed.

3.4 Evaluation Criteria

The methodology described in Section 1.3 gives us a data set consisting of code metrics and maintenance effort measures for each release of each file. Since we want to analyze the relationship between increasing or decreasing code metrics and maintenance effort, we transform these numbers so that they reflect the change in that value from the previous release. To this end, we take the metric value of each file in a release, and divide it by the equivalent value in the

[9] http://www.dwheeler.com/sloccount/

[10] http://www.spinellis.gr/sw/ckjm/

previous release. If the resulting quotient is less than 1, it means the value has decreased; if it is greater than 1, it has increased; if it equals 1, the value remains unchanged. We do an analogous calculation for effort values: since the same file can be involved in multiple issues for the same release, we divide the effort measure for each issue in the new release by the average effort measure per issue in the previous release. These derived data points each represent the change in a metrics or effort measure between two consecutive releases of a given source file.

Table 2: Example Data Extracted from Project Files

	File name	Releases	LOC	Actions	Group	Relative LOC	Relative Actions
1	client.java	1.0	100	40	-	-	-
2	client.java	1.0	100	60	-	-	-
3	client.java	1.1	140	75	1	1.4	1.5
4	client.java	1.2	70	60	2	0.5	0.8
5	library.java	1.1	60	75	-	-	-
6	library.java	2.0	72	150	1	1.2	2.0
7	server.java	1.0	200	40	-	-	-
8	server.java	1.2	240	52	1	1.2	1.3

The calculation of relative values is exemplified in Table 2 (ignore the group column for now). This table shows how the relative values for one file metric, LOC, and one effort measure, *actions*, are calculated. The relative LOC value of "client.java" in Row 3 (1.4) is the quotient of its current LOC value (140) and its LOC value in the previous release (100). We calculated relative effort values in a slightly different way. This is because our effort measures are calculated on a per-issue basis. For example, if "client.java" was changed in Release 1.0 to address two separate issues, it takes two separate rows (Rows 1 and 2 in Table 2).

To calculate the relative *actions* value of "client.java" in release 1.1 in Row 3 (1.5), where "client.java" was involved in two issues in the previous release, we divide its current value (75) by the average of all previous actions values ((40 + 60) / 2 = 50). The file "library.java" in Rows 5 and 6 has no action data associated with it for Release 1.0 or 1.2, so the entry for Release 1.1 is used as the first data point, and the entry for Release 2.0 as the second data point. We calculated relative values in this manner for each file with all 27 x 3 = 81 metric-effort pairs.

3.4.1 Spearman Analysis

Each line in Table 2 is considered as a data point. We thus calculated the data points for each individual file of each project as explained above, to understand the correlations between code metric variation and effort measure variation. Since the effort values do not follow a normal distribution, we have used the Spearman rank correlation test.

However, Spearman assumes independent measurements in the data set subject to the test. Since our data reflects deltas of the same measurements over time for each file, we cannot assume such independence, and we must not include data points pertaining to the various releases of the same file in the same data set.

We thus segment our data points into groups (see Table 2) according to the following procedure: we first skip all data points which come from the first release we have on record for a file,

because without a previous release, we cannot calculate meaningful relative values. After that, all data points that belong to the second release of each file go in Group 1, all data points that belong to the third release of a file go in Group 2, etc. This means that data from different releases of the same file never appear in the same group. We can then apply the Spearman correlation test separately to each group.

Once more, Table 2 shows an example of how the segmentation of data into groups is done. Row 1 and 2 belong to the first release of file "client java" for which we have information." Because we use values that are relative to the previous release, we cannot use these values in our analysis. Row 3 belongs to the second release of "client.java," so it goes into Group 1. Row 4 belongs to the third release of "client.java" that we have information for, so it goes into Group 2.

Row 5 contains the first release of "library.java" that we have information for, thus it has no relative values. Please note that Release 1.1, to which the data point for Row 5 belongs, does not have to be the first release in which library.java existed in the project; rather Release 1.1 is the first release where we have maintenance effort data for "library.java." Row 6 belongs to the second release of "library.java" for which we have information, so it goes into Group 1.

Row 7 is the first release of "server.java" for which we have information, so we cannot add it to a group. Row 8 is the second release of "server.java" for which that we have information, so we add it to Group 1. Please note that for Row 8 (Release 1.2), the relative values are not relative to a data point from the previous release (1.1), but from two releases before (1.0). This is because there is no effort data recorded for server.java in Release 1.1.

In our analysis we exclude issues that did not affect source files. For a file to show up in at least one group, it must be changed in more than one release. As a result, 2888 of the total 9733 issues from all five projects were used to generate usable data points.

We ended up having nine groups from this aggregated data set; the individual groups have the following number of data points: 87, 602, 47, 028, 21, 686, 9, 614, 5, 804, 3, 167, 1,388, 374, and 348. Since groups with higher numerical IDs will have an increasingly smaller population of data points (that is, fewer files have that many rounds of changes over releases), which is detrimental to the accuracy and reliability of the statistical analysis, we decided to only use Groups 1 to 7.

In summary, each data point represents how one type of maintenance effort and one type of file metrics of a file vary over two successive releases where the file was changed. Investigating all the groups will reveal how these two aspects change together over multiple releases.

3.5 Results

Since we have performed a Spearman analysis on a set of 81 (27 x 3) code metric type versus maintenance effort type pairs, over nine different groupings of files, we have run a total of 729 Spearman tests and therefore obtained 729 p and rho values. Due to the large number of tests, we allow a maximum p-value of 0.01 to ensure the significance of the results. We thus obtained 17 significant correlations from the aggregated data set. In addition, we performed the same Spearman tests on each project. The five projects have nine (Derby), three (FtpServer), seven (Ivy), eight (Lucene), and four (PdfBox) groups, yielding over 2500 additional data points. To

answer the two research questions proposed in Section 1.1, we report significant results ($p < 0.01$) obtained from the aggregated data set and from individual project data in Table 3 and Table 4 respectively.

Table 3: Metric-Effort Correlation for Aggregated Data Set

Group	Metric Type	Effort	P	Rho
3	CKJM-WMC	actions	0.00045	0.3910
4	CKJM-RFC	actions	0.00106	0.5158
4	raw LOC	actions	0.00167	0.4425
4	CKJM-Ca	actions	0.00180	0.3436
5	PROP-OUT-20-N	churn	0.00293	0.6378
5	PROP-OUT-5-N	churn	0.00899	0.5654
5	PROP-OUT-10-N	churn	0.00294	0.6378
6	CKJM-CBO	churn	0.00228	0.6694
6	PROP-OUT-1-N	actions	0.00008	0.7586
6	CKJM-NPM	actions	0.00006	0.7154

Our research question Q1 asked which metrics are significantly correlated to which types of maintenance effort. To answer this question, in Table 3, we list all significant results ($p < 0.01$) with rho value of at least 0.3, obtained from a sample size of at least 15. All of the top results show a strong correlation with just two effort types: actions and churn. In addition, it is interesting to note that 7 of the top 10 results show a strong correlation with coupling-based metrics: 4 variants of PC and 3 C&K metrics (Ca, CBO, and RFC).

The values in Table 3 are ordered in terms of their groups. It is encouraging that the rho value tends to increase as the group number increases for two reasons. First, a good predictive model should improve as you accumulate more data. Second, the accuracy of a predictive measure of maintainability, if it is a good measure, should increase over time as maintenance concerns and technical debt accumulate in a project. The idea is this: in the early stages of a project there tends to be a low level of technical debt and so virtually any structure of the software will not be overly problematic. As technical debt accumulates, complexity concerns tend to overwhelm a programmer's cognitive abilities, and maintenance costs go up. This is exactly what we can see from the results presented in Table 3.

In Table 4, we list the top 3 most significant results ($p< 0.01$) with rho value of at least 0.3 and sample size of at least 15, for each individual project. Our research question Q2 asked whether these correlations would differ between different projects. The project-by-project results shown in Table 4 suggest that this is indeed the case.

While actions and churn show up as important measures in all five projects—highly correlated with the C&K and PC metrics—discussion is highly correlated with PC in Derby and with LCOM (Lack of Cohesion of Methods) in PDFBox. Similar to the results obtained from the aggregated data set, coupling-based metrics are strongly correlated with effort in 12 out of 15 cases.

Table 4: Metric-Effort Correlation for Individual Projects

Project	Group	Metric Type	Effort	P	Rho
Derby	6	CKJM-NPM	actions	0.00007	0.81
Derby	3	CKJM-WMC	actions	0.002220.00001	0.55
Derby	1	PROP-IN-20-N	discussion		0.3
Lucene	1	PROP-IN-5-N	actions	0.0001	0.38
Lucene	1	PROP-IN-1-N	actions	0.000220.0001	0.38
Lucene	1	CKJM-Ca	actions		0.35
PDFBox	2	CKJM-LCOM	discussion	0.000620.002080.00	0.58
PDFBox	2	PROP-IN-1-N	churn	841	0.5
PDFBox	1	CKJM-RFC	actions		0.48
Ivy	3	CKJM-RFC	churn	0.000310.001570.00	0.71
Ivy	3	PROP-IN-20-N	churn	157	0.46
Ivy	3	PROP-IN-10-N	churn		0.46
FtpServer	1	PROP-IN-5-N	actions	0.007410.0091	0.5
FtpServer	1	PROP-IN-20-N	actions	0.0091	0.49
FtpServer	1	PROP-IN-10-N	actions		0.49

3.5.1 Discussion

In this section, we discuss the results, threats to validity and future work. Based on the analysis results presented in the previous sections, we can now answer the research questions proposed in Section 1.1:

1. *Which file-level code metrics are significantly correlated to which types of maintenance effort?* LOC, several variants of PC as well as RFC, CBO, WMC, NPM, and Ca have been shown to be significantly correlated to the maintenance effort measures of actions and churn. Previous work has shown that the eight metrics from the C&K metric suite are good predictors of maintenance effort. Our work supports these prior results and furthermore provides empirical evidence that PC metrics are also good predictors of effort.

2. *Does the correlation between file-level code metrics and maintenance effort differ between projects?* We found meaningful differences in the correlations between metrics and measures in the different projects that we studied. This is not surprising, since projects have very different characteristics (e.g., size, age, domain, inherent complexity) as shown in Table 1, as well as very different styles of leadership and cultures. For example, discussions are highly correlated with complexity metrics in just two of the five projects we studied: Derby and Lucene. Such correlation emphasizes the need for a large and varied corpus of data on which to base a predictive model of maintenance, since the most effective metrics and measures may be project specific.

3.5.1.1 Threats to validity

Now we discuss possible threats to validity caused by the choice of maintenance effort measures, the limitations of data extraction, and the choice of projects to study.

We employed the three types of file-level effort measures because developers in open source projects do not log their hours for maintenance work. Even in industrial settings, accurate effort logs are hard to obtain. Moreover, we need maintenance effort at the file level, which is even more difficult to obtain, if not impossible. We hypothesize that the three measures (churns,

actions, and discussions) can adequately approximate the maintenance effort spent on a file. This is, however, a hypothesis that is impossible to test directly within our existing research framework.

We considered several other possible maintenance effort proxies, such as the time used to resolve an issue, the number of discussions by email, or the number of defects. We chose not to include them for the following reasons: The time elapsed from a ticket was open until it was closed can easily contain slack time that does not reflect the actual effort. A ticket may, for example, remain open for a long time because of its low priority rather than its inherent difficulty. It is possible that discussions of an issue can be conducted through a developers' email. However, there is no reliable and automatic way to link email discussions with the files involved in the maintenance activities being discussed. Bacchelli and colleagues have made progress recently on the linkage between email contents and source code [Bacchelli 2012]. As part of our future work, we would like to strengthen our results further by including email discussions, if such tools for doing so are available and reliable.

There are many other works that use similar measures, such as the number of changes to predict defects. By contrast, the main focus of our work is the direct measure of maintenance effort, of which defect fixing is only a part. If a defect is fixed, then the fix is usually manifested as changes in code, patches, commits, and discussions, which will be captured by our three effort measures. These effort measures can also capture the effort spent on other kinds of maintenance activities. However, if an issue is not resolved, our approach won't count it as incurring any effort.

We also ignored files that had more than one class defined in them. However, as we showed earlier, only 7% of the files are excluded for this reason. Another threat to validity is that we only examined open source projects. While we expect that similar results will be obtained in industry projects, we cannot guarantee it.

Furthermore, we have only investigated five projects and we only investigated a subset of the possible metrics that we could have considered. For example, we could have considered PC metrics with different path lengths and different decay factors. A larger study employing more projects and more metric types would improve the validity of our conclusions.

3.6 Publications and Presentations

None thus far.

3.7 References

[Alshayeb 2003]

Alshayeb, M. and Li, W. "An empirical validation of object oriented metrics in two different iterative software processes," *IEEE Trans. Software Engineering, 29*, 11 (Nov. 2003): 1043–1049.

[Arisholm 2006]

Arisholm, E. "Empirical assessment of the impact of structural properties on the changeability of object-oriented software," 1046–1055. *Information and Software Technology 48,* 11, 2006.

[Anbalagan 2009]
Anbalagan, P. and Vouk, M. “On predicting the time taken to correct bug reports in open source projects.” 523–526. *IEEE International Conference on Software Maintenance (ICSM 2009).* Edmonton, Alberta, Canada, Sept. 2009. IEEE, 2009.

[Bacchelli 2012]
Bacchelli, A., Dal Sasso, T., D’Ambros, M., and Lanza, M. “Content classification of development emails,” 375–385. *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*, Piscataway, NJ, U.S.A., 2012. IEEE, 2012.

[Binkley 1997]
Binkley, A. B. and Schach, S. R. *Inheritance-based metrics for predicting maintenance effort: An empirical study*, Technical Report TR 9705, Computer Science Department, Vanderbilt University, 1997.

[Briand 1997]
Briand, L., Devanbu, P., and Melo, W. “An investigation into coupling measures for C++,” 412–421. *Proceedings of the 1997 (19th) International Conference on Software Engineering*, Boston, MA, May 1997.

[Brooks 1975
Brooks, Jr., F.P. “The Mythical Man-Month,” *Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975

[Chidamber 1994]
Chidamber S. and Kemerer, C. “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering, 20,* 6 (Jun 1994) 476–493.

[D’Ambros 2010]
D’Ambros, M., Lanza, M., and Robbes, R. “An extensive comparison of bug prediction approaches.” 31–41. *7th IEEE Working Conference on Mining Software Repositories (MSR)*, Cape Town, South Africa, May 2010.

[Demeyer 1999]
Demeyer, S. and Ducasse, S. “Metrics, do they really help,” 69–82. *Proceedings of Langages et Modeles a Objets LMO (LMO’99)*, Paris, France, Jan.1999.

[Dolado 2001]
Dolado, J. “On the problem of the software cost function,” *Information and Software Technology, 43*, 1 (2001): 61–72.

[Halstead 1977]
Halstead, M. H. *Elements of Software Science* (Operating and programming systems series). New York, NY, U.S.A.: Elsevier Science Inc., 1977.

[Harrison 1998]
Harrison, R., Counsell, S. J., and Nithi, R. V. "An investigation into the applicability and validity of object-oriented design metrics," *Empirical Software Engineering 3*, 3 (Sep. 1998): 255–273.

[Li 1993]
Li, W. and Henry, S. "Object-oriented metrics that predict maintainability," *Journal of Systems and Software 23*, 2 (1993): 111–122.

[MacCormack 2006]
MacCormack, A., Rusnak, J., and Baldwin, C. Y. "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science 52*, (July 2006): 1015–1030.

[McCabe 1976]
McCabe, T."A complexity measure," *IEEE Trans. Software Engineering SE-2*, 4 (Dec. 1976): 308–320.

[Misra 2005]
Misra, S.C. "Modeling design/coding factors that drive maintainability of software systems," *Software Quality Control 13*, 3 (Sep. 2005): 297–320.

[Mockus 2003]
Mockus, A., Weiss, D. M., and P. Zhang. "Understanding and predicting effort in software projects," 274–284. *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, Washington, D.C., U.S.A., 2003.

[Nagappan 2005]
Nagappan N., and Ball, T. "Use of relative code churn measures to predict system defect density," 284–292. *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. New York, NY, U.S.A. ACM, 2005.

[Riaz 2009]
Riaz, M., Mendes, E., and Tempero, E. "A systematic review of software maintainability prediction and metrics," 367 –377. *3rd International Symposium on Empirical Software Engineering and Measurement*, *(ESEM 2009)*. Lake Buena Vista, Florida, U.S.A., Oct. 2009.

[Ware 2007]
Ware, M. P., Wilkie, F. G., and Shapcott, M. "The application of product measures in directing software maintenance activity," *Journal of Software Maintenance and Evolution 19*, 2 (March 2007): 133–154.

[Welker 1997]
Welker, K. D., Oman, P. W., and Atkinson, G. G. "Development and application of an automated source code maintainability index," *Journal of Software Maintenance: Research and Practice, 9,* 3,(1997): 127–159.

[Xenos 2000]
Xenos, K. Z. Michalis, Stavrinoudis, D., and Christodoulakis, D. “Object-oriented metrics–a survey.” *Proceedings of the FESMA 2000, Federation of European Software Measurement Associations*. Madrid, Spain, Oct. 2000.

[Zhou 2008]
Zhou, Y. and Xu, B. “Predicting the maintainability of open source software using design metrics,” *Wuhan University Journal of Natural Sciences, 13*, (2008): 14–20.

4 Semantic Comparison of Malware Functions

Sagar Chaki
Cory Cohen
Arie Gurfinkel
Jeffrey Havrilla

4.1 Purpose

Malware infections continue to pose an ever increasing threat to both DoD and non-DoD software. One of the most effective means of defending against malware is to collect, catalog, and analyze existing malware samples to develop better techniques for preventing, detecting, diagnosing, and eliminating future infections. An important tool in the fight against malware is the ability to compare malware samples and determine the degree to which they are similar. For our purposes, two malware samples are similar if they have been compiled from the same source code with different versions or optimizations of a compiler, or slightly different source code. We call this provenance similarity [Chaki 2011]. Detecting provenance similarity between malware has at least two potential applications:

1. **Partition:** The number of malware samples collected grows rapidly over time. For example, CERT collects millions of syntactically different samples every three months or so. Clearly, all these samples do not correspond to new malware—there simply aren't enough malware authors to go around. One purpose of a similarity detection tool would be to partition a large collection *X* of syntactically distinct malware samples into a much smaller number of subsets, where each subset consists of "semantically" (or behaviorally) similar malware. This could enable more efficient cataloging and lookup.
2. **Query:** One of the first steps performed by an analyst is to determine if the malware sample under study *S* is similar to something in a large collection *X* of previously studied samples. The result of this step provides important clues about the malware's origin, purpose, and structure. Currently, this step is largely manual, and quite time consuming. An automated similarity detection engine would reduce the effort required for this step and free up resources for the remaining phases of malware analysis.[11]

A good malware similarity detector must satisfy two criteria. First, it must be *scalable*, and capable of partitioning large malware collections (e.g., comprising tens of millions) into subsets of similar ones in a reasonable amount of time (e.g., a few days). Second, it must be *precise* and able to detect samples similar to a target malware from a large collection of potential candidates quickly and with few false matches.

[11] L. Zeltser, "Analyzing Malicious Software," CyberForensics: Understanding Information Security Investigations, pp. 59–83, 2010.

For the purpose of our project, we focused on comparing malware "functions" for similarity. A binary function is a concrete machine code fragment that is produced by compiling a procedure or method at the source code level of abstraction. Binary functions are a natural unit of organization of malware in terms of both syntax and semantics. It is therefore logical to assume that similar malware executables will share many similar functions. Therefore, a precise and scalable technique to compare functions would lead to a correspondingly precise and scalable approach for detecting malware similarity. Thus, the goal of our project was to develop a precise and scalable technique for semantic comparison of binary functions in malware.

4.2 Background

We assume that a binary function is expressible in terms of its control-flow graph. Informally, a control flow graph is a directed graph whose nodes correspond to control-flow points and whose edges represent flow-of-control (i.e., jumps) between them. Nodes are labeled with assembly instructions that are executed after the program reaches the corresponding control-flow point. In the case that an edge represents a conditional jump, it is labeled by the condition under which the jump occurs.

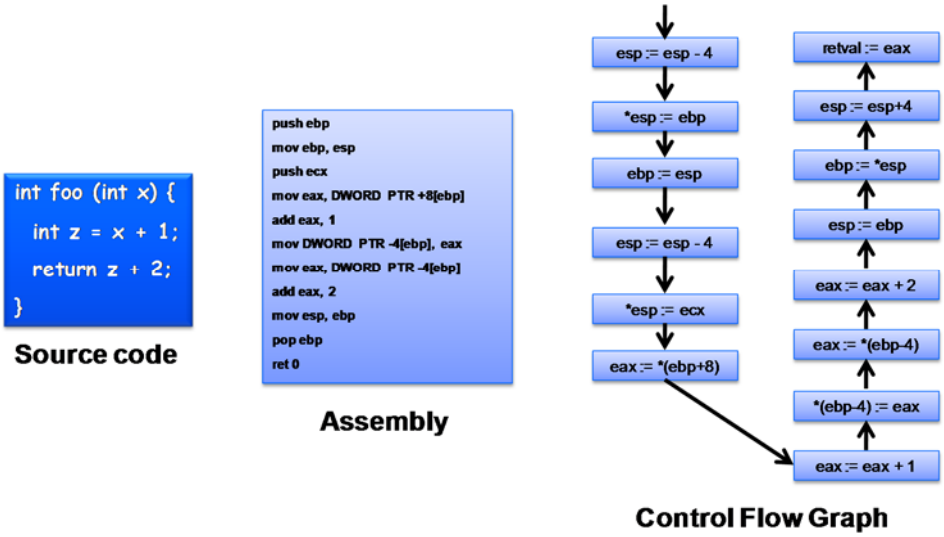

Figure 3: An Example Binary Function—Source Code, x86 Assembly, and Control Flow Graph

For our prototype implementation, we used the ROSE infrastructure to convert a binary function into its control flow graph. Figure 3 shows the C source code of a function, its x86 disassembly listing, and its control flow graph.

In earlier work, we had explored the use of supervised machine learning (also known as classification) to detect similarity between binary functions [Chaki 2011]. However, that approach required comparison between pairs of functions. This means that applying the approach to detecting similarity between n functions requires $O(n^2)$ comparisons, which makes this approach non-scalable to large function collections. In this project, we aimed to overcome this limitation by a combination of semantic clustering and semantic difference analysis. The details of this combination are presented in the next section.

4.3 Approach

We decomposed our goal into two broad sub-problems:

1. **Function Similarity:** Given a large collection of binary functions, partition them efficiently into subsets of semantically similar functions. The partitioning would be conservative in the sense that a cluster might contain functions that are not similar, but two similar functions would always belong to the same cluster.

2. **Semantic Function Comparison:** Given a pair of functions, determine if they are semantically equivalent (i.e., if they have identical input-output behavior in terms of the system state).

Our overall function similarity approach combines solutions to the above two problems as follows. First, partition the target binary function collection into subsets of similar functions using the solution for sub-problem 1. The idea is that each subset would be much smaller than the original collection. However, as mentioned, a subset might still contain semantically dissimilar functions. Next, compare each function pair from a subset using the solution for sub-problem 2 to refine each subset into smaller subsets. Since each subset is small, the pair-wise comparison would still be tractable. We now discuss our solutions to sub-problems 1 and 2.

4.3.1 Detecting Function Similarity

Our partitioning solution is based on *semantic hashing*. In essence, we hash each function into a semantic signature, and then use the signature as a unique cluster id. The signature is designed to guarantee that two semantically similar functions always yield the same signature. This ensures that the clustering is conservative—similar functions always belong to the same cluster.

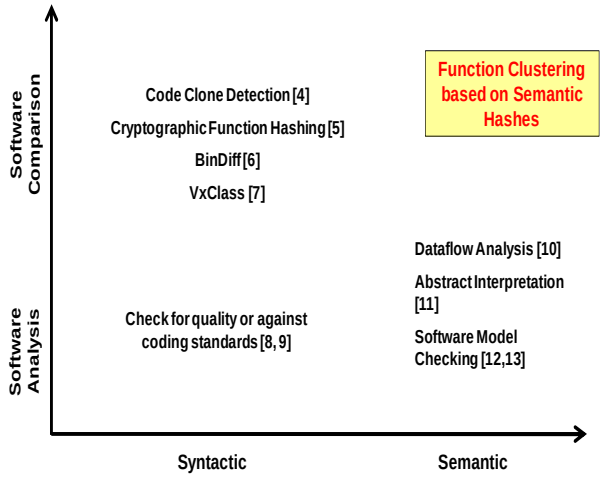

Figure 4: Related Work in Function Similarity Detection

Figure 4 shows a survey of related work in the area of function similarity detection. Note that our approach is distinguished by the fact that it is a semantic approach to software comparison. Other approaches are either syntactic or targeted at software analysis (i.e., more computationally expensive).

The semantic signature of a binary function was computed as follows:

1. Extract the set of basic blocks from the control flow graph of the function. A basic block is a sequence of assembly instructions without any conditional jumps.

2. Hash each basic block to a signature to reduce each block's feature set to single values. We explored two ways of extracting basic block signatures—one based on symbolic summaries and another based on sampling the result of executing the basic block symbolically.

3. Output a locality-sensitive MINHASH of the basic block signatures as the signature of the function.

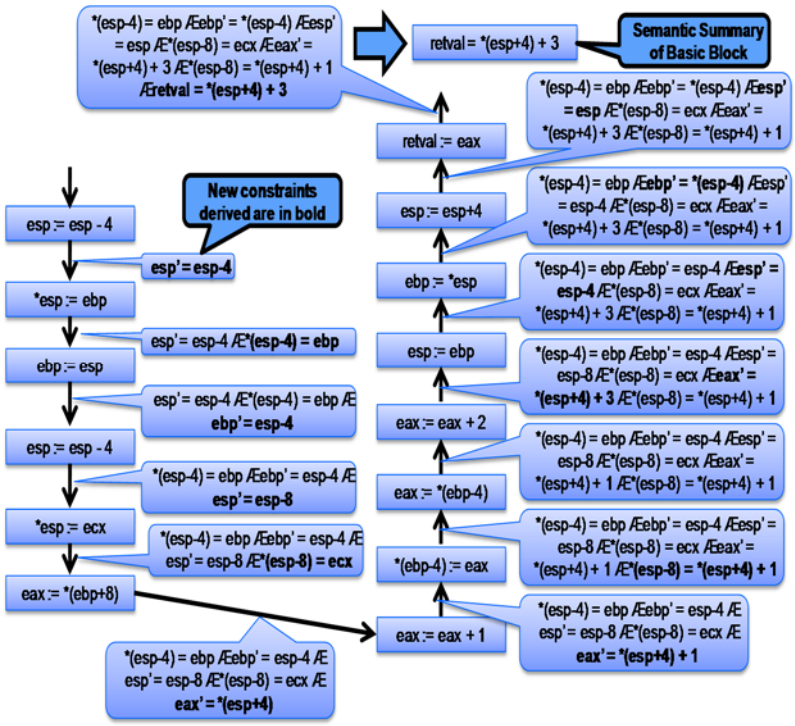

Figure 5: Example of Extracting Symbolic Summary of a Basic Block

Figure 5 shows an overview of computing the symbolic summary of a basic block. Full details of this process, as well as relevant experimental results, are available in our publication [Jin 2012].

4.3.2 Performing Function Comparison

Our approach to comparing functions was based on a paradigm known as regression verification [Godin 2009]. At a high level, this approach works as follows:

1. Suppose we want to compare functions $f_1()$ and $f_2()$. Let i_1 and o_1 be the input and output variables of $f_1()$. Compute a logical formula F_1 over i_1 and o_1 that represents the effect of executing $f_1()$. Similarly, let i_2 and o_2 be the input and output variables of $f_2()$. Compute a formula F_2 over i_2 and o_2 that represents the effect of executing $f_2()$.

2. Prove that the following formula is logically valid:

$$(i_1 = i_2) \wedge F_1 \wedge F_2 \Rightarrow (o_1 = o_2)$$

Clearly, the above formula is valid iff $f_1()$ and $f_2()$ have the same input-output behavior, that is, if they are semantically equivalent. In other words, the formula is valid iff, given equal inputs, $f_1()$ and $f_2()$ produce equal outputs.

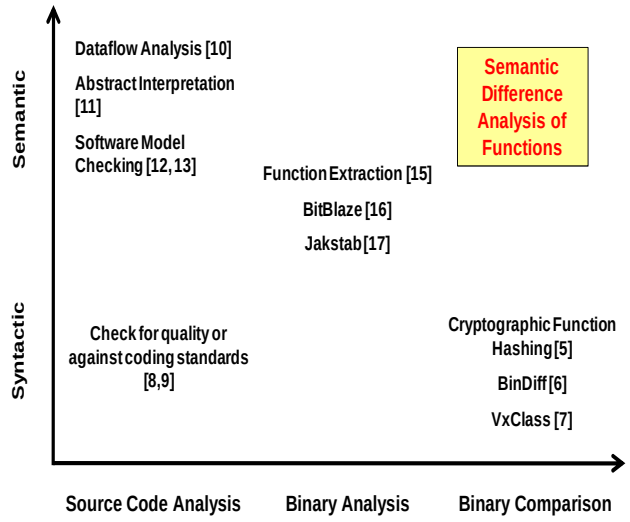

Figure 6: Related Work in Function Comparison

Figure 6 shows a survey of related work in the area of function comparison. Note that our approach is distinguished by the fact that it applies semantic techniques for binary function comparison. Other approaches are either syntactic, or aimed at binary or source code analysis.

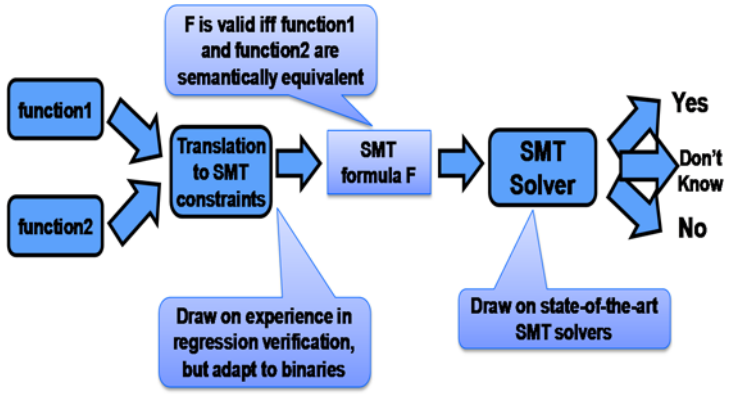

Figure 7: Overview of Function Comparison Approach

Figure 7 shows an overview of our function comparison approach. The challenge in making this approach practicable is to develop effective ways of constructing the formula F_1 and F_2. We were able to develop and implement algorithms to construct such formulas assuming that the functions $f_1()$ and $f_2()$ contained no loops. We were also able to validate our prototype on simple hand-

crafted examples. However, lack of available effort prevented further progress or more systematic validation in this part of the project.

4.4 Collaborations

The project was supported critically by two groups of collaborators:

1. members of the ROSE project, in particular Dan Quinlan and Robb Matzke
2. Professor Priya Narasimhan from CMU and her PhD student Wesley Jin

Robb helped us tremendously with feature updates and bug fixes to ROSE. Wesley was instrumental in developing and implementing some of the hashing algorithms, and providing feedback throughout the project. Our sincere thanks go to all of them.

4.5 Evaluation Criteria

We evaluated our semantic, signature-based, binary function similarity detection algorithm on a benchmark constructed from the CERT Artifact Catalog. The benchmark consisted of a set of sixteen test cases, each consisting of a collection of binary functions. The test cases were of two types—similar and dissimilar. A similar test case consisted almost entirely of (what we believe were) similar functions. In contrast, a dissimilar test case contained only (what we believe were) dissimilar functions. We then clustered each test case using our approach. Successful clustering would lead to a small number of clusters for the similar test cases, and a large number of clusters for the dissimilar test cases.

Table 5: Experimental Results

Example	#Funcs	#Samp	#Summ	#Comb
FormCreate()	2048	2.24	2.49	2.49
Memcpy1()	10000	0.17	0.19	0.19
Memcpy2()	4479	0.44	0.51	0.51
Memmove()	355	6.47	8.16	8.16
Random1()	4615	86.04	96.74	97.52
Random2()	9390	83.02	96.80	97.48
DistName()	193	89.11	93.26	93.78

4.6 Results

Table 5 shows a snapshot of our results. The first four rows correspond to similar test cases, while the last three rows correspond to dissimilar test cases. The first column indicates the name of the test case. The second column indicates the number of functions in the test case. The next three columns indicate the number of clusters, as a percentage of the total number of functions. Each of three columns represents a different variation of the semantic signature scheme presented earlier. Note that we observe a small number of clusters for the similar test cases, and a large number of clusters for the dissimilar test cases. Thus, our results indicated that the semantic signatures produce a good clustering with low computational cost. Further details are available in our publication [Jin 2012].

