

Echo: Practical Formal Verification by Reverse Synthesis

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

in Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

(Computer Science)

by

Xiang Yin

May

2012

Copyright © 2012

Xiang Yin

All rights reserved

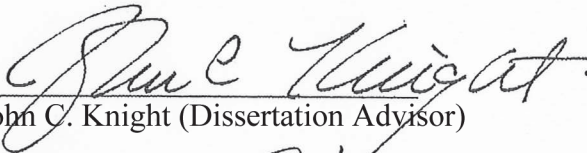
APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

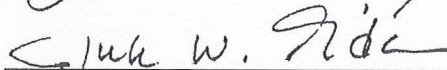


Xiang Yin (Author)

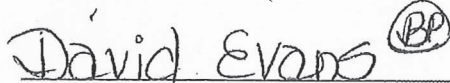
This dissertation has been read and approved by the Examining Committee:



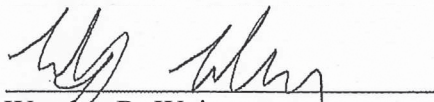
John C. Knight (Dissertation Advisor)



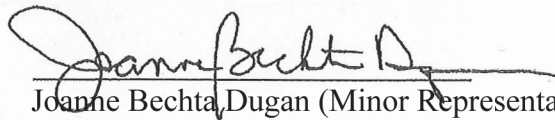
Jack W. Davidson (Committee Chair)



David E. Evans

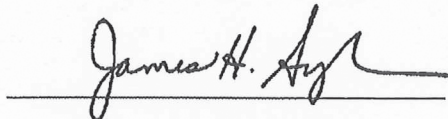


Westley R. Weimer



Joanne Bechta Dugan (Minor Representative)

Accepted for the School of Engineering and Applied Science:



James H. Aylor (Dean, School of Engineering and Applied Science)

May 2012

Abstract

Safe operation is crucial to safety-critical systems, such as fly-by-wire flight-control systems in aircraft. Software in these systems has to be correct, or, more precisely, the risk of being incorrect has to be reduced to an acceptable level. Ensuring the absence of implementation flaws by testing only is inadequate because it is infeasible to conduct the number of test cases required to establish a very high level of confidence. Thus formal specification and formally verified implementation are desirable.

Current approaches to formal verification are powerful but not sufficiently practical. Verification techniques are difficult and time-consuming to apply, tools are of limited capabilities and cannot be integrated, and the process requires high levels of expertise. Current approaches also impose limitations on developers that make it difficult to fit formal verification into the existing development cycle.

To address these problems, I develop a practical and effective approach to formal verification of the functional correctness of computer software. My approach, named Echo, relies upon and incorporates a number of powerful existing notations, tools and techniques, and distributes the verification burden over separate levels. At the core of the approach is a process called reverse synthesis in which a high-level abstract specification is extracted from the source code coupled with a low-level, detailed specification of a software system. Formal verification then involves two proofs each of which is either generated automatically or mechanically checked. These proofs are: (1) a proof that the software source code implements the low-level specification correctly; and (2) a proof that the extracted specification implies the original software system specification. The two proofs can be tackled with separate specialized techniques.

In order to facilitate both proofs, a variety of semantics-preserving transformations are used to refactor the implementation. These refactorings reduce the complexity of verification caused by program refinements and optimizations that occur in practice. They are either effected or checked mechanically. Refactoring transformations are used to simplify both of the proofs, in some cases making proofs feasible that otherwise would not be.

The introduction of a low-level specification as an intermediate point and the application of semantics-preserving refactorings allow the approach to dovetail with standard development processes more easily than existing approaches to formal verification. In effect, verification in Echo proceeds in a direction opposite to that of traditional verification approaches, exploiting automated reasoning and program transformation to dramatically increase the practicality of verification. Relatively few limitations are imposed on developers and many existing software engineering development methods can continue to be used, yet formal verification and all of its benefits can be applied. The proof structure introduced in Echo is designed to scale for large software systems.

In addition, when combined with other types of analysis such as run-time checks, Echo increases the expressive power of formal verification, allowing whole-system assurance arguments to be constructed for richer properties.

Acknowledgments

It is my honor to first thank my advisor, John Knight, for the years of direction and support to my graduate study and research work. John led me into the world of dependable software systems, inspired my on research ideas, and guided me on how to approach research problems and how to write research articles. He has always been there when I needed him. I am grateful to Wes Weimer, for providing constructive insights along the way of my work and broadening my view of the related and state-of-the-art works. I am also grateful to the other members of my dissertation committee—Jack Davison, Dave Evans, and Joanne Bechta Dugan—for the effort they invested in reviewing my writings, for their criticism and suggestions on how I should improve my work.

I have been in a research group that embraces wonderful professional and social relationships among everybody. It was very delightful to collaborate with Elisabeth Nguyen in developing the earlier work that preceded my dissertation research. She was incredibly valuable to me on helping me form the earlier ideas, tutoring me to get familiar with related tools and technologies, and even spending time to correct and improve my English. Tony Aiello, Patrick Graydon, Ben Taitelbaum, Ren Xu have all been very helpful by providing insightful suggestions, and extending my work in one way or another.

A special thanks go to my wife, Lingling Cui, who has been very supportive and very understanding throughout the whole period of time, and has helped me learn to look from a different point of view.

Finally, I thank my parents for educating me and supporting me to pursue the degree. They have always been providing constant and boundless encouragement along the way.

I am grateful to Altran Praxis for their technical support on the usage of SPARK Ada. This work was sponsored, in part, by NASA grants NAG-1-02103 & NAG-1-2290, and NSF grant CCR-0205447.

Table of Contents

1. Introduction.....	1
1.1. Software Verification.....	1
1.2. The Echo Approach to Formal Verification	2
1.3. Thesis Statement	4
1.4. Evaluation of the Echo Approach	5
1.5. Thesis Roadmap.....	6
2. Formal Verification	8
2.1. Safety and Security Critical Systems	8
2.2. Formal Methods	9
2.3. Traditional Formal Verification	10
2.3.1. Correctness Proof.....	10
2.3.2. Refinement.....	11
2.4. Complementary Verification Techniques	12
2.4.1. Static Analysis	13
2.4.2. Theorem Proving	13
2.4.3. Model Checking.....	14
2.4.4. Annotations & Code-level Verification	14
3. The Echo Approach to Formal Verification.....	16
3.1. Motivation.....	16
3.2. Concept and Requirement.....	17
3.3. Overall Process	19
3.4. Current Instantiation	22
3.5. Practicality	23
3.6. Complex Properties and Challenges	24
3.6.1. Real-time Constraints.....	24
3.6.2. Floating-point Arithmetic	25
3.7. Extended Usages.....	25
3.7.1. Synergistic Assurance.....	25
3.7.2. Model-Based Development	26
3.7.3. Property Proof.....	27
4. Proof by Parts.....	28
4.1. Structural Matching Hypothesis	28
4.2. The Proof Structure.....	29
4.3. Restructuring Transformation.....	31
4.4. Matching Metric.....	32
4.5. Approach to Proof.....	33
4.5.1. Type Lemmas.....	34
4.5.2. State Lemmas.....	36
4.5.3. Operation Lemmas.....	37
4.5.4. Implication Theorem.....	38

4.6. Proof Process	38
5. Specification Extraction	40
5.1. Specification Extraction Requirement	40
5.2. Structural and Direct Mapping	41
5.2.1. General extraction rules	42
5.3. Extraction Types	43
5.3.1. Extraction from Annotation	43
5.3.2. Direct Extraction from Code	44
5.3.3. Skeleton Extraction	45
5.4. Other Extraction Techniques	45
5.4.1. Component Reuse	45
5.4.2. Model Synthesis	46
6. Verification Refactoring	47
6.1. Motivation for Verification Refactoring	47
6.1.1. Support for the Implication Proof	48
6.1.2. Support for the Implementation Proof	49
6.2. Definition of Refactoring	50
6.3. The Refactoring Process	53
6.4. Code Metrics	55
7. Synergistic Analysis	57
7.1. Synergistic Assurance	57
7.2. Proofs in Synergistic Assurance	58
7.3. Integration with the Echo approach	61
7.3.1. Verifying the Static Obligations	61
7.3.2. Verifying the Dynamic Checker Functions	61
7.4. Complementary to Echo	62
8. Verification Argument	64
8.1. Soundness Justification	64
8.2. Soundness Involving Complex Properties	65
8.2.1. Real-time Properties	65
8.2.2. Floating-point	66
9. Evaluation Overview	68
9.1. Research Approach	68
9.2. Evaluation overview	69
9.3. Specimen Systems	70
10. Case Study: Verification of AES	73
10.1. Efficacy and Utility Assessment	73
10.2. The Advanced Encryption Standard	74
10.3. AES Verification	75
10.4. Verification Refactoring	76

10.5. Complexity Metrics Analysis	79
10.6. Implementation Proof	83
10.7. Implication Proof	85
10.8. Summary	86
11. Case Study: Defect Detection in AES Verification	88
11.1. Reverse Synthesis and Defect Detection	88
11.2. The Seeding Process	88
11.3. Defect Location.....	89
11.4. Case Study Results.....	90
11.4.1. Defect Detection	91
11.4.2. Defect localization	92
11.5. Summary	92
12. Case Study: Verification of Tokeneer	94
12.1. The Tokeneer system	94
12.2. Echo Proof of Tokeneer	96
12.3. Scalability Evaluation	100
13. Case Study: Verification of MBCS.....	102
13.1. MBCS System Description	102
13.2. MBCS Verification	105
13.2.1. The Artifacts Employed.....	105
13.2.2. Reverse Synthesis	106
13.2.3. Implementation Proof	106
13.2.4. Implication Proof.....	108
13.3. Summary and Evaluation.....	110
14. Application in MBD.....	113
14.1. Model-Based Development	113
14.2. Motivation for Echo Verification in MBD	114
14.3. Application of Echo in MBD.....	115
14.4. Simulink Case Study.....	117
15. Related Work	120
15.1. General Verification Approaches	120
15.2. More Related Approaches.....	121
15.3. Hardware Verification	123
15.4. Reverse Engineering	124
15.5. Automatic Code Generation	125
15.6. Other Related Work	125
16. Conclusion	127
16.1. Conclusion	127
16.2. Contributions	128
16.3. Limitations	130

16.4. Future Work	131
Bibliography	133
Appendix A.Example Specification Extraction and Implication Theorem.....	140
A.1. Original Specification of MBCS.....	141
A.2. Extracted Specification of MBCS.....	147
A.3. Implication Theorem of MBCS	159
Appendix B.Example Verification Refactoring	167
B.1. PVS Theorems for Example Transformations.....	168
B.1.1. Loop Rerolling.....	168
B.1.2. Lift-if.....	169
B.2. Example Transformation of AES.....	171
B.2.1. Original AES Encrypt Function.....	171
B.2.2. AES Encrypt Function after Loop Rerolling	175
B.2.3. Final AES Encrypt Function.....	176
Appendix C.Example Direct Specification Extraction from Code.....	178
C.1. Specification Extraction of Tokeneer	179
Appendix D.Example Obligation Partition in Synergistic Analysis.....	186
D.1. Synergistic Analysis of Tokeneer Requirements.....	187

List of Figures

Figure 1. Traditional formal verification	11
Figure 2. The Echo verification approach	20
Figure 3. Tool configuration for Echo instantiation	22
Figure 4. Proof structure	29
Figure 5. Proof by parts	35
Figure 6. The proof process	38
Figure 7. Sample of SPARK BNF	41
Figure 8. The verification refactoring process	53
Figure 9. Synergistic assurance	58
Figure 10. Proof process of synergistic assurance	59
Figure 11. Overall approach to experimentation	68
Figure 12. Metric analysis with AES verification refactorings	81
Figure 13. The Tokeneer ID Station	95
Figure 14. LifeFlow structure	103
Figure 15. MBD process	113
Figure 16. Application of Echo in MBD	116
Figure 17. Case study Simulink model	118
Figure 18. TIS requirement R_1 and associated obligations	188
Figure 19. TIS requirement R_2 and associated obligations	190

List of Tables

Table 1. Present instantiation of Echo	23
Table 2. AES implementation proof summary	84
Table 3. Annotations in implementation proof	84
Table 4. Defect detection for setup 1	91
Table 5. Defect detection for setup 2	91
Table 6. TIS implementation proof summary	98
Table 7. Magnetic bearing control software requirements	104
Table 8. MBCS implementation proof summary	107

Acronym List

ABD	Assurance Based Development
ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
API	Application Programming Interface
BNF	Backus–Naur Form
CHF	Congestive Heart Failure
DAC	Digital-to-Analog Converter
EAL	Evaluation Assurance Level
FIPS	Federal Information Processing Standards
JML	Java Modeling Language
LVAD	Left Ventricular Assist Device
MBCS	Magnetic Bearing Control Software
MBD	Model-Based Development
NIST	National Institute of Standards and Technology
NSA	National Security Agency
RTW	Real-Time Workshop
TCC	Type Correctness Condition
TIS	Tokeneer ID Station
VC	Verification Condition
WCET	Worst Case Execution Time

Chapter 1. Introduction

The ideal situation with verification of any particular system is the development of a complete formal proof that the implementation implies the specification. This has always been the case with verification since the original concepts were introduced [36]. Recently, it has become the core of a “grand challenge” posed to the computer science community [80]. The Echo approach I introduce in this thesis is a comprehensive approach to formal verification of functional correctness of software with the goal of making such verification readily available, applicable, practical, and useful to the community that needs it.

1.1. Software Verification

In general, verification is the process of showing that one representation of a software artifact is equivalent to another. The most common instance of this is showing that an implementation in a high-level language matches its specification. Clearly, this is important because, in principle, verification provides confidence that mistakes in software development have been avoided or eliminated. There are five types of verification that can be applied to different levels: code inspection, testing, static analysis, dynamic analysis, and formal verification. In many cases, verification is undertaken by testing the developed software artifact. Various testing techniques such as 100% statement coverage and modified condition/decision coverage [33] are used, but while these ameliorate the problem, no algorithm for test selection other than exhaustive testing can guarantee the absence of errors [40]. Exhaustive testing, however, is not feasible for most large systems, hence testing is not adequate for high levels of assurance [15]. Static analysis analyzes a program

before it runs and dynamic analysis checks a program at run-time, both of which have yielded excellent results. However, they are both limited to establishing certain properties of the program rather than equivalence to a specification. Under such circumstances, formal verification, i.e., verification based on mathematics, becomes an attractive alternative [15], especially for safety-critical systems. In some cases, formal verification is required, such as at Evaluation Assurance Level 7 (EAL 7) of the Common Criteria [55].

Nevertheless, although it is desirable, formal verification is not commonly used in practice. Many valid reasons have been put forward to explain this [44]. One important problem is the difficulty of applying current verification techniques and their fragmented nature. Verification approaches are difficult and time consuming to apply; current techniques and tools are limited in their capabilities; high levels of experience are required from the engineers involved; and often rigid, prescribed and unfamiliar processes must be followed. The problems with formal verification have to be solved—many critical systems are being built and assurance of their correct operation is becoming increasingly important.

Various attempts have been made to address the practicality of formal verification. Model checking [37], for example, provides automatic analysis on top of system models. It has been adopted in practice and proven cost-effective, especially at verifying hardware and temporal properties. Such techniques, however, usually do not target full functional correctness for software systems and can not produce a full assurance argument.

1.2. The Echo Approach to Formal Verification

In order to make formal verification more practical, I have developed a comprehensive approach to formal verification, that allows developers the maximum freedom possible in

building a system, yet requires minimal human intervention possible to apply the verification. This contrasts with current formal verification techniques that require developers to change their familiar practices and adopt new processes, or require significant efforts and skills to be accomplished.

The approach, named Echo, relies upon and integrates a number of powerful existing notations, tools and processes. It also introduces a new approach to analysis in which semantic-preserving transformations are applied on the software implementation to reduce the verification complexity, and in which a high-level, abstract specification (referred to as the extracted specification) is synthesized mechanically from a combination of the software source code and a low-level, detailed specification of the software to support the verification. This new approach to analysis is referred to as reverse synthesis, and it fills in a major gap in existing verification techniques. It alleviates the difficulty by separating the verification proofs, and it enables developers to continue to use existing software development methods, i.e., they are not limited solely to tools and processes that support verification, yet formal verification and all its benefits can be applied.

The Echo approach involves two verification proofs: (1) the *implementation proof*, a proof that the source code implements the low-level specification correctly; and (2) the *implication proof*, a proof that the extracted specification implies the original system specification from which the software was built. It introduces a scalable proof structure named proof by parts to facilitate the verification of large systems. The major implication proof is carried out by matching static operational structure of the extracted specification to that of the original specification, and organizing the proof as the conjunction of a series of lemmas about the specification structure. By setting up a different lemma for each distinct ele-

ment and proving each lemma independently, benefit is obtained that the proof scales for large systems.

The basic Echo approach imposes no restrictions on how software is built except that development has to start with a formal system specification, and developers have to create the low-level specification documenting the source code. The current instantiation of Echo uses: (1) PVS [57] to document the system specification and the extracted specification; (2) the SPARK subset of Ada [7] for the source program; and (3) the SPARK Ada annotation language to document the low-level specification. The implementation proof is discharged using the SPARK Ada tools, and the implication is constructed and proved using the PVS theorem prover.

1.3. Thesis Statement

The fundamental theme of this thesis is practicality. More precisely:

THESIS: *The Echo approach will make formal verification of functional correctness a practical yet comprehensive technique for real software systems of at least moderate length.*

By formal verification of functional correctness, I refer to verification of the functionality of the software in the sense of Floyd and Hoare [28]. In addition, this thesis focuses on verification of source programs against formal specifications, admitting but not considering here the additional problems that arise in translation to and extra detail in machine code.

By real software systems, I refer to software systems that are useful and realistic, that can benefit from formal verification, and that are built entirely by others without the application of the Echo verification approach in mind.

By moderate length, I refer to software systems that are at least several thousand source lines long, and expect that the approach scales for systems that are even larger.

1.4. Evaluation of the Echo Approach

To assess and evaluate the thesis statement, I have conducted case studies on three separate specimen systems.

The Advanced Encryption Standard is a symmetric, iterated block cipher. The algorithm is specified in the Federal Information Processing Standards Publication 197 [26]. I translated necessary artifacts involved to the notations adopted by Echo and applied the Echo approach to verify an optimized reference implementation against the official specification that are both publicly available. A further case study was also conducted on the same implementation to assess Echo's defect detection capability by randomly seeding defects into the source code.

The Tokeneer enclave protection system is a hypothetical system that was defined by the National Security Agency (NSA) to act as a challenge problem for security researchers [8]. The system consists of a secure enclave containing a number of workstations having access to files with various restrictions. The challenge to researchers is to construct implementations with appropriate security properties. Altran Praxis (formerly Praxis High Integrity Systems) implemented the Tokeneer ID station software using SPARK Ada and a process designed to comply with the Common Criteria's EAL5 security assurance requirements [17]. I conducted the case study to verify Praxis's implementation using the Echo approach and assessed the scalability of the approach.

The University of Virginia LifeFlow Left Ventricular Assist Device [75] is a prototype artificial heart pump that utilizes active magnetic bearings and a carefully designed

flow path to reduce damage to blood cells. Using the Echo approach, I conducted the last case study to verify an implementation of the pump's magnetic bearing control software that was developed by a colleague.

1.5. Thesis Roadmap

This thesis is organized as follows:

Formal verification

- Chapter 2 describes the need and the background of formal verification, as well as traditional and complementary formal verification techniques.

The Echo approach to formal verification

- Chapter 3 outlines the overall concept and structure of the Echo approach to formal verification.
- Chapter 4 presents the details of the proof by parts structure that is used in the major implication proof in Echo for it to scale.
- Chapter 5 illustrates the way that the extracted specification is synthesized from the implementation and low-level specification in Echo.
- Chapter 6 describes the verification refactoring process and how it facilitates both proofs in Echo.
- Chapter 8 discusses and justifies the soundness of the Echo approach.

Extended usage of the Echo approach

- Chapter 7 explains the concept of synergistic analysis and how Echo is extended and integrated into it to provide the static part of the assurance argument.

- Chapter 14 demonstrates the extended usage of Echo in verifying synthesized code from Model-based development.