4.7 Publications and Presentations

Our research report on solving the function similarity problem based on semantic hashing was accepted for publication and presentation at a peer-reviewed conference in machine learning [Jin 2012].

4.8 References

[Chaki 2011]
Chaki, Sagar, Cohen, Cory, and Gurfinkel, Arie. "Supervised learning for provenance-similarity of binaries," 15-23. *Conference on Knowledge Discovery and Data Mining (KDD) KDD'11, August 21–24, San Diego, CA, U.S.A.* ACM, 2011. http://users.cis.fiu.edu/~lzhen001/activities/KDD2011Program/docs/p15.pdf

[Ball 2001]
Ball, Thomas, Majumdar, Rupak, Millstein, Todd D., and Rajamani, Sriram K. "Automatic Predicate Abstraction of C Programs," 203–213. *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* Snowbird, Utah, June 2001.

[Cordy 2011]
Cordy, James R. and Chanchal, K. Roy. "DebCheck: Efficient Checking for Open Source Code Clones in Software Systems." *International Conference on Program Comprehension (ICPC-2011)*. Kingston, ON, Canada, June 2011. IEEE Computer Society, 2011,

[Cohen 2009]
Cohen, Cory and Havrilla, Jeffrey. "Function Hashing for Malicious Code Analysis." *CERT Research Annual Report*. (2009) 26–29. Software Engineering Institute, Carnegie Mellon University. http://www.cert.org/research/2009research-report.pdf

[Cousot 1977]
Cousot, Patrick and Cousot, Radhia. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," 238–252. *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages.* Los Angeles, CA, U.S.A. January 1977.

[Godlin 2009]
Godlin, Benny, and Strichman, Ofer. "Regression verification," 466–471 *Proceedings of the 46th Annual Design Automation Conference (DAC 2009)*. San Francisco, CA, July 2009.

[Gurfinkle 2008]
Gurfinkel, Arie and Chaki, Sagar. "Combining Predicate and Numeric Abstraction for Software Model Checking," 1–9. *Proceedings of the Formal Methods in Computer-Aided Design, 2008.* Portland, Oregon, November 2008. https://es.fbk.eu/events/fmcad08/

[Jin 2012]
Jin, Wesley, Chaki, Sagar, Cohen, Cory, Gurfinkel, Arie, Havrilla, Jeffrey, Hines, Charles, and Narasimhan, Priya. "Binary Function Clustering using Semantic Hashes." *Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA*), Boca Raton, Florida, U.S.A. December 2012.

[Kinder 2008]
Kinder, Johannes and Veith Helmut. "Jakstab: A Static Analysis Platform for Binaries," 423–427 International *Conference on Computer Aided Verification (CAV 2008)* Princeton, NJ, U.S.A., July 2008.

Linger 2011]
Linger, Richard C., Sayre, Kirk, Daly, Tim, and Pleszkoch, Mark G. "Function Extraction Technology: Computing the Behavior of Malware" 1–9. *44th Hawaii International Conference on System Sciences (HICSS),* Koloa, Kauai, HI, January 2011.

[Reps 1995]
Reps, Thomas W., Horowitz, Susan, and Sagiv, Shmuel. "Precise Interprocedural Dataflow Analysis via Graph Reachability," 49-61. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, CA, January 1995.

[ROSE 2013]
ROSE Compiler Infrastructure. *ROSE@LLNL*, 2013. http://www.rosecompiler.org

[Song 2008]
Song, Dawn Xiaodong, Brumley, David, Yin, Heng, Caballero, Juan, Jager, Ivan, Kang, Min Gyung, Liang, Zhenkai, Newsome, James, Poosankam, Pongsin, and Saxena, Prateek. "BitBlaze: A New Approach to Computer Security via Binary Analysis," 1–25. Fourth International Conference on Information Systems Security (CISS) December 2008. JNTU, Hyderabad, India. http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf

[Wikipedia 2013]
Wikipedia. "lint (software)." http://en.wikipedia.org/wiki/Lint_(software) (2013).

[Wikibooks 2009]
Wikibooks. "AdaControl." 2009.
http://en.wikibooks.org/wiki/Ada_Programming/Coding_standards

[Zynamics 2011]
Zynamics. *Zynamics BinDiff*, 2011. http://www.zynamics.com/bindiff.html

[Zynamics 2013]
Zynamics. *Zynamics VxClass*. http://www.zynamics.com/vxclass.html (Accessed April 2013).

5 Architecture-Focused Testing

Peter Feiler
John Hudak
David Gluch
John McGregor
Lisa Brownsword

Commercial and DoD software-intensive safety-critical systems, such as aircraft, are not delivered and upgraded in a timely manner despite current best verification and exhaustive testing practices. Studies [NIST 2002, Galin 2004, Dabney 2003] show that up to 70% of errors in embedded safety-critical software are introduced in the requirements (~35%) and architecture design phases (~35%), while 80% of all errors are not discovered until system integration, acceptance test, or operation causing unexpected system failures, loss of equipment and life [KnoxNews 2011]. Discovery of such system-level faults late in the lifecycle results in high rework cost, as much as 300-1000 times that of in-phase correction. A recent study of the National Research Council (NRC) on "Dependable Systems: Sufficient Evidence?" states that testing and good development processes are indispensable but are not sufficient to ensure high levels of dependability [Jackson 2007].

5.1 Purpose

The objective of Architecture-focused Testing (AFT) is to reduce development cost by making tests more efficient. This is achieved by taking an architecture-centric model-based approach to complement traditional testing practices. The purpose of this project was to develop a compositional AFT methodology and to demonstrate improved testing efficiency by applying it to actual customer systems.

We have developed the Architecture-Focused Testing (AFT) method based on the concept of virtual integration and repeated analysis of architecture fault models throughout the development lifecycle to discover faults close to when they are introduced. This leads to a reduction in faults leaking to and uncovered by testing. It also leads to identification of faults through analysis and simulation that are hard to test for, for example, loss of data due timing race conditions, and potentially unhandled faults (e.g., corruption of air data due to transient hardware malfunction).

Our approach takes advantage of advances in architecture modeling languages, in particular the SAE International Architecture Analysis & Design Language (AADL) [SAE 2004/2012], as the basis of an architecture-centric model-based engineering practice designed to be an integral part of continuous verification and validation (V&V) in the qualification of safety-critical software-intensive systems. The AFT method is based on an architecture fault model framework that combines composable architecture fault profiles with an error propagation ontology to efficiently construct an analyzable architecture fault model for a specific system. Consistency analysis of the architecture fault model leads to identification of unhandled faults. Impact analysis of the

architecture fault model identifies potential undesirable system behavior in the interaction between system components during nominal operation and under fault conditions. Root cause analysis of the architecture fault model identifies fault sources due to mismatched assumptions, leading the way towards a solution that actually corrects the problem.

We have applied the AFT method to several customer systems to demonstrate the feasibility of early discovery of system-level problems and early verification of proposed solutions to correct such problems. A comparison with current testing practice shows that earlier detection and correction of architecture-induced faults results in measurably fewer failing tests and more effective use of finite testing resources.

5.2 Background

Today's practice of developing software-reliant systems can be characterized as "build-then-test," with an increasing use of architecture modeling in the process. UML-based models are used to document the architecture and design of a system. Performance, safety, security, and reliability models are created independently with varying abstractions by different teams, resulting in analyses that are inconsistent with one another and with the actual system architecture [Redman 2010]. The resulting systems are "tested into submission" until testing budgets are exhausted.

Much of current testing practice consists of testing the implementation through unit test, system integration test, and acceptance test. Some improvement efforts of the testing practice have focused on improving test coverage, as exemplified by the progression from structural path coverage to decision coverage (DC), and modified condition decision coverage (MC/DC) [FAA 2002]—as required by DO-178B for different levels of criticality. Other improvement efforts have focused on reducing the amount of testing through regression testing [Richardson 2005] and firewall testing [White 2008].

Aerospace Recommended Practices (ARP) ARP4754 and ARP4761 provide guidelines for development of civil aircraft and systems and methods for conducting safety assessment, such as Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA). These ARPs together with Software Considerations in Airborne Systems and Equipment (DO-178B/ED12B) have gone through revision to address changes in practice, such as tool qualification, formal methods, object oriented technology, and model-based development and verification.

Despite best design practices and the use of fault tolerance techniques, most system-level faults are not discovered until late in the development process or even until operation. Typically, the cause of this pattern can be traced to mismatched assumptions between application system components and with the underlying runtime system [Feiler 2009]. A number of recent studies have identified this problem and recommended a paradigm shift towards architecture-centric predictive analysis through increasing use of formal analytical frameworks [Leveson 2004; Jackson 2007; GAO 2008].

The SAE AADL is a unifying framework for capturing the static modular software architecture, the runtime architecture in terms of communicating tasks, the computer platform architecture on which the software is deployed, and any physical system or environment with which the system interacts [Feiler 2012]. An architecture model expressed in AADL is annotated with information

that is relevant to the analysis of various operational characteristics, such as timing, performance, safety, reliability, and security. Through auto-generation of analysis models from the same annotated architecture model, architectural changes are consistently reflected in the different analysis dimensions (Figure 8). Similarly, application code and efficient tailored runtime system can be auto-generated from such validated and verified AADL models.

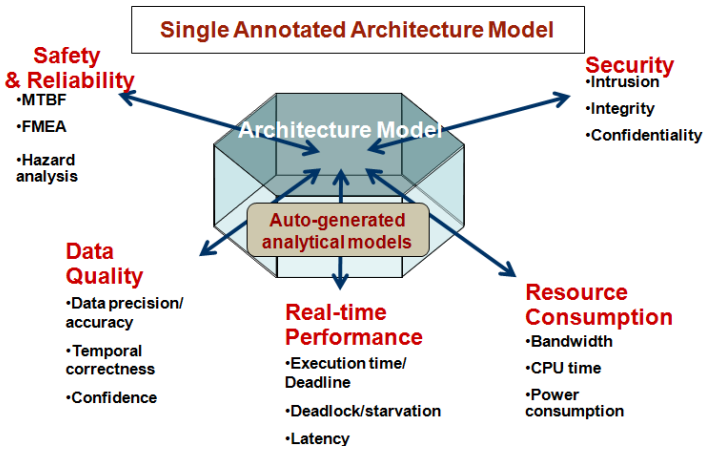

Figure 8: Multiple Analysis Dimensions Based on Architecture Reference Model

The concepts provided by AADL, such as threads, processes, or devices, as well as port-based sampled and queued communication, shared data access, and service request interaction concepts have well-defined execution semantics that support both lightweight and formal analyses of systems. Strong typing of the language ensures consistency and protected address space enforcement of processes. In addition, resource allocation enforcement of virtual processors ensure time and space partitioning, and the AADL mode concept represents dynamic changes in the architecture structure and connection topology. The enforcement of types and semantic rules on AADL models alone already enables discovery of potential issues with the architecture.

The built-in execution semantics allow for schedulability, latency, and performance analysis early in development. In addition, using its extensibility constructs, custom specification and analysis techniques can be incorporated with core AADL capabilities to address project and system specific needs. The Behavior Annex Standard extension allows for specification of component and interaction behavior [SAE 2011]. The Error Model Annex Standard supports fault modeling [SAE 2006]. Other standardized extensions include the ARINC653 Annex for partitioned architectures, and the Data Modeling Annex for integrating data models with architecture models.

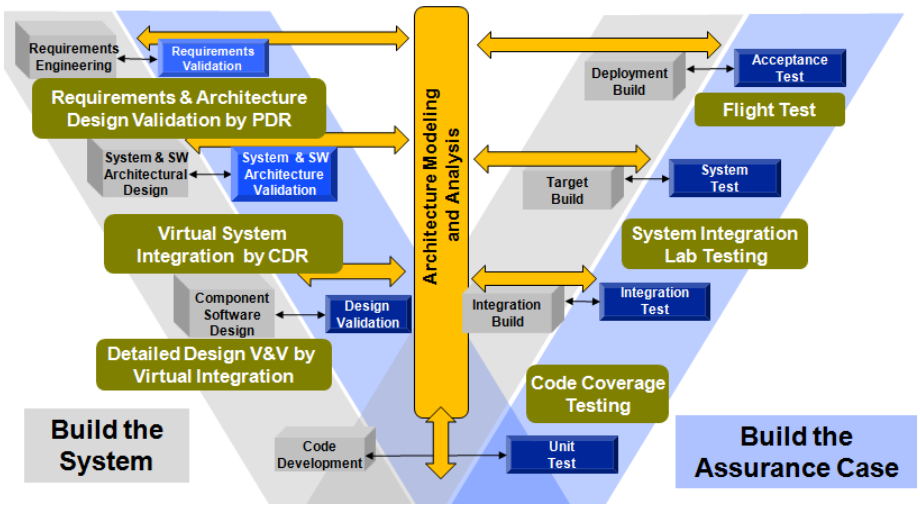

Figure 9: Incremental, End-to-End System Validation & Verification

The Aerospace industry has launched the System Architecture Virtual Integration (SAVI) program, which is a multiyear and multimillion dollar effort focused on developing a capability for system validation and verification through virtual integration of systems [Redman 2010]. AADL was selected as a key technology in the SAVI virtual integration approach that has been demonstrated to lead to discovery of system-level integration issues well before physical system integration. This approach of incremental end-to-end validation and verification throughout the lifecycle in parallel with system construction leads to major cost savings through rework reduction (Figure 9).

5.3 Approach

Our approach to the development of an AFT method has three components:

1. selection of an architecture notation that supports multi-dimensional analysis of architecture models, enhancing its architecture fault modeling capability, and prototyping tool support for the enhancements
2. development of an AFT method for the efficient creation of architecture fault models and their analysis in an incremental fashion
3. application of the AFT method to actual customer systems

The SAE AADL standard and its Error Model Annex standard has provided the best starting point for the AFT work. The Error Model Annex Standard is under revision by the SAE AS-2C (AADL) Standard Committee based on user experiences since its original publication in 2006 [SAE 2006]. This provided an opportunity to immediately impact a practitioner community by incorporating results from the AFT project into the draft Error Model Annex standard revision.

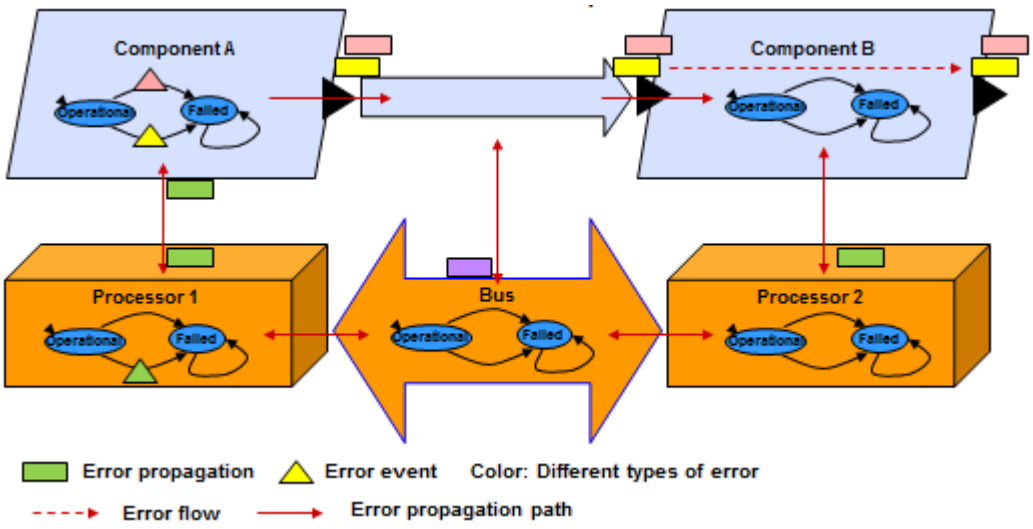

Figure 10: Composable Architecture Fault Models

AADL supports composability through its support for component types (component spec sheets) with multiple component implementations (component blue prints) composed of subcomponents, and the ability to instantiate each multiple times. The Error Model Annex supports composability

- through reusable error type and behavior specifications
- by associating error specifications with component types and implementations
- by leveraging the architecture structure and connection topology to identify potential error propagation paths within the software, within the hardware, and between the two (see Figure 11).

This support of composability leads to support for different aspects of safety analysis, such as Functional Hazard Assessment, Failure Mode and Effects Analysis, and Common Cause Analysis.

Our technical contributions to the Error Model Annex Standard revision are

- the introduction of a multi-set type system,
- a common error propagation ontology
- support for compositionality of error behavior.

The error type system allows users to define error types and associate them with error events, error behavior states, and error propagations. Error types can be defined in a type hierarchy, for example, an error type *TimingError* may have a subtypes *Early* and *Late*. Subtypes within the same type hierarchy are considered to be mutually exclusive. Error types can be grouped into type sets, effectively power sets over the types in each of the type hierarchies named in the type set. For example, when associated with an error propagation, the type set *{TimingError, ValueError}* represents all possible combinations of timing and value errors, such as the timing error propagation *{Late}*, the value error propagation *{OutOfRangeValue}*, or an error propagation that is the result of a late message with corrupted data *{Early,CorruptValue}*.

The error type system provides a concise way of specifying component error behavior and error propagation behavior between components. This allows for specification of outgoing error propagations and of incoming error propagations as well as specification of error types that are expected not to be propagated. By performing type checking on the specified error type sets, we discover inconsistencies in the specification of error propagation contracts and assumptions, such as propagation of an error type to another component that does not expect the error type.

Our second contribution, an error propagation ontology, draws on work by Bondavalli, Powell, and Walter [Bondavalli 1990; Powell 1992; Walter 2003]. It provides a set of error type hierarchies that are commonly encountered in error propagation between interacting components. Each type in the type hierarchy has a formal specification that is based on the concept of service with a service provider sending a sequence of service items. Figure 10 shows service-related-error types, omission and commission of the service as a whole, as well as omission and commission of individual service items.

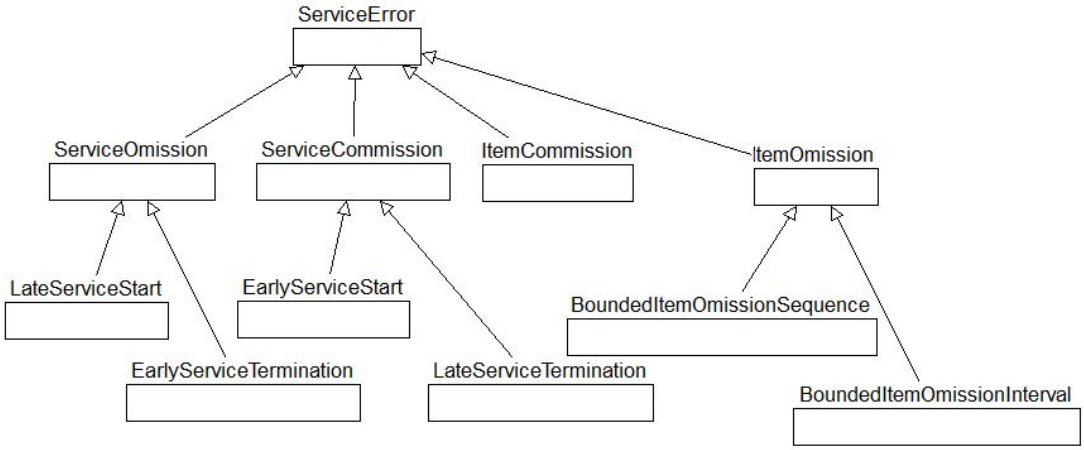

Figure 11: Service-Related-Error Types

Two type hierarchies represent service-item-specific errors: *Value Error*, and *Timing Error*. Value errors fall into locally detectable *Benign Error*, such as out of range, and *Subtle Error*, which require additional information such as CRC to be detectable. Timing errors falls to *Early Delivery* and *Late Delivery*. Two error type hierarchies represent error related to sequences of service items: *Rate Error*, and *Sequence Error*. Rate errors relate to the expected service item rate, namely *High Rate*, *Low Rate*, and *Variable Rate*. Sequence errors are *Out Of Order* and *Value Change Bound*, that is, exceeding a maximum acceptable value change between successive service items, such as set points to a control system.

One type hierarchy addresses errors related to replicated system operation. *Replication Error* divides into *Asymmetric* aka. *Inconsistent Omission*, *Timing*, and *Value*, with inconsistent value further refined into *Inconsistent Approximate Value* and *Inconsistent Exact Value*.

Each error type has a precise specification, for example, Item Omission is defined as XXX. The error propagation type ontology is becoming part of the revised Error Model Annex Standard. Users can adapt the ontology to a particular domain by defining aliases, such as *No Power* for

Service Omission, and extend it through additional error subtypes and additional type hierarchies as needed.

Our final contribution to the revised Error Model Annex is support for compositionality of error behavior. This means a component in a system hierarchy has two error behavior specifications: an abstracted error behavior model and a composite error behavior model. The abstracted error behavior model includes incoming and outgoing error propagations, the component as error source, and error behavior state transitions to reflect the impact of activated faults and propagations and their expected mitigation. The composite error behavior model defines the error behavior states of a component in terms of the error behavior states of its subcomponents. This hierarchical composition allows for fault tree analysis, and when annotated with occurrence probability properties, for reliability and availability analysis.

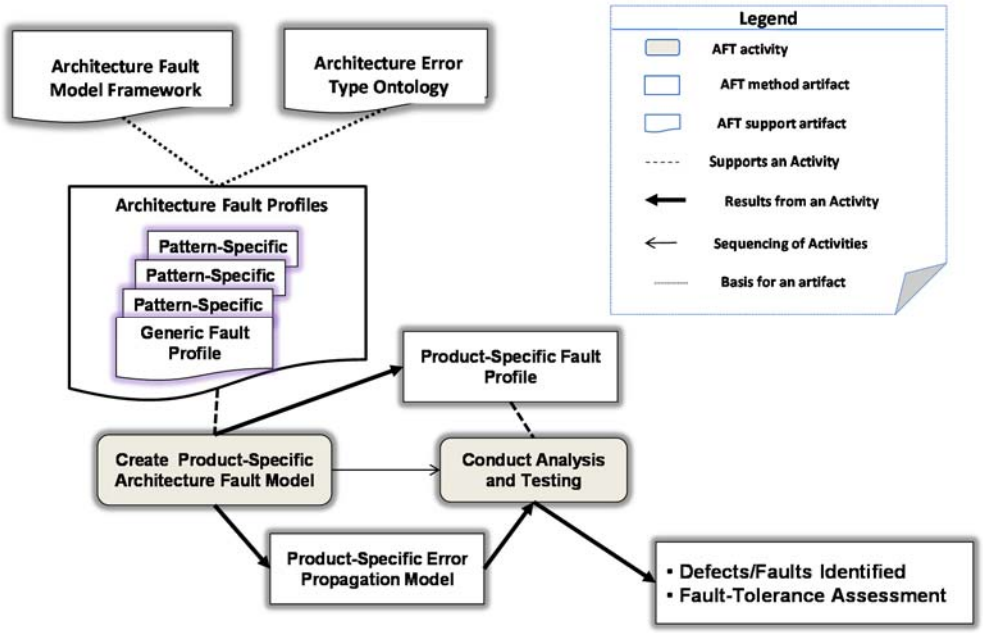

Figure 12: AFT Method

The AFT method has its roots in the Virtual Upgrade Validation (VUV) method [DeNiz 2012]. This method codifies several root cause areas of software-induced system-level errors and utilizes AADL to identify potential mismatches in expectations between the embedded application software due to changes in the hardware/OS platform. The AFT method adds the concept of reusable architecture fault profiles for specific architecture patterns, utilizing the AADL-based architecture fault model framework and the ontology of error propagation types and component intrinsic error types (illustrated in Figure 12). We defined one generic fault profile, and fault profiles for

- the sampled processing pattern, for example, found in control systems
- the interacting state machine pattern, found in interaction and coordination protocols and systems with multi-modal subsystems
- the replication pattern, found in fault-tolerant systems

This initial set of patterns was sufficient for the application of the AFT method to several actual customer systems. As will be elaborated in the results section, we were able to discover unhandled faults, faults that are difficult to test for, and testable faults earlier in development. We also started to investigate a strategy for reprioritization of existing tests by taking into account fault discovery through model-based architecture analysis, and the identification of new tests to validate assumptions made in the architecture fault models and analyses.

5.4 Collaborations

The SEI team collaborated with Brendan Hall and Kevin Driscoll of Honeywell Technology Center in the application of architecture fault models to the verification of safety-critical networks, with Luke Thomas and Julie Hawk of Aero Engine Control in the application of AFT to an engine control timing issue and the evaluation of alternative solutions. The SEI team also collaborated with members of the SAE AADL committee to incorporate the enhancements into the revised Error Model Annex draft standard, including Myron Hecht of Aerospace Corp., Bruce Lewis of the U.S. Army AMRDEC, Brian Larson of Multitude Corp., Steve Vestal of Adventium Labs, Serban Gheorghe of EdgeWater Computer Systems, Jerome Hugues of the Institute for Aeronautic and Aerospace Engineering (ISEA), Thomas Noll of the University of Aachen (RWTH), and Julien Delange of the European Space Agency (ESA).

5.5 Evaluation Criteria

The AFT project was to develop a framework of composable fault profiles that lead to the efficient creation of analyzable architecture fault models, and to develop and apply the AFT method to an actual customer system. In this pilot application, we were to track the types and number of errors we were able to discover earlier in the lifecycle and compare the data to customer project data using existing test practices. A quantitative success criterion was to show considerable reduction in rework cost. A final criterion is that the work finds its way into actual practice.

We have been able to apply AFT to four application system scenarios. We have been able to identify the root cause of a hard-to-test-for timing issue and evaluate two proposed solutions. Industry data shows that the cost of detecting and correcting an architecture issue during system integration is 100-300x the cost of in-phase detection and correction—turning a hypothetical $10,000 investment in architecture fault model analysis and in-phase correction into a $1-3M cost savings. Our application of AFT is leading to follow-on customer activity in the use of architecture fault modeling and analysis and into the incorporation of relevant architecture fault modeling concepts into the revision of an international industry standard.

5.6 Results

In an avionics subsystem, we are able to demonstrate that a major hazard encountered in an aircraft after several years of operation can be captured and identified in an architecture fault model. In the original system, the flight management system (FMS) was designed to handle service omission by the EGI (indicated as *NoData* in Figure 12). The FMS was to operate as fail silent (i.e., operate correctly without causing a stall unless it stopped working). Later it was

discovered that a transient hardware error of two boards, that did not meet specification, touched under certain vibration condition, corrupting airspeed data sent to the FMS. An architecture fault model could have discovered the issue earlier than actual flight operation in two ways. First, the ontology helps identify error propagation types to be considered and explicitly recorded in the model as expected to be propagated or not expected to be propagated. We would have discovered that the System Safety Assessment (SSA) report was silent on propagation of corrupted airspeed data. Second, once an engineer encounters an instance of the EGI with such transient error behavior in lab tests, that engineer records the observation as an update to the architecture fault model of the EGI, which, when reintegrated with the FMS, would have resulted in an analysis report highlighting an inconsistency in the outgoing and incoming error propagation specifications (see bottom left of Figure 13).

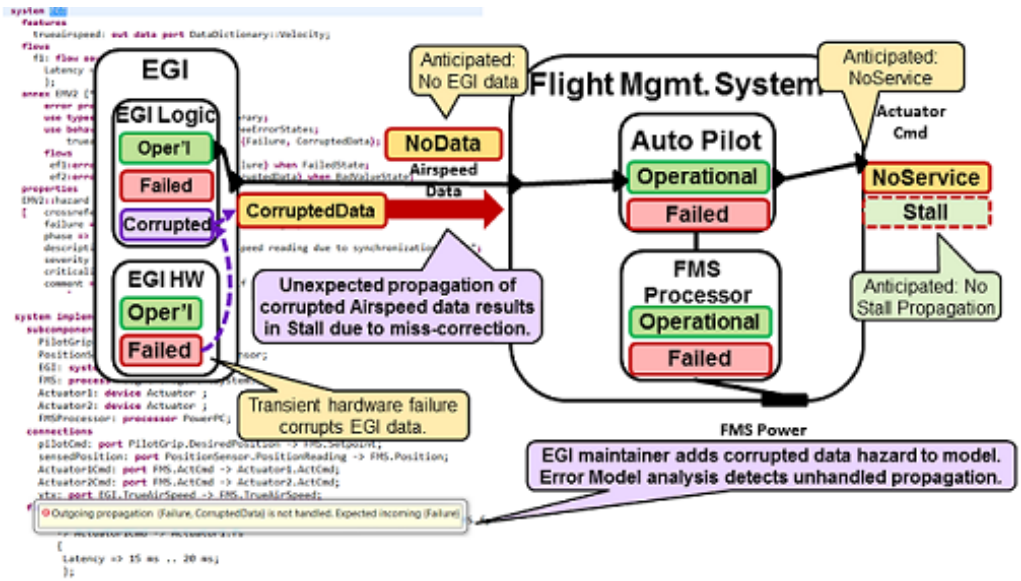

Figure 13: Impact Analysis of Unhandled Hazard

The second application of the AFT method was to an engine control system, in which a timing issue was causing a stepper motor to miss some steps. This problem was incidentally discovered through hardware in the loop testing without a test specifically designed to discover such issues. Initially the customer engineering team had proposed a solution to correct the problem by managing the jitter, and later a hardware solution by buffering commands sent to the stepper motor control interface. Using the pattern-based architecture fault profiles, we were able to quickly represent the system and identify the root cause of the step loss, namely excessive jitter due to control signal output at highly variable control task completion time, quantify acceptable jitter, and assess whether either of the two proposed solution was actually addressing the problem.

The third application of the AFT method was to improve on a verification effort for a safety-critical avionics network. The AADL virtual bus concept lent itself to capturing the network as a set of layered protocols on top of hardware. The error model was used to specify the services each protocol layer offered in terms of tolerating faults from the network hardware, including

Byzantine faults, and in terms of assumptions the protocol made on lower level protocols about masking certain hardware errors and assuring time alignment of replicated data being transmitted. The particular network supported dual channel operation of application systems that provided detection of asymmetric value and omission errors, mapping them into symmetric omission error propagation to the receiver pair. The error-model-based specification allowed us to investigate different use cases for the replicated application system. We could analytically determine that a dual-redundant application system could not operate in one channel standby mode. It would appear as an asymmetric omission error propagation to the network, causing it to propagate omission on both channels. We could also investigate whether the application system was communicating symmetric value errors, which would not be detected and mitigated by the network protocol, but passed through to the recipient pair. The layered architecture model allowed the customer to pursue a more incremental approach to verification instead of a single handcrafted formal specification encompassing the whole protocol.

We investigated the interaction of operational modes with failure modes in a subsystem with the objective of providing a specification of the nominal operational mode behavior expressed through AADL modes and transitions, separate from the failure mode specification expressed through error behavior states and transitions. This led to an enhancement of the Error Model Annex, which allows for automatic generation of a combined behavior model reflecting operational mode behavior under the various failure modes. We did so for a dual redundant flight control system with primary/backup operation in non-critical flight phases, and duplex operation in critical flight phases.

One of the SEI team members has incorporated the improvements proposed as the result of this project into the revised Error Model Annex draft standard. The document is currently in review for ballot consideration in the Spring of 2013.

Finally, we are integrating the AFT work with the Value-Driven Incremental Development (VDID) initiative to demonstrate the value added of AFT over current testing practice by comparing the faults discovered through the use of architecture-centric model-based analysis against existing project data with one of our collaborators. In this context, we are also establishing a measure of increased confidence gained through the evidence produced by the application of AFT.

5.7 Publications and Presentations

Peter Feiler. “Analyzing Fault Tolerant Architectures with the SAE AADL Error Annex,” *SAE 2012 Aerospace Electronics and Avionics Systems Conference*. Phoenix, AZ, U.S.A. Nov. 1, 2012.

Feiler, Peter. “Analytical Architecture Fault Models,” Invited Keynote Talk at the *3rd Analytic Virtual Integration of Cyber-Physical Systems Workshop*, San Juan, Puerto Rico, Dec. 4, 2012. Co-located with Real Time System Symposium (RTSS 2012).

Feiler, Peter. “Architecture Fault Models for Resilient Systems,” Dagstuhl Seminar on *Engineering Resilient Systems: Models, Methods, and Tools.* Wadern, Germany, Jan., 2013.

Feiler, Peter. *SAE Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model Annex V2,* Revised Error Model Annex Draft document, currently in review by the SAE AS-2C (AADL) standard committee for ballot in Spring 2013.

Gluch, David, McGregor, John, Hudak, John, Feiler, Peter, and Brownsword, Lisa. *The Architecture-Focused Testing Method,* SEI Technical Report, 2013.

Peter Feiler, John McGregor, David Gluch, John Hudak, Lisa Brownsword, "Architecture Fault Modeling Case Studies," SEI Technical Report, 2013.

5.8 References

[Bondavalli 1990]
Bondavalli, A. and Simoncini, L. *Failure Classification with Respect to Detection, Specification and Design for Dependability*. Esprit Project N3092 (PDCS: Predictably Dependable Computing Systems). First Year Report, May 1990.

[Dabney 2003]
Dabney, J. B. *Return on Investment of Independent Verification and Validation Study Preliminary Phase 2B Report.* NASA, 2003.

[DeNiz 2012]
DeNiz, D., Feiler, P., Gluch, D., and Wrage, L., *A Virtual Upgrade Validation Method for Software-Reliant Systems* (CMU/SEI-2012-TR-005). Software Engineering Institute, Carnegie Mellon University, 2012. http://www.sei.cmu.edu/library/abstracts/reports/12tr005.cfm

[FAA 2002]
FAA Certification Authorities Software Team (CAST). *What is a* Decision *in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?* Position Paper CAST-10, 2002.
http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf

[Feiler 2009]
Feiler, Peter H. "Challenges in Validating Safety-Critical Embedded Systems," 09ATC-0271. *Proceedings of SAE International AeroTech Congress*. Seattle, WA, Nov., 2009. SAE International, 2009. https://www.sae.org/technical/papers/2009-01-3284.

[Feiler 2012]
Feiler, P. and Gluch, D. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, Sept. 2012. ISBN-10: 0-321-88894-4.

[Galin 2004]
Galin, D. *Software Quality Assurance: From Theory to Implementation*. Pearson/Addison-Wesley, 2004.

[GAO 2008]
General Accounting Office. *DOD's Goals for Resolving Space Based Infrared System Software Problems Are Ambitious* (GAO-08-1073). 2008. http*://www.gao.gov/new.items/d081073.pdf*

[Jackson 2007]
Jackson, Daniel, Thomas, Martyn, and Millett, Lynette I., eds. *Software for Dependable Systems: Sufficient Evidence?* National Research Council of the National Sciences, 2007.

[Leveson 2004]
Leveson, Nancy G. "The Role of Software in Spacecraft Accidents." *Journal of Spacecraft and Rockets 41*, 4 (July 2004): 564–575.

[NIST 2002]
National Institute of Standards and Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Planning Report 02-3). NIST, 2002.

[Powell 1992]
D. Powell, "Failure Mode Assumptions and Assumption Coverage," *22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*. Boston, MA, July 1992.

[Redman 2010]
Redman, D., Ward, D., Chilenski, J., and Polari, G. "Virtual Integration for Improved System Design," *Proceedings of The First Analytic Virtual Integration of Cyber-Physical Systems Workshop*, San Diego, CA, November 2010.
https://wiki.sei.cmu.edu/aadl/images/d/de/SAVI_Virtual_Integration-AVICPS2010.pdf

[Richardson 2005]
Richardson, D., Dias, M., and Muccini, H. "Towards software architecture-based regression testing." *Proceedings of the 2005 Workshop on Architecting Dependable Systems.* Yokohama, Japan, June–July 2005.

[SAE 2006]
Society of Automotive Engineers International. *Architecture Analysis & Design Language (AADL) Annex Volume 1: AADL Meta model & XML Interchange Format Annex, Error Model Annex, Programming Language Annex.* SAE International Standards Document AS5506/1, 2006.
http://standards.sae.org/as5506/1

[SAE 2011]
Society of Automotive Engineers International. *Architecture Analysis and Design Language (AADL) Annex Volume 2*: *Behavior Annex, Data Modeling Annex, ARINC653 Annex.* SAE International Standards Document AS5506/2, 2011. http://standards.sae.org/as5506/2

[SAE 2004/2012]
SAE International. *Architecture Analysis & Design Language v2.1* (Standard AS5506B). SAE International, 2004–2012. http://www.sae.org/technical/standards/AS5506B

[Walter 2003]
Walter C. and Suri, N. “The Customizable Fault/Error Model for Dependable Distributed Systems.” *Theoretical Computer Science 290* (2003):1223–1251.

[White 2008]
White, Lee, Jaber, Khaled, Robinson, Brian, and Rajlich, Václav. “Extended firewall for regression testing: an experience report,” *Journal of Software Maintenance and Evolution: Research and Practice 20,* 6 (Nov/Dec 2008):419–433. John Wiley & Sons, Ltd., 2008.

[Willet 2011]
Willet, Hugh. “Suit filed, chopper blamed in accident that killed 2 Tennessee pilots in Iraq.” Knoxville *News Sentinel.* March 10, 2011. http://m.knoxnews.com/news/2011/mar/10/chopper-accident-2-Tenn-deaths-pilots-iraq/

6 Architecture Decision Making for Rapid Lifecycle Development in an Agile Context

Stephany Bellomo
Robert L. Nord
Ipek Ozkaya
Mary Ann Lapham

6.1 Purpose

There is great pressure to reduce release cycle time—the time from which software requirements are handed off to the development team to the time the capability is deployed on Department of Defense (DoD) software projects. The cycle time for DoD projects is often months or years. Industry and government alike increasingly have been adopting Agile software development practices to reduce lifecycle time [Hotle 2012]. Adopting agile software development practices alone will not address these challenges; more research is needed to reduce lifecycle time through improved architectural decision making. In that vein, we focused this work on identifying success factors for rapid fielding.

Why study success factors for rapid fielding? Many Agile success stories have been attributed to adoption of practices such as increased team communication, collective ownership, frequent customer-visible releases, backlog-driven requirements management, continuous integration, and shorter iterations. Yet, many teams in highly regulated environments, such as the DoD, struggle when adopting Agile or iterative methods. For example, on Agile projects we often see a pattern of high initial velocity for weeks or months due to schedule pressure followed by a slowing of velocity due to stability issues. There is lack of clarity around which factors truly contribute to the ultimate goal of rapidly fielding tested software functionality to its intended end-users [DDR&E 2010] (representing minimally marketable features [Denne 2003]). Our research in this area aims to gain better insight into these disruptive patterns and to gain insight into practices that successful practitioners apply to maintain a consistent and rapid fielding pace.

6.2 Background

New acquisition guidelines from the DoD emphasize reducing system lifecycle time for software development. For example, the DoD chief information officer (CIO)'s office released a 10 Point Plan to reform DoD information technology (IT) which stresses, "delivering useful capability every 6 to 12 months to reduce risk through early validation to users" [DoD 2012]. Program managers and acquisition executives are responding to this plan by applying industry practices, such as Agile methods. At the same time, related DoD guidelines encourage system development via a more open, modular software architectural style, such as loosely coupled service-oriented architectures. In this environment, the impact of architectural decisions becomes more critical

because they influence goals such as rapid feedback, ability to respond to change, and team communication.

Architectural decisions also influence agile development properties, such as the length of an iteration, the allocation of stories to iterations, and the structure of the teams and their work assignments. What is needed, therefore, is research on eliciting and employing these properties and determining their relative importance in promoting rapid lifecycle development. To address this need, our research study focuses on the implications of decisions made over the course of the software and systems lifecycle to validate the following hypotheses:

- The implications of the early decisions often surface in the final stages of the lifecycle. These often surface as disruptive patterns such as bug fixing sprints and major rework efforts. A better understanding is needed of the root cause for these disruptive patterns and the practices that positively or negatively influence them. Value may be gained by looking at programs that have experienced issues when trying to rapidly field software, such as significant reduction or increase in defects, significant reduction or increase in velocity, and significant reduction or increase in integration issues.

- The decisions made during the pre-engineering and manufacturing development phases have an impact throughout the lifecycle. Acquisition programs lack the techniques needed to elicit and analyze the implications of such fundamental architectural decisions. For example, architectural decisions that relate to decomposition and dependencies influence teaming structure, the capability to rapidly and confidently deliver features, and the ability to rapidly integrate new components, among other factors. Improved methods are needed to analyze architectural tradeoffs with respect to long-term lifecycle costs and rapid fielding goals.

6.3 Approach

Our approach was an empirical study of the factors in Agile projects that influenced lifecycle time. In this study, we interviewed software professionals from DoD and commercial organizations and analyzed the results. We based the interview structure around a guiding question: ***What are factors that enable or inhibit rapid fielding on your project?*** We followed a structured approach to capture and analyze the data. As we conducted interviews, we emulated Glaser's conceptual approach to grounded theory, which aims to let the theory emerge from the data [Dick 2005] while leveraging some of the structured steps described by Strauss [Barney 1995; Corbin 2008]. We leveraged grounded theory-based empirical study concepts for this work, including data collection, note taking, coding, memoing, and sorting. The general steps of our approach are summarized below:

- data collection
- memos and indicators
- coding (deriving concepts and categories)
- saturation and concept strength

This research method produced a set of concepts and categories used to organize information. The diagram below shows the conceptual relationship between categories, concepts, and indicators

along with an accompanying example. The example shows four indicators supporting the concept "Prototyping with quality attributes," which is within the "Practice" category. An elaborated example can be found in the study report.

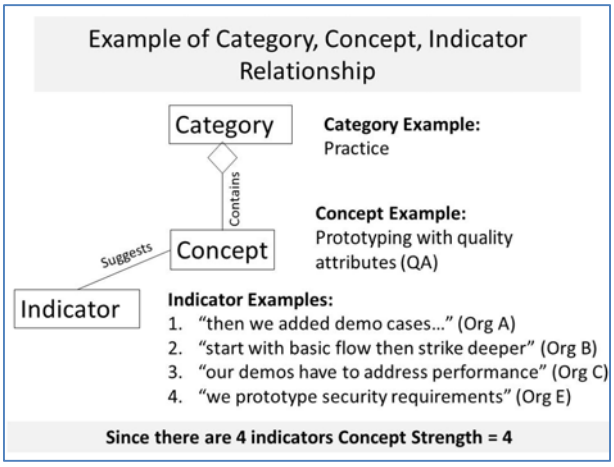

Figure 14: Example of Category, Concept, Indicator Relationship

6.4 Collaborations

We collaborated with Dr. Philippe Kruchten, University of British Columbia, who provided guidance on conducting the empirical study and architectural decision modeling. Software professionals from five government and commercial organizations provided us with the technical input to the empirical study. We also collaborated with Dr. Kevin Sullivan from the University of Virginia to explore, as future work, the potential application of dependency structure matrix visualization and analysis to modeling architecture decisions in complex, collaborative, agile team structures.

6.5 Evaluation Criteria

The ability to identify influential factors improves the ability of projects to make informed decisions and to avoid disruptive patterns that often result in delays in fielding software. Therefore, the evaluation criteria for the project are focused on the following outcomes for evaluation:

- a study that leverages semi-formal, structured data analysis methods to produce a set of factors (in this case practices) contributing to rapid fielding as well as insights as to how those practices are incorporated into an incremental lifecycle
- a recommendation for improved methods needed to analyze architectural tradeoffs with respect to long-term lifecycle costs and rapid fielding goals

6.6 Results

We interviewed project teams with incremental development lifecycles from five government and commercial organizations to gain a better understanding of factors that enabled or inhibited rapid

fielding on their projects. We also explored how Agile projects deal with the pressure to rapidly deliver high-value capability while maintaining project speed (delivering functionality to the users quickly) and product stability (providing reliable and flexible product architecture). The most important discovery made through this work is that practitioners don't apply pure Agile or architecture practices alone. Architecture practices are often combined with other practices, such as Scrum practices, to avoid disruptive scenarios such as bug-fixing sprints and redesigns.

6.6.1 Interviewee Profile

We interviewed project teams that were developing software systems. The project characteristics are summarized in Table 6.

Table 6: Project Profile

Project ID	Time in Production	Release Management approach	Type	Product size	Team Size	Sprint length / Prod Release cycle
A-P1	Pre-release	Scrum	case management system	<10M SLOC	10-20	2 weeks/ TBD
B-P1	12 years	Scrum	analysis support system	<10M SLOC	10-20	2 weeks/ 6 months – 1 yr.
C-P1	3 years	Scrum	training simulator	1-10M SLOC	>30	4-6 weeks/ 2-6 months
D-P1	Pre-release	Scrum	enterprise information sharing portal	TBD	>30	2 weeks/ TBD
E-P1	12 years	Scrum	doc management system	10-20M SLOC	9	2 weeks/ 1-3 months
E-P2	14 years	Incremental (prior to Scrum)	SQLWindows tool	<10M SLOC	10-15	N/A/ 1 year
E-P3	8 years	Incremental (prior to Scrum)	hardware controller	<10M SLOC	5	2 weeks/ 2months
E-P4	1.5 years	Scrum	COTS customization	10-20M SLOC	6	2 weeks/ 3 months

6.6.2 Speed-Triggers-Stability Scenario

Through these interviews we observed that projects with a business goal to deliver capability rapidly must deal with a natural tension between the pressure to get functionality out quickly (speed) and the desire for a reliable, stable and flexible product (stability). We see evidence of this tension in our work with Agile projects. For example, we often see a pattern of high initial velocity for weeks or months due to schedule pressure followed by a slowing of velocity due to stability issues. This slowing in velocity negatively impacts business stakeholders as rate of capability delivery slows.

The nature of the tension is influenced by the software development state. Agile project teams recognize that there is a desired software development state that enables them to quickly deliver releases that stakeholders value (Figure 15) [Bachmann 2012; Leffingwell 2007]. When product development starts, this desired state has not yet been achieved, and teams go through a

Preparation phase focused on getting the infrastructure in place. This involves getting platforms and frameworks, as well as supporting tool environments, practices, processes, and team structures in place to support efficient and sustainable development of features. Once they have achieved the desired state, teams enter into a Preservation phase where the infrastructure is in place and they work to achieve a consistent velocity (avoiding major disruptions to speed). In this phase, the goal is to maintain balance. For example, it is important to neither over-optimize the supporting development infrastructure nor to quit working on it.

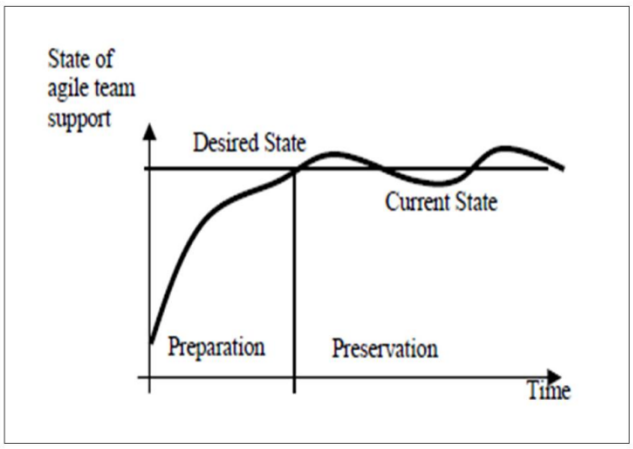

Figure 15: Software Development Support for Teams over Time

As we spoke with organizations, a scenario emerged that shed some light on how practitioners apply practices to stay within acceptable range of desired state. We refer to this as the Speed-Triggers-Stability scenario (Figure 16).

We explain this scenario by walking through the steps S1-S4 illustrated in the figure. (S1) Due to business needs, there is a significant pressure to field capability rapidly. We refer to this as a "Focus on speed." We also note that at S1 the project is within acceptable tolerance of "desired state." (S2) A stability problem occurs such as embarrassingly poor system performance during a stakeholder demonstration. The problem puts the project outside the acceptable tolerance of "desired state," which triggers a focus on improving stability to get back into acceptable range. (S3) The project team responds to address the problem by applying a single practice or combining practices. If the problem is very visible to stakeholders, the team responds quickly and the resulting practice change is incorporated into its software development support structure without major project disruption. We refer to this as the *incremental response cycle*.

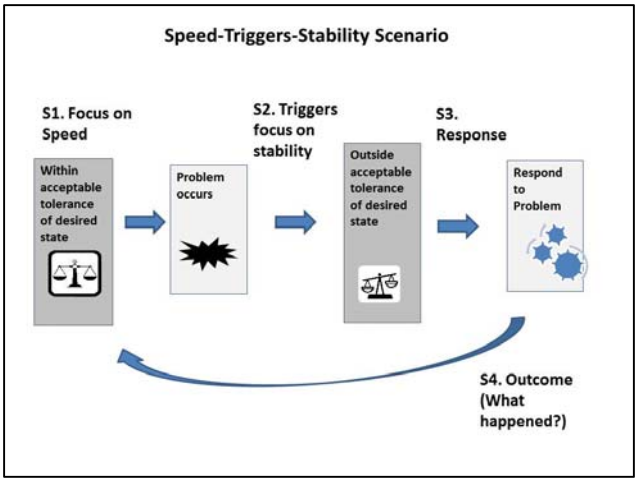

Figure 16: Speed-Triggers-Stability Scenario

If the problem is not visible (such as architectural problems that only the development team can observe), there may not be enough incentive on the business side to expend development effort in fixing it, so it is put off until the problem grows to such an extent that it becomes visible and can be put off no longer. We refer to this as the *"big bang"* response cycle. The big bang can result in significant disruption and effort. (S4). After the team responds, the outcome is either better or worse. If the outcome is good we observe that it will bring the project back toward the acceptable tolerance of desired state (for that particular problem).

Table 7: Summary of Enabling Practices within Acceptable Range of Desired State

Summary of Enabling Practices: Within acceptable range of Desired State
Vision Doc/roadmap (long-term release planning)
Scrum collaborative management style
Prototype/demo (community previews)
External Dependency Management
Use of collaborative tools foster communication
Scrum status meeting
Test-driven development
Continuous integration
Small dedicated team and limited scope
Incremental release cycle
End user involvement
Evolutionary design and documentation
Retrospective and periodic design reviews
Use of standards and ref models
Configuration Management
Story points for productivity tracking
Requirements to design traceability
Proof of concept (for unproven tech)

Pair programming

6.6.3 Summary of Enablers

This subsection summarizes the enabling practices that emerged from our interviews. We define *practice* as a repeatable way of accomplishing an activity related to software product development or delivery; for example, we consider *prototyping* (to validate requirements and gather user feedback) a practice. We observe that enabling practices fell into two groups. The first group, shown in Table 7, represents the set of practices that one would expect to find at the top of the list in any discussion with project teams about enabling Agile practices. The practices in this table and the ones that follow are ordered by frequency of occurrences.

These practices were typically described as enablers when projects were going well or "within acceptable range of desired state." The next group of practices, shown in Table 8, emerged as practitioners gave examples describing how they dealt with problem situations where they were "outside of acceptable range of desired state." We noted that often practitioners would not just apply an Agile or architecture practice, but rather would combine practices in creative ways to address the problem. We identify these combined practices in bold below.

Table 8: Summary of Enabling Practices Outside of Acceptable Range of Desired State

Summary of Enabling Practices Outside of Acceptable Range of Desired State
Release planning with arch considerations
Prototype/demo with quality attribute focus
Release planning with Joint prioritization
Test-driven development with quality attribute focus
Dynamic organization and work assignment
Release planning with legacy migration strategy
Roadmap/Vision with external dependency mgmt
Root cause analysis to identify architecture issues
Dedicated team/specialized expertise for Tech Insertion
Technical debt monitoring with quality attribute focus
Focus on strengthening infrastructure (runway)
Retrospective and periodic design reviews
Use of standards and ref models
Backlog grooming
Fault handling or performance monitoring
Vision document with architecture considerations

6.6.4 Enabling Practice Examples

In this section, we describe representative examples of the combined practices in more detail using the speed-triggers-stability scenario. In most of these examples an Agile practice is in use when a problem pushes the team outside the acceptable range of desired state. The experienced practitioner augments the Agile practice with another practice to address the problem with

minimal disruption to capability delivery. Abbreviated summaries of three combined practices are provided below.

Release Planning with Architecture Considerations. This practice extends the feature release planning process by adding architectural information to the feature description document and was provided by an architect from Organization C. The organization had adopted the Scrum release planning management process whereby the product owner prioritizes features ensuring a focus on speed [Schwaber 2011]. After the backlog is prioritized, the product owner hands the list of prioritized backlog to the developer team that develops the design for implementing the features. As the project migrated from a centralized development model to a geographically distributed work model with increasing focus on speed, the members realized that teams needed to work in parallel to meet schedule demands. The project eventually ran into challenges impacting speed because there was not enough architectural definition in the feature documentation to allow the teams to "go off and work independently." The team responded by augmenting the existing release planning practice, attaching a minimal design document containing architectural design information they called a "design memo" to the feature description document, prior to release prioritization. By extending release planning with architecture information the team was better able to identify tradeoffs to support parallel development. This practice was a widely supported practice based on the interview data.

Test-Driven Development with Quality Attribute Focus. This practice merges test-driven practices, such as automated test-driven development and continuous integration, with a focus on runtime qualities such as performance, scalability, and security. This example comes from Organization E. The team had developed a a set of test cases that effectively tested business functionality. They had a fixed deployment deadline, so due to schedule pressure, they did not have time to develop quality attribute-related test cases, such as security-related test cases. Late in the development lifecycle, the team became nervous that the software would not pass assurance testing and that late discovery of security vulnerabilities could cause schedule delay. Team members responded by removing some of the planned features from the release and refocusing that effort on shoring up the security-related design. As they did this, they also incorporated additional security-related test cases into their regression test case suite. By extending test-driven development to incorporate security considerations (a quality attribute focus) the team was able to improve confidence that security requirements were addressed, avoiding late discovery of schedule-impacting problems. All the organizations we spoke with gave examples supporting this practice. We also noted that several of the organizations appear to be struggling to make their test processes fit into a rapid release cycle (particularly within a sprint). We discuss this issue further in the Inhibitors section.

Technical Debt Monitoring with Quality Attribute Focus. The metaphor of technical debt is used to refer to accumulating degradation of quality due to intentional and unintentional shortcuts [Kruchten 2012]. During interviews we heard stories of problems due to unchecked technical debt leading to stability challenges and big bang response cycles. With this practice, practitioners described steps they are taking to put in place mechanisms to monitor technical debt. An architect with Organization B provided this example. In order to speed up development time, the team purchased a COTS tool to enable team members to easily add new fields to web pages. This tool

put in place a layer between the database and the application layers of the system. This appeared to be a change that would promote stability by encapsulating other layers from the database layer. The problem is that now each field that is added to a web page through the tool creates a new XML-based query. So, now there are a large number of these query interfaces to manage. In addition, making changes to the COTS tool requires a special skill set, so requests for changes back up waiting to be made. What seemed like a positive change resulted in a negative impact to modifiability. While team members would like to change this situation, they have difficulty making a case for change because the problem is only visible to the development team. Therefore, this problem still exists today. The team is waiting for a business-driven opportunity to try to work it in or until team performance drops to a point that is unacceptable to the business side. Discussion of technical debt is part of the everyday vocabulary for the projects of organization B and C (the more mature projects). Both these projects described how they often "hide" architectural change unrelated to features during feature development. This lack of transparency can result in incorrect productivity measures as well as unanticipated schedule impacts. The projects described how they were in the early stages of working on ways to better measure and monitor technical debt. They expressed the belief that if they were able to make technical debt more visible to stakeholders, they could avoid the potentially costly and disruptive big bang cycle.

6.6.5 Summary of Inhibitors

In this subsection we summarize inhibitors to rapid fielding collected during our interviews. Table 9 lists the inhibitors. We focus the discussion primarily on concepts with category strength of 2 or better shown in the white portion of the table.

Table 9: Summary of Inhibitors

Summary of Inhibitors
Desire for features limits requirements analysis or stability-related work
Slow business decision, feedback or review response time
Problems due to challenges with external dependency management
Stability-related effort not entirely visible to business
Limitations in measuring arch tech debt
Inadequate analysis, design or proof-of-concept
Testing practices with deficiency in quality attribute focus
Poor testing consistency
Runway or infra limitations
Resource limitations
Poor configuration Management limits reversibility
Over-dependency on Architect for arch knowledge
COTS selected limits flexibility
Organizational standards limit design options
Incompatible milestone lifecycles
Business didn't buy into Scrum
Arbitrary backlog grooming

Personnel issues limit ability to track individual productivity

We found that major inhibitors to rapid fielding generally fell into categories of constraints or practice deficiencies. As we analyzed the inhibitor data we saw relationships between some of the inhibitors and combined enabling practices. For example, when practitioners from Organizations B and C described incidents of applying the practice of technical debt monitoring with quality attribute focus they also often mentioned these influencing constraints: *Desire for features limits requirements analysis or stability-related work, Stability-related effort not entirely visible to business* and *Limitations in measuring architectural technical debt.* Taken together, this begins to paint a commonly observed picture that technical debt appears to build until refactoring will no longer addresses the problem.

Another constraint, *Slow business decision, feedback, or review response time*, is also at the top of the list. Several organizations said that they wasted a lot of time waiting on important management decisions and enterprise certification processes over which they had no control. Organizations A, B, C, and D all gave examples of this. Because constraints such as these appear to significantly influence rapid fielding, a future research interest area for us may be looking deeper into the role these constraints play in inhibiting project teams from achieving desired state.

We also noted several high-ranking inhibitors we categorized as practice deficiencies. For example, all of the organizations, except E, said that they were struggling with testing-related problems. Organization D struggled with developing test cases for complex and unpredictable functionality, such as user interaction, within a sprint or release cycle. We called this inhibitor *Inconsistent testing practices and/or deficiency in quality attribute focus*. Several teams said they wanted to fully leverage Agile test-driven development practices; however, the team's testing expertise and tool knowledge was limited. These inefficiencies in testing practices often resulted in inconsistency in applying testing practices. In other words, often there just was not enough time to do all the testing they said they knew they needed to produce the highest quality product. For example, Organization B acknowledged the need for performance and scalability testing on its project; however, because these tests take a lot of time, when there is significant businesses pressure they drop these tests. Organizations A and C both said their performance regression tests "take too long" so they conduct them when (and if) they can fit them in. We also note that there may be a relationship between this inhibitor and the high-ranking enabler *Testing practices with quality attribute focus*.

6.6.6 Key Takeaways and Future Work

Through this work we see evidence that software engineers don't apply pure Agile or architecture practices. In practice, they come up with innovative ways to combine practices to allow them to stay within (or get back to) an acceptable range of desired state for their project. We present several examples of the enabling and combined practices in this summary of our study. We observed that practice combinations allow teams to address problems with stability while still focusing on speed.

Business stakeholders prefer to avoid drastic measures that impact capability delivery like bug-fixing sprints or major redesigns (described as the big bang release cycle). We see evidence from

our interviews that improved visibility into technical debt may help projects avoid the disruptive big bang cycle.

Through the interviews, we identified several deficiencies that may warrant future attention. Testing appeared to be particularly problematic. The teams we interviewed struggled to develop test cases for complex situations and run quality-attribute focused tests (such as performance tests) within a sprint cycle. In addition, inconsistent test practices and delays due to enterprise processes such as assurance certifications processes also caused many problems. This raises an interesting question that we would like to explore in the future. Are we observing a lack of testing practice discipline or a rational and reasonable response to the need to balance speed and stability? We suggest that this area and implications of certification processes on rapid fielding may be areas for future study.

As an extension of our approach, we are looking at eliciting and modeling dependencies among the factors to improve decision making (with a focus on architecture-related decisions). Capturing key decisions with measurable attributes has only undergone early investigations. Emerging technology (e.g., frameworks) and practices (e.g., continuous integration) do not suffice to deal with integration issues at scale. In this work, we plan to investigate questions such as these: Can early design decisions be modeled in terms of generalizable measurement attributes across projects and taking advantage of dependency analysis? Can such a model provide insights about integration risks and achievement of rapid fielding business goals?

The results of this study appear to be supported by recent work in the community. In a recent blog posting, Ken Schwaber said he would like to change the mindset of **Scrum But** to **Scrum And, explaining that Scrum And** is a path of continuous improvement in software development beyond the basic use of Scrum. He gave this example to illustrate the concept of extending Scrum, "We use Scrum **and** we are continuously building, testing and deploying our increments every Sprint" [Schwaber 2012]**.** The work we did in our study supports the stance that practice extensions are needed and anticipated. Further investigation will reveal which practices combinations may prove most valuable to practitioners.

6.7 References

[Bachmann 2012

Bachmann, F., Nord, R. L., and Ozkaya, I. "Architectural Tactics to support rapid and agile stability." *CrossTalk: The Journal of Defense Software Engineering*, Special Issue on Rapid and Agile Stability. May/June 2012.

[Barney 1995]

Barney, G. and Glaser, I. "A look at grounded theory: 1984-1994." *Grounded theory 1984-1994*, *1* (1995):3-17. Sociology Press, 1995.

[Corbin 2008]

Corbin J. and Strauss, A. *Basics of Qualitative Research- Techniques and Procedures for Developing Grounded Theory, 3rd ed.* Sage Publications, 2008.

[Dick 2005]
Dick, B. *Grounded theory: a thumbnail sketch.*
http://www.uq.net.au/action_research/arp/grounded.html (2005).

[DoD 2010]
Director of Defense Research and Engineering, *Rapid capability fielding toolbox study*, Final Report. March 2010.

[Denne 2003]
Denne M. and Cleland-Huang, J. *Software by Numbers*. Prentice Hall, 2003.

[DoD 2012]
Department of Defense, Ms. Teri Takai. *DoD CIO's 10-point plan for IT Modernization*. 2012.
http://dodcio.defense.gov/Portals/0/Documents/ITMod/CIO%2010%20Point%20Plan%20for%20IT%20Modernization.pdf

[Hotle 2012]
Hotle, M., Norton, D., and Wilson, N. "The end of the waterfall as we know it." Gartner Research, August 20, 2012.

[Kruchten 2012]
Kruchten, P., Nord, R. L., and Ozkaya, I. "Technical debt: from metaphor to theory and practice." *IEEE Software*, 2012, pp. 18-21.

[Leffingwell 2007]
Leffingwell, D. *Scaling Software Agility*. Addison-Wesley, 2007.

[Schwaber 2011 10]
Schwaber K. and Sutherland, J. *Scrum Guidebook*. Scrum.org and Scrum Inc., 2011.

[Schwaber 2012]
Schwaber, K. (blog) "Telling it like it is," http://kenschwaber.wordpress.com/2012/04/05/scrum-but replaced-by-scrum-and/ , April 5 2012.

7 Semantic Analysis for Malware Code Deobfuscation

Cory Cohen
Charles Hines
Wesley Jin
Sagar Chaki
Arie Gurfinkel

7.1 Purpose

Obfuscation of malware code is a serious problem for the Department of Defense and the computing industry in general. Obfuscation techniques hinder manual and automated analysis of malware, increasing the expense of extracting actionable intelligence and thereby adding to detection, mitigation, and attribution efforts. This LENS project focuses on obfuscation techniques that cause sequences of instructions to be difficult for an analyst to understand.

When shown an example of obfuscated assembly code (Figure 17), an experienced malware analyst at the CERT Program took 550 seconds (more than 9 minutes) to determine its basic functionality. This example code, while created explicitly for this demonstration, shows several techniques commonly encountered in malware and represents a realistic scenario. The net effect of the code is equivalent to a single instruction (**`mov ecx, [ecx+4]`**) that, when deobfuscated, an analyst can understand almost immediately.

```
43E401: push 9387D2AF
43E406: mov  edx, F25C92BA
43E40B: pop  eax
43E40C: xor  edx, F21F75B2
43E412: and  eax, 37D28E56
43E418: jmp  43E501
…
43E501: push ecx
43E502: mov  ecx, eax
43E504: push edx
43E505: jnz  43E601
43E506: jmp  43E603
…
43E601: sub  eax, 13828202
43E607: ret  ret
…
43E708: pop  edx
43E709: mov  ecx, [eax+edx]
```

Figure 17: Obfuscated Malware Test

Several-hundred-fold increases in the effort required to analyze malware raises computer and network defense costs for the Department of Defense and prevents it from generating the necessary insights within reasonable budget constraints. This research explores the predominant obfuscation techniques in use by malware authors and proposes approaches to combatting them.

7.2 Background

Significant research has already been conducted on malware authors' obfuscation techniques, along with methods for defeating them. Schwarz, Debray, and Andrews describe the problems that obfuscations present for automated disassembly [Schwarz 2002]. Linn and Debray propose additional techniques for complicating analysis [Linn 2003]. Krügel and colleagues discuss improved methods of disassembling obfuscated malware [Krügel 2004]. Udupa, Debray, and Madou discuss obfuscation and deobfuscation transformations [Udupa 2005]. Cifuentes and Fraboulet describe the application of various program analysis techniques to binary executables [Cifuentes 1997].