Evaluating the Echo approach

- Chapter 9 overviews the evaluation process and three specimen systems upon which evaluation case studies are built.
- Chapter 10 presents the case study of applying the Echo approach to verify an implementation of Advanced Encryption Standard.
- Chapter 11 describes another case study on AES by randomly seeding defects into the source implementation.
- Chapter 12 presents Tokeneer ID Station verification case study.
- Chapter 13 describes a case study of verifying the magnetic bearing control software for a prototype artificial heart pump.

Finally, I discuss related work in Chapter 15 and conclude in Chapter 16.

Chapter 2. Formal Verification

In this chapter, I begin by reviewing safety and security critical systems, the use of software in these systems, and the need of formal verification for them. I then review formal methods, traditional formal verification techniques, some of the current approaches to verification, and their limitations.

2.1. Safety and Security Critical Systems

Safety-critical systems are those systems whose failure will result in catastrophic consequences on the users or the environment [69]. There are many well-known examples in application domains such as medical devices and flight control systems in aircraft. Failures in these systems will lead to loss of human life, significant property damage, and so on. The risk of unsafe operation of these systems must not exceed acceptable levels. Security has also become a crucial issue for systems such as cryptography and authorization in which protection of information and property from unauthorized activities is a major concern. Security vulnerabilities in these systems may lead to loss or untrustworthy use of critical data, computer programs, and other computer system assets. Hence high levels of confidence should be assured for systems in which safety and security are critical concerns.

There has been increasing use of software in safety- and security-critical systems because of the added functionality and flexibility that they can bring. However, because of its complexity, software is the least well understood technology involved. Myers estimates that there are approximately 3.3 software errors per thousand lines of code in large soft-

ware systems [54]. Software is quickly becoming a major cause of critical system failures and security vulnerabilities.

From the software perspective, assuring high levels of confidence requires significant advances in areas such as specification, architecture, verification, process, etc. Verification addresses software correctness issues, and it will be considered in the remainder of this thesis. By correctness here, I mean that the software's implementation adheres to its specification. Software verification is crucial to safety- and security-critical systems and is one of the most significant elements of risk reduction. If it is not completed, it will leave a notable point of weakness in software assurance arguments. In many cases, verification is undertaken by testing the developed software artifact against its specification. Testing, however, is not adequate for high levels of assurance [15]. Formal verification is an attractive alternative under such circumstances. It provides confidence with mathematical rigor that many classes of defects introduced in software development that manifest themselves as security vulnerabilities or safety hazards have been avoided or eliminated. In some cases—such as at Evaluation Assurance Level 7 of the Common Criteria [55]—it is required.

2.2. Formal Methods

Formal methods are mathematically rigorous techniques for system description and development. They grew out of program proving techniques, early examples of which are Edsger Dijkstra's predicate transformers [23] and Harlan Mills' function approach [49]. The mathematical background is a mixture of mathematical logic and set theory.

Formal methods emphasize formal specifications [59], which are mathematically based notations used to precisely describe a system. With formal specifications, formal

methods may be used at various levels and can provide different levels of assurance for the software developed by such methods [12]. It can be merely the formal specification, which provides a mathematical model of a system and allows reasoning about the model. The Z notation [68] has been widely used in this manner, and has been proved to be beneficial. It can also be formal verification, which is applied in the development of a system and involves the use of refinement techniques and proofs of correctness at each stage to insure that the current specification or implementation completely and correctly complies with the previous specification. For example, the B method [2] is specifically designed for this purpose, as opposed to being just a specification language. The process of formal verification can be undertaken manually or with mechanical tool support. Formal methods can also be used to ensure that unsafe or insecure states cannot arise in any system satisfying a formal specification.

Formal methods have been used in a wide variety of application domains, especially the safety-critical and security-critical domains, and have been demonstrated to reduce defects if used appropriately [1]. However, they are still not commonly adopted in industry due to their difficulty, and this is likely to take a while to change.

2.3. Traditional Formal Verification

In a general sense, traditional proof-based formal verification takes one of the two approaches shown in Figure 1: correctness proof and refinement.

2.3.1. Correctness Proof

In the correctness proof approach, a formal specification and the implementation developed from it are combined to form a correctness theorem. The basic form of a correctness theorem for a program is a Hoare triple, $\{P\} S \{Q\}$, in which the specification for the soft-

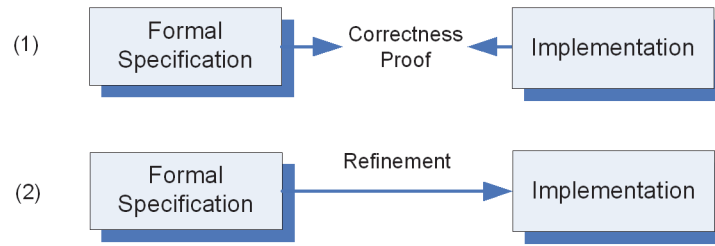


Figure 1 Traditional formal verification

ware, S , is the predicate $P \wedge Q$, with P and Q stand for the precondition and the postcondition of the software respectively. The general approach to establishing this theorem is to determine the verification conditions, derive the weakest precondition (WP) from S and Q , and show that $P \Rightarrow WP$.

This approach, often referred to as Floyd-Hoare verification after the originators of the technique [28, 36], requires generation and proof of many detailed lemmas and theorems. The main difficulty that arises with it is the complexity of the verification conditions and of the final implication. Although machine assistance has been developed, the details can easily overwhelm whatever machine resources are available, even for relatively small programs. The issue is not just the cumulative detail for the program, but also the complexity of individual predicates associated with elaborate or intricate source statements.

2.3.2. Refinement

In response to practical difficulties with correctness proofs, researchers have created approaches in which formal verification is tied closely to software development using a technique called refinement. Software development by refinement involves the transformation of an abstract specification to a concrete implementation by a series of refinement transformations. Each refinement adds details working towards an executable implemen-

tation. For each refinement, a proof that the refinement preserves application properties is established. This process of refinement continues until an implementation is reached that can be executed. The sequence of proofs constitutes the verification argument.

The best known and most comprehensive instantiation of refinement is the B Method [2]. The B Method is an extensive collection of techniques and notations that, taken together, provide a complete infrastructure for specifying, designing and implementing programs, including large ones, using refinement.

Creating a proof along with the program to which it applies is a laudable goal, but it leads to the following three significant limitations:

- Software development decisions often hinge upon a process of exploration, and many alternatives might be considered. Weaving the developments of the software and the proof limits this exploration.
- Many existing software development techniques cannot be used because software development is constrained by the simultaneous proof development.
- If changes to an existing program are required to meet performance goals, those changes invalidate the verification proof chain and require the whole refinement path to be revisited so that the proof can be updated.

These limitations essentially make refinement, including the B Method, either impractical or undesirable for the vast majority of software developments.

2.4. Complementary Verification Techniques

In addition to correctness proof and refinement, various attempts have been made to facilitate mechanical verification and to make formal verification more practical. Such verifi-

cation techniques include static analysis, theorem proving, model checking, and the use of annotations in code.

2.4.1. Static Analysis

Static analysis is the mechanical checking of software performed without actually running it. In most cases the analysis is performed on the source code and in the other cases the object code. A set of automated tools and techniques have been created for this purpose. Typical forms of static analysis include syntax checking and type checking. More advanced static analysis tools such as Splint [25] use lightweight formal methods to check certain properties of a program, sometimes with more user efforts in the coding process. However, although these tools have yielded excellent results, they are limited to establishing properties of a program rather than equivalence to its specification. Thus, for example, static analysis tools can be used to prove that a program will not access an uninitialized variable but not that the program computes the desired value with its initialized variables.

2.4.2. Theorem Proving

Properties, also known as putative theorems, can be stated and proved for a high-level specification. Despite some theoretical limits, practical theorem provers can solve many hard problems using the underlying logics, mechanically or interactively. For instance, PVS [57] is a higher-order logic specification language with an associated theorem prover. Proofs over the specification can be constructed using a collection of primitive inference procedures and more complex deductive strategies that can be applied interactively. Theorem provers have been used for specification analysis in several critical system domains. However, their application on hard problems usually requires a proficient user. Moreover,

although this technique is valuable for establishing confidence in the specification, it does not contribute to the goal of ensuring the accuracy of the implementation.

2.4.3. Model Checking

Model checking is an automated technique that, given a logical property and a model of a system, systematically checks whether this property holds for that model. Thus, when the system itself cannot be verified exhaustively, the user can build a simplified model of the system that preserves its underlying design characteristics but at the same time avoids known sources of complexity. The model can then be verified using model checking techniques. Model checkers such as SPIN [37] and Bogor [62] are becoming increasingly prominent in industrial practice. They provide very effective mechanisms for establishing various properties, usually temporal, of a program. Nevertheless, this technique, again, does not contribute to assurance that the program implementation's total compliance with its specification.

2.4.4. Annotations & Code-level Verification

Some languages are introducing annotations or “formal comments” to capture the code designer's intentions. These annotations are usually added to code in the form of comments, and can express declarative properties such as preconditions, postconditions, and invariants, thus can be used as a low-level specification to specify the desired behaviors or properties of the subprograms in the code. Following the idea of Design by Contract [53], these annotations are trying to provide rigorous formal semantics while still being accessible to ordinary programmer. Various code-level tools in the form of static checker and prover are developed to verify that a program is consistent with the design information included in its annotations. The code-level verification usually follows Floyd-Hoare style

in proof obligation generation and is largely automated, although a certain degree of human intervention is still needed. Typical examples are the SPARK Ada language with its annotations and toolset [7], and the Java language with JML annotations [47] and static checker such as ESC/Java [64] (and more recently ESC/Java2). The use of annotations and code-level tools makes verification at the low level straightforward. However, high-level specification languages are still preferred since they are more expressive, flexible, and amenable to more powerful analysis. The problem of verification of program implementations' compliance with high-level specifications still exists.

While all of these techniques are different from the core of my work, they are subset or complementary to it. As will be described in the following chapters, part of my work is to adopt, adapt, and integrate the different forms of analysis so that each can be used to best advantage in establishing software assurance.

Chapter 3. The Echo Approach to Formal Verification

In this chapter I introduce the concept of the Echo verification approach. I begin with the motivation and requirement for my work. I then go over the basic elements involved in the Echo verification process and the present language and tool instantiation of the process. Some extended usages of the approach are also be briefly discussed at the end of this chapter.

3.1. Motivation

In the development of a more practical approach to formal verification, I seek to address three goals:

- The first goal is to create a controlled process for formal verification that is reasonably practical to be adopted in industrial applications. This goal comes from the observation that the lack of application of formal verification in industry is due to the lack of a well-controlled process, the fragmented nature of current techniques, and the significant time and skill involved in the compliance proof. I aim to produce such a process and necessary tool infrastructure, which can be easily integrated into current software development practice, and with which the proof difficulty is mitigated and the verification is automated to the extent possible.
- The second goal is to allow developers the maximum freedom possible in building a software system. Showing compliance of an implementation with a specification should not necessitate a specific way for constructing the implementation. It does not

mean that formal verification should be open routinely to software that is written in a completely ad-hoc manner, with no attention to structure and with no attention to the need for verification. Rather, it means that development should be restricted as little as possible by the goal of verification. This is not the case currently with approaches such as the B method.

- The third goal is to use existing technology as much as possible to form a comprehensive verification approach. Many powerful notations and tools are available for mechanical analysis of software. Exploiting these notations and tools offers the opportunity to make progress more quickly. Moreover, existing notations and tools have taken a long time to develop and have solved very difficult problems. Thus, they both solve part of the problem and point in a positive technical direction.

3.2. Concept and Requirement

To address the above goals, I developed the Echo approach. At the heart of Echo verification is a process that I refer to as *reverse synthesis* in which a high-level, abstract specification (that I refer to as the extracted specification) is synthesized from a low-level, detailed specification of a system. Verification then involves two proofs: (1) the implementation proof, a proof that the source code implements the low-level specification correctly; and (2) the implication proof, a proof that the extracted specification implies the original system specification from which the software was built. Each of these proofs is either generated automatically or mechanically checked, and each can be tackled with separate specialized techniques and notations.

In principle, verification need not be undertaken with two proofs. The reason for the separation is to allow the mechanics of verification to be tailored to the underlying ver-

ification requirements and to take advantage of existing tools. The introduction of the low-level specification as an intermediate representation helps to bridge the gap between the semantic levels of the different representations, and thereby facilitates both proofs. It also allows the Echo approach to dovetail with standard software development tools and techniques more easily than existing approaches to formal verification.

The low-level specification is a crucial element of Echo verification. One of the requirements for the Echo approach to apply is that the developers create them together with the source code. The level that I define for this is an annotated implementation, i.e., an implementation supplemented with declarative property annotations such as preconditions, postconditions, and invariants. These annotations can be defined and inserted into the source code by the developers or partially generated directly from the code, to describe the desired behaviour of subprograms in the code. One might wonder whether developers would be prepared to annotate software that they create. If they are not, then the use of the Echo approach becomes problematic. Fortunately, the use of annotations in routine development is being pursued by industry. Praxis High Integrity Systems, the developers of SPARK Ada, have created a process called Correctness by Construction [32] that requires fully annotated source code development throughout. Their results with this process when applied to safety-critical applications have shown large reductions in average rates of defects per line of code. Microsoft also adopts the concept of annotations in the development of their latest products. Precondition and postcondition documentation using the SAL notation have been included in both the Vista operating system and the Office 12 tool suite [21]. Vista, for example, includes more than 500,000 function-level specifications

(annotations), and Microsoft researchers report that this technology has revealed more than 100,000 software defects in Vista.

Another requirement for the Echo approach to apply is the existence of a proper formal specification. It is required that (1) the system to be verified has a formal specification, and (2) the semantics contained in the formal specification has been restricted to those that can be implemented. This is important because formal specifications often abstract away detail, such as bounds of arithmetic operations, that in practice forms part of a program's semantics. If those semantics are not present in the specification, an implementation that truly complies with the specification cannot be built. This activity must be either performed or checked by a human. A simple way is to ask domain experts to supplement domain specific restrictions into the specification, such as type constraints, predicates, and axioms. However, it can also be put into a formal framework, which is referred to in the literature as *retrenchment* and has been studied in some depth [6].

3.3. Overall Process

Assume that the original specification from which the software was developed is complete and its semantics have been restricted to those that can be implemented, and assume a reasonable development practice has been followed to create an executable implementation together with proper annotations. Then the Echo verification approach, shown in Figure 2, consists of the following steps:

1. **Implementation Proof:** A mechanical proof (with certain human assistance) of the implementation against the declarative property annotations. It usually follows the Floyd-Hoare style and can be achieved using existing tools such as static code ana-

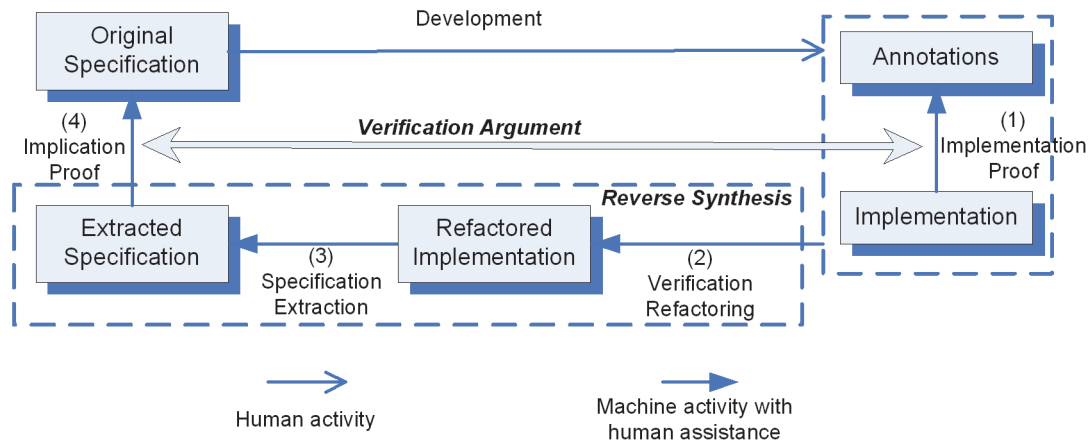


Figure 2. The Echo verification approach

lyzer and prover associated with the language. By exploiting the capability of existing mechanical tools, this step is largely automated. As described earlier, this is an established technique. Notions for annotations have been developed along with proof systems to provide comprehensive support for this type of proof. I just adopt them as part of Echo.

2. **Verification Refactoring:** A series of human-guided, mechanical, semantics-preserving transformations on the implementation to reduce the complexity of its verification. Software implementations are often optimized for various reasons such as efficiency of time and space. This often adds considerable complexity to the verification. Refactoring the program by removing the optimizations while keeping the semantics unchanged can, however, help reduce the complexity of verification. This process involves selecting and applying a number of helpful transformations that are proved to be semantics-preserving, and using the transformed program for subsequent verification. Commonly used transformations can be captured in a library and thus their proofs

- can be reused. I refer to the final transformed program as the refactored implementation.
3. **Specification Extraction:** An automatic or semi-automatic but mechanical checkable extraction of an abstract specification from the refactored implementation along with the declarative property annotations. The extraction process expresses the semantics of the implementation and declarative property annotations in a high-level specification language, which is amenable to more powerful analysis. Certain human guidance may apply to produce a desirable specification. I refer to the produced abstract specification as the extracted specification. This step, together with the previous verification refactoring step, is referred to as reverse synthesis, which is the core component of Echo verification argument.
 4. **Implication Proof:** A human-guided proof that the properties of the extracted specification imply the properties of the original specification. This is a major proof in Echo. For it to be scalable thus amenable for larger systems, I developed proof by parts, in which the proof is carried out by matching static operational structure of the extracted specification to that of the original specification, and is organized as a series of lemmas about the specification structure. This step can be set up and accomplished using a mechanical prover or proof checker associated with the specification language with human intervention. It checks proofs—whether they are constructed manually or automatically—ensuring the soundness of the proof.

Provided (1) that the implementation can be shown to implement the annotations;
 (2) that the verification refactoring and the specification extraction is either automated or

mechanically checked to preserve semantics; and (3) that the implication proof can be constructed and proved, we have a complete argument that the implementation behaves according to its specification.

3.4. Current Instantiation

The Echo approach imposes no restrictions on how software is built except that development has to start with an implementable formal system specification, and developers have to create the low-level specification documenting the source code. There are no limitations on design or implementation techniques nor on notations that can be used. The present instantiation of Echo uses: (1) PVS [57] to document the system specification and the extracted specification; (2) the SPARK subset of Ada [7] for the source program; and (3) the SPARK Ada annotation language to document the low-level specification. In the current instantiation, the proof that the extracted specification implies the system original specification is created using the PVS theorem prover, and the proof that the low-level specification is implemented by the source code is created by the SPARK Ada tools, including the Examiner, the Simplifier, and the Proof Checker [7]. Verification refactoring is also proved to preserve semantics in PVS theorem prover and is mechanically carried out in Stratego/XT toolset [13]. Specification extraction is created and performed by cus-

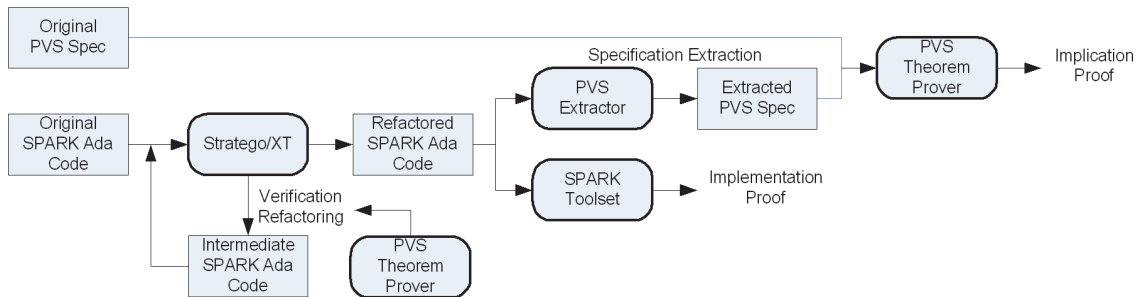


Figure 3. Tool configuration for Echo instantiation

tom tools. The language and tool configuration of present instantiation of Echo is shown in Table 1 and the whole Echo process with the instantiated tool configuration is shown in Figure 3.