Roundy and Miller describe obfuscation techniques in a dozen common packers [Roundy 2012]. They mention that it is valuable to reassemble the deobfuscated result into a format that can undergo additional manual analysis, a technique we feel is critical to improving the integration of deobfuscation into operational malware analysis. Roundy and Miller's work focuses on antidetection packing technologies, while our research is primarily concerned with antianalysis obfuscation in not packed malware. They also make use of the DynInst [Miller 2012] tool, which includes a dynamic component, for analysis, while our approach is purely static and carries no risk of executing malicious code.

7.3 Approach

Our approach to deobfuscation applies multiple semantic transformations to an obfuscated program to produce a new program that is functionally equivalent to the obfuscated program but easier to analyze.

Sufficiently robust reasoning about instruction semantics makes possible many simplifying transformations. A sample transformation might emulate instructions to determine whether a particular predicate is opaque [Collberg 1998] and, if it is, remove the conditional jump from the function. Another transformation might use definition and usage analysis [Allen 1976] to identify and remove instructions that write to values that are never read. Conceptually, a transformation could be as complex as replacing arbitrary sets of instructions with fewer but functionally equivalent instructions.

These semantic transformations may be run in multiple passes and in varying orders to simplify the deobfuscation of malware that applies techniques in sequence. A modular approach to the transformation should improve code maintenance and simplify the addition of new techniques.

To facilitate the integration of deobfuscation into other malware analysis activities, we propose a deobfuscation system that both reads obfuscated executables as input and writes deobfuscated code as output. This ensures that other malware analysis tools that process executable code will be able to load the output from the deobfuscation step for further analysis. An analyst should be able to load the deobfuscated code into the IDA disassembler[12] for further inspection. This integration is even more important in automated malware analysis systems, in which operating on the deobfuscated executable might significantly improve the results of downstream analysis.

---

[12] http://www.hex-rays.com/products/ida/index.shtml

Early in this research effort, we built a functioning prototype of our semantic transformation system using the ROSE compiler infrastructure.[13] ROSE provides capabilities including disassembly, instruction emulation, and control flow graph representation. We expanded on these capabilities by adding definition and usage analysis, dead code elimination, and other techniques specific to deobfuscation. We also implemented a rudimentary method for generating new executables as an output format by building on ROSE's assembly features.

We tested the prototype against several members of the *Ramnit*[14] malware family and hand-validated the deobfuscation results. The test demonstrated that use of the prototype significantly reduced the number of instructions in these functions (by more than 50 percent).

While we were conducting this research, we were also conducting operational analysis for which we wrote a program to address a specific string obfuscation technique. We felt that this program and our semantic transformation prototype represented a successful application of our approach. After building the prototype, we believed that implementing more transformations, testing more broadly, and enhancing the capabilities of our executable generation algorithm were all that remained to create an operational capability.

To our surprise, we found that this effort produced little evidence on which to judge the prevalence of obfuscations, as well as poor consensus on which transformations would yield the greatest benefit for the development effort expended. To better guide the allocation of our efforts, we decided to study obfuscation prevalence in our malware collection.

Conducting this study turned out to be at least as challenging as our prototype deobfuscation efforts. We found that many foundational issues that we took for granted or considered out of scope during the initial effort were now open for discussion and offered no firm evidence to support any particular position. For example, how do you know that the disassembly provided to the transformation was correct? If it was not, is the transformation expected to reduce the complexity of the code? If the code after transformation is simpler, how do you validate that it is still functionally equivalent? What metrics establish that code is obfuscated in the first place, and what degree of simplification is deobfuscation expected to produce?

These inquiries led us to some of the most interesting results of our research, and much of this chapter documents what we learned about obfuscation prevalence, disassembly correctness, and metrics for objectively measuring malware obfuscation.

7.4 Collaborations

A collaborative effort between the SEI's CERT® Program and Research, Technology, and System Solutions (RTSS) Program produced the bulk of this research. Principal researchers Cory Cohen and Charles Hines are senior members of the technical staff in the CERT Program. Principal researchers Sagar Chaki and Arie Gurfinkel are senior members of the technical staff in the RTSS Program. Wesley Jin is a graduate student currently pursuing his PhD in the Electrical Computer

---

[13] http://rosecompiler.org/

[14] http://www.f-secure.com/v-descs/virus_w32_ramnit_n.shtml

Engineering (ECE) department of Carnegie Mellon University. He also worked for the CERT Program during this project.

This work would not have been possible without the assistance and support of Dan Quinlan and Robb Matzke from the ROSE Compiler development team at Lawrence Livermore National Laboratory. Quinlan and Matzke answered numerous questions and made several enhancements to the ROSE compiler infrastructure in support of this research.

7.5 Evaluation Criteria

To develop objective criteria to evaluate deobfuscation success and correctness, we planned to compute basic complexity metrics, such as counting instructions and basic blocks, for the code before and after transformation. For example, having fewer instructions or nodes in the control flow graph would indicate that the code had been deobfuscated effectively. We computed these metrics for the samples processed by our prototype, and the samples showed significant complexity reductions after the deobfuscation transformations. However, what constitutes a reasonable expectation of complexity reductions for a particular sample remains an open question.

Establishing objective criteria for distinguishing between packed and obfuscated code became a significant challenge for this work. Other important pieces of the effort included determining the relative prevalence of obfuscation techniques, in order to prioritize development of different transformations, and understanding trends in malware authorship. Because much of our research ended up focusing on the complexity metrics and the difficulties of collecting them objectively, we will discuss evaluation criteria in more detail through much of the next section.

7.6 Results

7.6.1 Prototype Deobfuscation Tool

Early in this research effort, we developed a prototype system to demonstrate the viability of the overall proposed architecture and provide a base for future deobfuscation work. We built a tool that worked from end to end: reading executables, applying semantic deobfuscation transformations, and writing the results out as a new deobfuscated executable. This executable-in/executable-out paradigm ensured that the tool could be used in the typical operational malware analysis workflow.

To help drive the implementation (and later test) the prototype, we needed obfuscated malware samples that were simple enough that a few transformations would produce a correctly deobfuscated output file. We selected the *Ramnit* family of malware because it was relatively simple and consisted of only a few obfuscated functions. Figure 18 shows a small sample of the *Ramnit* code.

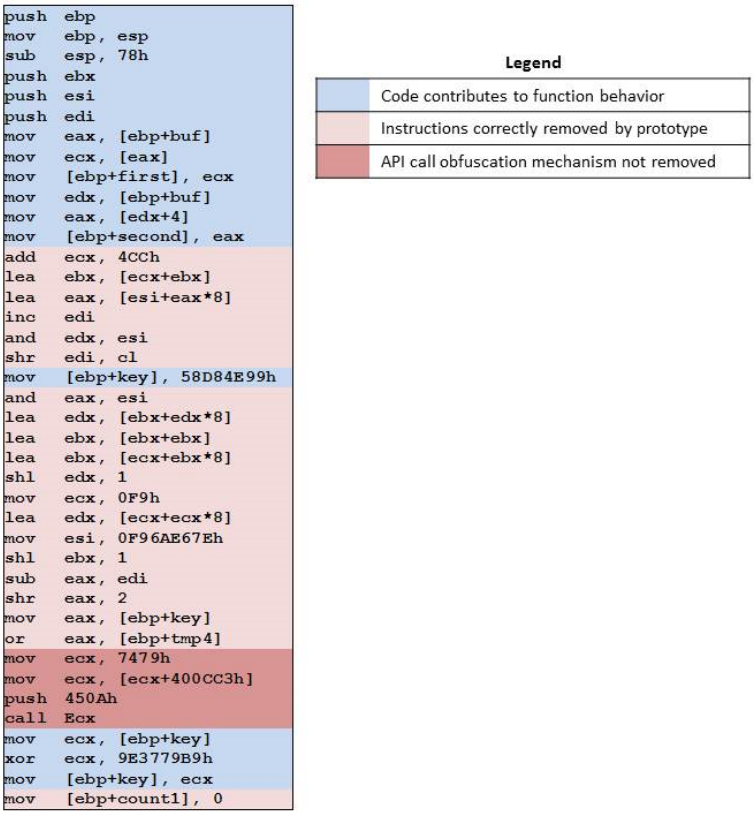

Figure 18: Obfuscated Ramnit *Malware Code*

Manual analysis revealed that the blue lines in Figure 18 contribute meaningfully to the function's behavior. The light pink lines represent instructions that our prototype correctly removed. The dark pink lines represent an API call obfuscation mechanism that the prototype did not remove.

The *Ramnit* functions use several obfuscation techniques, including

1. creation and use of stack variables with no net effect
2. addition of extraneous instructions that have no net effect
3. nonstandard use of uninitialized registers, violating calling conventions
4. needlessly complex arithmetic with constant values used in the function
5. calls to operating system APIs with invalid parameters that have no net effect

To address the first three obfuscations, we utilized definition and usage analysis to remove instructions that wrote to memory and registers, but whose values were never subsequently read (i.e., had a definition, but no usage). We implemented this analysis by using the instruction emulation feature in ROSE and adding callbacks for reads and writes involving registers or memory. To track stack variable usage (for obfuscation 1) correctly, we implemented a stack depth tracking feature similar to the one present in IDA. In the prototype, we supplied clues about the essential registers based on the function's calling convention, but we are confident that automatic detection of calling conventions is possible in the future. Thus, our first implemented deobfuscation transformation is a dead code eliminator.

For the executable transformation portion of the problem, we used the instruction assembly feature of ROSE to reassemble just the instructions on the critical list, skipping the junk instructions. We then created a new segment at the end of the program and placed the new instruction bytes in that segment. Thus an analyst could load a single executable into IDA to review both the obfuscated and deobfuscated code simultaneously.

A limitation of the prototype is that it processed a single function specified by the user and did not attempt to link the newly created deobfuscated function into the actual program control flow. A more complete tool must process all functions in the file and create control-flow linkages between the deobfuscated functions. For the deobfuscation transformations, processing all functions involves applying the transformation iteratively to each function; however, generating the deobfuscated program segment for all functions is more complex. We implemented some control flow removals and basic remapping of addresses to provide a proof of concept, but creating a more complete implementation will require more work.

The prototype did not remove the last two obfuscation techniques in the *Ramnit* family, hidden constants and extraneous system calls (4 and 5, respectively). However, we completed the deobfuscation manually to understand our progress toward a complete result. We also did not remove empty loops created by the elimination of all code inside the body of the loop. Figure 19 shows the reduction in complexity from the original code, to the output code from our prototype with the limited transformations, to complete deobfuscation.

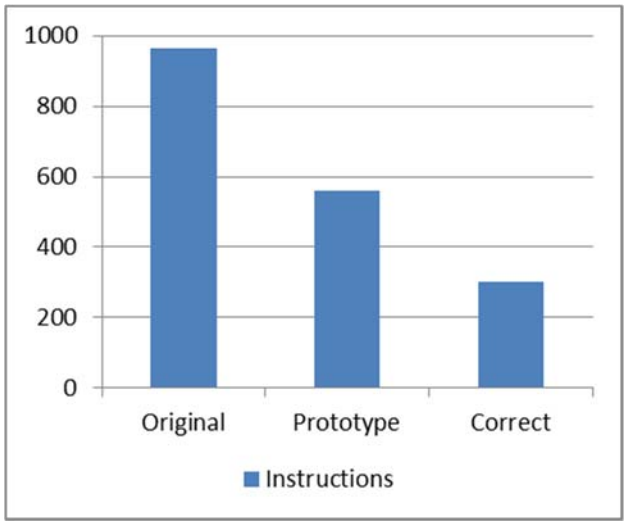

Figure 19: Effectiveness of Malware Deobfuscation Prototype

7.6.2 Application to String Deobfuscation

While working on the prototype tool, our team of analysts was operationally combating a specific type of string obfuscation. This technique involves moving immediate bytes into a local stack variable to construct a normal C-style null terminated string. The technique, which appears in a variety of malware families, prevents standard tools from easily recovering the strings. The bytes are typically moved onto the stack in an unusual order, sometimes by using temporary registers to

hold bytes that occur multiple times. Figure 20 shows an example from a malware file that obscures the string `INVALID PARAMETERS`.

```
mov   cl, 'I'
mov   edx, [esp+524h+u]
mov   [esp+524h+s], cl
mov   [esp+524h+s+5], cl
mov   cl, 'R'
mov   al, 'A'
mov   [esp+524h+s+0Ah], cl
mov   [esp+524h+s+10h], cl
mov   cl, 0Dh
mov   [esp+524h+s+3], al
mov   [esp+524h+s+9], al
mov   [esp+524h+s+0Bh], al
mov   al, 'E'
mov   [esp+524h+s+13h], cl
mov   [esp+524h+s+15h], cl
lea   ecx, [esp+524h+s]
mov   [esp+524h+s+0Dh], al
mov   [esp+524h+s+0Fh], al
mov   al, 0Ah
push  ecx
push  edx
mov   [esp+524h+s+1], 'N'
mov   [esp+524h+s+2], 'V'
mov   [esp+524h+s+4], 'L'
mov   [esp+524h+s+5], 'D'
mov   [esp+524h+s+7], ' '
mov   [esp+524h+s+8], 'P'
mov   [esp+524h+s+0Ch], 'M'
mov   [esp+524h+s+0Eh], 'T'
mov   [esp+524h+s+11h], 'S'
mov   [esp+524h+s+12h], '.'
mov   [esp+524h+s+14h], al
mov   [esp+524h+s+16h], al
mov   [esp+524h+s+17h], bl
call  sub_4018A0
```

Figure 20: Obfuscated String Example

Operationally, we required a detailed analysis of a malware family that used this technique frequently. While manually defeating the obfuscation technique for any given string was possible, it was time consuming. The analysts wanted to automatically recover all the strings of this nature from all known members of the family (several dozen at the time of the request). They would subsequently use this information to help assess the evolution of the family and group similar files together into variants within the family.

The diversity of instructions that can be used to accomplish the obfuscation presented an interesting challenge. Some progress had been made on developing an IDA script that would heuristically recover the string when interactively executed on one the instructions in the technique, but this approach did not scale well to number of files that needed to be processed. Also, it did not automatically find the strings, instead requiring manual invocation on each instance as the analyst encountered them.

Using ROSE, and the knowledge gained from creating our first prototype tool, our team constructed a specialized deobfuscator that recovered the strings and the starting address of the instructions that obfuscated them. The deobfuscator worked by emulating instructions and identifying contiguous stack memory locations that contain ASCII characters. Recovering the

value of the string was then simply a matter of iterating over the stack bytes in the order they occur in memory.

Our string deobfuscation tool worked well and met our operational needs. Our understanding of the variety and nature of technique grew, and we were able to process all members of the family on several occasions. By recovering the strings from each of the files in the family, we gained additional insights about the variants and evolution of the family much faster than we could have through a manual process. The successful application of our research efforts improved not only the final report on the family, but also increased our familiarity with ROSE and obfuscation techniques. We have found that such close interaction between research and operational application results in better results for both groups.

While the string deobfuscator has not yet been integrated into the transformation framework prototype described earlier, we believe it is quite feasible to do so. This technique could be turned into a code transformation pass that analyzes the code to identify variants of this obfuscation pattern, then removing the instructions that build the string as we did in the first transformation. To construct the new executable code, the string itself would be placed in the new segment, and any references to the stack variable would be replaced with a reference to the new address.

7.6.3 Obfuscation Prevalence Study

At this point in our research, we had demonstrated a working prototype and operational relevance. But we needed to implement more transformations to have a complete and generally applicable tool. The primary question we still faced was "Which transformations should be implemented next?" As we debated the issue, several other topics entered the conversation:

- Which obfuscation techniques are more or less difficult to implement transformations for?
- How effective is the technique at confounding analyst comprehension?
- What ordering constraints are there on the transformations, and how do they affect the results?
- How prevalent is each obfuscation technique in our malware catalog?

This last topic proved to be the most difficult to evaluate. Earlier in the project, we had created an incomplete catalog of obfuscation techniques documenting the nature of the technique and the addresses in specific malware files where the technique could be observed. This catalog proved useful for providing concrete examples when discussing specific techniques, clarifying exactly what was meant by a technique, and finding test cases for a technique. Unfortunately, the catalog provided no data about the prevalence of these techniques. We had developed some assumptions of which techniques were more common while developing the catalog, but it hardly settled the issue of how commonly each technique was used.

We decided to conduct an obfuscation technique prevalence study to gain some basic facts about the prevalence and distribution of the techniques in our catalog. We realized we had to write the code that detects obfuscations as part of the code that removed the obfuscations. Further, we concluded that the majority of the development effort was in the actual transformation or

executable generation, meaning that developing the detections first would allow us to better prioritize the remaining work.

Collecting objective data about obfuscation prevalence was more complicated than we had anticipated. Our malware collection contained approximately 43 million samples, so even a sample of 30,000 files is less than one tenth of 1 percent of the total. Running the prototype against tens of thousands of files uncovered a wide variety of problems in our code, the ROSE infrastructure, and even some misunderstandings about the nature of the problem. Consequently, the remainder of the project focused primarily on the prevalence study and interpretation of the results obtained.

7.6.4 Obfuscation Detection Tests

We chose to implement six tests for our initial study. We selected these tests for a variety of reasons, including a preference for detection tests that could be easily implemented and techniques that we perceived were common in our data set. We considered coverage for both control flow and noncontrol flow obfuscations to be important. Whenever possible, we chose general tests over specific tests with the expectation that we could refine the tests as needed in the future. Most of the detections are built using the disassembly, emulation, and analysis code in the ROSE framework.

Another concern was choosing tests that would minimize ambiguity about false positives and false negatives. All of the tests that we settled on implementing, described below, suffer from both false positives and false negatives to some degree. In most cases the false positives (incorrect detections) appear to be situations in which the code pattern was part of known compiler behavior or bad disassembly. When the detection code routinely found false identifications, we usually adjusted the algorithm slightly to eliminate that specific detection. For false negatives, (missed detections) the situation was even less clear.

Test 1: Suspicious Functions

This test looks for suspicious functions by constructing the post-dominator tree [Langauer 1979] from the control flow graph. Functions with malformed control flow graphs are not typically generated by compilers and might indicate a variety of obfuscation methods. If we are unable to construct the post-dominator tree, the control flow graph is likely malformed, and thus the function is possibly obfuscated. Also, if the function has multiple basic blocks, but none of the blocks are terminated with a RET instruction, then the function is malformed and may be obfuscated. In both cases, we consider this test to have detected an obfuscation for the purpose of our data collection. This test was partially intended to act as a backstop against arbitrary obfuscation techniques that so heavily permuted the control flow that we were unable to perform basic analysis.

Test 2: CALL/POP Instruction Pairs

One control flow obfuscation technique pairs a CALL instruction with a subsequent POP instruction that reads the return address off the stack. Many naïve disassemblers presume that the CALL actually returns, resulting in incorrect function boundaries and control flow. This test looks

for all POP instructions where the previous instruction writing to the same stack offset was a CALL instruction. However, this pattern can occur occasionally in compiler-generated code. For example, we identified a case where the instruction pattern was used to determine the failure address in abort-handling code from the compiler library. However, we believe that the majority of these detections will generally represent obfuscations. Missed detections can result from a variety of conditions, mostly related to failing to track stack depth correctly.

Test 3: Opaque Predicates

Malware authors use opaque predicates to obfuscate control flow by causing conditional jumps to always take (or not take) a given branch. This test looks for conditional jumps where the Boolean condition is always known based on the instructions in the same basic block as the conditional jump. This logic is implemented in the ROSE disassembly component, which puts such conditional jumps in the middle of a basic block rather than creating two basic blocks. Our test checks for conditional jumps that are not at the end of a basic block.

Test 4: Dead Stores

Malware authors frequently add extraneous instructions to complicate analysis. Often these instructions manipulate program values that are not relevant to the intent of the function. To construct this test, we adapted the detection logic from the prototype tool described earlier. The test uses definition and usage analysis to find instructions that overwrite memory and register values before being used by another instruction.

Test 5: PUSH/RET Instruction Pairs

Pairs of PUSH and RET instructions are used by malware authors to obfuscate control flow in much the same way as CALL and POP instruction pairs. This test looks for all RET instructions where the defining instruction was a PUSH (rather than the expected CALL). The implementation of this test and the expected outcomes are largely similar to Test 2, CALL/POP Instruction Pairs. We are unaware, however, of legitimate uses of the PUSH/RET code pattern.

Test 6: Constant Propagation

Malware authors sometimes obfuscate constants by performing mathematical operations on them to produce the desired value. For example, such techniques can hide the addresses of global variables or distinctive cryptographic constants. This test inspects instructions that perform common math operations (such as addition, subtraction, shifting, etc.) to see if both operands are constant values. If both operands are known constants, the instruction is likely to be an obfuscation technique because the compiler should have removed the instruction during optimization. A few common scenarios that otherwise meet this criteria have been excluded, such as subtracting a register from itself, which is a common way to clear a register. The test is conducted by emulating each instruction and testing the types of the operands.

7.6.5 Data Set Sampling

As stated earlier, our malware collection contained more than 43 million files at the time of this study, so we needed to select a reasonably sized subset. We chose sizes of the data sets to balance

analysis time and the accuracy of the representation of the larger collection. We selected several different data sets to evaluate various biases in our collection. Table 10 summarizes our data sets.

Table 10: Data Set Names and Summary Statistics

Data Set Name	Files	Functions
Random	27,947	1,862,329
Dates Proportional	31,447	2,171,742
Dates Equal	28,892	1,613,446
Pithos	27,984	1,970,192
No Match	10,000	810,527

We began by selecting a random sample of 27,947 files from the catalog. We were also interested in exploring the possible change in obfuscation prevalence over time, so we selected two data sets based primarily on collection date. The first contained 31,447 files selected from each month in proportion to the number of files in our catalog from that month. The second set contained 28,982 files with a fixed number of files from each month since January 2006. The first data set heavily favors more recent months because collection rates have been growing rapidly, and the second favors early years where collections were rather small.

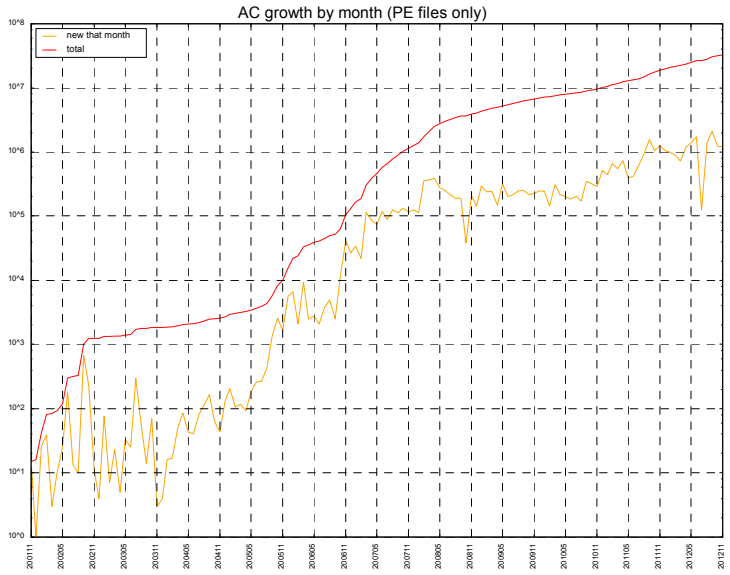

Figure 21: Artifact Catalog Growth by Month

A secondary concern in selecting the sample sets was to account for bias caused by the dramatically different prevalence of certain malware families and packing technologies. For example, we knew from previous analysis that there were more than 800,000 instances of the *Allaple* [F-Secure 2012b] family, and that more than 4 million of the files were packed with UPX [UPX Software 2012]. Because there was some concern that obfuscation prevalence might be higher among the malware families that were poorly understood, we wanted to ensure that we measured the prevalence among the rarer families as well as the most common.

This led us to generate a data set based on executable entry point signatures generated by our Pithos [Cohen 2008] clustering tool, which clusters files based on the first 100 bytes following the entry point. While this approach is imprecise, we have previously found it suitable as a starting point for similar tasks. We selected one file from each cluster in the collection. By comparison, the truly random data set contained files from only 7,966 of these signatures. The random set also included 696 of instances of the *Allaple* family.

Approximately 10% of the malware catalog had no Pithos signature match and was not represented in the previous data set. It seemed reasonable that heavy obfuscation might prevent Pithos from clustering files properly, so we wanted to contrast the unmatched set with other data sets. We selected 10,000 files randomly from those with no Pithos signature match for our "No Match" set. The random data set also included 3,274 files that had no Pithos signature match.

7.6.6 Problems Processing the Data Sets

Past experiments with bulk processing of malware has taught us to expect problems, and in this we were not disappointed. When run against the more than 149,000 files in our various data sets, the test tool and the ROSE framework encountered a number of problems, some expected and some not. However, after correcting for a variety of problems, 99.2% of the files in the selected data sets completed successfully.

The most common problems were related to excessively long runtimes in a small percentage of the files. We often encounter this in bulk analysis, so we set each run to abort after a certain amount of time. Additionally, modifying the test algorithms to abort a test for just the current function after an excessive number of instruction emulations significantly reduced this timeout problem and allowed the remaining functions in the file to be processed. Only 18,955 functions out of 15.5 million exceeded these thresholds, so the impact on our results is negligible.

Other problems we encountered included assertions thrown by ROSE, consumption of excessive stack space, and segmentation faults, among others. We corrected or otherwise accounted for the majority of these problems, but the few remaining errors stem from a wide variety of causes. This small percentage of remaining errors is unlikely to be biasing our results significantly.

Even when the execution is completed for a given file, the results are not guaranteed to be correct. The ROSE emulation code is currently unable to emulate certain classes of instructions, most notably floating point, MMX/SSE, privileged, and 64-bit. Rather than abort the processing of these functions, we treated the instructions as if they had null semantics and had the system generate a warning, which may produce a small number of incorrect conclusions. The number of affected functions varied from 5% to 11% of all functions depending on the data set. We intend to reduce the number of emulation failures in future work, which will improve the reliability of the results.

7.6.7 Obfuscation Prevalence

Obfuscation levels were found to be fairly low overall, as summarized in Table 11. We define a detection as a test observing an obfuscation technique at a specific address in the code, so a single test can return multiple detections for the same function in the same file. The number of functions

with detections ranged from 11.2% to 19.4% depending on the data set. The random data set and the two date sets had nearly identical levels of obfuscated files and functions, while the two Pithos-based sets showed slight increases in the levels of obfuscation over random sampling. The proportion of detections is also significantly higher in the two Pithos data sets.

Table 11: Obfuscation Prevalence with no Filtering of Detection Results

	Random	Dates Proportional	Dates Equal	Pithos	No Match
Files	27,947	31,447	28,892	27,984	10,000
Functions	1,862,329	2,171,742	1,613,446	1,970,192	810,527
Detections	1,587,980	1,690,210	1,289,109	5,018,247	790,081
Files with detections	20,009	22,436	21,126	23,461	8,679
Files with detections (%)	71.6%	71.3%	72.9%	83.8%	86.8%
Functions with detections	230,039	265,480	181,033	297,750	157,187
Functions with detections (%)	12.4%	12.2%	11.2%	15.1%	19.4%

The percentage of files having detections was quite high (71% to 87%) and did not seem consistent with the low percentage of functions having detections. Closer inspection revealed that the distribution of the detections was strongly long-tailed. For example, 69% of the functions from the random data set had only a single detection. Further, 43% of the detections came from the top 10% of the most heavily obfuscated functions. This distribution suggested that we might be encountering an occasional single false positive in many functions, while truly obfuscated functions were routinely positive for many detections in several tests.

To test our hypothesis that the detections in the long tail were false positives, we selected a couple dozen example malware files, each with only a single detection. On manual inspection of these files, we discovered that the most common cause of the detections was instructions that had been incorrectly created by the disassembler. When presented with these contextually nonsensical instructions, our tests often detected the code patterns that we were searching for, but characterizing these as obfuscations was incorrect. While the incorrect disassembly could have implications for the accuracy of our results, we found no evidence to contradict our suspicion that small numbers of detections were likely to be false positives.

We have previously investigated the disassembly correctness of ROSE and IDA and found them to be comparable. We found that ROSE often created more functions than IDA, and it appears that at least some of them are in error. We also found that, unsurprisingly, both tools have more difficulty with obfuscated malware. We were unable to accurately quantify the impact that this has on our current obfuscation prevalence study, but we will be investigating more correct disassembly techniques in a follow-on LENS in FY13 to help address this issue more generally.

Another problem related to incorrect disassembly involves traditional packing. Malware that is packed in encrypted layers can complicate static analysis based approaches like ours. In particular, packers that jumped into code that had been modified by earlier execution often lead ROSE to erroneously discover semi-random instructions. Determining whether a file is packed and then assessing the correctness and completeness of a static disassembly listing is a challenging problem that is not in the scope of our current research.

To address packing, we use other unpacking approaches to preprocess packed executables before applying our deobfuscation tool. As a side experiment, we applied this approach to the unpacked output of a conventionally packed subset of our data, and found only minor levels of obfuscations. A more detailed understanding of how packing correlates with post-unpacking obfuscation will require additional investigation.

While manually reviewing samples, we also encountered a legitimate use of the CALL/POP pattern as described previously. We also selected a sample of heavily obfuscated functions, and manual analysis confirmed that all of them met our subjective criteria for being obfuscated.

Taken together, these investigations indicated that we needed to remove a small amount of false-positive noise by developing objective criteria for identifying obfuscated functions. Because obfuscation level is an inherently subjective evaluation, any criteria will have deficiencies. The first version of our criteria was to accept as obfuscated only functions that had more than three detections. This had the effect of eliminating the singleton detections and generally improved the separation of clearly obfuscated files from nonobfuscated files. But closer inspection of the data revealed additional problems. Figure 22 shows the detections from each of the six tests in the random data set. Dead stores account for the overwhelming majority of the detections, often occurring hundreds of times in an obfuscated function. The post-dominator test and the PUSH/RET test were only ever positive once per function. While the former was expected (because it is a function-level test), the latter may be a consequence of the way ROSE groups instructions into functions. A simple count of detections clearly overrepresents dead stores and underrepresents other tests.

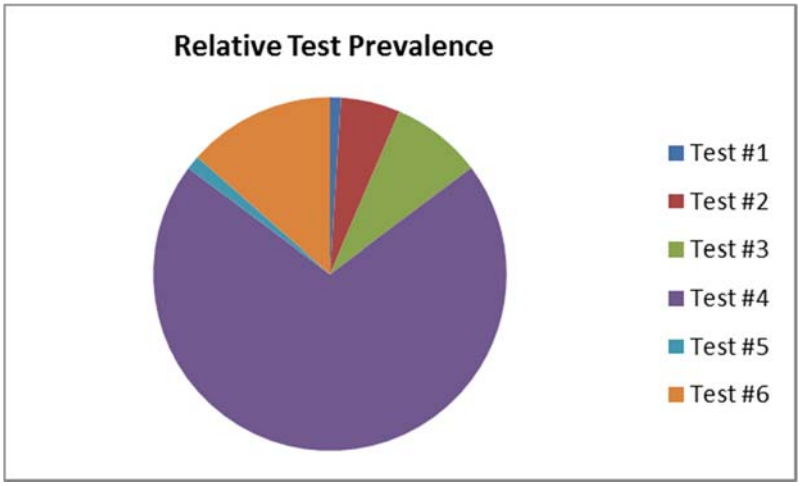

Figure 22: Relative Prevalence of Obfuscation Tests in the Random Data Set

Considering these factors, we established the following criteria for obfuscated functions, designed to reduce the impact of noisy false positives:

- A function must have at least one detection in each of two different tests.
- In the dead store test, a function must have at least four detections to correct for the test's overall higher detections and distribution.

- The function must have at least two detections of the CALL/POP technique, if this technique is present at all, to compensate for the known use of this technique in legitimate code.

Because these criteria would exclude a function that had a very large number of detections for a single test (e.g., dead stores) but which we would still consider an obfuscated function, we also accepted any function that had more than 10 total detections.

While we are unable to provide objective evidence for the legitimacy of these criteria, they largely matches our subjective definition of obfuscation. If obfuscation is understood to mean inhibiting the analyst's ability to understand the function's behavior, a few detections of a single a technique would not cause undue confusion. These criteria exclude at most a function with three dead stores, one CALL/POP pair, and one other detection. While such a function could be legitimately obfuscated, the criteria seem unlikely to exclude any seriously obfuscated code unless there are other deficiencies in our test suite.

Table 12: Obfuscated Files and Functions Based on Detection Filtering Criteria

	Random	Dates Proportional	Dates Equal	Pithos	No Match
Obfuscated files	6,533	7,341	6,662	10,600	4,257
Obfuscated files (%)	23.4%	23.3%	23.0%	37.9%	42.6%
Obfuscated functions	17,041	18,897	15,422	38,019	16,407
Obfuscated functions (%)	0.92%	0.87%	0.96%	1.93%	2.02%

Using this new standard for determining whether a function is obfuscated, we re-evaluated the data. Not surprisingly, the results, summarized in Table 12, show dramatically lower levels of obfuscation. We then defined an obfuscated file as a file containing one or more obfuscated functions. Using these criteria, the proportion of obfuscated functions fell to 1% or 2% depending on the data set. The percentage of obfuscated files ranged from 23% to 43%, less than half of the percentage under the stricter criteria.

7.6.8 Other Analyses

The Pithos-based data sets showed higher levels of obfuscation compared to the random and date data sets. Overall, the percentage of obfuscated files rose from 23% in the other data sets to 38% and 43%, respectively. The number of obfuscated functions doubled from less than 1% to about 2%. We suspect that this is related to the wider sampling of less common malware in those sets. This increase in obfuscation is most visible in the data set consisting of 10,000 files for which we have no Pithos signatures. Because heavy obfuscation is likely to vary the bytes following the entry point address, this is expected.

To test our ability to select obfuscated files from the catalog, we selected a set of 11 Pithos signatures that showed high levels of obfuscation based on the Pithos data set. We then randomly selected 3,497 files matching those signatures from the catalog. This data set resulted in 10% of functions being obfuscated and 91% of the files being obfuscated. These rates are dramatically higher than in any of the other data sets and suggest that when our detector is used in conjunction with Pithos, we are able to select significantly obfuscated files from our malware collection.

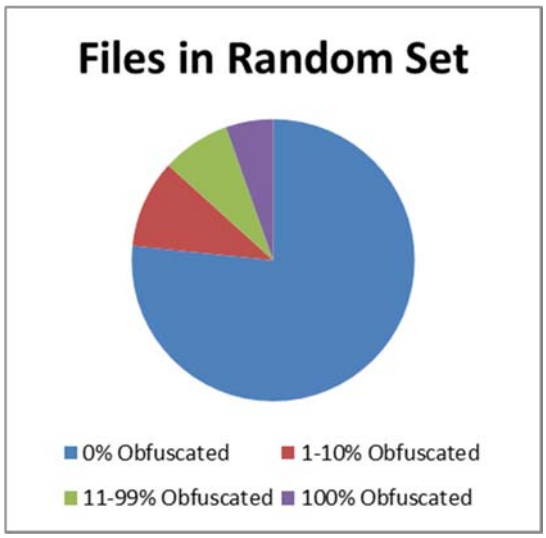

Figure 23: Different Obfuscation Levels for Malware Files in the Random Data Set

Figure 23 shows the proportion of files having various levels of obfuscation from the random data set. The levels of obfuscation were determined by the percentage of functions within a file that were obfuscated. Consistent with the previous results (Table 12), 76% of the files had no obfuscated functions. However, in 5% of the files, all functions in the file were obfuscated. Most commonly, this is caused by files with only one or two functions (possibly conventional packers), but there are occasional files with dozens of obfuscated functions. In 10% of the files, only a small proportion (<10%) of the functions were obfuscated. This phenomenon could represent files with narrowly targeted obfuscation goals (e.g., string decryptors), or the detections might simply be more false positives. The last 8% are the moderately obfuscated files, where 20% to 90% of the functions are obfuscated. This category might include file infectors or obfuscated malware with statically linked library code. Further research is required to understand the significance of varying levels of obfuscation.

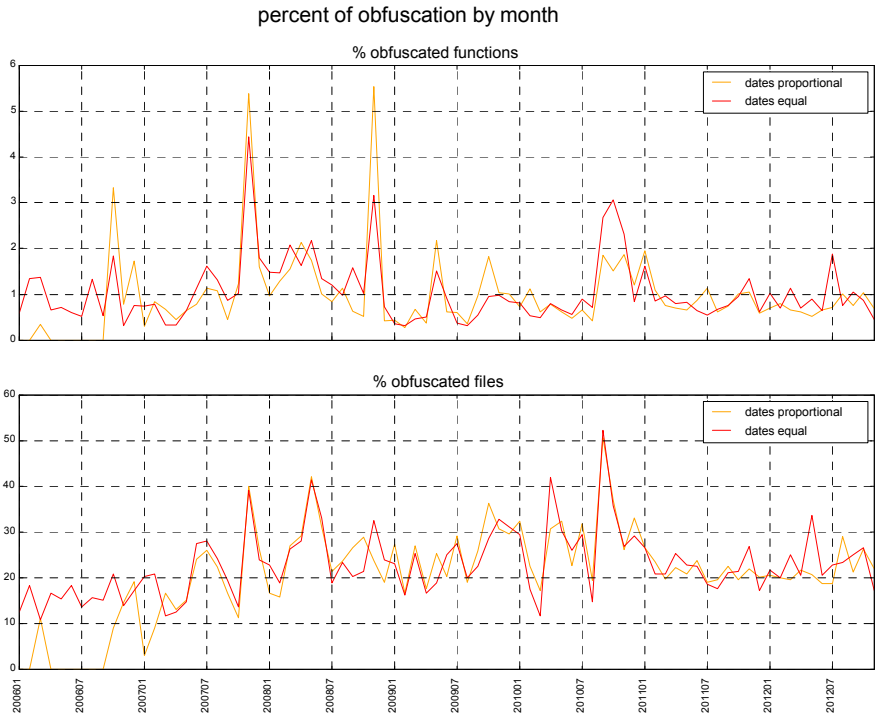

Figure 24: Obfuscation Level in Files and Functions Over Time

The date-based data sets were analyzed to detect trends in obfuscation levels over time. As Figure 24 demonstrates, our study detected no clear trend in obfuscation prevalence. The data set that selected an equal number of files from each month shows some additional volatility in the latter months, while the data set selected in proportion to the number of collected files has insufficient data in some early months. The results also give some sense of the month-to-month volatility of obfuscation levels, but they reveal no clear patterns.

7.6.9 Conclusions and Future Work

Obfuscation levels were found to be relatively low in our malware collection. Most functions have either at most one detection of the techniques that we tested for, and we suspect that many of the singletons are false positives. A small number of files that are heavily obfuscated contain functions with hundreds of detections per test, and they are often positive for multiple tests. The use of obfuscation does not appear to be increasing significantly over time, but rather linearly increasing in proportion to the overall number of malware samples collected.

Future work related to this research may include using machine learning algorithms, with the attributes we have identified, to automatically distinguish obfuscated malware from nonobfuscated malware. We intend to continue developing our deobfuscation tool by implementing additional transformations based in part on the prevalence results detailed in this report and on feedback from continued use in our operational malware analysis work.

This study had many challenges involving ground truth, confidence in our results, and objective measurement of hard-to-quantify factors. While changes in the testing infrastructure might cause some variation in the detected level of obfuscation, it seems unlikely that it would dramatically affect the overall conclusions. The largest outstanding problems are related to disassembly correctness and completeness. It is difficult to build confidence in results that are built on uncertain foundations. In the coming year, we intend to continue our research into automated static disassembly analysis.

Obfuscation remains an important problem for the Department of Defense, not because of the prevalence of obfuscation, but because of the difficulty of understanding malware behavior when obfuscation is present. We created a semantic-transformation-based framework for deobfuscating malware. The framework shows promise and worked well in limited tests. When used to deobfuscate malware, it can reduce analyst effort by a factor of several hundred.

7.7 References

[Allen 1976]
Allen, F. E., & Cocke, J. "A Program Data Flow Analysis Procedure." *Communications of the ACM 19*, 3, (1976): 137-147.

[Cifuentes 1997]
Cifuentes, C., & Fraboulet, A. "Intraprocedural Static Slicing of Binary Executables." *Proceedings of the International Conference on Software Maintenance*, 1997.

[Cohen 2008]
Cohen, C. F., & Havrilla, J. S. "Malware Clustering Based on Entry Points." *CERT Research Annual Report 2008*. Carnegie Mellon University, Software Engineering Institute, 2008.

[Collberg 1998]
Collberg, C.; Thomborson, C.; & Low, D. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs." *Principles of Programming Languages*, 1998.

[F-Secure 2012b]
F-Secure. *Net-Worm:W32/Allaple.A*. http://www.f-secure.com/v-descs/allaple_a.shtml (Accessed 11/30/2012).

[Krügel 2004]
Krügel, C.; Robertson, W. K.; Valeur, F.; & Vigna, G. "Static Disassembly of Obfuscated Binaries." *USENIX Security Symposium*, 2004.

[Langauer 1979]
Langauer, T., & Tarjan, R. E. "A Fast Algorithm for Finding Dominators in a Flowgraph." *ACM Transactions on Programming Languages and Systems*, 1979.

[Linn 2003]
Linn, C., & Debray, S. "Obfuscation of Executable Code to Improve Resistance to Static Disassembly." *ACM Conference on Computer and Communications Security*, 2003.

[Miller 2012]
Miller, B., & Hollingsworth, J. *Paradyn/Dyninst - Putting the Performance in High Performance Computing*. http://www.dyninst.org/ (Accessed 3/12/2012).

[Roundy 2012]
Roundy, K. A., & Miller, B. P. *Binary-Code Obfuscations in Prevalent Packer Tools*. 2012.

[Schwarz 2002]
Schwarz, B.; Debray, S.; & Andrews, G. "Disassembly of Executable Code Revisited." *Working Conference on Reverse Engineering*, 2002.

[Udupa 2005]
Udupa, S. K.; Debray, S. K.; & Madou, M. "Deobfuscation: Reverse Engineering Obfuscated Code." *Working Conference on Reverse Engineering*, 2005.

[UPX Software 2012]
UPX Software. UPX: The Ultimate Packer for eXecutables. http://upx.sourceforge.net/ (Accessed 11/30/2012).

8 Measuring Early Detection of Insider Threats

William Claycomb
Carly Huth
Brittany Phillips
Lori Flynn
David McIntire

8.1 Introduction and Purpose

A difficult and important problem in the study of insider threats involves identifying and correlating observable events, known as *indicators*, which could indicate potential malicious insider activity. We consider a malicious insider to be a current or former employee, contractor, or other business partner who has or had authorized access to an organization's network, system, or data and intentionally exceeded or misused that access in a manner that negatively affected the confidentiality, integrity, or availability of the organization's information or information systems. Given the complexity of possible interactions among the individual, organization, and IT system, we adopt a socio-technical approach to studying indicators, which combines both technical and nontechnical input for a more holistic analysis [Cappelli 2006]. One refinement of the development of indicators is the measurement of early detection. This includes research into pre-attack socio-technical indicators that may help the organization discover a potential attack, and quantifying how much time exists between observable events and the moment of attack.

Empirical analysis of existing instances of known malicious insider activity is a useful method for identifying indicators, including those for early detection [Claycomb 2012]. The process of examining actual insider threat cases involves several steps. One method is as follows:

1. Collect source data (e.g., documents, reports, etc.) on instances of insider crime.
2. Process case information using a repeatable and consistent process to store key events and information about the case.
3. Create chronological timelines from case data.
4. Identify key events in the chronology of the attack.
5. Examine case chronologies to identify patterns or other significant indicators of attack.
6. Compare results to baseline behaviors of assumed nonmalicious populations.

Much of the source data is limited because actual crimes are frequently unreported [CSO Magazine 2011]. However, since 2001, the CERT Insider Threat Center has used mainly public sources of information to collect a database of approximately 800 real-world insider threat cases. The focus of our work is not on the extraction of insider threat data; rather, we focus on initial efforts into steps 3 through 5 above.

8.1.1 Project Summary

We began by addressing step 3, creating chronological timelines. Though the cases in the source database we used already contained chronologies, they were not in a form conducive to our proposed analytical technique. We made a few minor changes to the structure and format of chronological events prior to analysis; those changes are detailed in this report. Next, we developed a set of descriptors and attached them to each chronology event.

Addressing step 4 involved identifying key events in the chronology of the attack. Our work in this project shows how we processed and analyzed the collected case information by applying a discrete and finite set of descriptors. This process included critical steps such as identifying the point of damage to the organization, or *zero hour* of the attack, as well as any malicious events prior to zero hour that enabled the attack but did not immediately cause harm. Finally, with descriptors and zero hour identified, we examined the chronology of events to identify patterns and indicators of attack, which is step 5 in the process above.

Our results from examination of 49 cases of insider IT sabotage show that 44 cases (89.8%) contained observable events caused by the insider prior to attack. Of those 44 cases, 34 (78%) contain behavioral actions by the insider prior to technical actions. This reflected the overall distribution of behavioral versus technical actions prior to attack for all cases: of 449 total events analyzed, 352 (78.3%) were categorized as behavioral and 97 (21.6%) were categorized as technical. Though most events caused by insiders appear to be malicious, we did not explicitly imply malicious intent for all events. Finally, we measured the time between the first observable event clearly enabling the attack and the moment of damage to the victim organization. Of the 49 cases, 18 (37%) had no observable event enabling the attack prior to damage occurring. Nearly the same number of cases, 17 (35%), had an identifiable event clearly enabling the attack less than one day prior to attack, and in 10 of 49 cases (20%) that moment was more than a day prior to attack—often a week or more in advance of actual cyber damage.

8.2 Background

The CERT insider threat database contains approximately 800 real-world insider threat cases and is the foundation for our research. The database contains three main types of crimes: fraud, intellectual property theft, and IT sabotage. The database includes information about the perpetrator, organizations involved, and incident details. This information is derived mainly from public sources (e.g., media reports and court documents) but also includes some data from nonpublic sources (e.g., law enforcement investigator notes). This approach, known as a comparative case design, has been employed by insider threat researchers studying technical, behavioral, and socio-technical aspects of insider threat. However, there are limitations to related work in this area. For example, Maybury and colleagues provide a socio-technical approach that focuses on detection after attack [Maybury 2005]. Greitzer and Paulson noted that few have attempted to focus on observable concerning behaviors prior to attack, and they are among the first to propose predictive modeling [Greitzer 2011]. However, they note that researchers lack real-world data to validate their approach. A recent insider threat literature review also called out the need for real-world data in analysis [Hunker 2011].

We chose to examine the issue of early-detection indicators of insider threat using cases of insider IT sabotage. Patterns among IT sabotage cases have been the topic of previous research efforts, including a 2005 joint study with the U.S. Secret Service, the CERT Program's MERIT project, and a collaboration with PERSEREC to compare espionage and sabotage [Keeney 2005, Cappelli 2006, Moore 2008]. These studies provide some initial insight into potential early indicators of IT sabotage. In the 2005 joint study, findings included the discovery that a majority of insiders planned out their activities in advance and acted out in a concerning way in the workplace [Keeney 2005]. The CERT Program and PERSEREC's comparison of espionage and IT sabotage yielded several findings, including that concerning behaviors were often observable before insider IT sabotage and espionage [Moore 2008]. This study also noted that insider technical actions could have alerted the organization to planned or ongoing malicious events.

8.3 Approach

8.3.1 Evaluation Criteria

The goal of this project was to identify patterns of behavior and measure potential detection time of malicious insider action on IT systems prior to the attack. The success criteria for the project were related to identification of observable events of insider disgruntlement prior to attack, with a hypothesis that 90% of cases have an observable indicator of the attack prior to the actual moment when damage to IT systems occurs. One definition of an observable event is, "anything that can be detected with current technology" [Brackney 2004]. While this is a simple definition, it did not simplify our event selection criteria. Questions arose concerning event detectability, such as the following:

- Was the event detectable by technology that was current at the time of attack?
- If event detection technology existed but was unavailable to the victim organization or prohibitively expensive, should the event be considered observable?
- If the event was detectable but highly unlikely, should the event be considered observable?

The answers to these questions depended largely on our analysis objective. We did not intend to focus on highlighting errors or missed signals in previous cases; rather, we were interested in characterizing insider behavior prior to attack. We chose a fairly broad set of criteria for event selection, to include events not necessarily detectable by technology at the time of attack and those with low detection probability. Our decision was guided by the following observations:

- Technology for event detection is rapidly evolving. For example, consider an event description of an unauthorized logon by a malicious insider who knows the username and password of the target account. This masquerade may seem undetectable by the host system, but recent work summarized by Killourhy and Maxion indicates that the identity of a user can be validated by measuring subtle differences in keystroke typing patterns [Killourhy 2009].
- Identifying events that represent key indicators of potential insider activity yet are undetectable by current technology can drive development of technologies to detect such events.
- By excluding currently undetectable events, we would limit the application and impact of this work as new technology emerges.

This study analyzed only events prior to the moment of cyber damage, with the goal of determining indicators prior to attack.

8.3.2 Hypotheses

Based on previous heuristic analysis of insider threat sabotage, we developed the following hypotheses of the insider IT sabotage cases available to test during the study:

- Hypothesis 1: Indicators of malicious insider activity represented in our cases are demonstrated in nontechnical, or behavioral, events prior to technical events in more than 50% of the cases.
- Hypothesis 2: The time difference between the first observable action that clearly enables the attack and the initial moment of damage to an organization's cyber systems is less than 1 day in more than 50% of the cases.
- Hypothesis 3: Behavioral indicators of malicious insider activity are more prevalent than technical indicators, among the cases studied.
- Hypothesis 4: Of the insider threat cases studied, 90% will have an observable event indicating potential malicious activity prior to the moment of actual damage to the victim organization's IT systems.

8.3.3 The Analysis Process

8.3.3.1 Using Existing Chronologies

This initial study selected 49 cases of malicious insider IT sabotage. These cases represented those with the greatest number of chronological events and diversity of information sources from the more than 140 available cases. Each chronology contained a sequence of discrete events including (when known) the attributes of date and time, place, and a detailed description of each of the known organizational and perpetrator actions, starting with any information known prior to the attack up through any known legal adjudication.

While chronologies were already present for each case, initial examination of event contents revealed issues that hindered development of a repeatable analysis methodology. For instance, a single event entry often described more than one distinct action by the subject of the entry, for example, "The insider logged on to the system and developed a logic bomb." In this example, "logged on" and "developed a logic bomb" are two distinct actions. Therefore, prior to analysis of chronological events, two trained analysts examined the existing chronologies to ensure that each event contained only one action.

In addition, some effort was made to streamline and regularize the free-text description of each event. For example, the difference between "Insider hired by the victim organization" and "Victim organization hired insider" can affect how analysts perceive certain data points and makes automated data extraction more difficult [Claycomb 2012]. To mitigate this issue, two trained analysts developed a standardized method for expressing common events such as hiring, firing, damage, and use of drugs. Figure 25 provides an example of a hypothetical partial chronology sequence.

Sequence	Date	Event
1	10/1/2008	Insider starts work at victim organization as an assistant system administrator.
2	11/30/2010	Insider receives a below-average performance review rating overall.
3	1/18/2011	Insider threatens co-worker, saying that insider could, "mess with your user account and make you look really bad."
4	2/3/2011	10:16 PM Insider logs into a shared workstation using co-workers userID and password.
5	2/3/2011	10:20 PM Insider logs off without doing anything else.
6	2/4/2011	Insider fired by manager.

Figure 25: Sample Chronology [Claycomb 2012]

8.3.3.2 Adapting Previous Codebook

We examined the existing *CERT Insider Threat Database Codebook*, which contains codes for personal predispositions, stressors, concerning behaviors, as well as organizational vulnerabilities, for suitability for the project. Because the *Codebook* lacked the expressiveness needed for in-depth analysis of the events prior to attack, we determined that a new structure should be developed for this study. We created a new structure following a *critical pathway* model of malicious insider behavior, first proposed by Shaw, Ruby, and Post [1998], and further refined, with Shaw's collaboration, into the *CERT Insider Threat Conceptual Framework*. Our effort included both the reorganization of event codes into a new hierarchical structure, as well as the addition of new event codes to describe further technical and behavioral observables that researchers hypothesized might be present in the chronologies.

We performed an initial pilot study to test the effectiveness of operational definitions for specific event codes. In this test, analysts were given an updated codebook, including definitions of the codes and examples for each code. Analysts were asked to identify appropriate codes for several chronology sequences. Agreement among analysts was low due to two significant observations: (1) operational definitions for event codes were ambiguous and (2) the actual chronology items lacked a standard structure, which led to different interpretations of the same events across multiple cases. We found that using the detailed hierarchical structure for codes, as well as requiring analysts to simultaneously interpret inconsistently worded events, was potentially too complicated to achieve consistent results on codes assigned to each event.

To aid the analysts, we began to create an automated, streamlined analysis process, with the goals of reducing cognitive load [Sweller 1998] and increasing agreement between analysts. We first developed a set of decision trees based on the hierarchical structure of the codebook. For example, the analyst would be asked, "Does this event indicate something that was known about the insider previous to employment?" If the analyst answered "yes," the analyst would be guided to codebook items addressing preemployment information. A pilot test was conducted using this decision tree approach. While initial results were promising, development and testing of a full-scale implementation, covering all codes required for this project, was outside the scope of time and resources available. We concluded that our new codebook was simply too complex and

unrefined to use in an accurate and timely manner, though future work includes creating the actual automated query process.

8.3.3.3 The Triad

To address the goals of the project using the time and resources available, we needed a more efficient way of parsing event data as an initial gateway into analysis. Through discussion, we developed a triad structure for describing incident events, loosely based on the method for describing events related to foreign policy described in East and Herman [East 1982]. Rather than assign a unique code to each event, we focused on accurately describing the contents of the event. Our proposed method applies an event descriptor, comprising three components, to each chronological event. The descriptor categories are the *actor*, *action*, and *target* (Figure 26).

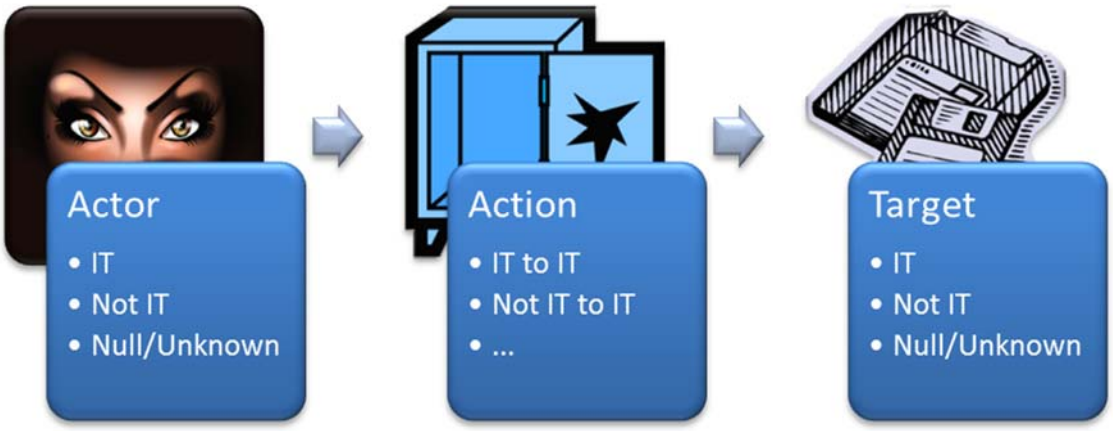

Figure 26: Triad Components

Prior to testing and full analysis of all events, a group of senior analysts developed the set of possible choices for each component, organizing them into a two- or three-level hierarchical structure. The goal of this activity was to develop a finite and discrete list of descriptors that adequately and accurately describe all possible actors, targets, and actions represented in the cases we studied. However, we did not want this set of descriptors to be so detailed that analysts would face the same challenges we faced during our first attempt to codify events. Our final working set of event descriptors contained 4 top-level *actor/target* options with 10 second- and third-level options as well as 6 top-level *action* options with 83 second- and third-level options. Not every category required three levels of detail to achieve the desired descriptive granularity. Table 13 illustrates each level of the triad structure, with sample input for a hypothetical event; Appendix A shows the full triad structure.

Table 13: Triad Values for the Example Event: "Insider threatened to harm co-worker."

Component	Actor Level 1	Actor Level 2	Actor Level 3	Action Level 2	Action Level 3	Target Level 1	Target Level 2	Target Level 3
Sample Input	*Not IT*	*Insider*	*Insider Individual*	*Interpers-onal Actions*	*Threatened*	*Not IT*	*Victim Org*	*Co-worker*

Analysts began the analysis process by selecting the appropriate top-level actor and target from the set: *Not IT*, *IT*, *Unknown*, *Null*. The top-level action category was automatically determined by the combination of top-level actor and target, for instance *Not IT* to *Not IT*. Based on that determination, the analyst could select from second-level action items, such as *Interpersonal*, *Criminal Justice*, or *Medical*. Analysis subsequently chose options from second- and third-level actor, target, and action options. The *Null* category is used for events without a target, such as life events (i.e., individual feelings, change in location, changes in financial situation, changes in health, change in family status)

8.3.3.4 Inter-Rater Reliability Testing of the Triad Method

Initially, three analysts independently examined the same seven cases, applying triad categories to each event, to test inter-rater reliability (IRR). Because the Fleiss' Kappa agreement (κ) [Fleiss 1971] was not as high as desired ($\kappa \geq .8$), additional training was provided, and a second IRR was undertaken with four cases and an additional analyst. Table 14 shows κ for both the first and second IRR tests. After reaching the desired level of agreement among analysts, the remaining cases were examined. Additional analysts were added throughout the project, initially examining the same cases as the analysts in the second IRR test. Inter-rater agreement was then calculated across all analysts to ensure a consistently high level of agreement.

Table 14: Fleiss' Kappa (κ) Values for First and Second Inter-Rater Reliability Tests

	Actor Level 1	Actor Level 2	Actor Level 3	Target Level 1	Target Level 2	Target Level 3	Action Level 2	Action Level 3
First IRR test	Inconclusive	.902	.806	.764	.738	.692	.631	Not calculated
Second IRR test	1.0	.902	.883	.869	.820	.804	.849	.904

During this process, we made a few minor changes to the analysis guidelines, to add or clarify specific actions. However, previously examined cases were not modified at the time. After analysis was complete, a senior analyst reviewed the initial round of results to determine if any changes should be made to earlier cases, based on the minor changes made during the analysis process. Very few event descriptors were changed.

Finally, we noted several lessons learned throughout the analysis processes.

- Project leaders found it helpful to meet often with the analysts to discuss problem areas in the analysis and develop solutions for future analysis.
- Analysts found it useful to develop a default method for addressing chronology items they felt were unclear.
- Analysts felt it would be useful to develop an automated processes for entering commonly observed events, both technical and nontechnical (e.g., logged on to system, insider hired).
- Emphasis on avoiding assumptions during the analysis process was crucial (e.g., if the chronology description did not explicitly state that IT systems were involved, the analyst should not assume IT systems were part of the action).

8.3.3.5 Identifying the Zero Hour

The next step in analyzing key points in each case was to identify the moment of cyber damage to the organization as a result of actions committed by the insider. We called this point the *zero hour*. An important distinction is that the insider's planning of or even staging of an attack is not considered the zero hour; we consider the event that clearly enabled the attack (but did not necessarily cause immediate harm) the point of *malicious action*. Using the previous example, the malicious action would be the insider testing or planting a logic bomb on an IT system, and the zero-hour event is when the logic bomb executes and IT damage occurs.

8.4 Results

Table 22, Table 23, and Table 24 in Appendix B outline the frequencies for each level of descriptor associated with case events. While the descriptors describe actions, both technical and behavioral, that took place during an insider attack, the actions were limited to what is available from our source material. That is, there may have been additional observables that are not reported in our sources. Also, this analysis does not account for whether or not these actions were actually observed or whether or not the intent of the insider was malicious for each specific behavior.

8.4.1.1 Events Prior to Attack

The attack zero-hour identifies the moment when the insider's action begins to harm the organization. With the caveat that the number of events recorded for each case is highly dependent on the amount of source materials available, we examined a total of 449 events prior to attack from 49 cases, for an average of 9.16 events per case. The smallest number of events prior to attack in a case was 1, and the greatest number was 44. Figure 27 shows a distribution of this value for all cases.

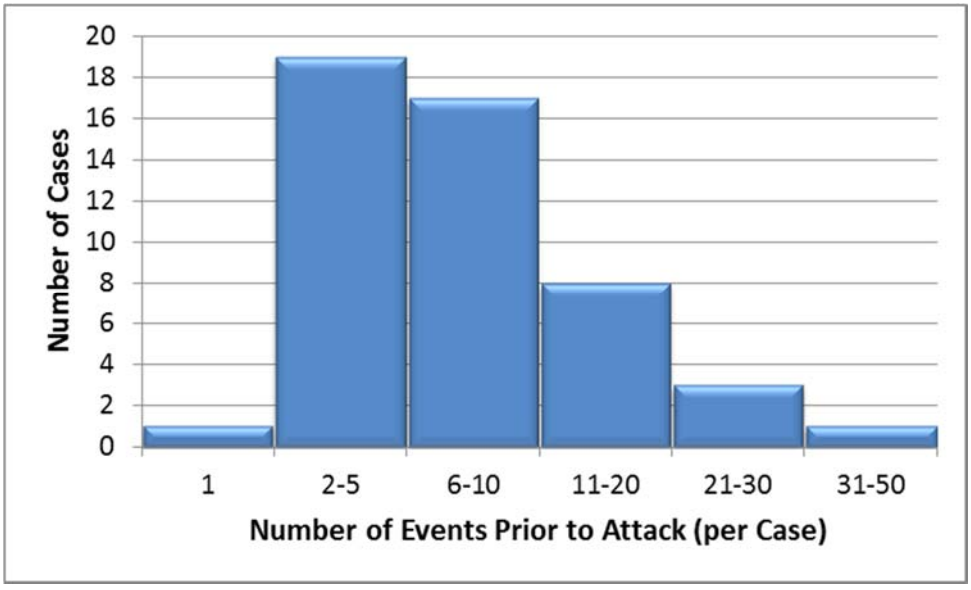

Figure 27: Distribution of Number of Events Prior to Attack (All Cases)

While analyzing all observable events prior to attack is important for understanding the relationship between the insider and the organization, identifying early indicators of attack for the insider alone requires us to distinguish events initiated solely by the insider. We identified insider-initiated actions in 44 of 49 cases, with an average of 5.39 events per case. Appendix B, Figure 30, shows the distribution of event counts for these cases.

8.4.1.2 Behavioral and Technical Actions

Events were determined to be *behavioral* or *technical* depending on the top-level actor or target categories as defined by the descriptor for each event. Events involving IT systems were denoted as *technical*, and those between individuals and/or organizations were denoted as *behavioral.* Of the 449 events analyzed, 352 (78.3%) were categorized as behavioral and 97 (21.6%) were categorized as technical.

8.4.1.3 Timelines of Behavioral and Technical Actions

In cases with observable insider-initiated actions prior to attack zero-hour, including four cases with only technical observable events prior to attack zero-hour, the insider demonstrated a behavioral action prior to any technical action in 34 of 44 (78%) cases. Of the cases with both behavioral and technical observable events prior to attack zero-hour, the insider demonstrated a behavioral action prior to a technical action in 34 of 40 (85%) cases, with an average of 2.85 behavioral events prior to the first observable technical event. Also, the first technical event occurred nearly two-thirds of the way through the timeline from first observable event to the point of attack.

All cases contained at least one technical observable event prior to, or including, the point of attack. Table 15 summarizes the results.

Table 15: Event Precedence (Behavioral vs. Technical)

Cases with the following order of event precedence:	Of 44 cases with observable events prior to zero hour	Of 40 cases with both behavioral and technical observable events prior to zero hour
Behavioral Prior to Technical	34	34
Technical Prior to Behavioral	10	6

8.4.1.4 Malicious Action and Zero Hour

In some cases, the malicious action occurred days or weeks before the attack zero-hour. In other cases, the malicious action occurred only hours or even minutes prior to attack zero-hour. And in other cases, there was no prior observable action that clearly signaled or enabled the attack. For cases where the malicious action data point could be determined, we measured the difference between the time of the malicious action and the zero hour. Table 16 shows detailed results.

Table 16: Time of Malicious Action (MA) with Respect to Zero Hour (ZH)

Time between MA and ZH	Number of Cases	Percentage of Cases Analyzed
≥1 day	10	20%

<1 day	17	35%
No MA prior to ZH	18	37%
Uncertainty in data	2	4%
Unknown	2	4%
Total	49	

Figure 28 shows the total number of days between malicious action and zero hour for cases with >1 day difference. Of the 10 cases represented, 5 cases have 20 or more days between malicious action and zero hour.