Echo verification elements	Present instantiation
Implementation language	SPARK Ada
Low-level specification language	SPARK annotations
High-level specification language	PVS
Implementation proof system	SPARK Examiner, Simplifier, and Proof Checker
Implication proof system	PVS theorem prover
Verification refactoring	Stratego/XT toolset
Refactoring proof system	PVS theorem prover
Specification extraction	Custom developed tools

Table 1: Present instantiation of Echo

3.5. Practicality

The Echo approach makes verification more practical. It does this in part by combining existing powerful techniques, and in part by introducing reverse synthesis and proof by parts.

By exploiting existing notations and tools, the approach offers the opportunity to make progress more quickly. Annotations are tightly coupled with the source code, thus are suitable to prove low-level functional correctness. High-level specification languages are more expressive and are better at reasoning about high-level properties. Reverse synthesis provides a mechanical link between annotations and high-level specification proofs thereby filling in the gaps left by tools already available.

With reverse synthesis, Echo allows an engineer to work with an existing implementation rather than requiring that an implementation be designed to show compliance. This gives developers as much freedom as possible to develop an implementation. Show-

ing compliance of an implementation with a specification in Echo does not necessitate a specific method for constructing the implementation: development decisions is minimally restricted by the goal of verification. With proof by parts, Echo breaks the major proof into a series of lemmas about the specification structure, making the whole proof structure scalable and amenable for large software systems.

3.6. Complex Properties and Challenges

As stated earlier, Echo focuses on verification of functional correctness only. For complex systems, as long as the properties to be verified can be specified functionally, Echo will be able to be applied. For instance, complex user interfaces (such as those that involve dialogues) can be specified functionally in a high-level specification language such as Z [68]. Verification of such specified user interfaces hence can be achieved by functional verification using Echo. However, for complex systems involving non-functional requirements such as real-time constraints and floating-point accuracy, Echo can only tackle with limitations or with support from other approaches. Properties involving concurrency, on the other hand, are not addressed in Echo. Such properties is well handled by model checking as the current state of art.

3.6.1. Real-time Constraints

Echo focuses only on functional correctness, but it can be integrated with dynamic checking in the synergistic assurance technique as will be discussed below. Real-time constraints will thus be checked at run-time in this framework. Presumably, if the processor/compiler selected for the system to be verified supports instruction cycle counts, real-time properties such as WCET may also be stated and verified statically in the Echo approach.

3.6.2. Floating-point Arithmetic

Floating-point arithmetic is known to be inaccurate. However, given a preset boundary and precision, some verification of properties involving floating-point types can still be achieved. The current instantiation of Echo uses the SPARK Ada language. Echo uses SPARK Ada's approximation of floating-point arithmetic and I verify functionalities of floating point types building on the SPARK Ada semantics.

3.7. Extended Usages

Echo is originally developed to be purely an approach for formal functional verification of an implementation against its formal specification. Nevertheless, during the development of Echo, it has shown more capabilities and extended usages.

3.7.1. Synergistic Assurance

My research on Echo and that of a colleague, Ben Taitelbaum, has led to the development of a new technique that integrates static verification using Echo with dynamic checking. This new technique provides a synergistic assurance argument in an approach called synergistic analysis. Dynamically checked properties are introduced as lemmas in the verification argument and the checking functions are statically verified.

The major benefit that synergistic analysis provides is that those properties not verified in Echo, such as some real-time properties and some floating-point arithmetics, can be checked dynamically and included in the final assurance argument. I will present the details of synergistic analysis in Chapter 7.

3.7.2. Model-Based Development

The creation of software by a synthesis tool uses a mechanical process to construct software from a formal specification. The output of synthesis is software in a traditional form, usually a high-level programming language. The most common usage of such code synthesis process is Model-based development (MBD), which is a software engineering methodology that producing source programs by first creating domain-specific models and then synthesizing code automatically from the models. The synthesis process relies upon the correctness of the synthesis tool to achieve the necessary quality in the resulting software. Examples of such synthesis tools in MBD are the SCADE Suite [65] and Simulink [66].

Echo permits formal verification of software built using a synthesis tool. Echo does not depend on actions taken during development, and so it can be applied immediately to synthesized software. The same tools and techniques that are used for software created by hand apply to synthesized software. In application domains for which the consequences of software failure are high, such as safety-critical applications, there is considerable concern about the possibility of the synthesis tool being defective. Traditional methods of avoiding or eliminating faults that are used when software is built manually, including existing methods of formal verification, cannot be applied because the action of the synthesis system is not subject to inspection or modification. Echo presents a comprehensive alternative.

My colleague, Ren Xu's research has extended the usage of Echo to MBD, and showed the feasibility of applying it to Simulink models and synthesized code. I will present some of the details in Chapter 14.

3.7.3. Property Proof

One of the requirements that Echo needs is the existence of a formal specification when a system is developed. However, many systems, especially legacy systems, may not have formal specifications when developed, yet certain level of assurance is still highly desired. This is usually not something about complete functional correctness. People may just want to reason about properties of the system, for example, whether a certain security property holds for the system. Echo is still applicable in such cases. With reverse synthesis, Echo starts from the implementation (augmented with low-level annotations if needed), and can still refactor the code and extract a high-level abstract specification from the code. Then instead of proving the extracted specification implies the original specification, engineers can state the security property in the specification language and try to prove it within the extracted specification. Since high-level specification languages are more abstract and amenable to more powerful analysis, this can usually be achieved with much less effort. Reverse synthesis preserves semantics, so if the security holds for the extracted specification, it holds for the original system.

Such property proof usage is also adopted in both of the above two extended application of Echo: synergistic analysis and model-based development, as will be illustrated in later Chapters.

Chapter 4. Proof by Parts

The implication proof that the extracted specification implies the original specification is the major proof in Echo. For Echo to be practical and amenable for large software systems, the proof structure involved must be scalable. I developed *proof by parts*, in which the implication proof is carried out by matching static operational structure of the extracted specification to that of the original specification, and organizing the proof as the conjunction of a series of lemmas about the specification structure. By setting up a different lemma for each distinct element and proving each lemma independently, it obtains the important benefit that the proof scales easily for large systems.

4.1. Structural Matching Hypothesis

The heart of proof by parts is the structural matching hypothesis. I hypothesize that many systems of interest have the property that the high-level structure of a specification is retained, at least partially, in the implementation. Ideally, a specification should be as free as possible of implementation detail. However, the more precise a specification becomes, the more design information it tends to include, especially structural design information. While an implementation need not mimic the specification structure, in practice an implementation will often be similar in structure to the specification from which it was built because: (a) repeating the structural design effort is a waste of resources; and (b) the implementation is more maintainable if it reflects the structure of the specification.

The hypothesis tends to hold for model-based specifications that specify desired system operations using pre- and post-conditions on a defined state. The operations reflect

what the customer wants and the implementation structure would mostly retain those operations explicitly. As an example, consider the following specification of a simple function that defines a `foo` operation over a `type state`:

```
state: TYPE = [# a: int, b: int #]
foo(st: state) : state
```

An intuitive implementation would almost certainly retain the `state` type as a data structure and the `foo` operation as a subprogram, such as:

```
type state is
  record
    a: Integer;
    b: Integer;
  end record;

procedure foo(st: in out state);
--# derives st from st;
```

The structural matching hypothesis is also implicitly assumed in the well-known Floyd-Hoare approach to verification, which requires a stepwise proof that a function implementation complies with its specification. This implicitly requires a mapping from functions and variables in the specification to those in the implementation. Thus, I have not added assumptions, only evaluated existing ones in more detail.

4.2. The Proof Structure

The proof of interest is the implication proof. Given implementation I and specification S , this proof has to establish the implication $I \Rightarrow S$. Specifically, let OP_I and OP_S denote the

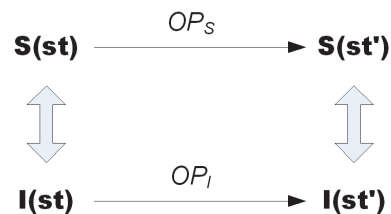


Figure 4. Proof structure

valid state transitions, and $I()$ and $S()$ retrieve the state representation in the implementation and specification respectively, we prove that all valid state transitions in the implementation should also be valid in the specification, and any meaningful state (that can validly transit to other states) defined by the specification should also be meaningful in the implementation:

$$\begin{aligned} \forall st, st': (I(st), I(st')) \in OP_I &\Rightarrow (S(st), S(st')) \in OP_S \\ \forall st: (\forall st': (I(st), I(st')) \notin OP_I) &\Rightarrow (\forall st': (S(st), S(st')) \notin OP_S) \end{aligned}$$

Alternatively, let pre and $post$ denote the sets of pre- and post-conditions, it can be also shown as:

$$I \Rightarrow S \Leftrightarrow pre(S) \subseteq pre(I) \wedge post(I) \subseteq post(S)$$

The implementation is said to imply the specification if and only if it weakens the pre-condition and decreases the non-determinism.

In Echo, the implication proof is not carried out directly between the implementation and the specification. Rather, an extracted specification is first produced by the reverse synthesis process. It is guaranteed that the extracted specification preserves the structure and the semantics of the implementation. The implication proof is then to prove the extracted specification implies the original specification in the same way as the above theory. Since the extracted specification will usually be more abstract than the implementation and will be in the same specification language as the original specification, it is usually easier to establish the implication proof this way.

In order to establish this proof for large software systems, the proof structure involved must be scalable. Proof by parts relies upon the structural matching hypothesis, and matches the static operational structure of the extracted specification created by

reverse synthesis to the original specification. The proof is then organized as a series of lemmas about the specification structure, i.e., proof by parts.

Each lemma is set up for declarative properties over a single distinctive element, e.g. type or operation, and is independently proved. The conjunction of all the lemmas then forms the whole implication theorem that the extracted specification implies the original specification. Since each lemma is over a different elements of the system and is proved independently without reference to the whole system, the proof easily scales for large software systems.

Clearly, the construction of a proof in this way is only possible for a system for which the structural matching hypothesis holds for the entire implementation. Inevitably, this will rarely if ever be the case for real software systems. For systems in which the two structures do not match, I employ a certain type of code transformation, within the verification refactoring technique in Echo, to restructure the implementation to match the specification structure.

4.3. Restructuring Transformation

There are many reasons why the structural matching hypothesis will only apply partially to a particular system. For example, software implementations are often heavily optimized to achieve efficiency in time and/or space, and optimization inevitably affects the structure of the implementation. To facilitate proof by parts under such circumstances, I use verification refactoring to restructure the implementation.

By verification refactoring I mean the transformation of a program in a way that preserves the functional semantics of the program and facilitates verification. Verification refactoring consists of selecting transformations, proving they are semantics preserving,

and applying them to the program before specification extraction. Transforming a program does change the program's execution time. However, by proving that each transformation which is applied preserves the functional behavior (input/output), the functional correctness which is my verification focus is not affected.

Echo's verification refactoring mechanism provides a set of semantics-preserving transformations that can be applied to an implementation to facilitate verification in various ways. A certain set of transformations restructures a program to align the structure of the extracted specification with the structure of the original specification, i.e., to make the matching hypothesis apply to more of the program. This set includes transformation to modify data structures to eliminate complexities related to efficiency, to reverse inlined functions, to split or merge procedures, etc.

Details of verification refactoring, and the restructuring transformations are discussed in Chapter 6.

4.4. Matching Metric

Proof by parts tries to match the static operational structure of the extracted specification to the original specification, organize the proof as a series of lemmas that sets of declarative properties over types and operations combine to imply the specification's properties. The structures of the two specifications do not have to be identical, but the closer the two specifications are in structure, the easier matching the elements between the two becomes and hence the easier the proof becomes.

This raises the question: "How close is close enough?" I have defined a matching metric to summarize the similarity of the structures of the original and the extracted specifications that indicates the feasibility of proof by parts. In Echo, this is done by comparing

the two through visual inspection and evaluation of a match ratio. The match ratio is defined as the percentage of key structural elements—data types, system states, tables, operations—in the original specification that have direct counterparts in the extracted specification. The match ratio does not necessarily imply the final difficulty of the proof, but the match ratio does provide an initial impression of the likelihood of successfully establishing the proof.

Establishing the match ratio is fairly straightforward in many cases. Some of the matching can be determined from the symbols used, because the names used in the original specification are often carried through to the implementation and hence to the extracted specification. Although it is currently evaluated mainly by visually inspection in Echo, significant tool support would be simple to implement.

More details of the matching metric, and other code metrics that evaluate different aspects of the proof difficulties, are discussed in Chapter 6.

4.5. Approach to Proof

The property that needs to be shown in the implication proof is implication not equivalence, hence the name. By showing that the extracted specification implies the original specification, but not the converse, I allow the original specification to be nondeterministic and allow more behaviors in the original specification than the implementation. The basic definition of implication I use for this is that set out by Liskov and Wing known as behavioral subtyping [50]. Behavioral subtyping was studied in the context of languages that permit inheritance in order to define what it meant for a subtype to comply with the type constraints of a supertype. Intuitively, the requirement is similar in verification: we want to ensure that the function implementation complies with the constraints defined in

its specification. While my instantiation is more general, not making assumptions on what is or is not required of a type system, the principles are the same. Then, by implication, I mean that the functions in the extracted specification are subtypes of the matching functions in the original specification.

The goal with proof by parts is to make formal verification relatively routine. Structural matching between the implementation and the specification, achieved with or without verification refactoring, provides the basis of the implication proof.

The way in which the extracted specification is created influences the difficulty of the later proof. In the case where the implementation retains the structural information from the original specification, a simple way to begin proof by parts is to also retain the structure by directly translate elements of the implementation language, such as packages, data types, state/operation representations, pre-conditions, post-conditions, and invariants, into corresponding elements in the specification language. The extracted specification will thus be structurally similar to the original specification. Such a strategy is straightforward, but it does have considerable potential.

The static operational structure of the extracted specification is matched to that of the original specification. For each pair of matching elements, I establish an implication lemma that the element in the extracted specification implies the matching element from the original specification. The final proof is organized as the conjunction of a series of such lemmas. There are three types of implication lemmas as shown in Figure 5.

4.5.1. Type Lemmas

Most of the data types in the specification extracted from the implementation are either equivalent to or refinements of their counterparts in the original specification. Although

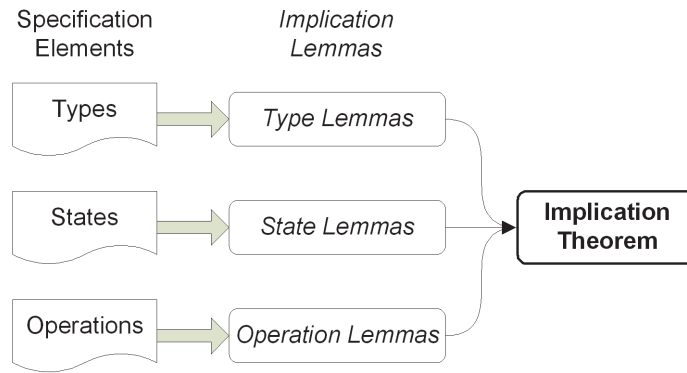


Figure 5. Proof by parts

this is not necessarily the case and proving these relationships is not necessary before tackling the proofs of functions, setting up and proving the relations between the types to the extent possible facilitates the proof of later lemmas.

For each pair of matching types, I define a retrieval function from the extracted type to the original type. When trying to prove the relation between each pair, two possibilities arise:

- **Surjective Retrieval Function.** If the retrieval function can be proved to be surjective, the extracted type is a refinement of the original type, i.e., all properties contained in the types in the original specification are preserved. If the retrieval function can be proved to be a bijection, the two types are equivalent.
- **Non-surjective Retrieval Function.** If the retrieval function is not surjective, then either: (a) there is a defect if the two types are intended to be matched; (b) certain values in the original specification can never arise; or (c) a design decision has been made to further limit the type in the implementation, i.e., to make the post-condition stronger. Upon review, if the user does not confirm that there is a defect or does not further

refine the specification, I postpone the proof by transforming the types into subtype predicates on the same base type (e.g. integer). These extra predicates are added as conjuncts in function pre-conditions or post-conditions depending on where they appear, and they are checked when the later lemmas regarding those functions are established.

4.5.2. State Lemmas

State is the set of system variables used to monitor or control the system. State is defined over types, thus earlier proved type lemmas can be used to facilitate proofs of state lemmas. As with type lemmas, I set up a retrieval function from the extracted state to the original state. For each pair, I prove the following two lemmas:

- **State Match.** As with the type lemmas, I prove that the retrieval function is surjective to show refinement (or equivalence in the bijection case). If it cannot be proved, indicating certain values of the original state cannot be expressed by the extracted state, I again present it for user review. It is either a defect or, by definition, certain values of original state cannot arise.
- **State Initialization.** For states that require initialization, the extracted specification will contain an initialization function. I prove that whenever a state is initialized in the extracted specification, the corresponding retrieved original state also satisfies the initialization constraints in the original specification. Given R as the retrieval function, st as the state, ext and org as extracted and original specification respectively, I prove:

```
FORALL st:
  Init_ext(st) => Init_org(R(st))
```

4.5.3. Operation Lemmas

System operations are usually defined as functions or procedures over the system state. Previously proved state and type lemmas can be used to facilitate the proof. When matching pairs of operations in the extracted specification and the original specification, I set up an implication lemma for each pair according to the concept of behavior subtyping. The operation extracted from the implementation should have weaker pre-condition and stronger post-condition than the operation defined in the original specification. Specifically, I prove:

- **Applicability.** The extracted operation has a weaker pre-condition than the original operation. In other words, the extracted operation is applicable whenever the original operation is. For any state st upon which the operation operates, given R as the retrieval function for st , I prove:

$$\text{FORALL } st: \\ \text{Pre_org}(R(st)) \Rightarrow \text{Pre_ext}(st)$$

- **Correctness.** The extracted operation has a stronger post-condition than the original operation if applicable. In other words, whenever an extracted operation is executed when the matching original operation is also applicable, the output must also be an allowed output of the original operation. Given any $st1$ and $st2$ as input and output for an operation f , R as the retrieval function for state, I prove:

$$\text{FORALL } st1, st2 \mid st2 = f(st1): \\ \text{Post_ext}(st2) \text{ AND } \text{pre_org}(R(st1)) \Rightarrow \text{post_org}(R(st2))$$

By reasoning over predicates such as the pre-conditions and post-conditions in the low-level specification, implementation details are avoided as much as possible when proving these implication lemmas.

4.5.4. Implication Theorem

The conjunction of all the lemmas forms the implication theorem. All the resulting proof obligations need to be discharged automatically or interactively in a mechanical proof system. Since the extracted specification is expected to have a structure similar to the original specification, the proof usually does not require a great deal of human effort. Also, by setting up the lemmas operation by operation rather than property by property and proving each operation independently, the proof structure easily scales. Defects can also be easily located if any of the lemmas fails to be proved, because it must be inside the corresponding structure or element.

4.6. Proof Process

Our process for applying the proof in practice is shown in Figure 6. In most situations, I choose to proceed with verification refactoring first to increase the match-ratio metric until it becomes stabilized through transformations. There are other types of refactoring and corresponding metrics I evaluate to facilitate the proof process, e.g., to reduce the size of proof obligations generated. Details of this verification refactoring and metric analysis process and the benefits it brings to the proof are discussed in Chapter 6.