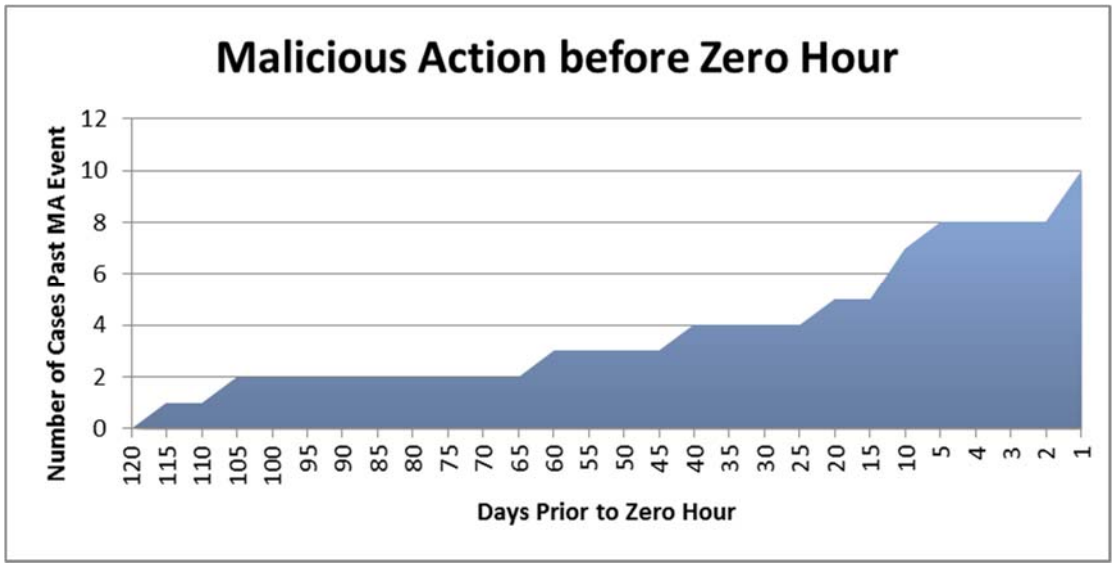

Figure 28: Time between Malicious Action (MA) and Zero Hour (ZH)

8.4.1.5 Event Descriptors

Table 17 shows the 26 most common event descriptors used, and Appendix B, Figure 29, shows the distribution of event descriptor usage. Organizational actions (hiring, firing, termination), insider feelings (almost exclusively negative), insider requests of the victim organization (often unmet), and insider access of victim organization IT systems are the most common descriptors. For the purposes of this study, we distinguish "Fired," "Terminated," and "Resigned," as follows:

- Fired: The point in time when the insider is notified his or her employment is being involuntarily terminated.
- Resigned: The point in time when the insider notifies the organization of a voluntary termination of employment.
- Terminated: The point in time when the insider's employment actually ends. This could be a point in time after notice of firing or resignation is actually given.

Table 17: The Most Common Event Descriptors Identified Prior to Zero Hour

Actor	Action	Target	# of Events	% of all Events
Victim Org	Hired	Insider	33	7.35
Insider	Accessed	Victim Org IT	30	6.68
Insider	Felt	N/A	23	5.12
Victim Org	Fired	Insider	12	2.67
Victim Org	Terminated	Insider	10	2.23
Insider	Requested	Victim Org	10	2.23
Insider	Other	Victim Org Property	10	2.23
Insider	Stole	Victim Org Property	8	1.78
Insider	Resigned	Victim Org	8	1.78
Insider	Installed	Victim Org IT	8	1.78
Victim Org	Contracted	Insider	7	1.56
Law Enforcement	Arrested	Insider	7	1.56
Victim Org	Denied	Insider	6	1.34
Insider	Other (Undefined Action)	Victim Org IT	5	1.11
Victim Org	Other Interpersonal Actions	Insider	5	1.11
Insider	Logged in	Victim Org IT	5	1.11
Insider	Posted	Internet	5	1.11
Victim Org	Demoted	Insider	5	1.11
Insider	Lied/Concealed	Victim Org	5	1.11
Insider	Changed Financial Status	N/A	4	0.89
Victim Org	Promoted	Insider	4	0.89
Insider	Was Tardy/Absent	Victim Org	4	0.89
Victim Org	Denied Physical Access	Insider	4	0.89
Court	Other Criminal Justice Actions	Insider	4	0.89
Victim Org	Expressed Concern	Insider	4	0.89
Insider	Executed	Victim Org IT	4	0.89

8.5 Analysis

Revisiting our hypotheses about the data, we can make the following observations:

- *Hypothesis 1: Indicators of malicious insider activity represented in our cases are demonstrated in nontechnical, or behavioral, events prior to technical events in more than 50% of the cases.*

  - Result: Supported. In cases with observable insider actions prior to attack zero-hour, the malicious insider demonstrated a behavioral action prior to technical action in 34 of 44 (78%) cases.
- *Hypothesis 2: The time difference between the first observable action that clearly enables the attack and the initial moment of damage to an organization's cyber systems is less than 1 day in more than 50% of the cases.*
  - Result: Unsupported. Of 49 cases, 17 (35%) had an identifiable event clearly enabling the attack less than one day prior to attack.
- *Hypothesis 3: Behavioral indicators of malicious insider activity are more prevalent than technical indicators, among the cases studied.*
  - Result: Supported. Of the 449 events analyzed, 352 (78%) were categorized as behavioral, and 97 (22%) were categorized as technical.
- *Hypothesis 4: Of the insider threat cases studied, 90% will have an observable event indicating potential malicious activity prior to the moment of actual damage to the victim organization's IT systems.*
  - Result: Supported. Of 49 cases, 44 (90%) contained observable events by the insider prior to attack.

While these results are preliminary and limited to the cases available, they do indicate focus areas an organization could address to detect early indicators of insider threat. With respect to behavioral actions, our analysis indicates that there is likely a great deal of individual interaction between the insider and the victim organization. Our findings reinforce previous literature on insider sabotage suggesting that insider behavioral actions often occur before technical actions [Claycomb 2012]. This suggests that organizational groups should receive further training or resources about detecting and reporting behavioral indicators.

With respect to technical actions, the second most frequent technical action event found after system access was installation of software on organization IT systems, which frequently enabled malicious actions to occur. This underscores the importance of situational awareness of an organization's IT systems, including hardware and software, for detecting unauthorized changes. Given the technology available today to monitor and manage organizational IT systems, further research could be performed to better understand this technical behavior as a potential early indicator.

8.6 Collaboration

We collaborated with several subject matter experts during this project. The primary purpose was to review the development of our methodology for analyzing cases. Collaborations included consultations with Dr. Roy Maxion from the Computer Science Department of Carnegie Mellon University and Dr. David Zubrow, Chief Scientist of Software Engineering Process Management from the Software Engineering Institute.

8.7 Publications and Presentations

Our initial work is detailed in an SEI blog post at http://blog.sei.cmu.edu/post.cfm/enabling-and-measuring-early-detection-of-insider-threats.

Initial findings were published in the proceedings of the *4th International Workshop on Managing Insider Security Threats* [Claycomb 2012]. At least one in-depth publication outlining the triad and initial results is also anticipated.

8.8 References

[Brackney 2004]
Brackney, R., & Anderson, R. *Understanding the Insider Threat: Proceedings of a March 2004 Workshop*. RAND Corporation, 2004.

[Cappelli 2006]
Cappelli, D.; Desai, A.; Moore, A.; Shimeall, T.; Weaver, E.; & Willke, B. J. *Management and Education of the Risk of Insider Threat (MERIT): System Dynamics Modeling of Computer System Sabotage*. Software Engineering Institute, Carnegie Mellon University, 2006. http://www.cert.org/archive/pdf/merit.pdf

[Claycomb 2012]
Claycomb, W.; Huth, C.; Flynn, L; McIntire, D; & Lewellen, T. "Chronological Examination of Insider Threat Sabotage: Preliminary Observations." *International Workshop on Managing Insider Security Threats (MIST 2012)*, 2012. http://www.cert.org/archive/pdf/CERT_CodingSabotage.pdf

[CSO Magazine 2011]
CSO Magazine, U.S. Secret Service, Software Engineering Institute, & Deloitte. "2011 Cyber Security Watch Survey." *CSO Magazine*, January 2011.

[East 1982]
East, M., & Herman, M. Ch. 5, "Targets in Foreign Policy Behavior," 115-132. *Describing Foreign Policy Behavior*, SAGE Publications, 1982.

[Fleiss 1971]
Fleiss, J. "Measuring Nominal Scale Agreement among Many Raters." *Psychological Bulletin*, *76*, 5: 378-382.

[Greitzer 2011]
Greitzer, F., & Paulson, P. "Modeling Human Behavior to Anticipate Attacks." *Journal of Strategic Security IV*, 2 (2011).

[Hunker 2011]
Hunker, J., & Probst, C. "Insiders and Insider Threats - An Overview of Definitions and Mitigation Techniques." *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications 2*, 1 (2011).

[Keeney 2005]
Keeney, M.; Kowalski, E.; Cappelli, D.; Moore, A.; Shimeall, T.; & Rogers, S. *Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors*. Carnegie Mellon University, Software Engineering Institute, 2005. http://www.cert.org/archive/pdf/insidercross051105.pdf

[Killourhy 2009]
Killourhy, K. S., & Maxion, R. A. "Comparing Anomaly Detectors for Keystroke Dynamics," 125-134. *Proceedings of the 39th Annual International Conference on Dependable Systems and Networks (DSN 2009)*. Estoril, Lisbon, Portugal, June 29-July 2, 2009. IEEE Computer Society Press, 2009.

[Maybury 2005]
Maybury, M.; Chase, P.; Cheikes, B.; Brackney, D.; Matzner, S.; Hetherington, T.; Wood, B.; Sibley, C.; Marin, J.; Longstaff, T.; Spitzner, L.; Haile, J.; Copeland, J.; & Lewandowski, S. "Analysis and Detection of Malicious Insiders." *Proceedings of the 2005 Intl. Conference on Intelligence Analysis*, 2005.

[Moore 2008]
Moore, A.; Cappelli, D.; &Trzeciak, R. *The "Big Picture" of Insider IT Sabotage Across U.S. Critical Infrastructures* (CMU/SEI-2008-TR-009). Carnegie Mellon University, Software Engineering Institute, 2008. http://www.sei.cmu.edu/library/abstracts/reports/08tr009.cfm

[Shaw 1998]
Shaw, E.; Ruby, K.; & Post, J. "The Insider Threat to Information Systems: The Psychology of the Dangerous Insider." *Security Awareness Bulletin 2*, 98 (September 1998): 27-46.

[Sweller 1998]
Sweller, J. "Cognitive Load During Problem Solving: Effects on Learning." *Cognitive Science 12*, 2 (1998): 257-285.

8.9 Appendix A

Table 18: Actor/Target Triad Categories

IT	Not IT	Criminal Justice	Outside	Unknown	Null and Other
• Victim Org IT • Other Org IT • Internet • Insider's IT • Unknown • Other	• Insider • Himself/herself • Property • Victim Org • Supervisor • Management • Co-worker • Subordinate • Organization Group (e.g., HR/IT) • Property • Unknown	• Court • Authorities	• People/Property • Organizations	• Insider • Victim Org • Outside • Other • Unknown	• Null: used for life events (no target) • Other: used for any other actor or target

Table 19: Not-IT-to-Not-IT Action Triad Categories

Interpersonal	Criminal Justice	Psych/Medical
• Made Agreement • Broke Agreement • Bullied/Intimidated/ Threatened • Argued • Bragged • Stole • Denied • Approved • Complained • Requested • Blamed • Mitigated • Contacted • Reported • Flirted • Regretted • Expressed Concern • Lied/Concealed • Communicated • Hired • Contracted • Terminated • Resigned • Fired • Laid Off • Transferred • Sanctioned • Rewarded • Promoted • Demoted • Discovered • Investigated • Reorganized • Was tardy/absent • Other	• Arrested • Pleaded • Sentenced • Reached Verdict • Searched • Identified • Sued • Seized • Investigated • Analyzed • Evaded • Indicted/Charged • Other	• Diagnosed • Treated • Other

Table 20: Not-IT-to-IT, IT-to-IT, and IT-to-Not-IT Triad Actions

Not IT to IT	IT to IT	IT to Not IT

• Logged in
• Logged off
• Accessed
• Executed
• Moved
• Copied
• Exploited
• Analyzed
• Posted
• Downloaded
• Printed
• Read
• Delayed
• Deleted
• Modified
• Created
• Disabled
• Recovered
• Installed
• Physical Contact
• Attacked
• Discovered
• Investigated
• Restricted Access
• Other
	• Executed Scheduled Programs
	  ○ Scripts
	  ○ Backup Jobs
	  ○ File Transfers
	  ○ Other
		• Alerted
		• Other

Table 21: Not-IT-to-Null Action Triad Categories

Changed Locations	Felt/Changed Health Status/ Other	Financial	Personal	Family
• Traveled • Moved	• Three separate categories used to code the insider's feeling, changes in mental or physical health, and other information about the insider, respectively	• Changed financial status • Sold • Purchased	• Drank • Did drugs • Gambled • Other	• Had family problem • Changed family status

8.10 Appendix B

Table 22: Level 1 Frequencies

Actor	Percent of Total	Target	Percent of Total
IT	.5%	IT	21.6%
NOT IT	99.5%	NOT IT	68.8%
		Null	9.6%

Table 23: Highest Level 2 Frequencies

Actor	Percentage of Total	Target	Percentage of Total	Action	Percentage of Total
Insider	52.7%	Insider	37.6%	Interpersonal Actions	64.1%
Victim Organization	36.8%	Victim Organization	25.2%	Accessed	7.3%
Outside	6.2%	Victim Org IT	18.0%	Felt	5.1%
Criminal Justice	3.6%	Blank	9.8%	Criminal Justice Actions	4.0%
Other	.7%	Outsider	3.6%	Installed	2.2%
		Unknown	2.2%	Other	17.3%
		Other	3.6%		

Table 24: Highest Level 3 Frequencies

Actor	Percent of Total	Target	Percent of Total	Action	Percent of Total
Insider Individual	52.6%	Insider Individual	37.2%	Blanks	28.3%
Organization Group	29.4%	Blank	32.7%	Hired	9.1%
Supervisor	4.2%	Organization Group	12.0%	Communicated	5.6%
Organizations	3.1 %	Property	5.8%	Fired	3.6%
Other	10.7%	Co-worker	3.3%	Requested	3.8%
		Other	9.0%	Other	49.6%

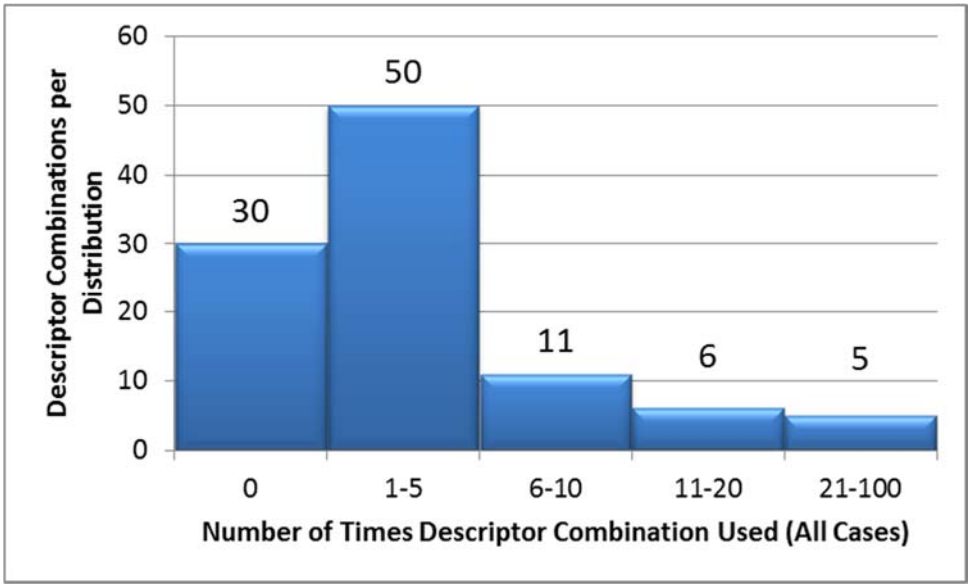

Figure 29: Distribution of Descriptor Usage for All Available Descriptor Combinations

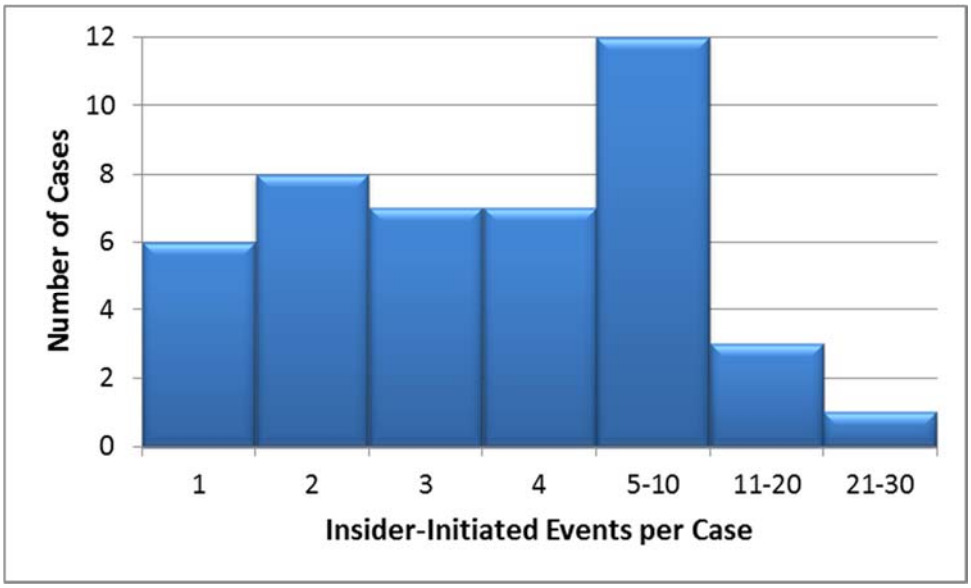

Figure 30: Distribution of Insider-Initiated Events per Case

9 Quantifying Uncertainty for Early Lifecycle Cost Estimation (QUELCE)

Julie Cohen
Robert Ferguson
Dennis Goldenson
Jim McCurley
Robert Stoddard
Dave Zubrow

9.1 Purpose

The Government Accountability Office (GAO) has frequently cited poor cost estimation as a significant contributor to cost overruns, and the problems are increasing [GAO 2011, GAO 2012]. The growth in major defense acquisition programs' (MDAPs) research, development, test and evaluation (RDT&E) costs from the first estimate rose from 27% in 2000 to 54% in 2011, and delay in delivering initial capabilities rose from 16 to 23 months. As depicted in Figure 31, difficult program cost experiences, such as with the Future Combat Systems program, led to the Weapons Systems Acquisition Reform Act (WSARA) in 2009. The WSARA requires cost estimates for Milestone A approval in an attempt to prevent and mitigate such cost growth [WSARA 2009]. However, program cost experiences since WSARA, such as the Ground Combat Vehicle depicted in Figure 31, demonstrate that traditional methods of cost estimation prove insufficient since they do not account for program execution change and uncertainty during the DoD acquisition life cycle. These traditional methods rely primarily on analogy, limited information, and idiosyncratic judgment. The estimation process the SEI is developing, referred to in this report as Quantifying Uncertainty for Early Lifecycle Cost Estimation (QUELCE), overcomes these challenges by representing uncertainty—and therefore risk—in the assumptions and explicitly modeling them. This allows DoD decision makers reviewing major defense acquisition programs MDAPs to make more informed choices and fund programs at levels consistent with the magnitude of risk to achieving success, leading to fewer and less severe program cost overruns [RAND 2007].

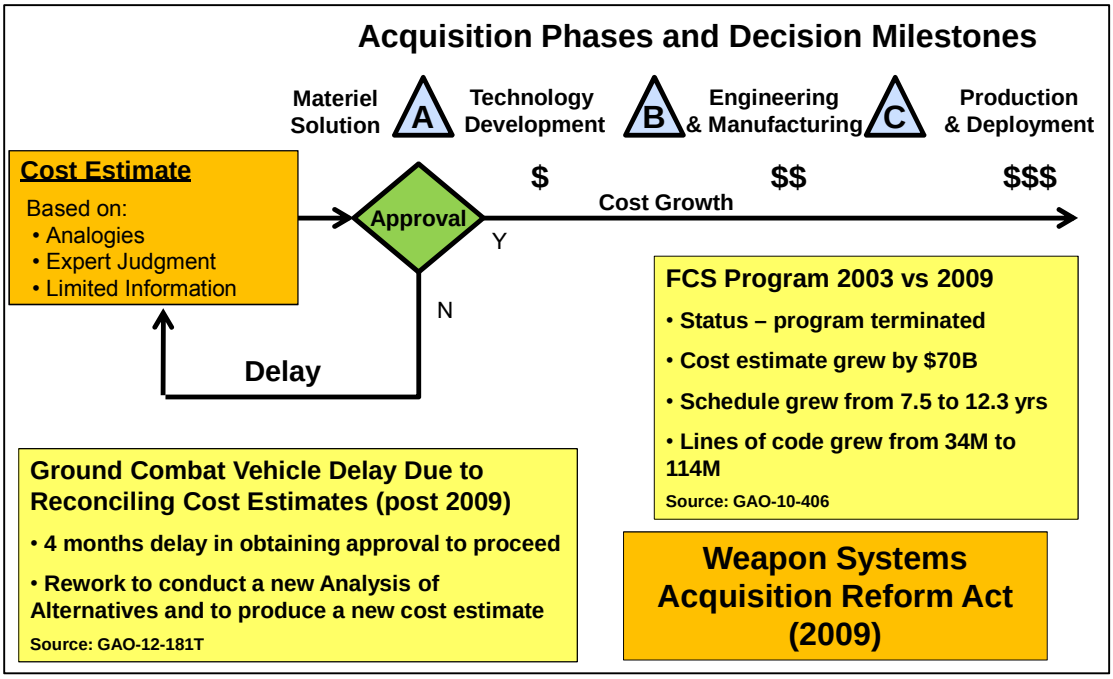

Figure 31: DoD 5000 Acquisition Lifecycle

Building on the early use of experts in estimating unprecedented weapons systems, QUELCE also focuses on the development and validation of methods for calibrating expert judgment. The research literature and experience show that experts are often overly confident and optimistic in their judgments. Consequently, QUELCE uses proven training techniques to improve experts' accuracy in judgments and predictions while also capturing information regarding their uncertainty, which is also useful in the QUELCE cost estimation method.

9.2 Background

Figure 32 depicts recent work applicable to the challenge of developing cost estimates early in the acquisition life cycle. The primary criteria differentiating this research from the related work consists of the adequacy of the cost inputs at pre-Milestone A and the degree to which likely sources of change are accounted for in the cost estimation modeling. Notably, much of the recent DoD research into early lifecycle cost estimation surrounds the concept of capability-based cost estimation [Book 2009, Garvey 2011, Roper 2010]. Related research to the DoD capability-based cost estimation includes identification of Monte Carlo simulation as a modeling technique and use of analogy during cost estimation [GAO 2009, Angelis 2000]. Research involving dependence on expert judgment includes a number of recent studies [Don 2007, Dysert 2006, Gino 2011, Hubbard 2010, Jorgensen 2005]. Additionally, aspects of the QUELCE solution to the early lifecycle cost estimation problem point to recent research into Bayesian Belief Network modeling, design structure matrices, and scenario planning [[Hamdan 2009, Lindemann 2013, Lindgren 2009]. Our research pursues a solution to meet the gap shown in Figure 32 while also leveraging the existing body of knowledge of cost estimation and the assortment of existing cost estimation tools.

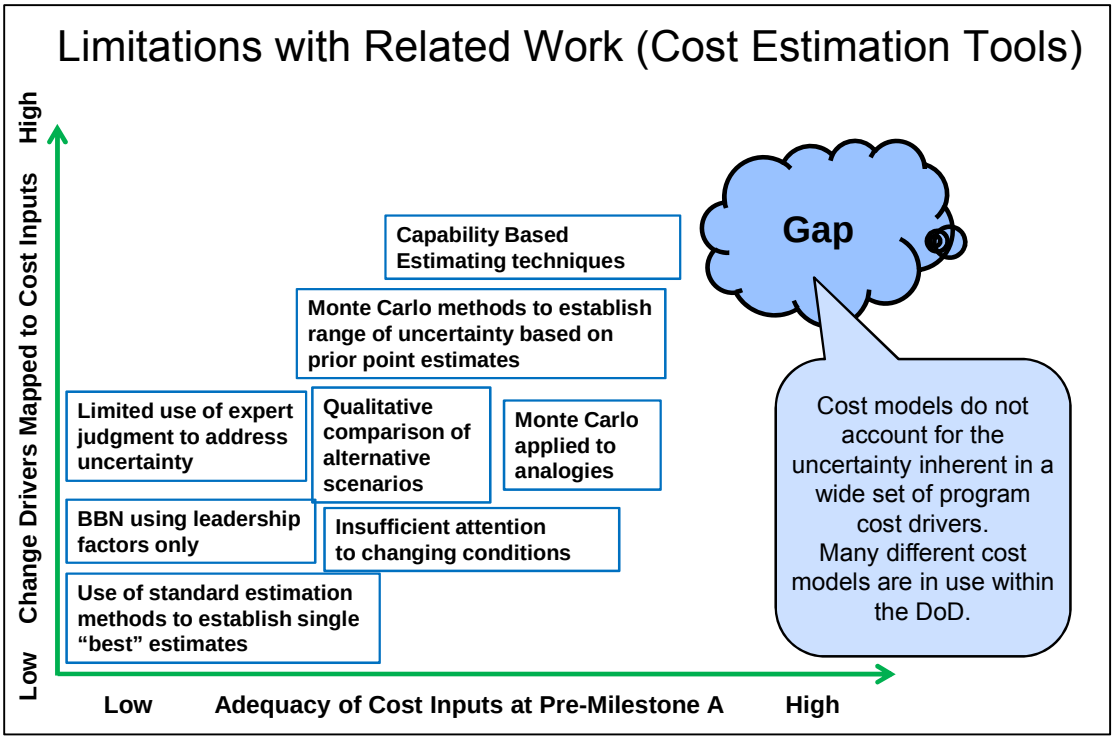

Figure 32: Gap in Related Cost Estimation Research

9.3 Approach

The primary criteria differentiating this research from the related work consists of the adequacy of the cost inputs at pre-Milestone A and the degree to which expected sources of change are accounted for in the cost estimation modeling. This research pursues a solution to meet this gap while also leveraging the existing body of knowledge of cost estimation and the assortment of existing cost estimation tools.

The QUELCE method, as shown in Figure 33, incorporates a number of innovative aspects as compared to traditional parametric and analogy-based cost estimation.

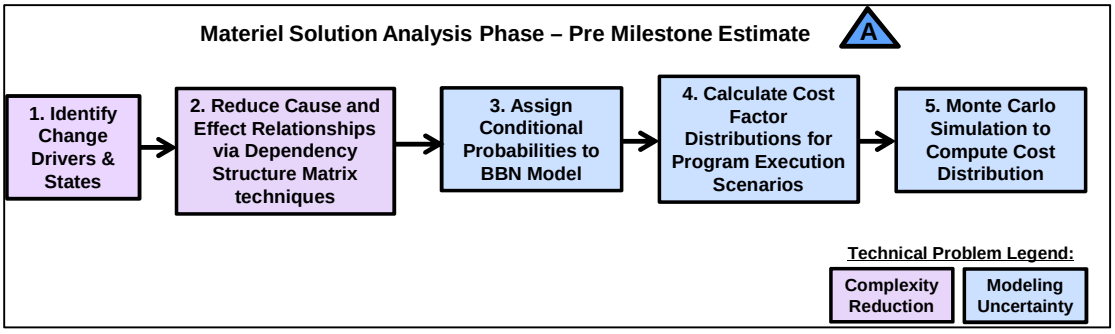

Figure 33: Innovative Portions of the QUELCE Method

The QUELCE method seeks to model the uncertainty on the input side of the cost estimation analysis, rather than on the output of the cost estimate. In this fashion, the method uses information

about the drivers of change, which can cascade into cost growth. To accomplish this, a crucial step of eliciting such change drivers from domain experts, as well as the interrelationships of these drivers, occurs using a dependency structure matrix (DSM) shown in Figure 34. The DSM matrix, leveraged from the scenario planning methodology popularized by Shell Oil, directly enables the construction of a Bayesian Belief Network (BBN) model (shown in

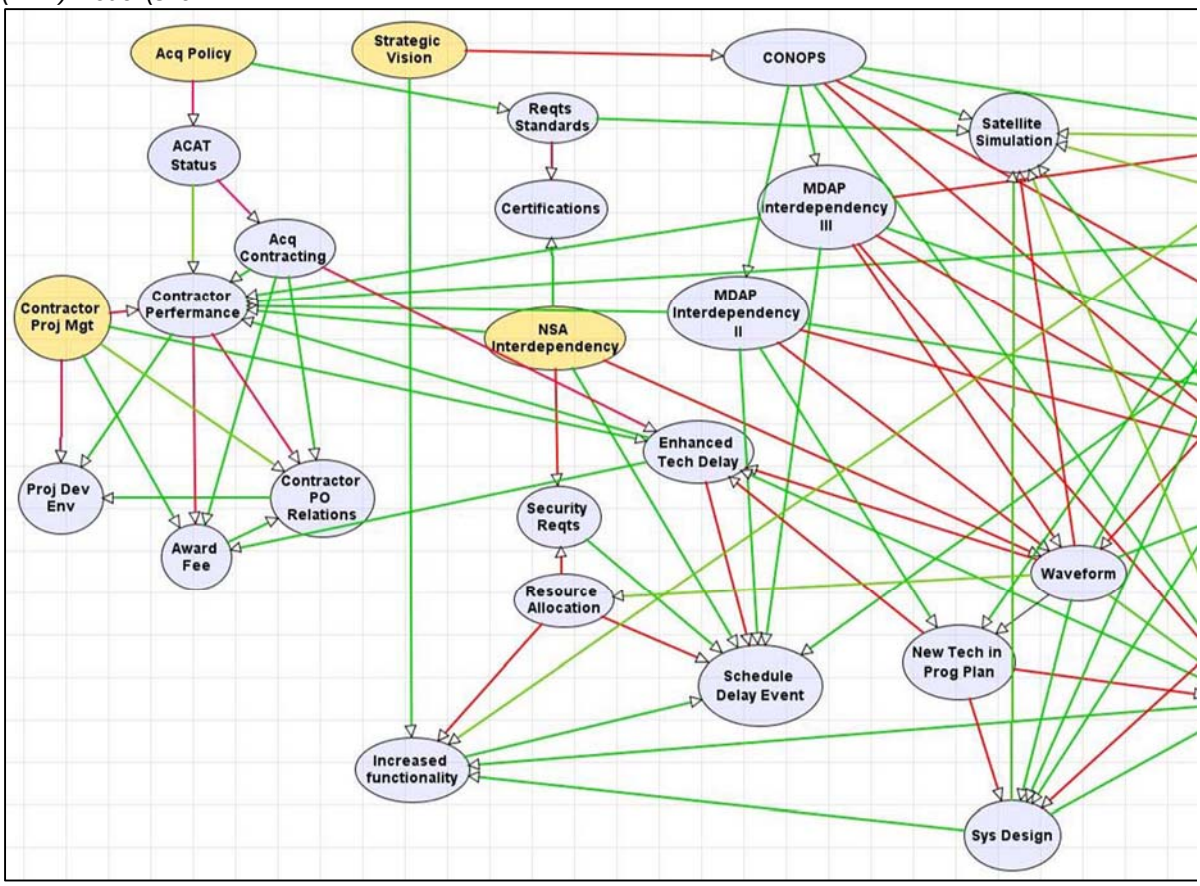

Figure 35), thereby providing a probabilistic approach to understanding the cascading effects of change on cost. The BBN determines output factors which are associated mathematically with the input factors of the MDAP's cost estimation models. Most notable at this point is the use of probability distributions within the BBN output nodes, thereby enabling probability distributions for each of the input factors of the client's cost estimation tool.

The capture of uncertainty as represented by the conditional probabilities in the BBN nodes represents a major innovative practice. By affording probability distributions of the input factors, QUELCE drives the existing cost estimation model to produce a probability distribution for the cost estimate (shown in Figure 36) allowing decision makers to select the cost estimate that meets their risk tolerance. This method allows alternative scenarios of program execution to be estimated and compared. The approach provides detailed insight as to the sources of risk and their influence on the cost estimate.

Note, however, that the QUELCE approach evokes a set of tough technical challenges including (1) data mining sensitive cost variance data from DoD MDAP programs;(2) characterizing the

uncertainty of expert judgment that QUELCE so fundamentally depends on; and (3) connecting the BBN to a host of vendor cost estimation tools and DoD cost estimating relationships. The research team remains well positioned to overcome these challenges based on internal expertise, collaboration with the university research community, and a close relationship with DoD and Service cost analysis organizations.