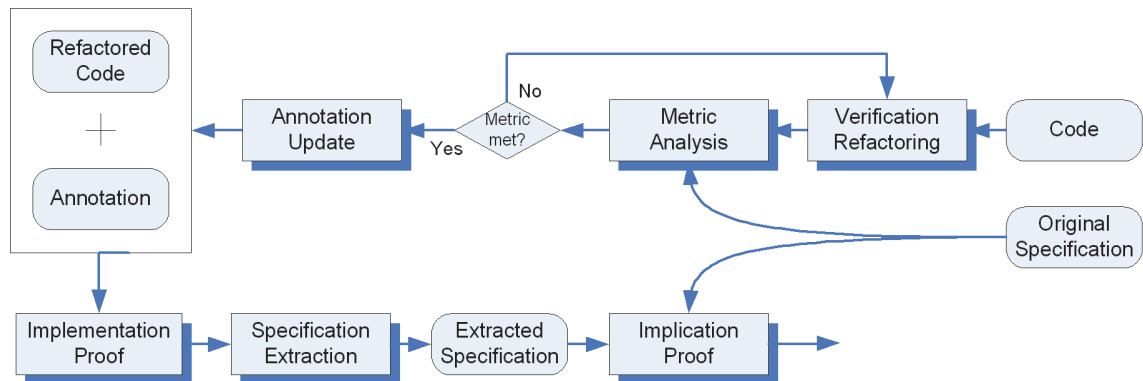


Figure 6. The proof process

After applying verification refactoring, we have a version of the implementation from which a specification can be extracted that shows structural similarity to the original specification. I then: (a) update and complete the low-level specification (documented as annotations) that might have become erroneous during the refactoring process (e.g. by splitting a procedure); and (b) prove that the code conforms to the low-level specification, i.e., create the implementation proof. Recall that the implementation proof is completed mechanically with minimal human intervention. This technology is well established, and I do not discuss it further here.

From the refactored code and the associated low-level specification, I extract a high-level specification, i.e., the extracted specification, using custom tools. I then establish the three types of implication lemma and the implication theorem following the approach discussed earlier. Finally, I prove that the extracted specification implies the original specification by discharging all the implication lemmas in the theorem prover.

Chapter 5. Specification Extraction

The specification extraction fills the gap between the implementation proof at the code level and the implication proof at the specification level. To flesh out the implementation details that are already proved in the implementation proof and that are irrelevant to the verification, the specification extraction step extracts an abstract specification from the annotated implementation to be used in the proof of implication with the original specification. In this chapter I demonstrate how the extracted specification is synthesized from the implementation.

5.1. Specification Extraction Requirement

The specification extraction is the key to the automation of the whole Echo process. The way in which the extracted specification is created influences the difficulty of the implication proof.

The specification extraction seeks a mapping from the implementation to a specification. Since the specification language usually is more expressive than the implementation language, the structure and modularity of the implementation can often be easily maintained through the specification extraction. Given the structural matching between the implementation and the original specification, achieved with or without verification refactoring, the extracted specification can be similar in structure to the original specification so that the implication proof can be facilitated.

5.2. Structural and Direct Mapping

The specification extraction tries to retain the structural information of the implementation in the extracted specification. A simple method for automating the process is to write straightforward rules for translation of elements of the annotated implementation language, such as data types, state/operation representations, preconditions, postconditions, and invariants to corresponding elements in the specification language. Such a strategy, which I call structural and direct mapping, is not always applicable, but it does have considerable potential in the Echo process and thus is intensively used.

In the PVS-SPARK instantiation of the Echo verification process, I implemented an extraction tool that extracts a PVS representation from given SPARK Ada code. I based the extraction on the extended BNF grammar of the SPARK language, developed a parser in Java, and designed a set of extraction rules on top of it to emit corresponding PVS structures when SPARK declarations, definitions, and annotations are parsed. An example of a portion of the BNF is shown in Figure 7.

```

<subprogram_declaration> ->
    <procedure_specification> ;
    <procedure_annotation> |
    <function_specification> ; <function_annotation>

<procedure_specification> ->
    procedure <defining_identifier> <parameter_profile>

<parameter_profile> -> [<formal_part>]

<formal_part> ->
    ( <parameter_specification>
      { ; <parameter_specification> } )

<parameter_specification> ->
    <defining_identifier_list> : <mode> <subtype_mark>

<mode> -> [ in ] | in out | out

```

Figure 7. Sample of SPARK BNF

5.2.1. General extraction rules

Structuring the extraction so that the extracted PVS specification reflects the structure and modularity of the implementation in SPARK Ada is essential for practicality of the implication proof. General extraction rules from SPARK to PVS are categorized as below:

- **Software structure.** I identify packages in SPARK Ada and create a PVS theory for each package in SPARK. Relations among SPARK Ada packages, such as inheritance and compilation order, are extracted into importing relations among PVS theories. Scoping and visibility can also be maintained in the PVS theory structure. The following is a sample of the structure extraction, where `theory_A` and `theory_B` are the corresponding theories of `package_A` and `package_B`, respectively.

SPARK	PVS
<code>with package_A;</code>	<code>theory_B: THEORY</code>
<code>--# inherit package_A;</code>	<code>BEGIN</code>
<code>package package_B</code>	<code>IMPORTING theory_A</code>
<code>...</code>	<code>...</code>
<code>end package_B;</code>	<code>END theory_B</code>

- **Non-function type definitions.** Basic types such as integers, real numbers, enumeration types, record types, and also their subtypes can be directly extracted into carefully selected corresponding type representations in PVS.
- **Variable and constant declarations.** Variable and constant declarations can also be extracted directly into appropriate forms, usually system state representations in PVS. Variables that are local to subprograms are usually not extracted, since they will be masked by the introduction of pre- and post-condition annotations.
- **Function / procedure declarations and definitions.** SPARK Ada functions or procedures are extracted into PVS functions. Not only the direct input / output variables, but also the global variables accessed by that function / procedure will be extracted into

the input / output of the corresponding PVS function. Pre- and post-condition annotations, if available, are extracted into appropriate predicates or type restrictions over the input / output. Detailed implementation of the function / procedure is usually left out if pre- and post-condition annotations exist to provide a level abstraction.

Different extraction types and examples are discussed below.

5.3. Extraction Types

5.3.1. Extraction from Annotation

For functions that are annotated with proved pre- and post-condition annotations, the annotations provide a level of abstraction. This is helpful for programs that contain a lot of computation. For instance, when one specifies a function, the property that one cares about would be correctness of the output. The actual algorithm used is not important. If the function is annotated and the annotation is proved using code-level tools, I extract the specification from the annotations and leave out the unrelated implementation details. The proof of the annotations by the code-level tools can be introduced as a lemma that is proved outside the specification proof system. As an example consider the following code fragment that is written in SPARK Ada:

```

type state is
  record
    a: Integer;
    b: Integer;
  end record;

procedure foo(st: in out state)
--# derives st from st;
--# pre      st.a = 0;
--# post     st = st~[a => 1];
is
begin
-- procedure body
...
end foo;
```

Leaving out the details in the procedure body but introducing an additional lemma, the extracted specification in PVS will be:

```
state: TYPE = [# a: int, b: int #]

foo_pre(st: state): bool = (st`a = 0)
foo_post(st_, st: state): bool = (st = st_ WITH [`a := 1])
foo(st: state): state
foo: LEMMA FORALL (st: state):
    foo_pre(st) => foo_post(st, foo(st))
```

The additional lemma indicates that the postcondition will be met if the precondition is met. The lemma is marked as proved outside and can be used directly in subsequent proofs.

5.3.2. Direct Extraction from Code

The expressive power of the SPARK Ada annotations with which the low-level specification is documented is limited. Certain properties either: (a) cannot be expressed by the annotation language; (b) are expressible but not in a straightforward way; or (c) are expressible but are not helpful in abstracting out implementation details. In such situations I extract the specification directly from the source code. This usually happens in control systems where system state updates, the order of events (function calls) are mostly coded, but not a lot of computation involved. Consider the following example again in SPARK Ada:

```
procedure foo(st: in out state)
is
begin
    foo1(st);
    st.a = 1;
    foo2(st);
end foo;
```

This procedure will be extracted directly to:

```
foo(st: state): state =
    LET st1 = foo1(st) IN
```

```

LET st2 = st1 WITH [ `a := 1 ] IN
LET st3 = foo2(st2) IN
  st3

```

Extracting directly from source code means that a lot of implementation detail need to be managed. As of now in the SPARK-PVS instantiation of Echo, direct extraction from source code is used in limited circumstances, i.e. when there is no iterative control flow. I rely for the most part on annotations to provide an extra level of abstraction to divide the verification burden between the implication proof and the implementation proof.

5.3.3. Skeleton Extraction

A lightweight version of specification extraction is used to facilitate verification refactoring and estimate the likely difficulty of the implication proof. When verification refactoring is applied, I extract a skeleton specification from the transformed code. I refer to this specification as a skeleton because it is obtained using solely the type and function declarations and contains none of the detail from the annotations or the function definitions. The skeleton specification, however, does reflect the structure of the extracted specification. I can then compare the structure of the skeleton extracted specification with that of the original specification and evaluate the match ratio as defined in section 4.4 to determine whether further refactoring is needed. Details on the usage of such match ratio as a metric to guide the verification refactoring process is discuss in Chapter 6.

5.4. Other Extraction Techniques

5.4.1. Component Reuse

Software reuse of both specification and code components is a common and growing practice. If a source-code component from a library is reused in a system to be verified and that

component has a suitable formal specification, then that specification can be included easily in the extracted specification.

5.4.2. Model Synthesis

In some cases, specification extraction may fail for part of a system because the difference in abstraction used there between the high-level specification and the implementation is too large. In such circumstances, I use a process called model synthesis in which the human creates a high-level model of the portion of the implementation causing the difficulty. The model is verified by conventional means and then included in the extracted specification.

All extraction techniques described above combine and contribute to the goal of effective specification extraction.

Chapter 6. Verification Refactoring

Informally, by verification refactoring, I mean the transformation of a program in such a way that the functional semantics of the program (but not necessarily the temporal semantics) are preserved and verification is facilitated. The reverse synthesis process in Echo makes extensive use of verification refactoring, and it is a critical part of the way in which Echo is made more broadly applicable. In this chapter, I discuss the motivation for verification refactoring in terms of the difficulties that it helps to circumvent in the two Echo proofs, the process of the refactoring, semantics-preserving proof and code metrics to facilitate refactoring.

6.1. Motivation for Verification Refactoring

A lot of effort in software development goes to making sure that the software is adequately efficient. The result of this effort is careful treatment of special cases, compact data structures and efficient algorithms, with the inevitable introduction of complexity into the control- and data-flow graphs. Much of the difficulty that arises in formal verification results from the complexity of the source program. One of the reasons for the use of verification refactoring is to reduce this complexity.

A second reason for the use of verification refactoring is to align the structure of the extracted specification with the structure of the system specification. This alignment permits the implication proof to be structured as a series of lemmas and allows an efficient overall proof structure.

The transformations used and the mechanism of their selection is different for the two proofs in Echo, and so each is discussed separately in this section.

6.1.1. Support for the Implication Proof

The implication proof is the proof that the extracted specification implies the original specification from which the program was written. In principle, if the software is indeed a correct implementation of the specification, then it is always possible to construct such a proof. The challenge in Echo, however, is to make the construction of the proof relatively routine.

The feasibility of this proof rests in large measure on the form, content and structure of the extracted specification. Echo uses several techniques to synthesize this specification, but the key in Echo to making the proof practical lies in the structural and direct mapping technique as discussed in previous chapters. This technique rests on the hypothesis that the high-level structural information in a specification is frequently retained in the implementation. There is no experimental evidence to support this hypothesis, but my rationale for believing it is discussed earlier.

Structural and direct mapping provides the basis of the implication proof. The structure of the proof is based on the structure of the specification. The basic approach that I use is to try to match the static function structure of the extracted specification to the original specification, and to organize the proof as a series of lemmas about the specification architecture.

With this approach to proof, the closer the extracted specification's structure comes to that of the original specification, the higher the chance of the proof being completed successfully and in a reasonable time. The transformations that are selected to apply

to the source program are those which will align the extracted specification's structure more closely with that of the original specification.

6.1.2. Support for the Implementation Proof

The implementation proof is the proof that the implementation implies the low-level specification. In the current instantiation of the Echo system, the implementation proof is carried out using the SPARK Ada toolset. The preferred approach to developing SPARK Ada software is to use correctness by construction [32]. In correctness by construction, the SPARK Ada tools are often able to complete proofs with either no or minimal human intervention. The proof process is repeated as the software is constructed thereby ensuring that each refinement leaves the software amenable to proof.

By contrast in Echo, since there are no restrictions on development techniques, the SPARK Ada tools frequently fail when they are applied to software after development is complete. The low-level design of software that is not developed using correctness by construction is unlikely to be in a form suitable for proof. The reasons are many but, as with the implication proof, they typically fall under the heading of complexity introduced to achieve some specific design or performance goals.

The difficulties with the SPARK proof system take one of three forms: (1) the required annotations for function pre- and post-conditions can be many dozens of lines long, lengths that are impractically complex for humans to write; (2) the implementation proof exhausts available resources, usually memory, even though the SPARK tools are quite efficient and typically adequate for proofs that are needed for correctness by construction; and (3) the verification conditions sometimes are sufficiently complex that they cannot be discharged automatically, and human guidance becomes necessary.

Verification refactoring addresses all three of these difficulties without limiting the development process. Because verification refactoring does not need to maintain any aspect of efficiency, any transformation that addresses the three types of difficulty can be used.

6.2. Definition of Refactoring

The Echo verification argument relies upon refactoring, and so it is essential that there be a precise definition of refactoring and a mechanism for ensuring that refactoring complies with this definition in practice. Since Echo is verifying functional behavior, I make the following two simplifying assumptions: (1) refactoring does not preserve the execution time of the program; and (2) refactoring need not preserve the exact sequence of intermediate program states as long as the initial state and final state are unchanged. It is only focused on the initial and final states. The transformation from program P to program P' is semantics preserving if, given the same initial state, both P and P' will generate the same final state when they halt.

In order to prove that any given transformation is semantics-preserving, I define the semantics of the elements of the programming language that are needed in PVS and use the PVS theorem prover to discharge the following theorem:

$$\begin{aligned} \text{init_state}(P) &= \text{init_state}(P') \\ \Rightarrow \text{final_state}(P) &= \text{final_state}(P') \end{aligned}$$

To make the proofs reusable, I have identified some common refactoring transformations, characterized them into templates, and developed a preliminary library for which the necessary properties have been proved. Similar libraries of semantics preserving transformations exist in the domains of compilation, software maintenance, and reverse engineering. I have included some common transformations in the library, but few existing

transformations can be adapted because they have different goals. Compilation transformations, for example, are usually targeted at performance improvement. Transformations in verification refactoring are designed to reduce the complexity and size of verification conditions, and so frequently reduce software's efficiency.

Verification refactoring involves both computation and storage. Programs can be made more amenable to verification by adding redundant computation or storage, by adding intermediate computation or storage, or by restructuring the program. Here I itemize some of the refactorings that I have developed for the prototype library and discuss how each affects the goal of verification:

- **Splitting procedures.** Long procedures usually result in verbose and complex verification conditions. By splitting a procedure into a set of smaller sub-procedures, the verification conditions become vastly simpler and easier to manage.
- **Moving statements into or out of conditionals.** This type of transformations moves statement blocks into or out of conditional statements provided no side effects will result. This transformation can help to simplify execution paths and to reveal certain properties.
- **Adjusting loop forms.** Loops are frequently defined to promote efficiency and ease of use. Adjustment of the loop parameters can facilitate verification by, for example, permitting loop invariants to be inserted more easily thereby allowing verification conditions to be simplified.
- **Reversing inlined functions or cloned code.** Reversing inlined functions involves identifying cloned code fragments and replacing them with function definitions and

calls. Function definitions can be provided by the user or be derived from the code. This transformation aligns the code structures with the specification and removes replicated or similar verification conditions so as to facilitate proof. Furthermore, by reversing the inlining of functions, if an error is identified in a particular inlined function, only that function needs to be re-verified rather than all of the inlined instances.

- **Rerolling loops.** Rerolling loops allows generated verification conditions to be simplified by recovering the loop structure and permitting the introduction of loop invariants.
- **Separating loops.** Loops that combine operations can be split so as to simplify the associated loop invariants.
- **Modifying redundant or intermediate computations or storage.** These transformations modify the program by adding or removing redundant or intermediate storage or computation. This can facilitate proof by: (a) storing extra but useful information; (b) shortening the verification condition by removing redundant or intermediate variables; or (c) merely tidying the code so as to facilitate understanding and annotation of the code.

With these transformation, refactoring is based on the following four stages: (1) identify candidate refactoring transformations—since refactoring might address certain optimizations and refinements introduced during development, this usually needs guidance from developers to identify the occurrences of optimizations, although some can be found mechanically; (2) determine the order to apply the transformations—the order matters if there are dependencies among the transformations; (3) prove the transformations are semantics-preserving if they are not selected from the proved library—all transforma-

tions should be proved to preserve the semantics and should not require the user to discharge complex proof obligations; and (4) apply the transformations to the code—all of the transformations should be applied mechanically to avoid introducing errors.

6.3. The Refactoring Process

The process for applying verification refactoring in practice is shown in Figure 8. A semantics-preserving transformation from the library is selected by the user (or suggested automatically), and the transformer then checks the applicability of the selected transformation mechanically and applies it mechanically if it is applicable. When all of the selected transformations have been applied, a metrics analyzer collects and analyzes the code properties of the transformed code, and presents the complexity metrics to the user. If the metric results are not acceptable, or if they are acceptable but later verification proofs cannot be established, the process goes back to refactoring and more transformation are performed.

The role of the source-code metrics is to give the user insight into the likely success of the two Echo proofs. I hypothesize that the metrics I use are an indication of rela-

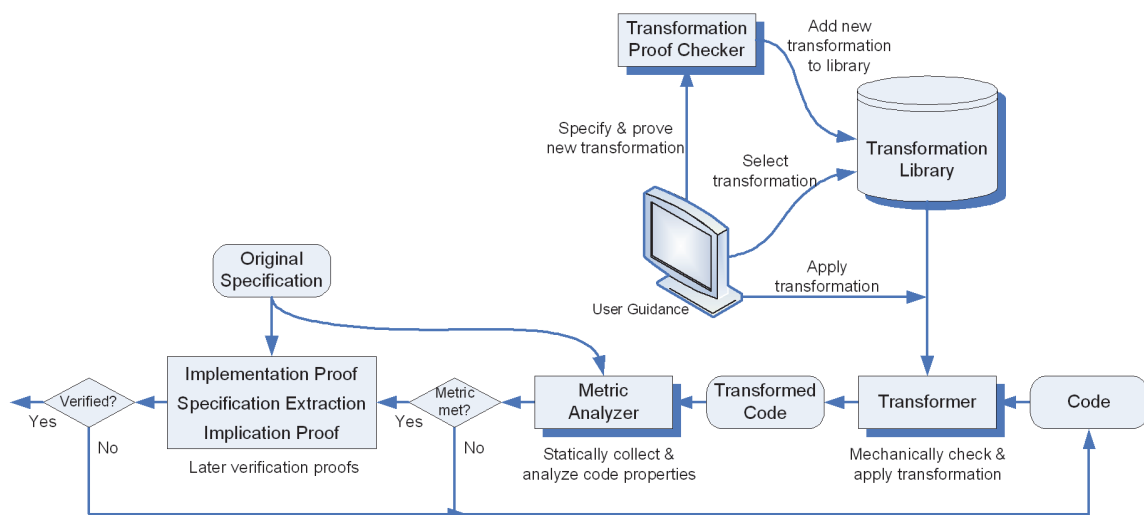


Figure 8. The verification refactoring process

tive complexity and therefore of likely verification difficulty, and I present some support for this hypothesis in a later case study chapter.

Verification refactoring cannot be fully automatic in the general case, because recognizing effective transformations requires human insight except in some special cases. Furthermore, some software, especially domain specific applications might require transformations that do not exist in the library. In such circumstance, the user can specify and prove a new semantics-preserving transformation using the provided proof template and add it to the library.

To facilitate exploration with transformations, if the user has confidence in a new transformation, the semantics-preserving proof can be postponed until the transformation has been shown to be useful or even until the remainder of the verification is complete.

In most cases, the order in which transformations are applied does not matter. Clearly, however, when two transformations are interdependent, they have to be applied in order. A general heuristic is that those transformations that change the program structure and those that can vastly reduce the code size should be applied earlier.

I am not aware of any transformations or circumstances of their application in which a transformation would have to be removed, and I make no explicit provision for removal in the current tools and process. In the event that it becomes necessary, removing a transformation is achieved easily by recording the software's state prior to the application of each transform.

All the user activities, especially the design and selection of transformations, have to be mechanically checked, and these two activities need to be supported by automation to the extent possible. The transformer is implemented using the Stratego/XT toolset [13].