The QUELCE method relies on expert judgment at several steps, including (1) the identification of likely "program change drivers" that can affect the execution of a given project over its lifecycle; (2) the identification of states within a change driver for building scenarios; (3) the probability of a change driver departing from a nominal (planned) status; and (4) the nature and strength of the cause–effect relationships between and among change drivers. Consequently, expert judgment must be consistently dependable and repeatable to be used credibly within the QUELCE method. A method is needed to ensure that expert judgment may be satisfactorily calibrated before experts participate in the QUELCE method.

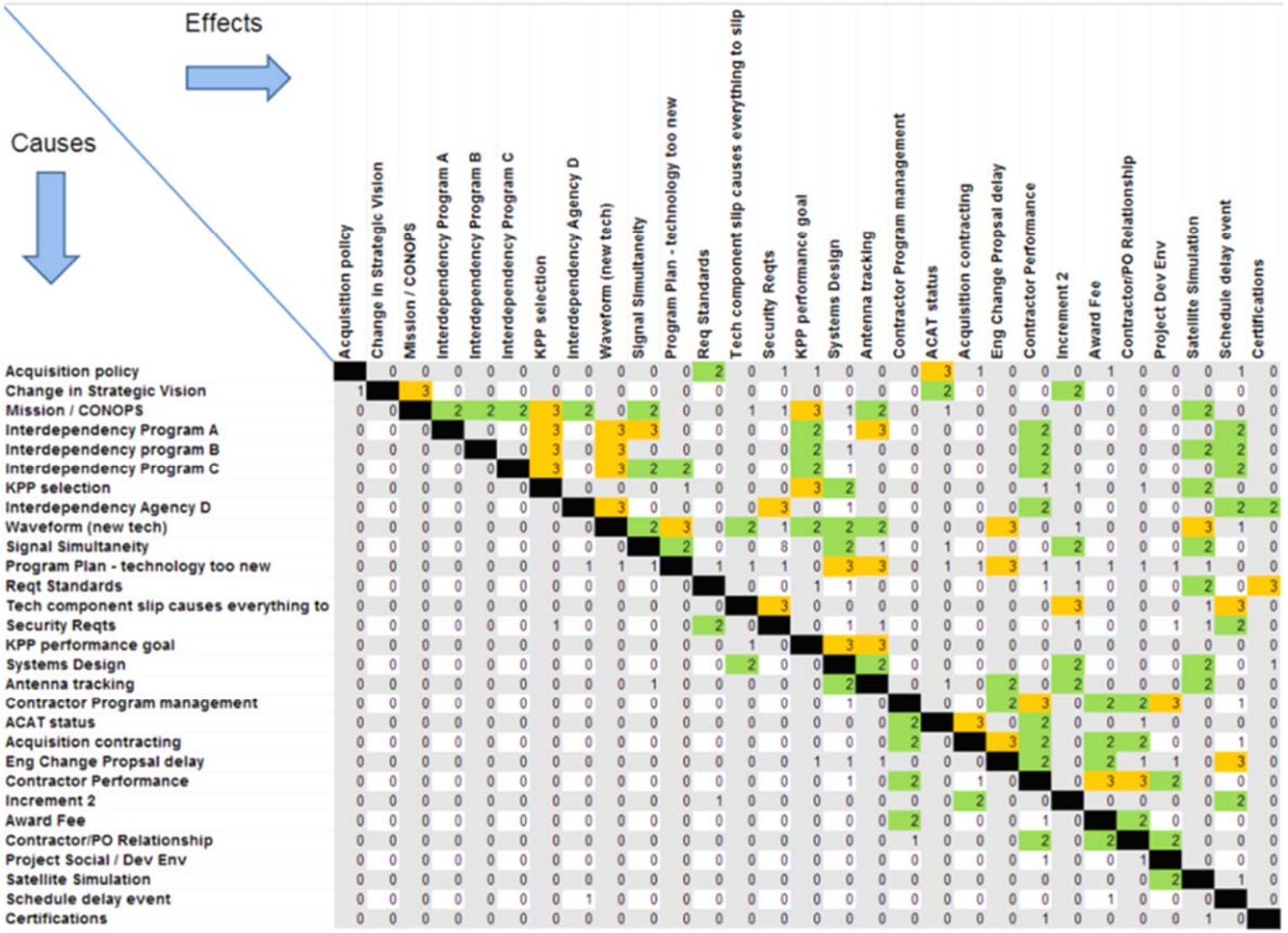

Figure 34: The QUELCE DSM Matrix

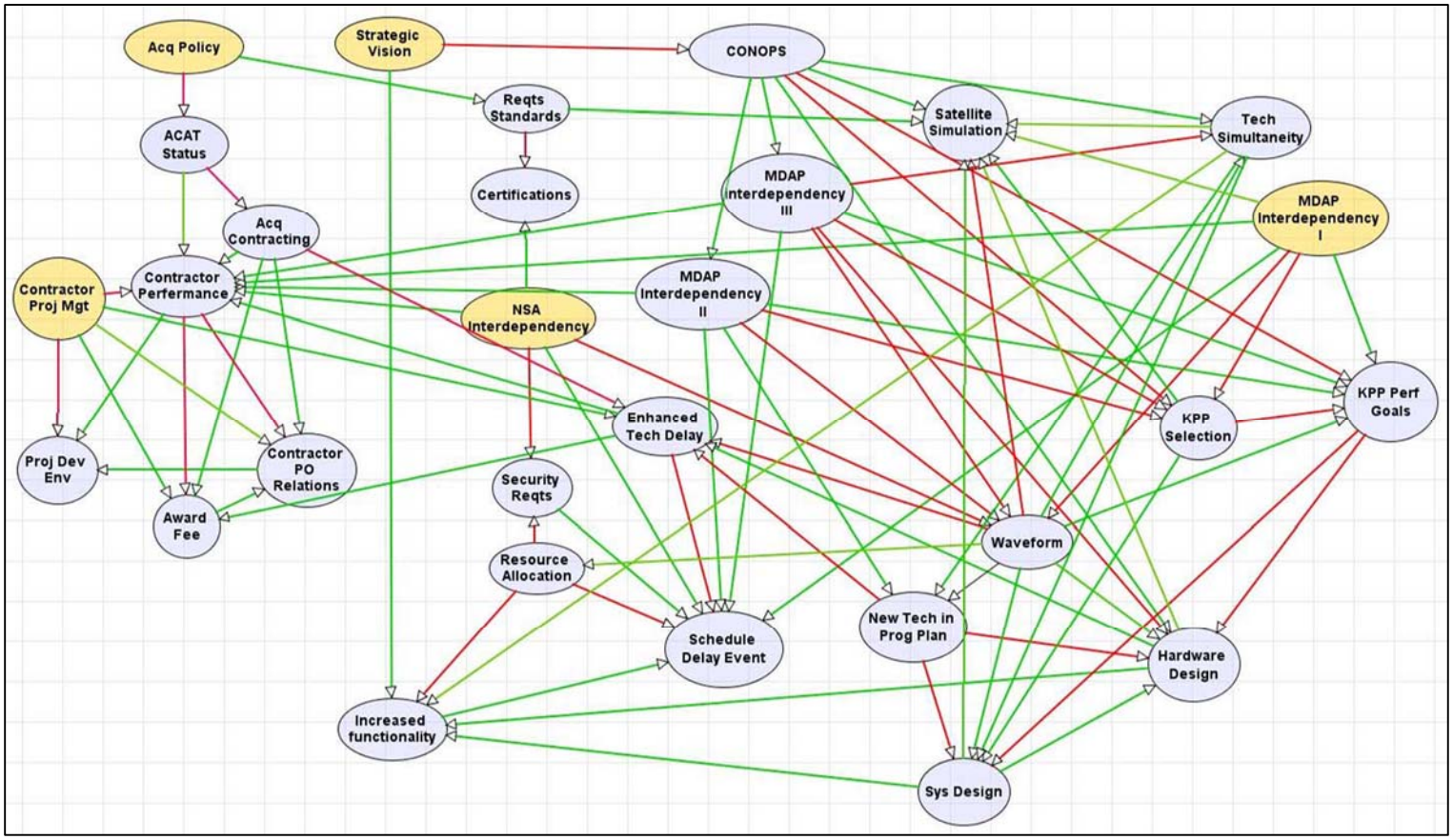

Figure 35: The QUELCE Bayesian Belief Network

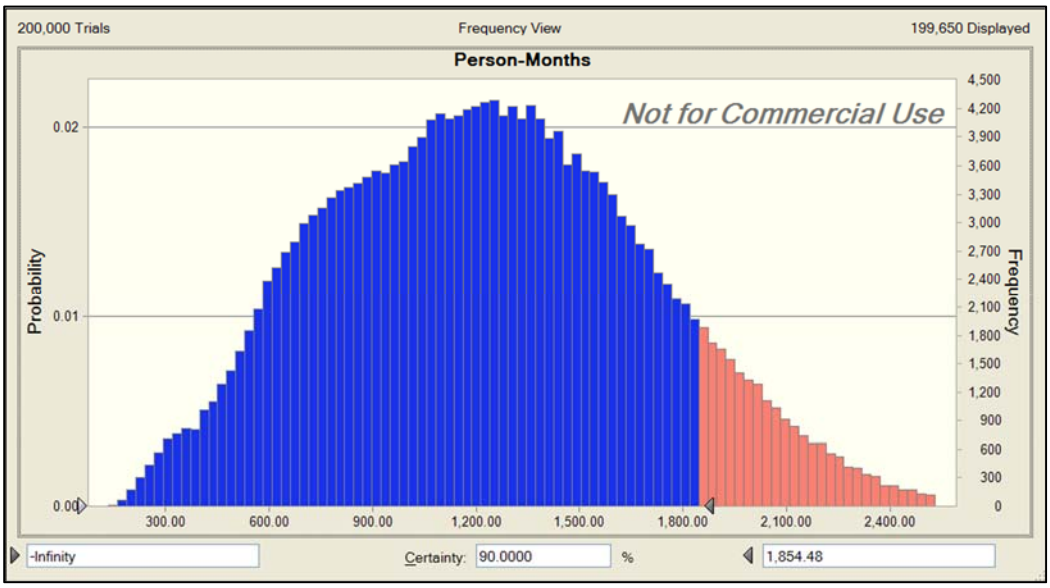

Figure 36: Monte Carlo Simulation Output of Cost Estimate

Although our goal is to apply information about uncertainty that is generated in the conceptualization of an MDAP, domain experts have a detailed knowledge and understanding of the specific issues. This information is not currently utilized in forming cost estimates. QUELCE collects past MDAP experiences regarding program change drivers to construct a repository of such information to form domain specific "reference points." These domain reference points comprise data about changes in a program and the subsequent difference in estimated costs versus actual costs. The domain experts will be prompted at different points throughout the QUELCE method to access the domain reference points housed in the QUELCE repository to aid their judgment regarding estimates of probabilities and values. Prior to accessing the repository in a QUELCE workshop, experts will access just-in-time virtual training for calibration by domain. For example, domain experts may be participating in QUELCE to develop a cost estimate for a new communication system that involves satellite technology. If the domain experts have not recently completed virtual calibration training for that domain, they may receive a refresher course consisting of a two- to four-hour online exercise. This approach to improving expert judgment will help domain experts during the following three steps in the QUELCE method:

1. **Identifying pertinent change drivers**. After completing the training, domain experts will be asked to participate in a workshop exercise that anticipates which change drivers will most likely be relevant to a particular program. In the workshop, domain experts will query for communication programs or specific technology names related to particular programs. In the example involving the new communication system above, the results should yield information related to historical communication programs or technologies and domain reference points, explaining why certain aspects went over budget or schedule.
2. **Populating the change driver cause-and-effect matrix**. The next judgment involves the change driver dependency structure matrix. The domain experts will evaluate each change driver and rate the probability, on a scale of 0 to 3, that the change driver will cause any other change drivers on the list to switch from a nominal to an off-nominal condition. This exercise requires judgment about the relationships between change drivers. The domain

expert will receive assistance from querying the repository before rendering this type of judgment. For example, reference points might include historical information about a change driver going off-nominal and subsequently causing three other change drivers to go off-nominal. The reference points therefore give the domain expert a basis to understand the relationships between change drivers and their probability of cascading.

3. **Establishing probabilities for the Bayesian Belief Network (BBN)**. The BBN models the change drivers as nodes in a quantitative network, including probabilities that changes of state in one node will create a change of state in another node. Every change driver has a parent table that presents all the possible combinations of the parent change driver states. For example, in our BBN, we have change driver A and change driver B, and both have an influence on change driver C. If change driver A has nominal and off-nominal states and change driver B has nominal and off-nominal states, there are four different combinations of A and B change driver states that may affect change driver C. These combinations of change driver states can be thought of as different scenarios of program execution. Once calibrated through the online calibration training, the subject matter experts will then participate in a one-day QUELCE workshop to discuss relevant program execution change drivers and their relationships while accessing a repository of domain reference points.

The repository will include a searchable database of domain reference points that helps the experts exercise better judgment during cost estimation. Experts will be able to query the reference points using keywords based on search technology. Search results will show the key reference points in relation to the domain and technology challenge. The expert(s) will then review those reference points before formulating their quantitative inputs (e.g., probabilities and values) for the QUELCE model.

In summary, our research into the QUELCE method for pre-Milestone A cost estimation represents a significant advance by enabling the modeling of uncertain program execution scenarios that are dramatically different from the traditional cost factor inputs of cost estimation models currently employed in the DoD acquisition life cycle. By synergizing the latest advancements in proven methods, such as scenario planning workshops, cause–effect matrices, BBNs, and Monte Carlo simulation, we have created a novel and practical method for early DoD acquisition lifecycle cost estimation.

9.4 Evaluation Criteria

Two hypotheses were identified at the beginning of this research to validate the solution significance from both an internal and external validity standpoint.

Hypothesis 1: Modeling uncertainty of interdependent cost drivers produces more credible distributions of cost compared to current methods.

The original plan consisted of up to three DoD pilot uses of the QUELCE method producing cost estimates that would be compared to cost estimates conducted using traditional methods. The lack of access to current MDAP programs to pilot QUELCE caused the team to exercise a backup plan and conduct a Problem Challenge workshop with SEI staff followed by a retrospective of a recently terminated MDAP program in which several SEI staff had previously worked. Results from both activities were used to confirm that the research team was working on the right problem

and had a compelling technical solution capable of overcoming the recognized challenges for early lifecycle cost estimation.

Hypothesis 2: Cost-related judgment is improved using generic and domain-specific reference points by at least 30 and 60 percentage points, respectively.

This hypothesis was tested using a series of expert judgment calibration experiments as follows:

- Three experimental groups with the same domain-specific final test
- Group 1: Generic calibration training and testing
- Group 2: Domain-specific calibration training and testing
- Group 3: Generic calibration training and testing, followed by domain-specific training and testing

The team confirmed the improvement of both the generic and domain-specific judgment calibrations using nonparametric hypothesis tests. Most notably, the team confirmed improvement leaps of 30 and 60 percentage points as hypothesized above, providing solid confidence in the value of providing experts with access to domain reference points within a QUELCE repository.

The following three hypotheses were not tested in the FY12 research because of the delays in accessing DoD MDAP program-execution data and difficulties in locating a current MDAP to pilot QUELCE:

- Hypothesis 1: 80% of experts report higher confidence in the QUELCE estimates.
- Hypothesis 2: The cost probability distribution produced by a reduced set of drivers is not statistically different from one that considers all drivers.
- Hypothesis 3: Teams of experts with calibration training will reach consensus 50% faster.

The SEI plans to test all three of these hypotheses in FY13 as part of continued line-funded research of this project.

9.5 Results

In all, there were four focus areas for this research to include: (1) building an initial QUELCE repository; (2) conducting expert judgment calibration experiments; (3) connecting the BBN model to the front end of a cost estimation tool; and (4) conducting a retrospective in lieu of a live MDAP pilot use of QUELCE.

9.5.1 QUELCE Repository

A special server was established to house the wealth of information gained from documents labeled “for official use only” (FOUO) from many disparate sources within the OSD and the Services. These documents will be used to formulate program change drivers that include detailed information regarding what was planned or expected versus what actually occurred during program execution. Once an initial data mining of these early artifacts began, the team also developed a template for the domain reference points to structure the necessary context data needed in the repository.

Sources of data for the QUELCE Repository include:

- initial capability document
- analysis of alternatives (pre-Milestones A,B,C)
- capability-based assessment
  - functional area analysis
  - functional needs analysis
  - functional solutions analysis
- service cost estimate
- independent cost estimate
- capability development document (pre-Milestone B)
- capability production document (pre-Milestone C)
- preliminary design review
- critical design review
- engineering change proposals
- software resources data reports
- contractor cost data reports
- earned-value-management reports
- defense acquisition executive summaries
- selected acquisition reports
- integrated master plan schedules
- engineering change proposals
- technology readiness reports
- probability of program success gate reviews

In FY12 work on the repository focused on extracting information from these documents as part of the initial effort to develop program change drivers and build domain reference points for use in calibration.

9.5.2 Expert Judgment Calibration Experiments

With regard to the second focus of expert judgment calibration, the team conducted three expert judgment calibration experiments. These experiments enabled the team to quantify the expected learning curve of judgment calibration, confirmed the experts' benefit of domain-specific reference points, prompted the automation of the expert judgment tests, and simplified the procedure to elicit expert judgments (e.g., it addressed the language of probability and ensured the right question would be answered). Lastly, the experiments confirmed the need for future experiments focused on measuring the calibration of judgments from group settings of experts.

A total of 36 individuals from three separate groups participated in the experiments: (1) Carnegie Mellon graduate students from the School of Computer Science's Master of Software Engineering program and a few members of the SEI technical staff; (2) members of a master class of adult learners in Australia; and (3) graduate students from Carnegie Mellon's Heinz College

concentrating on software engineering and information technology along with two more computer science students. All of the participants had previous industrial experience.

The calibration training provided guidance about how to make more realistic judgments, tempered with a degree of confidence that reflected the participants' actual knowledge. That guidance was followed by a series of calibration exercises, each of which included a battery of factual questions that asked the trainees to provide upper and lower bounds that they were 90 percent certain included the correct answer to each question. Each test battery was followed immediately by a brief review of the correct answers. A short discussion at the end of the training provided further guidance about ways to explicitly consider interdependencies among related factors that might affect the basis of one's best judgments under uncertain circumstances.

A total of 29 individuals from all three groups completed three batteries of software engineering domain-specific test batteries. A total of 14 participants from the first study group also completed four batteries of generic knowledge questions.

Results from both sets of questions showed improvement over the test batteries with respect to recognition of the participants' true uncertainty. The domain-specific training was accompanied by notable improvements in the relative accuracy of the participants' answers when we introduced additional contextual information to the questions along with reference points about similar software systems. Moreover, the additional contextual information in the domain-specific questions and reference points helped the participants improve the accuracy of their judgments while also reducing their uncertainty in making those judgments.

Examples of the questions from the generic tests used with permission from Hubbard include

- How many feet tall is the Hoover dam?
- How many inches long is a 20-dollar bill?
- What percentage of aluminum is recycled in the U.S.?
- What year was Elvis Presley born?
- What percentage of the atmosphere is oxygen by weight?
- What is the latitude of New Orleans? Hint: Latitude is 0 at the equator and 90 at the North Pole.
- In 1913, the U.S. military owned how many airplanes?
- The first European printing press was invented in what year?
- What percentage of all electricity consumed in U.S. households was used by kitchen appliances in 2001?

The following are examples of questions from software engineering domain-specific calibration tests and questions from those tests.

- **Apache JAMES Project**: a complete and portable enterprise mail engine based on open protocols; also a mail application platform that allows processing emails (e.g., to generate automatic replies, update databases, filter spam, or build message archives)
  - What is the project's current codebase size in lines of code (LOC)?

- **LibreOffice**: a multi-platform, integrated office suite based on copyleft licenses and compatible with most document formats and standards; includes spreadsheet, word processor, chart, business productivity, presentation, database, Linux, C++ and other applications
  - How much total effort in person years has been spent on this project?
- **WebKit**: an open source web browser engine. The project`s HTML and JavaScript code began as a branch of the KDE (K Desktop Environment) libraries. WebKit is also the name of the engine used by Safari, Dashboard, Mail, and many other OS X applications. KDE is a GUI-based user interface primarily for Unix and Linux machines, but also available for Windows and Macintosh.
  - What is the current codebase size in LOC?
- **TkCVS**: a Tcl/Tk-based graphical interface to the CVS and Subversion configuration management systems. It will also help with RCS. The user interface is consistent across Unix/Linux, Windows, and MacOS X. TkDiff is included for browsing and merging changes.
  - How much total effort in person years has been spent on this project?

As may be seen in Figure 37, the experiments confirmed the learning curve of experts across a series of three or four calibration tests. The median score for the generic information results improved by 40% while the domain-specific results improved by 60%. The learning curve on the generic tests corroborated data seen by Hubbard in his research. Most notably from Figure 37, the testing of experts with software engineering domain questions with access to domain reference points resulted in significant improvement that was four times greater than the learning curve improvement of the generic calibration tests.

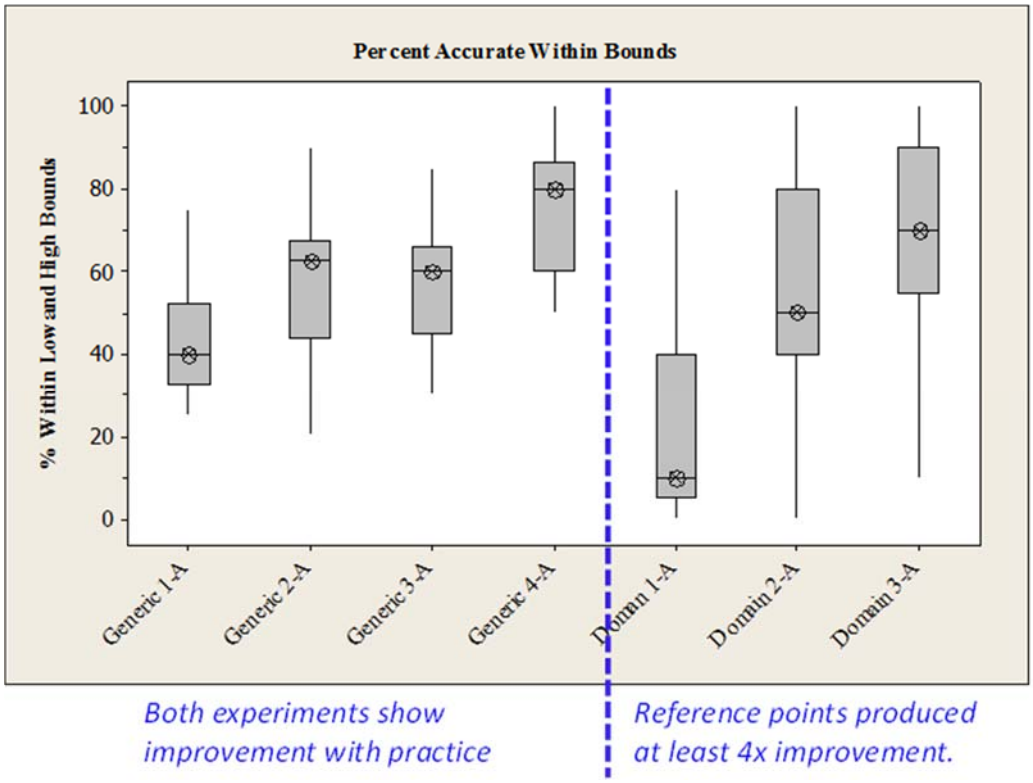

Figure 37: Accuracy-Within-Bounds by Test Battery

Lastly, Figure 38 depicts the post-experimental feedback from the subjects with regard to their perception of value added by access to the domain-specific reference points. The data clearly show a perceived strong value of the reference points when taking calibration tests. A more complete description of the experiments and their results can be found in "Quantifying Uncertainty in Expert Judgment: Initial Results" [Goldenson 2013].

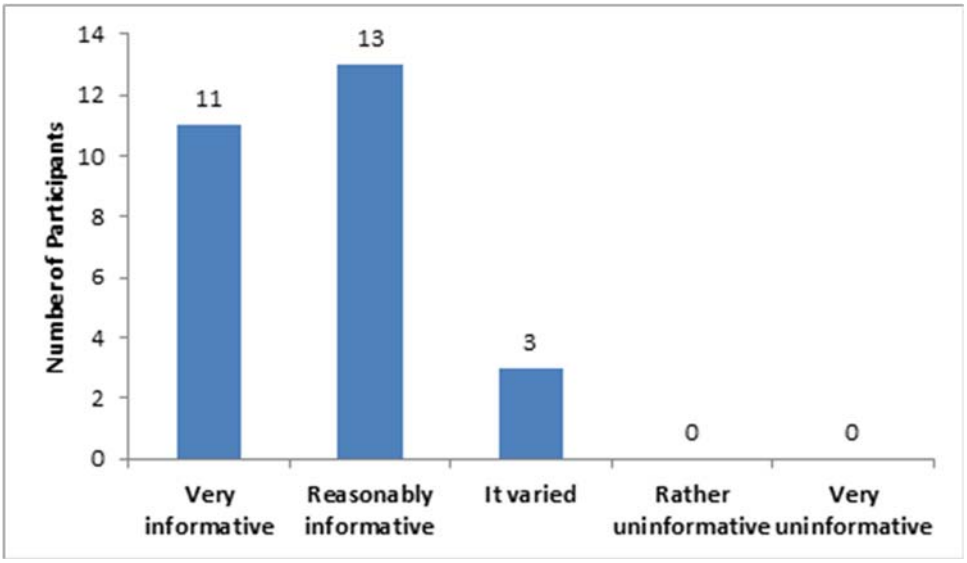

Figure 38: Informativeness of the Reference Points Tables

9.5.3 Connecting the BBN Model to Cost Estimation Models

In the third focus area of connecting the BBN to cost model tools, the team progressed beyond the initial connection or mapping of BBN output nodes to the COCOMO software cost estimation model by also developing a connection with the SEER-SEM cost model. This activity was more complex given the approximately 80 input factors to the SEER-SEM tool. However, collaboration with the SEER-SEM vendor enabled the connection to the BBN model setting the stage for validation of the connection in FY13.

9.5.4 Retrospective MDAP Analysis

The fourth focus area of applying QUELCE to a full-scale MDAP is considered vital to establishing QUELCE as a significant improvement to existing early life cycle cost estimation methods. After a live MDAP could not be located in a reasonable timeframe, the team decided to pursue unprecedented access to documentation for the Family of Advanced Beyond Line-of-Sight Terminals (FAB-T) program to construct a full-scale retrospective QUELCE analysis for an MDAP. The team accessed more than 4,100 FAB-T program files, which documented virtually all of the program's history. In addition, the team obtained more than 135 official contractor submissions of software resource data reports (SRDR) and earned value management (EVM reports to the Defense Automated Cost Information System (DACIMS) administered by the Defense Cost and Resource Center (DCARC) as part of OSD's Cost Assessment and Program Evaluation (CAPE). We also obtained 83 acquisition reports from the Defense Acquisition Management Information Retrieval (DAMIR) Purview repository, which included the FAB-T selected acquisition reports (SAR) and the defense acquisition executive summary (DAES)

reports. The documentation confirmed that much information generated early in the acquisition life cycle addresses programmatic and technical uncertainties and that such information was not captured by the tools and models used to establish the program cost estimate.

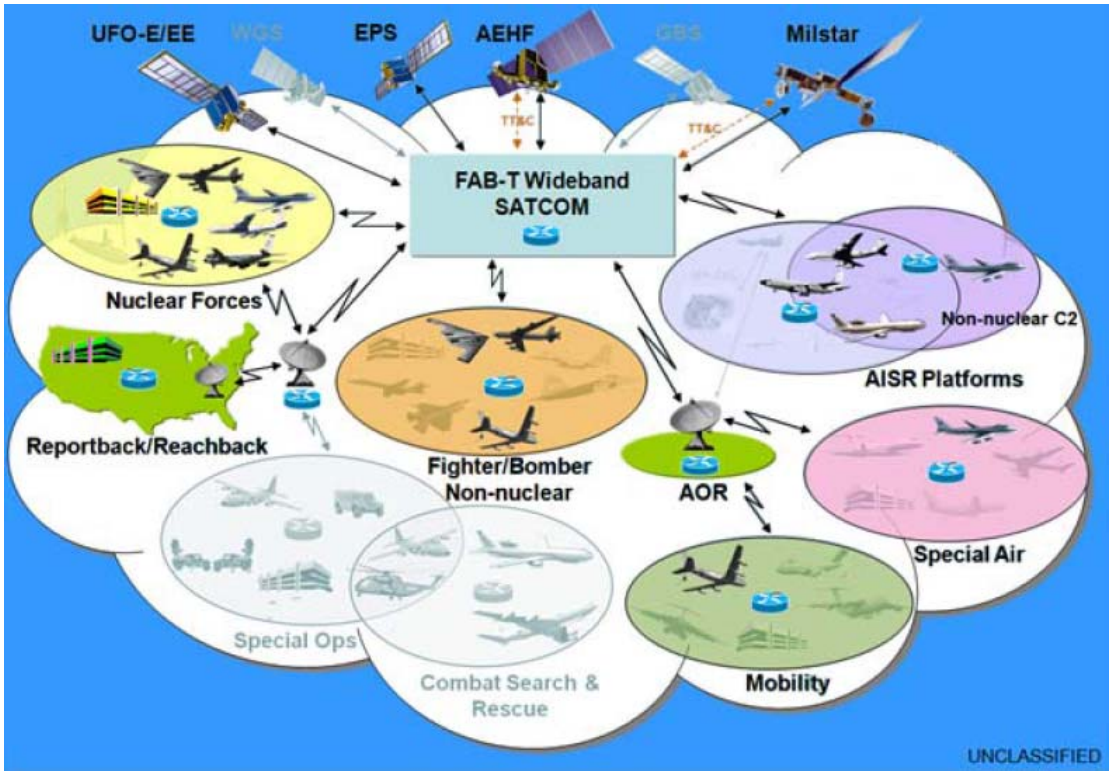

Figure 39: Overview of FAB-T

Figure 39 provides an overview of FAB-T showing the large number of systems the terminal must support. Frequent changes occurred during the development program that affected the deployment and interfaces to FAB-T but did not significantly change the size of the end product. UFO was replaced by commercial UHF satellites. AEHF waveforms and crypto took a long time to stabilize. Different airborne antennas were needed for the different apertures and sizes depending on which airframe was employed. These changes required significant rework and retesting. With few exceptions, the SEI team was able to reconstruct the timeline of important changes to the program, thus providing verification of the expert judgment solicitation of change drivers (see Table 25).

Table 25: FAB-T Retrospective Timeline

Aug 2002	FAB-T contract award. Planning begins. Award is protested and resolved by Jan 2003. Budget projection FY2002=$10M, FY2003=$52M, FY2004=$96M.
Late 2004	NSA requires very large increase in #keys supported. New crypto card announced. AEHF dealt directly with NSA on crypto concerns but terminals were not in the loop. Systems engineering staff numbers and skills did not meet initial plan. SE for

	communications switches was particularly weak.
Dec 2005	Program Re-Plan. Recognition of consequential risk associated to simultaneous development of AEHF, Milstar changes, NSA crypto updates. Modem not meeting weight requirements (simultaneous LDR, XDR requirements). Tiger Team established Dec 2005. Complexity of system level integration for hardware/software from eight vendors. Requirement for both airborne and command post terminals (with different antennas and different user interfaces). Systems engineering experience is also cited as a problem. Senior leadership of the prime contractor is replaced by a new team.
Feb 2005	Budget projection FY2004=$112M, FY2005=$147M, FY2006=$224M, FY2007=$203M. Plan to merge High Data Rate Radio Frequency Airborne with FAB-T in 2006. Plan to merge Multi-band Communications Antenna with FAB-T in 2006.
Mid-2006	Major engineering change and re-plan for ECP-20 $525M including $225M cost overrun. New baseline plan published in early 2007.
Jan 2007	Risk Radar charts from this point forward show a consistent pattern recognizing the concerns of concurrent engineering. The risk mitigation date is moved forward every few months for several years.
Feb 2007	Conduct CDR for Low Data Rate System. Budget projection FY2006=$230M, FY2007=$249M, FY2008=$365M, FY2009=$352M. By 2007 the projected cost is nearly 4x the original contract of 2002. A Technology Readiness Assessment conducted at CDR concluded that only the radome was not yet TRL 7. This conclusion in spite of Lincoln Labs assessment of XDR Modem software as difficult and not ready for field use. CDR also claims that contractor has resolved its systems engineering staffing problems.
Dec 2007	Acquisition Performance Baseline (APB) published. Program review documents continue to show cost overrun and schedule slip because of technical problems. Earned value metrics show cost overrun but not schedule slip.
Mid-2008	ECP-24 adjusts for changing specifications in power, reliability, et al. Result of an oversight in earlier design reviews. Block 8 re-plan is in ECP-24

Late 2008	ECP-35 makes adjustment for communications support for multiple levels of security. Acquisition support strategy published Dec 2008 cites interface to AEHF as a significant risk but only rates it as a moderate (yellow) risk. At this point AEHF is already late.
May 2009	Budget projection FY2008=$326M, FY2009=$305M, FY2010=$210M.
Oct 2009	MITRE ITA counts 17 ECPs/CCPs totaling $739M from 2004-2009. ITA specifically cites the schedule delay caused by concurrent block development because of competing access to test facilities and skilled employees. ITA also cites the Lincoln Labs finding about XDR software.
Dec 2009	Air Force Operational Test and Evaluation Center (AFOTEC) prepared an operational assessment report indicating some operational performance difficulties.