Stratego checks the applicability of the selected transformation, and carries it out mechanically using term rewriting. I use the PVS theorem prover as the transformation proof checker and provide a proof template. When the user specifies a new transformation, an equivalence theorem will be generated automatically, and the user can discharge it interactively in the theorem prover.

6.4. Code Metrics

To my knowledge, there is no verification complexity metric available that could guide the user in selection of transformations, and so I present a hybrid of metrics to the user for review using a commercial metric tool [24], the SPARK Examiner, and my own analyzer. The metrics include:

- **Element metrics.** Lines of code, number of declarations, statements, and subprograms, average size of subprograms, logical SLOC, unit nesting level, and construct nesting level.
- **Complexity metrics.** McCabe cyclomatic complexity, essential complexity, statement complexity, short-circuit complexity, and loop nesting level.
- **Verification condition metrics.** The number and size of verification conditions, maximum length of verification conditions, and the time that the SPARK tools take to analyze the verification conditions.
- **Specification structure metrics.** The user is presented with a summary of the structure of the original and the extracted specifications. A match ratio as defined in section

4.4 can be evaluated. This allows comparison so as to get an idea of the difficulty of implication proof.

All the metrics are subjective, and I do not have specific values that would give confidence in the ability of the PVS theorem prover to complete the implication proof.

I developed the following heuristics to both select transformations and determine the order of application: (1) dependent transformations are applied in order; (2) transformations that impact the major sources of difficulty, such as code and verification condition size, are applied first; (3) transformations that affect global structure are applied earlier and those that affect local structure are applied later; (4) refactoring proceeds until all proofs are possible; and (5) refactoring proceeds until all subprograms can easily annotated, and the program structure “matches” the specification’s.

In practice, if specification extraction or either of the proofs fails to complete, or if either proof is unreasonably difficult, the user returns to refactoring and applies additional transformations.

Chapter 7. Synergistic Analysis

The Echo approach focuses solely on static formal verification of source programs against formal specifications. With the static proof capability of Echo, my colleague, Benjamin Taitelbaum, introduced a comprehensive approach named *synergistic analysis* in which software confidence is obtained by a hybrid use of static and dynamic analysis. In general with this technique, for each desired property of the software, a static and a dynamic method are combined such that the combination ensures that the desired property is achieved. In this chapter I briefly explain the concept of synergistic assurance and how Echo is integrated into it to provide the static part of the assurance argument.

7.1. Synergistic Assurance

The ideal situation for verification of any system is the development of a complete static formal proof that the implementation implies the requirements. However establishing a complete proof for a given program within a single formal framework is difficult. Some functional properties might be established by proof, but temporal properties are often more easily established by model checking and some other properties by testing.

The fundamental goal of synergistic assurance is to facilitate the static proof of a program's properties by making some proof obligations into runtime checks. The obligations that are moved are those which make the proof either infeasible or impractical in some way. By making an obligation into a runtime check, the static proof can treat the obligation as a proved lemma, and the remainder of the proof can then be completed. The obligations treated this way are referred to as dynamic obligations. The synergistic

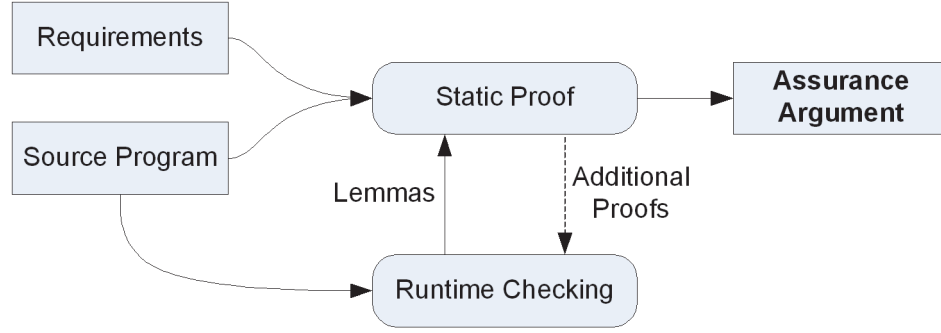


Figure 9. Synergistic assurance

approach to proof is shown in Figure 9. In synergistic assurance, the theorem that needs to be proved is a formal statement of Safe Programming [4] behavior, which includes the notion of a program either working correctly or halting. The assurance theorem for a particular system states that the implementation either implies the system’s requirements or halts, specifically:

$$I \Rightarrow (R_1 \wedge R_2 \wedge R_3 \wedge \dots \wedge R_n) \vee Halt$$

This is carried through to the dynamic obligations, each of which has the following form and becomes a new derived requirement that the program being verified has to meet:

$$Predicate \vee Halt$$

Since the static proof treats the dynamic obligations as lemmas upon which it can rely, additional checks must be performed to ensure correct implementations have to be developed, and there has to be assurance that they will be invoked correctly during execution.

7.2. Proofs in Synergistic Assurance

The assurance theorem is formalized and partitioned into a set of smaller proof obligations, to be either statically proved or dynamically checked. These obligations might be partitioned into even smaller obligations during the proof following the divide and con-

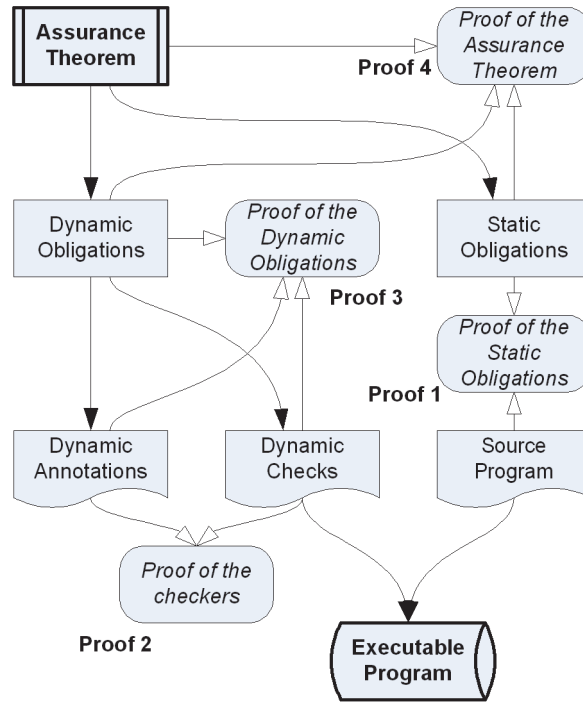


Figure 10. Proof process of synergistic assurance

quer approach. The partition must be justified to produce a sound argument that the assurance theorem holds for the system. To do the justification, all the obligations are introduced as lemmas into a theorem prover, and are then used to statically prove that the conjunction of them implies the assurance theorem using conventional proof of correctness approaches. This can either be done directly after the partition or be postponed until the obligations are statically proved or dynamically checked.

The general approach to construction and proof of the synergistic assurance theorem for a given system is shown in Figure 10, where black arrows denote the development process and white arrows show the proof involved. The whole approach contains a sequence of seven steps:

1. The assurance theorem is examined and obligations required for the proof are partitioned between static obligations and dynamic obligations.

2. The static obligations are statically verified against the implementation as shown by proof 1 using formal verification mechanisms.
3. The dynamic obligations are restated as derived requirements for the program being verified, and are documented in the form of dynamic annotations.
4. Additional software which carries out the dynamic checks is implemented to meet the additional requirements. These dynamic checks will be compiled into the implementation to generate a guarded executable program.
5. The dynamic checker functions are verified against the derived requirements using formal verification mechanisms as shown by proof 2.
6. The program being verified is analyzed statically to ensure that the newly implemented checks will actually be made whenever there is any prospect of the associated requirement being violated. This is shown by proof 3.
7. Finally, to justify the partition of static and dynamic obligations and prove the assurance theorem, all the obligations are used as proved lemmas to construct a proof that the conjunction of these obligations implies the original assurance theorem. It is proved statically just as with a conventional proof of correctness as shown by proof 4.

Given proof 1, we have all static obligations verified. Given proof 2 and proof 3, we have all dynamic obligations ensured. And then given proof 4, we establish the assurance theorem for the system developed.

7.3. Integration with the Echo approach

The Echo approach allows verification proofs to be established statically. Hence it can be perfectly integrated into the synergistic assurance proof process to provide the static verification support. It can be adopted in proof 1: verifying the static obligations and proof 2: verifying the dynamic checker functions as in Figure 10 during the whole proof process of the assurance theorem.

7.3.1. Verifying the Static Obligations

For each of the static obligations partitioned from the assurance theorem, we formalize it as a theorem to be proved and then use Echo to formally verify it. To do this, we mechanically extract an abstract specification from the implementation (which is annotated) using Echo. This extracted specification represents the behaviors of the implementation. If there exists a formal specification for the system, we first prove that all the static obligations are satisfied by the original specification and then prove that the extracted specification implies the original one. Otherwise, we try to prove the static obligations directly over the extracted specification as the extended usage of Echo for property proof in Chapter 3 section 3.7.2. All the proofs are constructed and proved in a theorem prover (PVS as in the current Echo instantiation). Given the Echo approach all the proofs involved are static constructed and proved, we can statically establish that the obligations hold for the implementation.

7.3.2. Verifying the Dynamic Checker Functions

By definition, the runtime checks that are required during execution for the proof of the dynamic obligations have to be added to the source program. This means that dynamic checker functions have to be implemented and inserted into the source program, their

implementations verified, and the implementations tied into the overall proof argument. All the checker functions must be verified, otherwise the runtime checking process is not sound and risks can still be introduced. The verification of the checker functions is also a static proof that can adopt the Echo approach.

In the prototype runtime assurance support mechanism in synergistic analysis, we formalize all the dynamic obligations and use declarative annotations to document the desired behaviors of the dynamic checker functions. To integrate it together with the static support in Echo, SPARK annotations are used to match the current instantiation of Echo. SPARK proof functions are also used as a special kind of annotation to represent dynamic obligations and annotate the relevant source code.

The desired dynamic obligations are essentially the postconditions for the checkers. These obligations could be hard to check with the original code, but since the checker functions just check against these obligations directly, they can be easily verified statically and straightforwardly by applying the Echo verification approach.

In addition, there has to be a high level of assurance that the checker functions will always be invoked correctly when needed during execution. This is done through flow analysis and proof but is outside the Echo scope. Therefore I skip the details here.

7.4. Complementary to Echo

If the assurance goals of a system can be stated formally in the style of Safe Programming, synergistic assurance and analysis can distribute the burden of establishing properties of software between static proof and dynamic checking. By combining both static and dynamic techniques, it is able to exploit the benefits of each in a complementary manner.

The Echo verification approach focuses solely on static proof. When integrated with synergistic analysis, it allows Echo to be applied as appropriate to the assurance proof but without having to force everything into the Echo proof. In addition, a single complex obligation can be broken down into parts and those parts that are not suited for the Echo approach can be allocated to dynamic checking.

Clearly, even with dynamic analysis available, a static verification proof is always preferable if it can be achieved. The practical success of synergistic analysis can depend on identification of dynamic obligations that facilitate the desired static proof substantially and that can be implemented correctly. We can always begin by attempting to complete a static proof using the Echo approach. Difficulties with completing the proof thus suggest candidate dynamic obligations. For example, proving numeric bounds of floating-point arithmetic, and proving real-time constraints are not fully supported in Echo, yet can easily be checked during execution. In this way, synergistic analysis greatly extends the application domain of the Echo verification approach.

Chapter 8. Verification Argument

With all the major pieces in the Echo process presented in detail in previous chapters, I justify the soundness of the whole approach in this chapter. I also discuss the soundness argument when complex properties such as real-time and floating point properties are involved.

8.1. Soundness Justification

The verification argument in Echo, including the use of proof by parts for the implication theorem, is as follows:

1. The implementation proof establishes that the code implements the low-level specification (the annotations).
2. The transformations involved in verification refactoring are applied mechanically and proved to preserve semantics.
3. The specification extraction is automated or mechanically checked.
4. The implication theorem is constructed and proved.
5. The combination of (1) through (4) provides a complete argument that the implementation behaves according to its specification.

The verification argument is sound and will not generate false-positives. All defects in the code will be exposed in either the implementation proof or the implication proof. Any inconsistency between the code and the annotations will be detected by the

code-level tools in the implementation proof. An inconsistency could arise because of a defect in either or both. If both are defective but the annotations match the defective code, it will not be detected by the implementation proof. However in that case the annotations will not be consistent with the high-level specification, and so will be caught in the implication proof. The use of annotations does introduce an extra source of defects, however the use of them in routine development is being pursued by industry and is demonstrated to be beneficial.

8.2. Soundness Involving Complex Properties

8.2.1. Real-time Properties

Echo focuses only on functional correctness. If the processor selected on the target platform for the system to be verified support instruction cycle counts, some real-time properties such as WCET can be stated and annotated statically. They can thus be verified using the Echo approach. The soundness argument above still applies.

However, in most situations real-time properties cannot be stated like that and thus cannot be handled directly by the Echo approach. Such properties will be checked completely or partly at run-time in the synergistic assurance framework presented in Chapter 7. Details of the soundness for synergistic analysis is discussed elsewhere but in short, provided that:

1. The run-time checker is verified to be correctly implemented (using Echo);
2. The run-time checker is proved to be correctly invoked whenever necessary; and
3. The conjunction of the obligations that are run-time checked and the obligations that are statically proved (using Echo) is proved to imply the original property,

we can justify the property to be checked will be ensured.

8.2.2. Floating-point

Functionality correctness for software is often conceived of in terms of real-valued or integer-valued arithmetic, but digital computers can only implement arithmetic on finite types.

In the case of integer arithmetic, the practical distinction is well understood by programmers, who take care to allocate enough storage to handle the largest and smallest values a given variable might take on. Potential problems such as overflows can be precisely checked by current tools. In the Echo approach, such checks are done in implementation proof using the SPARK Examiner. It checks errors including array index out of range, type range violation, division by zero, and numerical overflow. All these ensure the implementation proof is sound.

On the other hand, the distinction between real-valued arithmetic and a floating-point approximation is less well understood. Even if each step is required to conform to the IEEE-754 standard, the floating-point semantics (rounding and exceptions) might still make the source programs' behavior difficult to foresee and analyze. The Echo approach treats floating-point arithmetic as if it were real-valued arithmetic with a bounded range. The soundness with regard to real-valued arithmetic is the same as the above argument. It checks for instance that the implementation does not use one variable when another was meant or multiplication where addition was meant. It cannot, however, tell whether the adopted floating-point arithmetic is adequately precise for the target software. By adopting SPARK Ada as the implementation language in the present instantiation of Echo, it adopts and accepts SPARK Ada approximation.

Verifying floating-point computation to adequately substitute for real-valued arithmetic is quite complicated. Formal methods have been successfully used both for hardware-level and high-level floating-point arithmetic. If the rounding and approximation semantics are built into verification condition generation for the source code adopted by Echo, one might be able to reason about not only properties about the actual floating-point values but also about the ideal real number values, for instance the bound of the difference of the actual output to the exact output without rounding. Such a technique does exist [11] and has been demonstrated to be useful.

Chapter 9. Evaluation Overview

In this chapter I summarize the evaluation of the Echo approach to formal verification. I then briefly introduce the three specimen systems that I selected to conduct case studies, and describe how each fits into the overall evaluation. Details of the case studies and evaluation are in the next several chapters.

9.1. Research Approach

After the initial proof-of-concept stage, my research on the Echo approach was based on incrementally prototyping and assessment using specimen systems. I used specimen systems to help define the details of the approach and to evaluate concepts and techniques as they were developed (see Figure 11). For each specimen system, I built the necessary artifacts and applied formal verification using Echo. I used the experience to continuously refine and assess all aspects of the Echo approach. Following this approach, I conducted several case studies aiming to ensure that the technical concepts developed were based on practical considerations and to ensure that the resulting techniques were suitable for practical use.

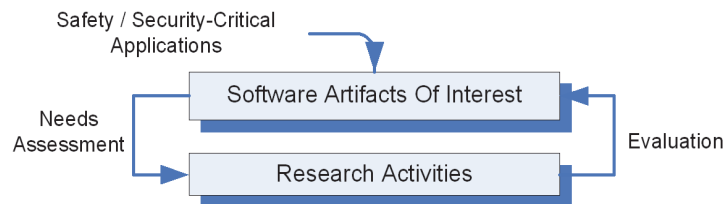


Figure 11. Overall approach to experimentation

9.2. Evaluation overview

The approach to evaluation that I followed was to conduct case studies using the specimen systems available at the time to answer a range of questions pertaining to the research goals [82]. Recall that my thesis statement for the Echo approach is:

THESIS: *The Echo approach will make formal verification of functional correctness a practical yet comprehensive technique for real software systems of at least moderate length.*

And one of the goals for the Echo approach is to allow developers the maximum freedom possible in building a system in order to make the verification process practical. I sought a way to assess the success in meeting the goals as well as the utility of the overall technique by applying the technique to important yet publicly-available system written entirely by others. Clearly, the system's development would not be constrained by the verification requirements of Echo. To assess my thesis statement, evaluation was conducted towards the end of the verification cycle of each specimen system with a view to determining the following criteria:

- (1) Applicability
- (2) Scalability
- (3) Efficacy
- (4) Efficiency

I evaluated the Echo approach by conducting case studies on three separate specimen systems. In the next section, I introduce the three specimen systems and describe how each fits into the overall process of evaluation.

9.3. Specimen Systems

I selected the following three specimen systems for assessment and evaluation of the Echo approach and its constituent parts. They were chosen to provide challenges that are both realistic and diverse.

Advanced Encryption Standard: The Advanced Encryption Standard (AES) is a symmetric, iterated block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. The AES algorithm is specified in the Federal Information Processing Standards Publication 197 [26]. A reference ANSI C implementation of AES was developed by Rijmen et al. [20] which is publicly available. The implementation is 1258 lines of code excluding all the APIs and contains several documented optimizations to enhance performance.

I translated necessary artifacts involved and applied the Echo verification to determine the various aspects of the approach and provide some fundamental assessment of the efficacy and utility, with major focus on the reverse synthesis process, which is the core process of the Echo approach. This case study is presented in Chapter 10.

A second case study was also conducted on AES, which evaluated the defect detection and localization effectiveness by randomly seeding defects into the implementation. This is presented in Chapter 11.

Tokeneer ID Station: The next specimen system is known as Tokeneer Enclave Protection system[8], and it was developed under the direction of the National Security Agency (NSA). Tokeneer is an authentication system for a secure enclave that contains secure workstations. Users have security tokens that are used to gain access to the enclave and to the workstations. Different users have different privileges. The subject Tokeneer ID Sta-

tion (TIS) is a stand-alone entity responsible for performing biometric verification of the user and controlling human access to the enclave. A formal specification of TIS has been developed in Z and an implementation has been developed in SPARK Ada by Altran Praxis. The implementation is 30,278 lines of annotated, commented source code.

My colleagues and I were offered technical support by NSA and Altran Praxis in working with the existing Tokeneer artifacts when the case study to verify the TIS implementation was conducted. At the time of this thesis is written though, all Tokeneer artifacts (documents, specifications, implementations, etc.) have been made publicly available.

With this case study, I mainly aimed to evaluate the practicality and scalability to apply the Echo approach to large software systems, details of which is presented in Chapter 12.

LifeFlow Magnetic Bearing Control Software: The University of Virginia’s LifeFlow Left Ventricular Assist Device (LVAD) [72, 73, 75] is a prototype artificial heart pump designed for the treatment of Congestive Heart Failure (CHF). The UVa LVAD is an axial-flow design that uses magnetic rather than mechanical bearings to avoid damaging blood cells, thus reducing both the potential for the formation of a fatal blood clot and the need to take blood thinning medications. The device operates with several control loops, including the control of the magnetic bearings. Control of the magnetic suspension bearings is provided, in part, by a digital control algorithm running on a microcontroller.

The team developing the UVa LVAD project supplied full system details and access to domain experts. My colleague, Patrick Graydon, developed an implementation of the LifeFlow Magnetic Bearing Control Software (MBCS) in SPARK Ada during his

research on Assurance Based Development [30, 31]. As part of the project, I applied the Echo approach to verify the functional correctness of the control loop in the MBCS digital implementation. The involved code is not large (around 800 lines of annotated code excluding hardware interface and logging) but contains the usage of floating point calculations. With this case study, I once again aimed to evaluate Echo's applicability and efficacy, but also focused on assessing the impact of floating-point properties.

Chapter 10. Case Study: Verification of AES

In this chapter I present the first case study of applying the Echo approach to a small but non-trivial specimen system. This case study illustrates the various aspects of the approach and provides some fundamental assessment of Echo's feasibility and the efficacy and utility of the reverse synthesis process, which is the core process of the Echo approach.

10.1. Efficacy and Utility Assessment

The issues that affect the efficacy and utility of reverse synthesis include: (1) the ease with which developers can select verification refactorings; (2) the ease with which developers can add domain specific refactorings and prove them to be semantics preserving; (3) whether selected refactorings do facilitate the specification extraction and necessary proofs; and (4) whether reverse synthesis impedes development in some way.

Issues 1, 2, and 3 are tied closely to the use of code metrics in the verification refactoring process, since I anticipate the values of metrics being the basis for developers' decisions. I sought to determine: (1) the impact on code metrics of individual types of refactoring and of series of refactorings; and (2) the values of the metrics for software that was amenable to proof and refactorings that were suggested by the values of metrics. This chapter focuses on the experience with reverse synthesis in application of the Echo approach to verify of the subject application and provides information about the first three issues. The fourth issue will be addressed in Chapter 11.

10.2. The Advanced Encryption Standard

The subject of study was the Advanced Encryption Standard. I used artifacts from the National Institute of Standards and Technology (NIST) and applied the Echo approach to verify the functional correctness of the complete AES implementation.

The Federal Information Processing Standards Publication 197 [26] specifies the AES algorithm. It is a symmetric, iterated block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. The number of rounds employed is a function of the key lengths. Each encryption round is composed of four critical subroutines: SubBytes, ShiftRows, MixColumns, and AddRoundKey. Each decryption round is composed of the reverse of these steps. AES also specifies what is referred to as a key expansion routine that is used to generate a series of different keys, one for each round, from the cipher key. The basic unit used in the specification is a byte (8 bits).

The AES artifacts that I employed were:

FIPS 197 specification. The specification is 26 pages long and describes the AES algorithm, mostly in natural language with mathematical statements and pseudo code for some algorithmic elements [26].

ANSI C implementation. Developed by Rijmen et al. [20], this optimized implementation is written in ANSI C. It is 1258 lines of code excluding all the APIs and contains several optimizations to enhance its performance.

Both the specification and the implementation were written entirely by others, and so there were no constraints on the development process imposed by the subsequent application of Echo. I assume that these artifacts were created by a traditional software devel-

opment process, and that the developers took no actions that would make formal verification infeasible or very difficult.

10.3. AES Verification

I supplemented these artifacts as necessary to apply the Echo approach by translating the notations into the ones used in the current Echo instantiation. I developed a formal version of the FIPS 197 specification in PVS by formalizing all the behaviors and constraints described in the FIPS specification and including them in a formal PVS specification. I also translated the ANSI C implementation into SPARK Ada by translating each C statement into corresponding Ada statement. The PVS specification is 811 lines long, excluding boilerplate constant definitions. The SPARK Ada implementation is 1365 lines without annotations and excluding all the APIs. In practice, such formal specification and annotated implementation might be produced by developers, making this type of translation unnecessary.

The verification of AES employed the complete Echo process:

1. A series of refactoring transformations were applied;
2. The final refactored version was documented using the SPARK Ada annotation language;
3. The code was shown to be compliant with the annotations;
4. A high-level specification was extracted from the refactored, annotated code; and
5. The extracted specification was shown to imply the original specification.

10.4. Verification Refactoring

According to the original AES documentation [20], the AES implementation employs various optimizations, including:

1. Implementing functions using table lookups;
2. Fully or partially unrolling loops;
3. Packing four 8-bit bytes into a 32-bit word; and
4. Inlining functions.

These optimizations improved performance but also created difficulties for verification. For instance, the SPARK tools ran out of resources on the original program because the unrolled loops created verification conditions that were too large.

I applied a total of 50 verification refactoring transformations in eight categories. Of those 50, the following 38 transformations from six categories were selected from the prototype Echo refactoring library (the number after the category name is the number of transformations applied in that category):

- Rerolling loops (5);
- Reversing inlined functions or cloned code (11);
- Splitting procedures (2);
- Moving statements into or out of conditionals (3);
- Adjusting loop forms (4);

- Modifying redundant or intermediate computations (2); and
- Modifying redundant or intermediate storage (11).

Each transformation was proved to be semantics-preserving using PVS and was applied mechanically using Stratego. The goal was to reverse the optimizations that were causing difficulties for verification as well as to help match the code structure to the specification. The detailed rationale and use of these transformations are discussed in the next section. Among them rerolling loops and reversing inlined functions were aimed directly to reverse the documented optimizations:

- **Rerolling loops (5).** Rerolling loops involved locating the repeated code, redefining it as a for-loop, and changing literal references to use the new loop induction variable. This transformation introduced two new loop induction variables and dramatically shrank the code size as will be shown in next section since vast amount of repeated code were removed. After the transformation, loop invariants could be annotated to facilitate the verification. This transformation assisted the implementation proof, because by introducing new loop invariants and removing replicated loop bodies, it substantially reduced the states involved in the proof.
- **Reversing inlined functions (11).** In the AES implementation, inlined functions obscured events that are explicitly required by the specification. Reversing such inlining aided both the implementation proof and the implication proof. By finding cloned code fragments, it removed replicated or similar proof obligations in the implementation proof. By reversing the inlining, it aligned the code structure with the specification structure so that the implication proof was easier to be constructed. After undoing

inlining, source code size might actually increase due to the addition of verbose function definitions, but the conceptual complexity was reduced.

In addition to the transformations above, I also added two new categories of transformations specifically for AES:

- **Adjusting data structures (2).** 32-bit words were replaced by arrays of four bytes, and sets of four words were packed into blocks or states as defined by the specification. Constants and operators on those types were also redefined accordingly to reflect the transformations. This type of transformation was targeted to undo the word-packing representation optimization. The AES standard describes encryption in terms of bytes, but the original implementation packs the bytes into 32-bit words to utilize efficient word-level operations. The AES implementation includes utility functions to split and combine 32-bit words; the bytes inside a word are referenced by bit shifting. After transformation by references to 32-bit words were replaced by arrays of four bytes. Thus splitting, combining, and references to bytes used native array operations. Specialized procedures for manipulating packed data were removed, but every line of code that referenced packed data had to be updated to use the new representation. This type of transformation assisted the implication proof since the code and the specification used the same basic type to refer to data after it and were thus easier to verify.
- **Reversing table lookups (10).** Ten table lookups were replaced with explicit computations based on the original documentation and the precomputed tables removed. The AES implementation combined different cryptographic transformations into a single set of table lookups. The tables contain pre-computed outputs and thus reduce the runtime computation. The properties of those tables have been documented [20], hence

the AES implementation maintains the invariant `Table[i] = computation(i)`. After this transformation, references to these tables in the form of `Table[i]` were replaced with inlined instances of the appropriate computations `computation(i)`. As a result, all tables were removed causing a dramatic code-size reduction as is shown in next section. It also made the implication proof easier since the specification was phrased in terms of the computations, not the tables.

Both of these two added transformation types were driven by the goal of reversing documented optimizations and matching the extracted specification to the original specification. Reversing table lookups and reversing function inlining were dependent though since the table entries encoded part of the defined functions, so reversing table lookups happened before reversing function inlining.

The final refactored AES program contained 25 functions and was 506 lines long.

10.5. Complexity Metrics Analysis

Using the heuristics mentioned earlier in Chapter 6, I selected and ordered transformations to use with AES. Rather than examining the effects of each transformation separately, I grouped the transformations into the following 14 blocks:

1. Loop rerolling for major loops in the encryption and decryption functions;
2. Reversal of word packing to use four-byte arrays;
3. Reversal of table lookups;
4. Packing four words into a state/block;
5. Reversal of inlined subroutines for major the encryption and decryption functions;

6. Reversal of inlined functions for key expansion subprograms;
7. Moving statements into conditionals to reveal three distinct execution paths followed by procedure splitting;
8. Adjustment of loop forms;
9. Reversal of additional inlined functions;
10. Loop rerolling for sequential state updates;
11. Procedure splitting;
12. Adjustment of intermediate variables;
13. Adjustment of loop forms; and
14. Additional procedure splitting.

Blocks 7-11 were for the subprogram that set up the key schedule for encryption, and blocks 12-14 were for the subprogram that modified the key schedule for decryption. As well as the main transformations, each block of transformations involved smaller transformations that modified redundant or intermediate computations and storage.

As part of determining whether further refactoring was required, I periodically attempted the proofs and determined the source-code metrics. Some of the results of the effect of applying the transformations on the values of the metrics are shown in Figure 12. The histograms show the values of different metrics after the application of the 14 blocks of transformations where block 0 is the original code.

As the transformations were applied, the primary element metric, code size, dropped. The non-comment, non-annotation lines dropped from over 1365 to 412 mainly because loops were rerolled and precomputed tables were removed. The other transformations had little additional effect on length. I hypothesize that fewer source code lines will usually result in shorter annotations and verification conditions.

The average McCabe cyclomatic complexity also declined as transformations were applied, dropping from 2.4 to 1.48. Statement complexity, essential complexity, etc. also declined. There is no evidence that these complexity metrics are related to the difficulty of

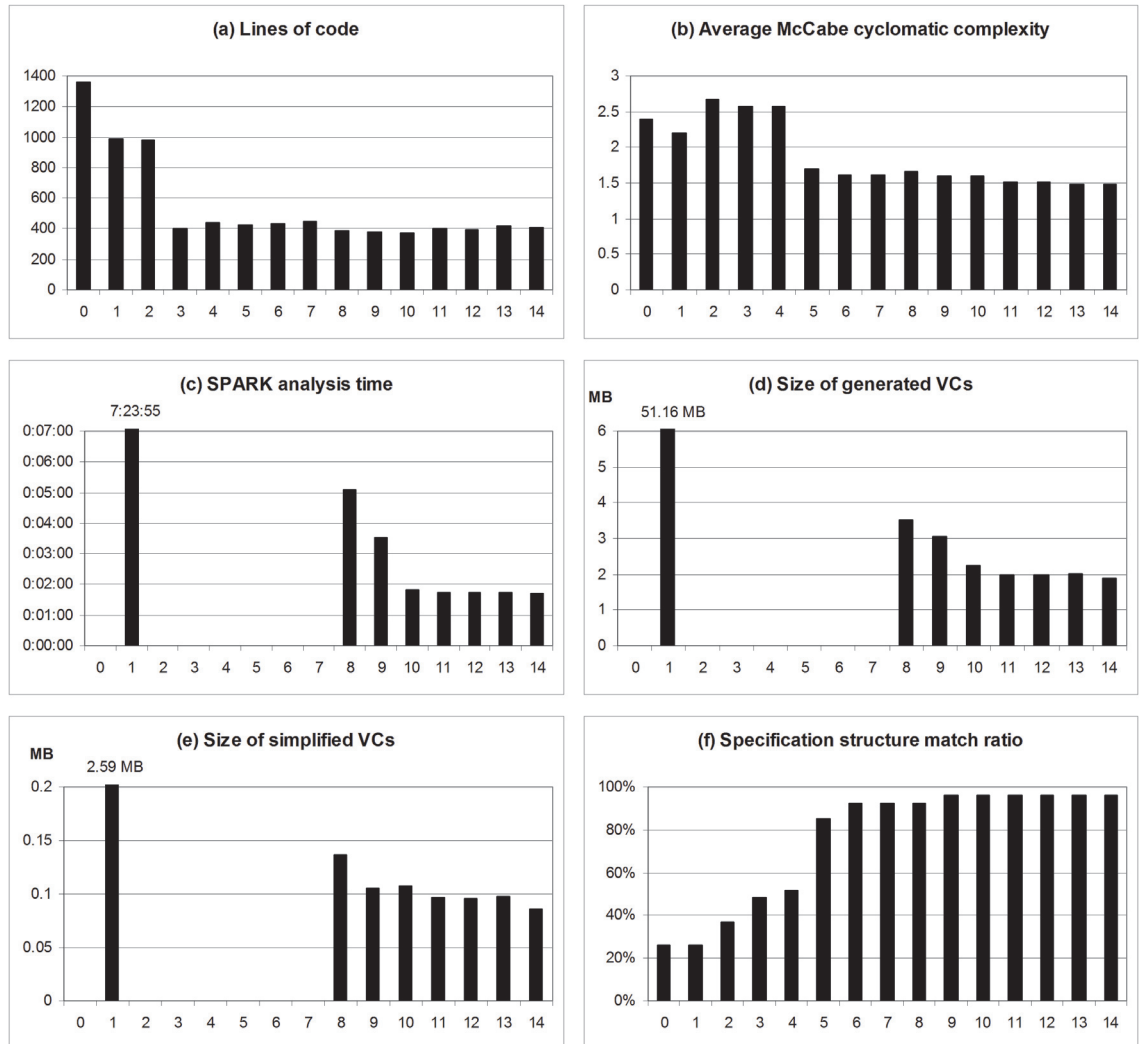


Figure 12. Metric analysis with AES verification refactorings

verification, but their reduction suggests that the refactored program might be easier to analyze.

Since in the Echo approach I would not undertake full annotation until refactoring was complete, I had no way to assess the feasibility of the proofs. To gain some insight, I set the postconditions for all subprograms to *true* for each version of the refactored code, generated verification conditions (VCs) using the SPARK examiner, and simplified the generated VCs using the SPARK simplifier. I then measured the number of VCs, the size of VCs, the maximum length of VCs, and the time that the SPARK tools took to analyze the code. These data did not necessarily represent the actual proof effort needed for the implementation proof, but they were an indication.

The times required for analysis with the SPARK tools after the various refactorings are shown as Figure 12(c). Some blocks are shown with no value because the VCs were too complicated to be handled by the SPARK tools. After the first loop rerolling at block 1, the tools completed the analysis but took 7 hours and 23 minutes on a 2.0 GHz machine. At block 2 with word packing reversed, the analysis again became infeasible. It was not until block 8, when the loop forms in the key schedule setup function had been adjusted that analysis by the SPARK tools became feasible again. The required analysis time gradually decreased and reached 1 minute 42 seconds for the final refactored program (a reduction of more than 99%).

The size of VCs showed the same declining trend. In block 1, 51.16 MB VCs were generated and 2.59 MB VCs were left after simplification. For the final refactored code, 1.90 MB VCs were generated and 86 KB were left after simplification (Figures 12(d) and 12(e)).

The simplified VCs were those that needed human intervention to prove. After block 1, the maximum VC length was over 10,000 lines, far beyond what a human could manage. In the final refactored code, the maximum was 68 lines. When the implementation annotation was complete, the maximum length of VCs needing human intervention was only 126 lines.

I extracted a skeleton specification from the code after applying each block of transformations. These specifications were skeletons because they were obtained before the code had been annotated and so contained none of the detail from the annotations. I compared the structure of the skeleton extracted specification with that of the original specification by visually inspection and evaluated the match-ratio metric, which is the percentage of key structural elements—data types, operators, functions and tables—in the original specification that had direct counterparts in the extracted specification as defined in Chapter 4. This measure may indicate the likelihood of successfully establishing the implication proof.

The values of the match ratio are shown in Figure 12(f). The ratio increased gradually from 25.9% to 96.3% as the transformation blocks were applied. There is only a small increase in its value after the block 8 transformations were applied, and the implication proof could have been attempted at that point. However, although some metrics had stabilized after the block 8 transformations, the time required for the SPARK analysis was still declining, and so I chose to continue refactoring until that metric stabilized also.

10.6. Implementation Proof

After refactoring, the code was examined and annotated manually. The actual numbers of annotations are shown in Table 2:

Type	Lines
Pre-conditions	8
Post-conditions	123
Loop Invariants & Assertions	54
Proof Functions, Proof Rules, & Other	32

Table 2: Annotations in implementation proof

The implementation proof was carried out using the SPARK Ada toolset. The SPARK examiner generated 306 verification conditions for all 25 functions, of which 265 verification conditions (86.6%) were discharged automatically by the SPARK simplifier in 145 seconds on a 2.0 GHz machine. All of the verification conditions for 15 of the 25 functions were discharged automatically.

The remaining verification conditions required manual intervention using the SPARK proof checker. The manual intervention was quite straightforward, usually involving either the application of pre-conditions or induction on loop invariants. The interactive proof process for each remaining verification condition was finished within a few minutes by a single individual who has a good level of SPARK Ada experience.

Total VCs by type: ----- Proved By Or Using -----					
	Total	SPARK Examiner	SPARK Simplifier	Proof Checker	Review
Assert or Post:	82	0	54	28	0
Precondition check:	5	0	2	3	0
Check statement:	0	0	0	0	0
Runtime check:	219	0	209	10	0
Refinement VCs:	0	0	0	0	0
Inheritance VCs:	0	0	0	0	0
Totals:	306	0	265	41	0
% Totals:		0%	87%	13%	0%

Table 3: AES implementation proof summary

A summary of how all the VCs are discharged are listed in Table 3. Throughout the implementation proof process, the length of the verification conditions remained completely manageable. No difficulties were encountered in reading or understanding them, or in manipulation of them with the SPARK tools.

10.7. Implication Proof

The extracted specification (in PVS) produced by the Echo specification extraction tool was 1685 lines long. It was much larger than the original specification because the implementation contained two tables for computing multiplication in the $GF(2^8)$ field which were not present in the original specification.

When typechecking the extracted specification, the PVS theorem prover generated 147 Type Correctness Conditions (TCCs), of which 79 were discharged automatically by the theorem prover in 23.5 seconds on a dual 1.0 GHz machine and the remaining 68 were all subsumed by the proved ones.

As a result of verification refactoring, the architecture of the extracted specification was sufficiently similar to the architecture of the original specification that I was able to identify the matching elements easily. To prove the extracted specification implied the original one, I created an implication theorem using the proof by parts process in Echo as explained in Chapter 4.

There were 32 major lemmas in the implication theorem. Type checking of the implication theorem resulted in 54 TCCs, 29 of which were discharged automatically in 4.2 seconds on a dual 1.0 GHz machine and 25 were subsumed by the proved ones.

In most cases, the PVS theorem prover could not prove the lemmas completely automatically. However, the human guidance required was short and straightforward, typ-

ically including expansion of function definitions, introduction of predicates over types, or application of extensionality. In some cases, introducing other previously proved supporting lemmas and structuring the proof as cases was required. Each lemmas was established and proved interactively in a few minutes (thus the implication theorem discharged).

10.8. Summary

The AES implementation chosen for this case study is moderate-sized program written by others and not designed for formal verification. The off-the-shelf SPARK toolset could not even analyze it and generate verification conditions. Instead it quickly exhausted heap space and stopped, presumably because the generated proof obligations were too large.

By applying the Echo approach to verify the AES implementation, it showed that Echo is feasible to be applied on real systems. In particular, it overcomes the problem of unworkably large verification conditions and does not require the developers to follow a rigid development process necessary for refinement.

Efficacy and utility of the Echo approach, especially that of the reverse synthesis process have been demonstrated in this case study. The verification refactoring process was guided by a set of complexity metrics that helped both select transformations and determine when the refactored program was likely to be amenable to proof. Off-the-shelf verification was impossible using conventional tools, but the addition of refactoring made the task both feasible and straightforward as shown in sections 10.4 and 10.5.

Identification of refactoring transformations in this case study was done with manual guidance. The process is straightforward. With certain level of verification knowledge, one could identify common transformations that could make proof obligation simpler, e.g. loop rerolling, reversing function inlining, procedure splitting, etc. Domain specific trans-

formation was easily identified in this case as the specific optimizations were documented. In the general case, one might need some domain knowledge of the subject system to be able to accurately identify these. However, many could be deduced from specification, i.e. reversing the word packing to match specification. And I also expect the conventional software development artifacts may well record why and where domain specific optimizations were applied. With further tool support, identifying many of the transformations can be done automatically [45, 48, 63]. Such support will be pursued in future work. On the other hand, selecting the transformation spots, performing the transformation, and proving the preservation of the semantics were all machine checked using Stratego and PVS.