With the participation of two in-house experts who worked with the FAB-T program, we established a provisional set of approximately 57 program change drivers specific to FAB-T. We also elicited their judgment on the likelihood of change for each of the program change drivers and their potential cascading effect on the other drivers. These judgments formed the basis for implementing dependency structure matrix (DSM) techniques to reduce the complexity and capture the cascading effects of the interdependencies among the program change drivers.

DSM reduced the number of program change drivers to those that the experts considered to have moderate or high likelihood of change during program execution. We realize that asking the experts to mentally reconstruct what potential changes might have been considered at the early stages of the program brings the potential for problems of bias resulting from the experts' later experience. But if implemented at pre-Milestone A as envisioned, these judgments represent the reality of the early lifecycle estimation process. DSM techniques yielded 30 program change drivers that formed the acyclic graph required for the construction of the BBN.

In assigning the required conditional probabilities for the BBN to each change driver, we utilized both the experts' elicited judgment of probability and the ranges of variance produced from the expert calibration experiments performed earlier. These variances provided an innovative and novel means of including the uncertainty in the probabilities elicited from the experts. For purposes of demonstration, we relied on the results of those experiments, but in a "live action" MDAP we would use the actual program experts' calibration results, which would be obtained through a calibration test. The technical workshop with the MDAP experts would then serve to both elicit their required judgments as described earlier and their participation in a series of calibration training exercises. The exercises sharpen expert abilities to exert less overconfident and less overoptimistic judgment while also producing the required data for use in capturing uncertainty within the BBN.

Another challenge to overcome was the complexity of filling in all of the required conditional probabilities in the BBN. To meet this challenge the SEI developed an algorithmic method to produce a consistent set of conditional probabilities that also included the uncertainties associated with the expert judgment as described in the preceding paragraph. This made the elicitation of

probabilities from the experts tractable while at the same time incorporating the observed degree of confidence in their judgments.

The resulting retrospective BBN enabled the output of probability distributions used as inputs to the cost estimation tool. We constructed linkages to the SEER-SEM cost estimation tool used by the FAB-T program for the 13 system software components. Monte Carlo techniques allowed us to generate confidence intervals for these distributions, which were then used for input to the cost model.

The team's ability to adapt to changing conditions resulting from a delay in access to DoD MDAP execution data, along with issues in identifying a current MDAP to pilot QUELCE, enabled progress in the work and positioned the team to launch a significant FY13 continuation of the research. The team fortuitously landed an industry pilot opportunity of QUELCE, which provided an early exercise of the DSM matrix and the first experimental activity regarding expert judgment calibration. This early industry pilot enabled the team to create and test the QUELCE job aids and process before launching into the DoD space.

9.6 Publications and Presentations

Our research team published the following articles and technical reports as part of the FY12 LENS effort:

- *Quantifying Uncertainty in Early Lifecycle Cost Estimation (QUELCE)*, CMU/SEI-2011-TR-026, Dec 2011.
- "An Innovative Approach to Quantifying Uncertainty in Early Lifecycle Cost Estimation," DACS Journal of Software Technology, Feb 2012, 24-31.
- *Quantifying Uncertainty in Expert Judgment: Initial Results*, CMU/SEI-2013-TR-001, Jan 2013.

The team also delivered the following presentations, tutorials, and podcasts:

- QUELCE presentation, CoCoMo Forum, Nov 3, 2011
- QUELCE half-day tutorial, SEPG North America, Mar 2012
- QUELCE presentation, SSTC, Apr 2012
- QUELCE podcast for iTunes U, June 2012
- QUELCE presentation at Practical Systems and Software Measurement User Group Conference, July 2012

9.7 Acknowledgments

We would like to thank Eduardo Miranda of the Carnegie Mellon School of Computer Science and Ricardo Valerdi of the University of Arizona for their assistance with the expert judgment experiments. Doug Hubbard graciously gave us permission to reuse some of his large catalog of generic test questions. Finally, we would like to thank Michael Cullen, Keith Miller, and their colleagues at Tektronix who provided valuable insights for implementing and refining our methods during an early site visit there.

9.8 References

[Angelis 2000]
Angelis, L. and Stamelos, I. "A Simulation Tool for Efficient Analogy Based Cost Estimation." *Empirical Software Engineering 5*, 1, March 2000.

[Book 2009]
Book, S. A. "Estimating Probable System Cost." *Crosslink 2*, 1,Winter 2000/2001.

[Don 2007]
Don, A. M. and Deborah, A. S. "Error and Bias in Comparative Judgment: On Being Both Better and Worse Than We Think We Are." *Journal of Personality and Social Psychology 92*, 6: 972, 2007.

[Dysert 2006]
Dysert, L. R. "Is 'Estimate Accuracy' an Oxymoron?" *AACE International Transactions*, EST.01 AACE International, 2006.

[GAO 2009]
"GAO Cost Estimating and Assessment Guide: Best Practices for Developing and Managing Capital Program Costs." GAO-09-3SP, March 2, 2009. www.gao.gov/products/GAO-09-3SP.

[GAO 2011]
"Trends in Nunn-McCurdy Breaches and Tools to Manage Weapon Systems Acquisition Costs." GAO-11-499T, Mar 29, 2011. www.gao.gov/products/GAO-11-499T.

[GAO 2012]
"Defense Acquisitions: Assessments of Selected Weapon Programs." GAO-12-400SP, March 2012. www.gao.gov/assets/590/589695.pdf

[Garvey 2011]
Garvey, P. R. and Flynn, B. J. "Enhanced Scenario-Based Method (eSBM) for Cost Risk Analysis." DoD Cost Analysis Symposium, February 2011.

[Gino 2011]
Gino, F., Sharek, Z., et al. "Keeping the Illusion of Control Under Control: Ceilings, Floors, and Imperfect Calibration." *Organizational Behavior and Human Decision Processes 114*, 2: 104, 2011.

[Goldenson 2013]
Goldenson, Dennis R. and Stoddard, Robert W. *Quantifying Uncertainty in Expert Judgment: Initial Results* (CMU/SEI-2013-TR-001). Software Engineering Institute, Carnegie Mellon University, 2013.

[Hamdan 2009]
Hamdan, K.; Bibi, S.; Angelis, L.; Stamelos, I. "A Bayesian Belief Network Cost Estimation Model that Incorporates Cultural and Project leadership Factors." 2009 IEEE Symposium on

Industrial Electronics and Applications (ISIEA 2009), October 4-6, 2009, Kuala Lumpur, Malaysia.

[Hubbard 2010]
Hubbard, D. W. *How to Measure Anything: Finding the Value of Intangibles in Business.* Hoboken, New Jersey: John Wiley & Sons, Inc., second edition, 2010.

[Jørgensen 2005]
Jørgensen, M. "Practical Guidelines for Expert-Judgment-Based Software Effort Estimation." *IEEE Software,* May/June 2005.

[Lindemann 2013]
Lindemann. Design Structure Matrix. Retrieved from http://www.dsmweb.org.

[Lindgren 2009]
Lindgren, M. and Bandhold, H. *Scenario Planning—Revised and Updated Edition: The Link Between Future and Strategy*. Palgrave Macmillan, 2009.

[RAND 2007]
"Evaluating Uncertainty in Cost Estimates." Research Brief, RAND Project Air Force, 2007. http://www.policyarchive.org/handle/10207/bitstreams/4595.pdf.

[Roper 2010]
Roper, M. "Pre-Milestone A Cost Analysis: Progress, Challenges, and Change." Defense Acquisition University, January 2010. www.dau.mil.

[SAR 2011]
Selected Acquisition Report (SAR): FAB-T, Department of Defense, Defense Acquisition Management Information Retrieval (DAMIR), RCS: DD-A&T (Q&A) 823-199. December 31, 2011.
www.dod.mil/pubs/foi/logistics_material_readiness/acq_bud_fin/SARs/DEC%202011%20SAR/FAB-T%20-%20SAR%20-%2031%20DEC%202011.pdf

[WSARA 2009]
Weapons Systems Acquisition Reform Act 2009 (DTM-09-027). acc.dau.mil/wsara.

10 Real-Time Scheduling on Heterogeneous Multicores

Bjorn Andersson

Many things, both common and not so common, contain embedded computers. For example, a normal car today has more than 100 computers inside. Pacemakers and hearing aid contain computers. All missiles have embedded computers. In addition, robots, both industrial and domestic, both humanoid and non-humanoid, have embedded computers. Many things have embedded computers and it is hard to come up with anything that will not have an embedded computer.

Typically, these things with embedded computers interact with the physical environment. They do so by taking readings from sensors (for example, cameras, radars, thermometers, pressure sensors) and then computing commands to actuators (for example motors, hydraulics, electrical relays, transmission on an antenna, opening/closing a valve). Often, the interaction is about controlling the physical environment, and this requires correct timing. Typically, it is required that the time between two sensor readings is not too extensive, because too long a time between sensor readings makes it possible for the physical world to change in a way that the computer cannot detect. And typically, it is required that the time from taking a sensor reading until actuation is not too long; otherwise the actuation is based on stale information. Computer systems that require correct timing are referred to as *real-time systems*.

In order to understand the role of timing in real-time systems, consider, for example, that if an airbag in a car is inflated too late, then the airbag is not functioning (it fails to save a life in case of an accident). If a pacemaker computes the right stimuli but delivers the stimuli at the wrong time, then the pacemaker is not functioning (it fails to keep the human alive). If the computer in a feedback control loop in a flight control system of a modern airplane has too large delay from sensing to actuation, then the airplane becomes unstable and may crash. If a countermeasure of a military jet in battle is released too late, then the countermeasure is not functioning. Finally, if an anti-missile, launched to intercept an incoming nuclear ballistic missile, computes the trajectory of a ballistic missile correctly but does so at the incorrect time, then a major city (e.g., New York) may be destroyed. In many applications, incorrect timing causes a mission failure or the failure to complete a task. In some applications, incorrect timing is a safety risk.

10.1 Purpose

Among engineers developing real-time systems and among researchers aiming to create better methods for developing and analyzing real-time systems, it is common to describe the computer system as a set of tasks where

1. a task is one part of the software (e.g., one task may control temperature, whereas another task may control pressure)
2. a task generates a sequence of jobs, and the task describes all of the following:
    a. the times at which jobs can arrive

b. how much execution a job requires in order to finish

c. how late a job is allowed to finish execution (relative to the arrival time of the job)

The times at which a job can arrive is determined either by the physical environment (e.g., a car crashes so an accelerometer indicates a very high deceleration, and therefore a job for computing how to inflate airbags arrives) or is determined by the software itself (a task "sleeps" and computes a time in the future when it "wakes up," and when it "wakes up" a job arrives). The latter is common in feedback control systems (e.g., controlling the altitude of an airplane). How much execution a job must perform before it finishes is determined by how fast the processor is and the structure of the software. How late a job is allowed to finish execution (relative to the arrival time of the job) is a number that an engineer chooses; it is called the *deadline* of the job. The engineer must choose the deadline by considering what is needed in order for the interaction with the physical world to be correct. The interaction between the computer system and the physical world is often very complex. Under certain circumstances, a very large deadline can be allowed (for example, for an autopilot software in an airplane, a deadline can be allowed to be large for a job if the airplane is flying at a high altitude). However, under other circumstances, the deadline of a job must be small (for example, for an autopilot software in an airplane, a deadline must be small if the airplane is landing). Software engineers typically consider the worst-case circumstance and derive a deadline based on that. Hence, if all jobs finish before their deadlines then we say that all jobs meet their deadlines, and then the interaction between the computer system and the physical environment will be correct. The problem that a software engineer then faces is how to ensure that all deadlines are met.

A processor cannot execute two or more jobs simultaneously. This complicates system design because in most systems, the number of tasks exceeds the number of processors and this implies that there exist instants where the number of jobs with unfinished execution exceeds the number of processors and hence not all these jobs can execute simultaneously. At these instants, it is therefore necessary to select which job should execute. Clearly, if during a short time interval (e.g., a duration of a few milliseconds), a certain job with a tight deadline (e.g., a few milliseconds) is never selected for execution during this time interval, then it can happen that this job misses its deadline. A missed deadline can, as mentioned above, have severe consequence. Hence, it is important to (i) create algorithms for carefully selecting which job should execute at a given instant, and (ii) create algorithms for proving mathematically that all jobs meet deadlines, assuming that a given scheduling algorithm is used and assuming a model of the times at which jobs can arrive. The former is referred to as *real-time scheduling*; the latter is referred to as *schedulability analysis*. Note that if the times when jobs arrive are known in advance then one can generate a static schedule (similar to a bus schedule) before the computer system is used. Then it becomes trivial to perform schedulability analysis. In many systems, however, one does not know in advance when a job will arrive (for example, typically, one does not know when a car will crash and hence cannot schedule the job for inflating the airbag in a car in advance). For such systems, real-time scheduling must be performed when the software runs and schedulability analysis becomes non-trivial.

The scientific community has created sound mathematical frameworks for real-time scheduling and schedulability analysis, for tasks that generate jobs where the arrival times of jobs are not known in advance but a model providing bounds on the arrival times is known. The most well-known such framework is called the *Rate Monotonic Analysis* (RMA) [Klein1993]. In this

framework, an engineer assigns a priority to each task; this priority is a number and it does not change. For example, task1 may be assigned the priority 7 and task2 may be assigned the priority 5. The priority assigned to a task does not necessarily reflect criticality; instead, the priority of a task is used to instruct the real-time scheduling algorithm at runtime on how to take decisions. Specifically, a job is assigned the same priority as the task that generates the job. When the system runs, the job selected for execution is the one with the highest priority among the jobs that have arrived and have unfinished execution. The most well-known algorithm for assigning priorities is called *Rate-Monotonic* (RM) [Liu 1973]; it assigns high priority to a task if the task generates jobs often (with high rate).

To understand RM, consider two tasks: task1 and task2,

where task1 generates a sequence of jobs with two consecutive jobs of task1 having arrival times being separated by at least 10 ms and

where task2 generates a sequence of jobs with two consecutive jobs of task2 having arrival times being separated by at least 100 ms.

Since task1 can have jobs arriving at a higher rate than task2, RM will assign a higher priority to task1. For example, RM could assign priority 7 to task1 and priority 5 to task2. The RMA framework provides schedulability analyses for systems that use RM; this is desirable because it makes it possible for a designer to prove that all deadlines are met before a system is put into operation. It therefore becomes possible to prove that the interaction between the computer and the physical world is correct before the computer system is put into operation. Consequently, the RMA framework has been an extraordinary success, and

- is taught in undergraduate courses in real-time systems at major universities throughout the world
- has been adopted in high-profile critical systems (such as the Apollo 11 mission landing on the moon in 1969 [Liu 1969] and the Mars Path Finder in 1997 [Jones 1997])
- is supported by many design tools and is a standard engineering practice today (in the industries of avionics, automotive and medical devices) [Klein1993].

Unfortunately, the success of RMA was limited to computer systems with a single processor because most of the results of RMA applied only to a single processor. This limitation was not severe during the time that RMA was invented, because at that time, most computers had just a single processor. But over the past ten years, this has changed, and today all major chip makers sell chips comprising many processors (called *multicore processors*), and most computers today use such chips.

A recent trend among chip makers is that they have started to sell *heterogeneous multicore processors*, that is, multicore processors where processors are of different types. For example, many chips have normal processors and a graphics processor. Examples of this include Intel Ivybridge and AMD Fusion, which are used in many personal computers today but also in embedded computer systems. Some chips have other types of processors, for example, a movie-decompression accelerator and/or a processor for filtering of an image. In a computer with a heterogeneous multicore chip, some tasks can only be assigned to a certain type of processor and

some other tasks can be assigned to any of the types of processors but the execution speed of the task is dependent on the type of processor to which a task is assigned.

Clearly, with one assignment of tasks to processors, it may be possible to guarantee that all timing requirements are fulfilled whereas with another assignment of tasks to processors, timing requirements be violated. Therefore, the goal of this project has been to develop algorithms for assigning real-time tasks on a heterogeneous multiprocessor platform. Ideally, such an algorithm should

1. always run fast, and
2. be optimal. That is, it should find an assignment of tasks to processors so that all timing requirements are fulfilled whenever such a task assignment exists.

10.2 Background

The problem of assigning tasks to processors can be described as follows: A task is described with its processor utilization, but it has different processor utilizations for different processors. For example, if a given task is assigned to a graphics processor, then the task will have a utilization of 10 percent. If the task is assigned to a normal processor (a general purpose CPU), the task will have a utilization of 70 percent. We are interested in assigning each task to exactly one processor such that for each processor, the sum of utilization of all tasks assigned to this processor will not exceed 100 percent. If we find such an assignment then we know that if tasks have deadlines described with the model implicit-deadline sporadic tasks—and if the scheduling algorithm Earliest-Deadline-First (EDF) is used—then all deadlines will be met at runtime (with a minor modification, we can use Rate-Monotonic scheduling as well) [Liu 1973].

The task assignment problem belongs to a class of problems that are computationally intractable, meaning that it is highly unlikely to be possible to design an algorithm that

1. always runs fast and
2. is optimal. That is, it finds an assignment of tasks to processors so that all timing requirements are fulfilled whenever such a task assignment exists.

So either we create an algorithm that achieves 1 but not 2; or we can create an algorithm that does not achieve 1 but achieves 2. If we are interested in achieving 2 but not necessarily 1, then task assignment can be modeled as integer-linear programming (ILP) as follows:

Minimize z
subject to the constraints that
for each processor p: $x_{1,p}*u_{1,p} + x_{2,p}*u_{2,p} + \ldots + x_{n,p}*u_{n,p} \leq z$
and
for each task i: $x_{i,1} + x_{i,2} + \ldots + x_{i,m} = 1$
and
for each pair (i,p) of task i and processor p: $x_{i,p}$ is either 0 or 1

In the optimization problem above, n is the number of tasks, m is the number of processors, and $u_{i,p}$ is the utilization of task i if it would be assigned to processor p. $x_{i,p}$ is a decision variable with the interpretation that it is one if task i is assigned to processor p; otherwise it is zero. Unfortunately, solving this integer linear program takes a long time. To design an algorithm that

always runs reasonably fast, there are several algorithms, as described in a research paper by Sanjoy K. Baruah [Baruah 2004], that transform the ILP into the following linear program (LP):

Minimize z
subject to the constraints that
for each processor p: $x_{1,p}*u_{1,p} + x_{2,p}*u_{2,p} + \ldots + x_{n,p}*u_{n,p} \leq z$
and
for each task i: $x_{i,1} + x_{i,2} + \ldots + x_{i,m} = 1$
and
for each pair (i,p) of task i and processor p: $x_{i,p} \geq 0$

Solving this LP and performing certain tricks gives us a solution to the ILP; this solution is feasible but may not be optimal. Although the time required to solve this LP is less than the time required for solving the ILP above, solving this LP can still take a long time. To design algorithms that run faster, we would like to perform task assignment in a way that does not require solving an LP.

10.3 Approach

Our approach is to consider the special case of a heterogeneous multiprocessor with only two types of processors. We will initially assume that tasks are allowed to migrate between processors of the same type (but not between processors of different types). We will use a so-called swapping argument to reason about a structure of the problem and, using this structure, devise an algorithm for task assignment that is very fast. We will then show that this task assignment can be changed so that each task is not only assigned to a processor type but is also assigned to a processor. We will now present a high-level view of this idea. Consider the following LP:

Minimize z
subject to the constraints that
$x_{1,1}*u_{1,1} + x_{2,1}*u_{2,1} + \ldots + x_{n,1}*u_{n,1} \leq z*m_1$
and
$x_{1,2}*u_{1,2} + x_{2,2}*u_{2,2} + \ldots + x_{n,2}*u_{n,2} \leq z*m_2$
and
for each task i: $x_{i,1} + x_{i,2} = 1$
and
for each pair (i,p) of task i: $x_{i,1} \geq 0$
and
for each pair (i,p) of task i: $x_{i,2} \geq 0$

where m_1 is the number of processors of type-1 and m_2 is the number of processors of type-2.

In this LP, $u_{i,1}$ indicates the utilization of task τ_i if it is assigned to a processor of type-1 and $u_{i,2}$ indicates the utilization of task τ_i if it is assigned to a processor of type-2. The variables $x_{i,1}$ and $x_{i,2}$ indicate the type of processor task to which τ_i is assigned. Specifically, $x_{i,1}=1$ indicates task τ_i is entirely assigned to type-1 processors, but it has not been assigned to any specific type-1 processor; $x_{i,2}=1$ indicates task τ_i is entirely assigned to type-2 processors but it has not been assigned to any specific type-2 processor; if $x_{i,1}<1$ and $x_{i,2}<1$ then it indicates that task τ_i has been fractionally assigned to type-1 and fractionally-assigned to type-2.

In a solution to this optimization problem, $z \leq 1$ indicates that all tasks assigned to type-1 processors utilize at most 100% of the processing capacity of type-1 processors and that all tasks assigned to type-2 processors utilize at most 100% of the processing capacity of type-2 processors.

We are now facing two challenges (1) how to solve the LP efficiently, and (2) how to change a solution that involves fractionally assigned tasks to a solution that involves no fractionally assigned tasks. If we can succeed in doing so, then we have a solution to the LP above, so that for each task τ_i it holds that either $x_{i,1}=1$ or $x_{i,2}=1$ and we can obtain this solution efficiently. Then it is straightforward to create an assignment of tasks to processor-types based on this solution: for each task with $x_{i,1}=1$, assign it to type-1 processor; for each task with $x_{i,2}=1$, assign it to type-2 processor. We then face a third challenge: (3) given that a set of tasks have been assigned to the set of type-1 processors, assign each task to a processor. Ditto for type-2. In this project, we have solved each of these three challenges and received the outstanding paper award at a top-conference for this work.

10.3.1 Solving the Challenge How to Solve the LP Efficiently

We will discuss a property of the linear program and then use this property to design a solution. Suppose that we had a solution to the linear program (LP) above and suppose that there are two or more tasks that are fractionally type-assigned. Let task *i* and task *j* denote these tasks. We can choose the indices of these two tasks so that it holds that

$$u_{i,2}/\, u_{i,1} \;\geq\; u_{j,2}/\, u_{j,1}$$

that is, task τ_i has a relatively higher benefit of being assigned to a processor of type-1 than being assigned to a processor of type-2. This solution can be modified as follows:

$$x_{i,1} \leftarrow x_{i,1} + \delta_1$$

$$x_{i,2} \leftarrow x_{i,2} - \delta_1$$

$$x_{j,1} \leftarrow x_{j,1} - \delta_2$$

$$x_{j,2} \leftarrow x_{j,2} + \delta_2$$

This modification of the solution can be interpreted as moving some execution of τ_i from type-2 processors to type-1 processors and moving some execution of τ_2 from type-1 processors to type-2 processors. By choosing the values of δ_1 and δ_2 so that they are positive and the ratio between them is selected properly, one obtains a solution with a value of *z* that is lower. We can repeat this modification until at least one of the tasks is no longer fractionally assigned. Repeating this argument for each pair of tasks gives us a solution where there is at most one task that is fractionally assigned.

With this observation, let us now discuss how to solve the LP efficiently. Let us sort tasks such that

$$u_{1,2}/\, u_{1,1} \;\geq\; u_{2,2}/\, u_{2,1} \geq u_{3,2}/\, u_{3,1} \geq \;\ldots \geq u_{n-1,2}/\, u_{n-1,1} \geq u_{n,2}/\, u_{n,1}$$

We can start to consider tasks in the left-to-right order and assign each task to type-1 as long as the utilization of type-1 is at most 100% of the entire capacity of type-1 processors. Then consider

tasks in the right-to-left order and assign each task to type-2 as long as the utilization of type-2 is at most 100% of the entire capacity of type-2 processors. Assign the remaining tasks fractionally between type-1 and type-2.

10.3.2 How to Change a Solution Where There Are Fractionally Assigned Tasks to a Solution Where There Are No Fractionally Assigned Tasks

One can show that in the LP above, if there is an optimal solution then there is also an optimal solution such that there is at most one task being fractionally type-assigned. Let f denote the index of this task. Let $x_{f,1}$ and $x_{f,2}$ indicate the fractional assignment of this task. Then change the solution as follows:

if $x_{f,1}>1/2$ **then**

$\delta \leftarrow x_{f,2}$

$x_{f,1} \leftarrow x_{f,1} + \delta$

$x_{f,2} \leftarrow x_{f,2} - \delta$

else

$\delta \leftarrow x_{f,1}$

$x_{f,1} \leftarrow x_{f,1} - \delta$

$x_{f,2} \leftarrow x_{f,2} + \delta$

end if

In this way, a solution is obtained where the task with index f is assigned non-fractionally but z may be higher than what it would be for an optimal solution.

10.3.3 Given that a Set of Tasks Have Been Assigned to Types of Processors, Assign Each Task to a Processor

Consider the tasks with $x_{i,1}=1$ that we have obtained from the previous section and let us now assign them to processors. Consider the tasks one by one and assign the task to the first processor of type-1 as long as the utilization of this processor remains less than 100%. When assigning a task to the first processor would result in the utilization of this processor becoming greater than 100%, then split this task into two pieces where one piece of the task is exactly so large that assigning this piece to the first processor results in the first processor having utilization 100%; the second piece is assigned to the second processor of type-1. Then continue by considering tasks one by one and assign the tasks to the second processor of type-1 as long as the utilization of the 2nd processor of type-1 does not exceed 100% after the task is assigned. Eventually, it holds that a task is such that assigning this task on the second processor of type-1 would result in the utilization of the 2nd processor of type-1 having utilization greater than 100%. Then split this task into two pieces and assign one piece to the second processor of type-1 and assign the second piece to the third processor of type-2. Continue in the same way. We now end up with the following: Of those tasks with $x_{i,1}=1$, there are at most m_1-1 tasks that are split between processors and for the other tasks, each task is assigned to a single processor. Of those tasks that are split between

processors, we can make them non-split assigned using the same technique as in the previous section. We end up with the following: Of those tasks with $x_{i,1}$=1, each of them is assigned to a processor of type-1.

The same technique is applied for tasks with $x_{i,2}$=1. We also have to treat tasks that have $u_{i,1}$>1 or $u_{i,2}$>1 in a special way—-see publication by Raravi and colleagues for details [Raravi 2012a] (Section 10.7).

With these ideas, we obtain an algorithm with time-complexity O(n*(max(log n,m))). This is a very fast algorithm.

10.4 Collaborations

The team members of this project were Bjorn Andersson (Software Engineering Institute [SEI]), Dionisio de Niz (SEI), Gabriel Moreno (SEI), Jeffrey Hansen (SEI), Charles (Bud) Hammons (SEI), Ragunathan Rajkumar (Carnegie Mellon University), and Shinpei Kato (Nagoya University).

We have also collaborated with Gurulingesh Raravi, Konstantinos Bletsas, and Vincent Nelis (Polytechnic Institute of Porto).

10.5 Evaluation Criteria

An optimal task assignment algorithm OPT is defined to be an algorithm for which, for each task set, it holds that if there exists an assignment such that the task set meets all deadlines, then it holds that OPT assigns tasks to processors so that all tasks meet deadlines.

In this project, we evaluate the performance of our algorithms by proving a resource augmentation bound. The resource augmentation bound of algorithm A (denoted CPT_A) is defined as follows: if for which, for each task set, it holds that if there exists an assignment such that the task set meets all deadlines, then it holds that A assigns tasks to processors so that all tasks meet deadlines, assuming that each processor is CPT_A times faster.

10.6 Results

Based on the approach described above, we have developed an algorithm for assigning tasks to processors [Raravi 2012a]. This algorithm runs very fast (in a few microseconds), is easy to implement and has provably good performance. For our algorithm that assigns tasks to processor-types, the resource augmentation bound is 1.5. For our algorithm that assigns tasks to processors, the resource augmentation bound is 2. In our experiments, we have found that our algorithms offers very good task assignment (near optimal).

We have also developed an algorithm task assignments for the case that tasks also share other resources (e.g., sensors or actuators or data structures) proven performance bounds for these algorithms as well [Raravi 2012b].

Tasks executing on heterogeneous multiprocessors tend to have more complex interactions than tasks executing on a homogeneous multiprocessor and more complex interactions than those we have discussed so far. Typically, in a heterogeneous multiprocessor, a task arrives on one processor and then performs execution (e.g., reading a sensor executing on a general purpose

CPU) and then must execute on another processor (e.g., to perform data processing executing on a graphics processor), then execute on another processor (e.g., send a command to an actuator, executing on a general purpose CPU). Performing schedulability analysis of such tasks, with multiple stages, is challenging. Fortunately, we have developed, in a related project, an idea for analyzing more complex real-time systems using Mixed-Integer Linear Programming (MILP). In the later part of this LENS project, we used this MILP idea to develop an algorithm that can analyze multistage tasks running on a heterogeneous multiprocessor. This approach is very flexible and allows a wide range of real world effects to be modeled. Therefore, it has the potential to allow software engineers great confidence in their timing analysis (because the model that this analysis uses is close to reality) and to be able to exploit the high performance of heterogeneous multicore processors. We made this discovery in the middle of the execution of the project, and so we adapted the plans for the execution of the project. Although this new direction that we set appears very promising, it had a drawback: due to lack of time, we were not able to actually build a tool for performing schedulability analysis with this MILP idea. Further development is needed to create such a tool and further research is needed to make such a tool run fast.

10.7 Publications and Presentations

The following publications have been produced as a result of this project.

[Raravi 2012a]

Raravi, G., Andersson, B., Bletsas, K., and Nelis, V. "Task Assignment Algorithms for Two-type Heterogeneous Multiprocessors." *Proceedings of 24rd Euromicro Conference on Real-Time Systems*. Pisa, Italy, July 2012. Outstanding Paper Award.

[Raravi 2012b]

Raravi, G., Nelis, V., and Andersson, B. "Real-Time Scheduling with Resource Sharing on Uniform Multiprocessors." *Proceedings of 20th International Conference on Real-Time and Network Systems*. Pont à Mousson, France, November, 2012.
A paper has also been submitted to ECRTS'13. This describes the idea of performing schedulability analysis of complex real-time systems by solving MILPs.

10.8 References

[Jones 1997]

Jones, Mike. "What really happened on Mars?" December 7, 1997. http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html

[Klein 2012]

Klein, M. H., Ralya, T., Pollak, B., Obenza, R., and Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993, ISBN: 0-7923-9371-9.

[Liu 1969]

Liu, C. L. "Scheduling algorithms for multiprocessors in a hard real-time environment," *JPL Space Programs Summary 37-60*, (1969):28–31.

[Liu 1973]
Liu, C. L. and Layland, J. "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM 20,* 1 (Jan. 1973):46–61.

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 2013	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Results of SEI Line-Funded Exploratory New Starts Projects	FA8721-05-C-0003

6. AUTHOR(S)
Bjorn Andersson, Stephany Bellomo, Lisa Brownsword, Yuanfang Cai, Sagar Chaki, William Claycomb, Cory Cohen, Julie Cohen, Peter Feiler, Robert Ferguson, Lori Flynn, David Gluch, Dennis Goldenson, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, John Hudak, Carly Huth, Wesley Jin, Rick Kazman, Mary Ann Lapham, Jim McCurley, John McGregor, David McIntire, Robert L. Nord, Ipek Ozkaya, Brittany Phillips, Robert Stoddard, Dave Zubrow

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2013-TR-004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	ESC-TR-2013-004

11. SUPPLEMENTARY NOTES

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

13. ABSTRACT (MAXIMUM 200 WORDS)

The Software Engineering Institute (SEI) annually undertakes several line-funded exploratory new starts (LENS) projects. These projects serve to (1) support feasibility studies investigating whether further work by the SEI would be of potential benefit and (2) support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. Projects are chosen based on their potential to mature and/or transition software engineering practices, develop information that will help in deciding whether further work is worth funding, and set new directions for SEI work. This report describes the LENS projects that were conducted during fiscal year 2012 (October 2011 through September 2012).

14. SUBJECT TERMS	15. NUMBER OF PAGES
Line-funded exploratory new starts, research	146

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