After reverse synthesis, both implementation and implication proof were completely manageable by a single individual. Many proof obligations were discharged automatically and the remaining ones only required straightforward human intervention as shown in sections 10.6 and 10.7. It did require the person carrying out the verification to have knowledge of various analytic tools including the SPARK tools and the PVS theorem prover, and a good level of skill on performing the proofs.

Chapter 11. Case Study: Defect Detection in AES Verification

In this chapter I continue to use the previous verification of the AES specimen system for another case study for the Echo approach. This case study evaluates how effective defects can be detected and located during the Echo approach by randomly seeding defects into the AES implementation.

11.1. Reverse Synthesis and Defect Detection

When using formal verification, defects in the subject programs are revealed by a failure to complete the proof. Proof failures always present the dilemma that either the program or the proof could be wrong. But this dilemma is present with any verification method, including testing.

The reverse synthesis process especially the verification refactoring in the Echo approach, might make the dilemma worse or introduce other forms of difficulty in identifying defects. In order to investigate this issue, I seeded defects into the original AES implementation and then determined the effect of each defect on verification. I present the results of that case study in this chapter.

11.2. The Seeding Process

The seeding process was done by randomly choosing a line number and making a change in the code. Each defect in the program was a change in either:

1. A numeric value;

2. An array index;
3. An operator (for computation or predicate);
4. A variable or table reference; or
5. A statement or function call.

These types of defect are not equivalent to those introduced by programmers. However, they do reflect common errors that might be introduced when coding the AES implementation, and there is some evidence that simple seeded defects share important properties with actual defects [43].

Code and therefore the defects are closely tied into the annotations that document the low-level specification. The defective code could be annotated so as to either describe its desired behavior rather than its actual behavior, or vice versa. I used both scenarios in this experiment and evaluated them separately.

11.3. Defect Location

There are three stages in the proof process that could expose defects in the code:

- **Verification refactoring.** Technically the refactoring process is not supposedly to be a place that defects would be exposed. However, if the refactoring transformation is derived from a documented optimization, a defect could change the code such that it did not match a particular transformation template and the transformation could not be

applied. For example, a defect in only one iteration of an unrolled loop rather than in all interactions would make loop rerolling inapplicable.

- **Implementation proof.** Any inconsistency between the code and the annotations would be detected by the SPARK Ada tools. An inconsistency could arise because of a defect in either or both. If both were defective but consistent with each other, then the combined defect would not be detected by the SPARK Ada tools. In that case, the annotations would not be consistent with the high-level specification, and so the defects would be caught in the implication proof.
- **Implication proof.** Defects in the code or post-condition annotations that are too weak in the program used to create the extracted specification would cause the implication theorems to be unprovable and so would be detected by the implication proof.

11.4. Case Study Results

I seeded 15 defects, three defects of each basic type as described in section 11.2, one at a time into the AES implementation, and then I ran the Echo verification process twice for each defect.

In the first (setup 1), I assumed that the defects were caused by misunderstandings of the specification when implementing the code, and thus annotated the code so that the annotations corresponded to the functional behavior of the code.

In the second (setup 2), I assumed that the defects were introduced by implementation errors, and annotated the code so that the annotations corresponded to the high-level specification. The results are shown in Table 4 and Table 5.

11.4.1. Defect Detection

Verification Stage	Defects Caught	Defects Left
Initial state		15
Verification refactoring	4	11
Implementation proof in SPARK	2	9
Implication proof in PVS	8	1

Table 4: Defect detection for setup 1

For setup 1, most defects were caught during the implication proof since the annotation matched the code. Four defects that were caught in the verification refactoring because they made reversing documented optimization not applicable, i.e. loop rerolling or reversing table lookups. The two defects that were caught in the implementation proof were found during the proof of exception freedom because they caused possible out-of-bound array references. The remaining defect that was not caught at any stage was benign. It will be discussed later.

Verification Stage	Defects Caught	Defects Left
Initial state		15
Verification refactoring	4	11
Implementation proof in SPARK	10	1
Implication proof in PVS	0	1

Table 5: Defect detection for setup 2

For setup 2, most defects were caught during the implementation proof since the annotation did not match defective code. The four defects caught during verification refactoring were the same ones as caught in setup 1. The remaining defect was the same benign defect.

In both setups, verification caught the same 14 seeded defects. The remaining (benign) defect changed the key schedule, the array of keys used in AES sequential rounds. The length of the array had been set to accommodate the maximum number of rounds in the case of a 256-bit key length. However for key lengths of 128 bits or 192 bits,

the last several entries in the array were not used in the computation. This was purely an implementation decision, and the specification did not impose any restrictions. Thus, for shorter key lengths these entries could be allowed to have arbitrary values without affecting functional correctness.

11.4.2. Defect localization

Echo does require that the developer annotate the code, and, whenever there is an unprovable proof obligation, the user has to determine whether it is the result of a defect in the code or the annotations. However, the use of architectural and direct mapping in the creation of the extracted specification means that the location of defects can be restricted to the function that cannot be proved.

In the AES case study, architectural and direct mapping were strictly followed. Functions in the specification and functions in the refactored code had direct counter parts with each other. Whenever an unproved obligation arose in the implication proof of an implication lemma, it was very easy to go back and revisit the corresponding code fragment in the implementation. For all the defects detected in the Echo verification, I located the defects by manually inspecting the corresponding code and unproved proof obligation. With the verification refactoring applied in the Echo verification of AES, each function was quite small and manageable, making defect location quite simple.

11.5. Summary

By randomly seeding defects into the AES implementation to be verified, this case study demonstrated that the Echo approach is capable of detecting defects in different stages effectively. No false-negative was reported and benign defects were left undetected.

Verification refactoring in Echo did not create extra difficulties for the defect detection and localization. On the contrary, it helped break down the proof into smaller isolated pieces so that it was easier to locate a defect once detected.

Annotation did introduce another source of defects. However in the AES study it did not give much trouble on identifying the defects, presumably because the annotation size was much less than the code itself. In verification, the benefits overweigh the efforts added to locate the defects.

Chapter 12. Case Study: Verification of Tokeneer

In this chapter I present the case study of applying the Echo verification approach to a larger specimen system. It demonstrates how the Echo approach scales with the proof by parts infrastructure, and its applicability and practicality to be applied to real software systems of at least similar size.

12.1. The Tokeneer system

The second specimen system for which I conducted a case study of applying the Echo approach is a hypothetical system called Tokeneer that was defined by the National Security Agency (NSA) to act as a challenge problem for security researchers. The system consists of a secure enclave containing a number of workstations having access to files with various restrictions. The challenge to researchers is to construct implementations with appropriate security properties.

Tokeneer is a large system that provides protection to secure information held on a network of workstations situated in a physically secure enclave. The system has many components including an enrolment station, an authorization station, security resources such as certificate and authentication authorities, an administration console, and an ID station. I used the core part of the Tokeneer ID Station (TIS) in this research. TIS is a stand-alone entity responsible for performing biometric verification of the user and controlling human access to the enclave. To perform this task, the TIS asks the individual desiring access to the enclave to present an electronic token in the form of a card to a reader. The

TIS then examines the biometric information contained in the user's token and a fingerprint scan read from the user. If a successful identification is made and the user has sufficient clearance, the TIS writes an authorization onto the user's token and releases the lock on the enclave door to allow the user access to the enclave. The overall organization of the Tokeneer ID Station is shown in Figure 13.

Much of the complexity of the TIS derives from dealing with all eventualities. A wide variety of failures are possible that must be handled properly. Since Tokeneer is a security-critical system, key security properties such as unlocking only with a valid token and within an allowed time, and keeping consistent audit records need to be assured with high level of confidence.

A fairly complete implementation of major parts of the TIS has been built by Altran Praxis (formerly Praxis High Integrity Systems). The Praxis implementation includes a requirements analysis document, a formal specification written in Z, a detailed design, a source program written in SPARK Ada, and associated proofs. The complete Z specification is 117 pages long. The SPARK Ada implementation contains 9939 lines of

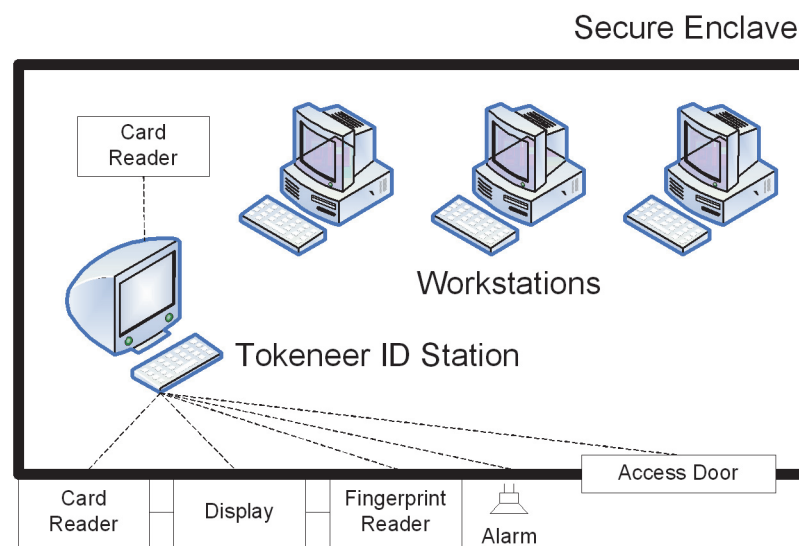


Figure 13. The Tokeneer ID Station

non-comment, non-annotation code. I was fortunate to be able to use these artifacts as the subject of study for this case study. Since the tools used in the current Echo approach operate with PVS, I translated the Z specification of the TIS into PVS. The final specification in PVS is 2336 lines long.

12.2. Echo Proof of Tokeneer

The Praxis implementation of Tokeneer was developed using Correctness by Construction [19] with the goal of demonstrating rigorous and cost effective development. High-level security properties were established by documenting the properties using SPARK Ada annotations, including them in the code, and then proving them using the SPARK Ada tools. By contrast, the Echo proof is of the full functionality of the implementation as defined by the original, high-level specification. Given the Echo proof of functionality, high-level security properties can be established by stating the properties as theorems and proving them against the high-level specification.

Turning now to the proof itself, upon review, I found that the TIS source program structure resembled the specification structure very closely, i.e., the architectural matching hypothesis was true. Almost all states and operations in the specification have direct counterparts in the source program. For example, for the `UnlockDoor` operation defined for system internal operations in the specification:

```
UnlockDoor(dla_i, dla_o: DoorLatchAlarm, c: Config): bool =
    dla_o`latchTimeout = dla_i`currentTime + c`latchUnlockDuration AND
    dla_o`alarmTimeout = dla_i`currentTime
                        + c`latchUnlockDuration + c`alarmSilentDuration AND
    dla_o`currentTime = dla_i`currentTime AND
    dla_o`currentDoor = dla_i`currentDoor
```

A corresponding procedure with the same name in the `Door` package could be easily identified in the source program:

```

procedure UnlockDoor
is
    LatchTimeout : Clock.TimeT;
begin
    LatchTimeout := Clock.AddDuration(
        TheTime      => Clock.TheCurrentTime,
        TheDuration => ConfigData.TheLatchUnlockDuration);
    Latch.SetTimeout(Time => LatchTimeout);
    AlarmTimeout := Clock.AddDuration(
        TheTime      => LatchTimeout,
        TheDuration => ConfigData.TheAlarmSilentDuration);
    Latch.UpdateInternalLatch;
    UpdateDoorAlarm;
end UnlockDoor;

```

If one examine the code in more depth, one could also find that there was also close correlation between statements in the specification and the code as shown above.

I excluded the peripheral interface functions in the Echo verification and performed a skeleton extraction. I found the match ratio to be 74.7%. The match ratio is not 100% (or at least close to it) because refinements carried out during the development added operations that were not defined in the specification. I concluded that the architectures of the specification and the source program were sufficiently similar that the necessary specification could be extracted effectively and easily without performing verification refactoring. The final extracted PVS specification is 5622 lines long. As an

example, the extracted specification using direct extraction from code for the above `UnlockDoor` procedure is:

```
UnlockDoor(st: State): State =
  LET LatchTimeout = Clock.AddDuration(TheCurrentTime(st),
                                         TheLatchUnlockDuration(st)) IN
  LET st1 = SetTimeout(LatchTimeout, st) IN
  LET st2 = st1 WITH [`AlarmTimeout := Clock.AddDuration(LatchTimeout,
                                                         TheAlarmSilentDuration(st))] IN
  LET st3 = UpdateInternalLatch(st2) IN
  UpdateDoorAlarm(st3)
```

Following extraction of the specification, I performed both the implementation proof and the implication proof.

The implementation proof was carried out using the SPARK Ada toolset to prove functional behaviors of those subprograms that had been documented with pre- and post-condition annotations. The SPARK Examiner generates verification conditions (VCs) that

Total VCs by type:	----- Proved By Or Using -----				
	Total	SPARK Examiner	SPARK Simplifier	Proof Checker	Review
Assert or Post:	1006	561	376	29	40
Precondition check:	67	0	60	3	4
Check statement:	1	0	1	0	0
Runtime check:	1337	0	1331	2	4
Refinement VCs:	212	182	2	9	19
Inheritance VCs:	0	0	0	0	0
Totals:	2623	743	1770	43	67
% Totals:		28%	67%	2%	3%

Table 6: TIS implementation proof summary

must be proved true to demonstrate that the code does indeed meet its specified postconditions, within the context of its preconditions. It also generates VCs that must be satisfied to ensure freedom of run-time exceptions. Altogether there were over 2600 VCs generated, among which 95% were automatically discharged by the toolset itself. The remaining 5% required human intervention, and were covered in the documents from Praxis' proof. A summary of how all the VCs are discharged are listed in Table 6.

The implication proof was established by matching the components of the extracted specification with those of the original specification. Identifying the matching in the case study was straightforward, and in most situations could be suggested automatically by pairing up types / functions with the same name as showed by the above example. For each matching pair, I created an implication lemma and altogether there were just over 300 implication lemmas. Typechecking of the implication theorem resulted in 250 Type Correctness Conditions (TCCs) in the PVS theorem prover, a majority of which were discharged automatically by the theorem prover itself. In 90% of the time, the PVS theorem prover could not prove the implication lemmas completely automatically. However, the human guidance required was straightforward due to the tight correlation between the original specification and source code, typically including expansion of function definitions, introduction of type predicates, or application of extensionality. Each lemma that was not automatically discharged was interactively proved in the PVS theorem prover by a single person with moderate knowledge about PVS (thus the implication theorem discharged). The total typechecking and proof scripts running is less than 30 minutes.

12.3. Scalability Evaluation

By successfully applying proof by parts to the functional verification of Tokeneer, I conclude that the Echo approach is both applicable and scalable on systems at least of similar size. Due to architectural similarity between the specification and the source program for the Tokeneer case, it worked very smooth and no major difficulties were met. In cases where the architectural matching hypothesis does not hold, verification refactoring can be applied to restructure the program and facilitate proof by parts. It is not demonstrated by the Tokeneer case study, however the previous case study in Chapter 10 has shown its feasibility. During the proof of Tokeneer, each implication lemma was individually proved, without interfering with other lemmas. Interactive proofs for many lemmas were similar in terms of proof strategy and commands. This also showed that the proof structure may scale for even larger systems without the need for extensive training of relevant knowledge.

By following proof by part, it also made defect localization easier since it must be inside the component whose corresponding lemma fail to be proved. During the proof of Tokeneer, I did find several mismatches between the source program and the specification but later found that they were documented changes. An example is the validity period in certificates. It is defined in the specification as a set of time which does not need to be contiguous. In the source code, however, it is defined as a pair of start and end time, which manifests as a contiguous period. The corresponding lemma could not be discharged since it was not a valid refinement. I later found that it was constrained to contiguous in design to reflect the nature of X509 certificates and was documented in the design document. As

future work, I plan to seed defects into the Tokeneer source program and then determine the effect on the proof to further evaluate its efficacy and utility.

Chapter 13. Case Study: Verification of MBCS

In this chapter I present another case study of applying the Echo verification approach during the development of the control software of a prototype artificial heart pump. This case study assesses the impact of floating-point calculations on the functional verification in Echo in addition to evaluation of Echo's applicability and efficacy.

13.1. MBCS System Description

The University of Virginia's LifeFlow Left Ventricular Assist Device (LVAD) [72, 73, 75] is a prototype artificial heart pump designed for the long-term treatment of Congestive Heart Failure (CHF). LifeFlow has a continuous, axial-flow design that uses magnetic rather than mechanical bearings to avoid damaging blood cells, thus reducing both the potential for the formation of a fatal blood clot and the need to take blood thinning medications. Magnetic bearings and a brushless DC motor keep the pump's impeller centered in the pump housing and turning without the need for mechanical bearings or shaft seals. The elimination of pinch points, coupled with careful design of the pump cavity, impeller, and blades, aided by computational fluid-dynamics simulations, minimizes the damage done to blood cells.

The device operates with several control loops, including the control of the magnetic suspension bearings. Control of the magnetic suspension bearings is provided, in part, by a digital control algorithm running on a microcontroller. In hard real-time, the controller must sample the position of the rotor as reported by a self-sensing circuit, com-

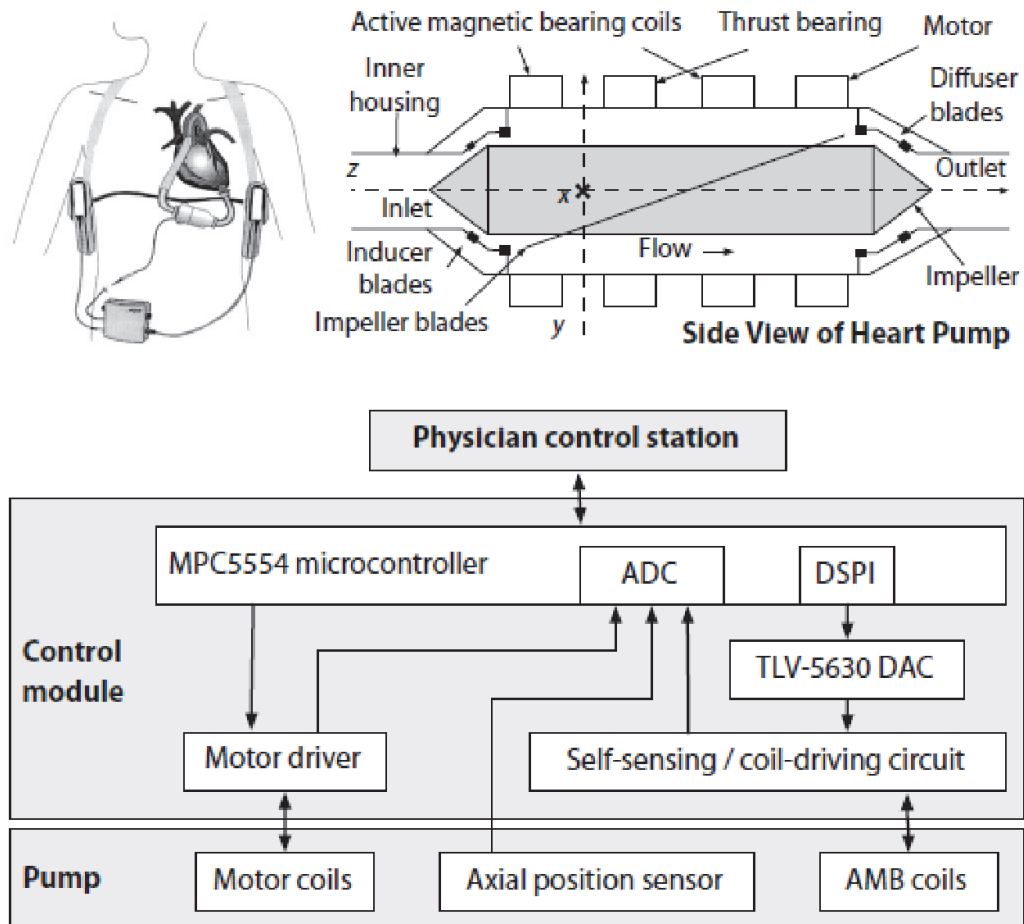


Figure 14. LifeFlow structure [30]

pute the coil currents necessary to keep the rotor adequately centered, and direct the coil driver to achieve those currents. Both individual magnetic coils and the wires connecting them to the control circuits can fail. Since such failures are anticipated to be more likely than is acceptable, the control software is also required to be capable of reconfiguring to a variety of backup modes in which rotor levitation is accomplished with only two of three coil pairs. Figure 14 shows the placement of the pump, the batteries and the controller, a cross-section of the pump, and the overall structure of the controller.

Table 7 briefly summarizes the requirements for the LifeFlow Magnetic Bearing Control Software (MBCS). My colleague, Patrick Graydon, developed an implementation

Functionality	<ol style="list-style-type: none"> 1. Trigger and read ADCs to obtain impeller position vector \vec{u} 2. Determine whether reconfiguration is necessary. If so, select appropriate gain matrices A, B, D, and E 3. Compute target coil current vector \vec{y} and next controller state vector \vec{x} $\vec{y}_k = D \times \vec{x}_k + E \times \vec{u}_k$ $\vec{x}_{k+1} = A \times \vec{x}_k + B \times \vec{u}_k$ 4. Update DACs to output \vec{y} to coil controller
Timing	This functionality must be provided in hard-real-time with a frame rate of 5 kHz
Reliability	No more than 10^{-9} failures per hour of operation

Table 7: Magnetic bearing control software requirements [30]

of the Magnetic Bearing Control Software in SPARK Ada as a case study for his research on Assurance Based Development [30, 31], which is a synergistic process for construction of both the critical software system and its assurance argument that demonstrates its fitness for use in given operating contexts. As part of the process, the Echo verification approach was chosen to provide evidence that the functionality requirement was enforced. I therefore applied the Echo approach to verify the functional correctness of the control calculation and the main cyclic execution structure in the MBCS digital implementation source code as indicated in the above table, and used it as a self-contained case study to get more assessment on Echo. Real-time requirement and reliability requirement in the table were determined by other approaches such as machine analysis of Worst-Case Execution Time (WCET). Correct compilation of the source code, correct execution of the binary on the target microcontroller, and correct usage of hardware interface were also out of the scope of Echo verification.

Complexity of functional correctness verification of the control software source code lies in the accuracy and correctness of control calculation. Since MBCS is a safety-critical system designed for long-term usage, the control calculation has to be correct, and any imprecision has to be limited to an acceptable level.

13.2. MBCS Verification

13.2.1. The Artifacts Employed

During the early stage of the project, an informal specification in natural language was developed by my colleague for the MBCS functionality. The specification refined the MBCS requirements and included control calculation functions, hardware interfaces, signal sampling, etc. For the Echo approach to be applied, I translated the portion to be verified into formal specification in PVS. The formal portion consisted of 226 non-comment lines, which specified control constants, needed types and arithmetic theorems, input and output conversion, control calculation, and the real-time frame synchronization mechanism. Hardware related interfaces were not formalized, but I translated bogus interfaces for them in the PVS specification and used them only to simulate the event sequence in the control frame.

An implementation of MBCS was also developed by the same colleague. The part to be verified by the Echo approach was completely done in SPARK Ada. The SPARK Ada portion of source code contained 2510 lines of annotated commented code, of which 579 lines implemented the control calculation, and 114 lines for the main cyclic execution program. These were the parts that were subjected to Echo verification. The rest of the lines were for hardware interfaces, logging, etc. and were minimally annotated and checked by means other than Echo.

13.2.2. Reverse Synthesis

Upon review, I found that the part of MBCS source code subject to verification had a structure resembled the specification structure closely and followed the architectural matching hypothesis completely. Presumably it was because the specification and implementation were both originated from the same person and the structural design was reused. All types, states, and operations in the specification had direct counterparts found in the implementation. I performed skeleton specification extraction from the SPARK Ada source code, compared the outcome with the original PVS specification, and found the match ratio to be exactly 100% if I excluded the helper arithmetic functions in the original specification. Also the involved SPARK Ada source code was already properly annotated during the development process and the verification conditions generated by the SPARK tools were of manageable size and complexity. I concluded that the verification refactoring might not be necessary for both implementation and implication proofs to proceed. I therefore directly applied specification extraction on top of the original code. Hardware interface and logging source code packages were not the subject of the Echo verification and were not fully annotated. Thus only skeleton extraction was performed on those packages and pre- and post- condition informations were not extracted. The final extracted PVS specification was 586 lines long in total.

13.2.3. Implementation Proof

The implementation proof was carried out using the SPARK Ada toolset, aiming to prove the functional behaviors of the subprograms conformed with their specified pre- and post-condition annotations, and also to prove the code was free of run-time exceptions. The

SPARK examiner generated 167 verification conditions (VCs) for all 17 subprograms involved. The ways to discharge these VCs are listed in Table 8:

Total VCs by type:	----- Proved By Or Using -----				
	Total	SPARK Examiner	SPARK Simplifier	Proof Checker	Review
Assert or Post:	54	12	13	13	16
Precondition check:	0	0	0	0	0
Check statement:	0	0	0	0	0
Runtime check:	93	0	92	0	1
Refinement VCs:	20	19	0	0	1
Inheritance VCs:	0	0	0	0	0
Totals:	167	31	105	13	18
% Totals:		19%	63%	8%	11%

Table 8: MBCS implementation proof summary

136 of the 167 VCs (82%) were discharged completely automatically by the SPARK examiner and simplifier.

Among the left 31 VCs that required human intervention, I proved 13 of them interactively inside SPARK proof checker. The proofs for these VCs were accomplished straightforwardly, each required less than 10 proof commands and many follows similar patterns due to the nature of matrix calculation in the source code. Corresponding proof script was saved so that it could be automatically invoked by the SPARK tools. The total length of the script was 196 lines.

The last 18 VCs could not be proved directly. However, upon review, 17 of them were generated from the hardware interface package which was not subject to Echo verification. 16 VCs were to check that the value in a control register conformed to the type of

the variable bound to that register, and the other one VC was to check that a hardware input routine was equivalent to the artificial proof function created to represent it. Since these were tightly coupled with the microcontroller hardware, and would be checked by other means, I asked my colleague to review them carefully and marked them as proved by review.

Besides the above VCs, there was one last VC that needed to be discharged but could not be proved directly. It originated from the subprogram that converted the control calculation results to update the DAC output. A simplified version of the VC (simplified by the SPARK simplifier) is:

```
H4:    (element(c, [loop__1__i]) + 2) * 1024 <= 4095 .
H5:    0 <= (element(c, [loop__1__i]) + 2) * 1024 .
C1:    round__((element(c, [loop__1__i]) + 2) * 1024) >= 0 .
C2:    round__((element(c, [loop__1__i]) + 2) * 1024) <= 4095 .
```

To prove this required knowledge of the hardware floating-point-to-integer conversion semantics. I had to assume that the hardware would follow the SPARK Ada's definition of the round operation. With that, it could immediately be deduced that C1 followed from H5, and C2 followed from H4, hence the VC could be discharged.

With all the proof script and proof review files, the SPARK tools generated, processed, and discharged all 167 VCs in 86 seconds on a 2.0 GHz machine.

13.2.4. Implication Proof

With the original PVS specification and extracted PVS specification, the Echo implication proof was carried out and checked by the PVS theorem prover to show that the low-level specification embodied in the SPARK annotations complies with the original PVS specification. The PVS implication theorem was constructed by importing both the original and

extracted PVS specifications, and setting up implication lemmas for each matching type, state, and operation. Excluding the hardware interfaces, the completed PVS theory file was 422 lines long, including 37 implication lemmas and a final implication theorem that was the conjunction of all the implication lemmas.

Type-checking and attempting to prove the implication theorem through the PVS theorem prover resulted in 55 formulas to be proved: 17 type correctness conditions (TCCs), 37 lemmas, and 1 theorem (which is the conjunction of the 37 lemmas). 46 of these formulas were successfully proved either automatically or by a single (`grind`) command in the PVS theorem prover in 210 seconds on a dual 1 GHz machine. 5 of them were proved interactively inside the theorem prover with human guidance. The remaining 4 formulas could not be proved and included 2 TCCs and 2 implication lemmas.

One TCC was related to the value type of the cells in the controller input, state, and output vectors. These were represented as single-precision floating-point variables in the SPARK Ada implementation but as real numbers in the PVS specification. The specification extracted the definition of single-precision floating-point type from SPARK Ada's target configuration file. This definition was an approximation of real numbers and had a bounded range. Obviously it could not be proved to precisely represent the unbounded real number as defined in the original specification. This unproved formula was presented but was discharged by review since the LifeFlow LVAD control engineers asserted that they could prove that these values would not exceed the limits of single-precision floating-point storage.

The other TCC and two implication lemmas were related to the time type. Time values derived from the MPC5554's 64-bit time base were represented in the SPARK Ada

implementation using a `mod 2 ** 64` type, which was a bounded integer type. However in the specification time type was specified as an unbounded integer to model infinite time base. Obviously the implementation time type could not cover all possible values in the specification and hence caused failure of the TCC's proof. Also, with a bounded time type implementation with 64-bit storage, it could overflow and "wrap around" when the time base advanced to a large enough number. Such overflow wouldn't occur with the specification's unbounded time type. The difference caused implication lemmas for two functions regarding the control frame failed to be proved since the "wrap around" semantics were not defined in the specification. The unproved formulas were presented and reviewed. Although valid, the difference for time type would not cause actual problem since the device was intended for 10-20 years of operation but 64-bit representation of the time base would not overflow and "wrap around" in centuries of operation.

13.3. Summary and Evaluation

Successful application of the Echo approach to the verification of MBCS functional correctness gives additional evidence of the Echo approach's applicability and practicality on real systems developed by others. Due to architectural similarity between the specification and the source program for the MBCS, it worked smoothly and efficiently. Two mismatch between the source code and the specification were identified indicating potential defects, although both were discharged upon review for domain specific reasons.

However, not all aspects of MBCS could be verified under the Echo approach, especially for the hardware interface routines. Echo could not be applied, for example, to prove that a loop waiting on a hardware flag indicating the completion of analog-to-digital conversion would terminate in bounded time. Such a proof would require knowledge that

the writes to memory-mapped variables that preceded the blocking loop would cause the hardware to set the flag in question in bounded time. Such information is available as natural-language text in the microcontroller manual, but not in any formal language that can be utilized by Echo. If formal models of the computing hardware behavior exist, the Echo approach can then be extended to related hardware verification.

Floating-point arithmetic also has an impact on the Echo approach and may limit its applicability. Requirement and specifications are often conceived of in terms of real-valued arithmetic, just as the MBCS specification did. However the source program on any target hardware has to use floating-point approximation as no hardware can implement infinite types. In the MBCS case, single-precision floating-point types and arithmetic were used. The Echo approach treats floating-point arithmetic as if it were real-valued arithmetic with a bounded range. For instance, as in the MBCS case study, the single-precision floating-point type in SPARK Ada was extracted from the target configuration file as in PVS:

```
Float: TYPE = {r: real | r >= -3.40282*10^38 AND r <= 3.40282*10^38}
```

This extraction ignored precision by accepting SPARK Ada's approximation and extracted the floating-point type as a bounded real number in PVS. This way it can easily verify the arithmetic calculations at a high level, for instance, whether one variable was used when another was meant or multiplication where addition was meant. It cannot, however, tell whether the adopted floating-point arithmetic is adequately precise for the target software. It has to assume that it is acceptable to adopt the rounding and approximation semantics of the implementation language. By choosing SPARK Ada as the implementation language in the present instantiation of Echo, it adopts and accepts the SPARK Ada's approximation

just as in the MBCS case study. If the assumption is not true, there will be a hole in the verification argument and the Echo approach cannot be applied.

Chapter 14. Application in MBD

My colleagues and I have extend the usage of the Echo approach to verify synthesized software in Model-based development. In this chapter, I briefly present the motivation and basic flow of how Echo is applied in MBD. My colleague, Ren Xu, has demonstrated the feasibility by using Echo to verify synthesized code generated from Simulink models.

14.1. Model-Based Development

Model-based development (MBD) is gaining increasing prominence, especially in domains such as control systems, including aerospace and automotive. In MBD, the creation of software consists of two steps. First, a platform-independent system model is created using a domain-specific modeling language, by which domain experts can design and reason about the system regardless of extraneous target platform and implementation details. Next, code and other target platform artifacts are generated from the model by automatic interpretation and transformation from the model using code synthesis tools. Code synthesis tools are designed to ensure the consistency between the models and the generated implementations. Well-known examples of such synthesis tools in MBD are the SCADE Suite [65] and Simulink [66]. The overall approach is illustrated in Figure 15.

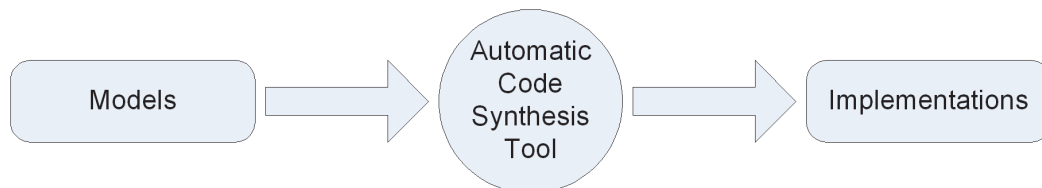


Figure 15. MBD process

With MBD, the usage of system models allows developers to analyze and design a system at the domain model level, leaving all implementation details to be addressed by the code synthesis tools automatically. At any point in the modeling process, the system model can be transformed into certain intermediate programs for direct simulation and testing using automatic tools. Furthermore, various verification techniques can be applied to the system model to ensure it satisfies the requirement, including static analysis and model checking.

14.2. Motivation for Echo Verification in MBD

Despite the advantages of MBD, it also introduces challenges when used in application domains for which the consequences of software failure are high, such as safety-critical applications. In such application domains, there is considerable concern about the possibility of the synthesis tool being defective. Traditional methods of avoiding or eliminating faults that are used when software is built manually, including existing methods of formal verification, cannot be applied because the action of the synthesis system is not subject to inspection or modification.

To ensure the correctness of the resulting software in MBD, one need to: 1) ensure the correctness of the system model, and 2) ensure the correctness of the code synthesis. Many approaches have been developed to certify the system models in MBD including simulation, abstract interpretation, static analysis, model checking, and theorem proving. However, little has been done on the code synthesis process to provide assurance on the synthesized code. The synthesis process relies upon the correctness of the synthesis tool to achieve the necessary quality in the resulting software. However, the state of the art is not sufficient to produce a fully verified code synthesis tool without significant research.

Given that the system model can be checked, but the code synthesis tool cannot, in order to have confidence on the software produced by MBD, one can choose to formally verify the synthesized code against the system model. As traditional formal approaches, such verification can be difficult and tedious to establish. The Echo approach presents a comprehensive alternative. Echo permits formal verification of software built using a synthesis tool. Echo does not depend on actions taken during development, and so it can be applied immediately to synthesized software. The same tools and techniques that are used for software created by hand apply to synthesized software.

14.3. Application of Echo in MBD

Although the same techniques in the Echo approach can be applied directly to verify synthesized software in MBD, it must be expanded to accommodate the languages and notations used by the system model and the code synthesis tool.

The output of synthesis is software in a traditional form, usually a high-level programming language. The Echo approach applies as long as the output code can be annotated (either manually or mechanically), and the annotations can be proved at the code level.

Models in MBD are typically presented in block-based graphical modeling notations. The semantics of graphical notations, albeit formal, cannot be used directly for proof without expressing them in some suitable languages for the proof tools, e.g. theorem prover. In order for Echo to apply, the semantics of the graphical notations must be captured or translated to a specification language such as PVS. Thus the synthesized code can be verified using Echo against the translated specification rather than the model directly. Such translations do exist, e.g. Bensalem et al. defined such a translation structure [10]

which has been adopted by Rockwell Collins Inc. to translate Simulink models into PVS. One might argue defects can also be introduced in such a translation, which is the case. However, since the semantics are on an abstract level, and usually there are only limited number of graphical notations that can be used to construct the model, checking the translations can be done with reasonable time and effort. Once checked, mechanical process can be adopted to automatically derive the specification from the model using the checked translation rules.

Sometimes complete functional correctness verification of the synthesized software is not required, and only certain safety/security properties need to be ensured. In such cases, translating the system model is not required. The subject properties can be defined in a suitable formal language (e.g. PVS), and then be established on the extracted specification when applying the Echo approach on the synthesized software.

Since the modeling notations in MBD are usually block-based and code synthesis tools usually generate code based on those blocks, it is highly likely that the synthesized

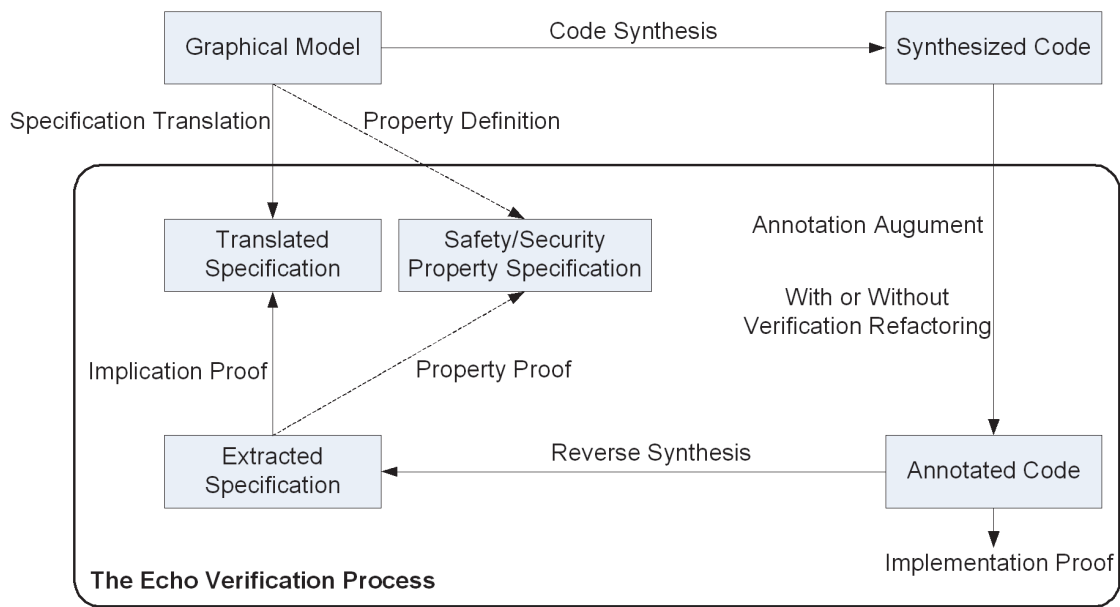


Figure 16. Application of Echo in MBD

code will exhibit similar structure with the model, which satisfies the structural matching hypothesis in Echo. If the synthesized code contains or has been augmented with optimizations (e.g. for efficiency) that cause complexities for verification, verification refactoring in Echo can also be applied before annotations are augmented to the synthesized code.

The expanded flow to apply the Echo approach for verification of synthesized software in MBD is shown in Figure 16.

14.4. Simulink Case Study

Following the general approach outlined in section 14.3, one of my colleagues, Ren Xu, developed necessary tools and techniques, and performed a case study for application of the Echo approach to verify synthesized code from Simulink [81]. Here I briefly present the case study as a demonstration of the feasibility for applying Echo to MBD. More details are discussed in Ren’s work [81].

In Simulink, models are described as graphical block diagrams. Real-Time Workshop (RTW) from MathWorks, Inc. [51] can automatically synthesize C source code for a system described by a given Simulink model. A demo application from MathWorks’ official help documentation was picked for Ren’s case study. The chosen application, albeit small, makes use of various Simulink blocks, feedback controls, and a subsystem. The Simulink model for this demo application is shown in Figure 17. “An 8-bit counter feeds a triggered subsystem parameterized by constants `INC`, `LIMIT`, and `RESET`. The I/O for the model is `Input` and `Output`. The `Amplifier` subsystem amplifies the input signal by gain factor `K`, which is updated whenever signal `equal_to_count` is true” [52].

An executable C program was generated from this model by the Real-Time Workshop code synthesizer. The generated C program contained 11535 non-comment, non-

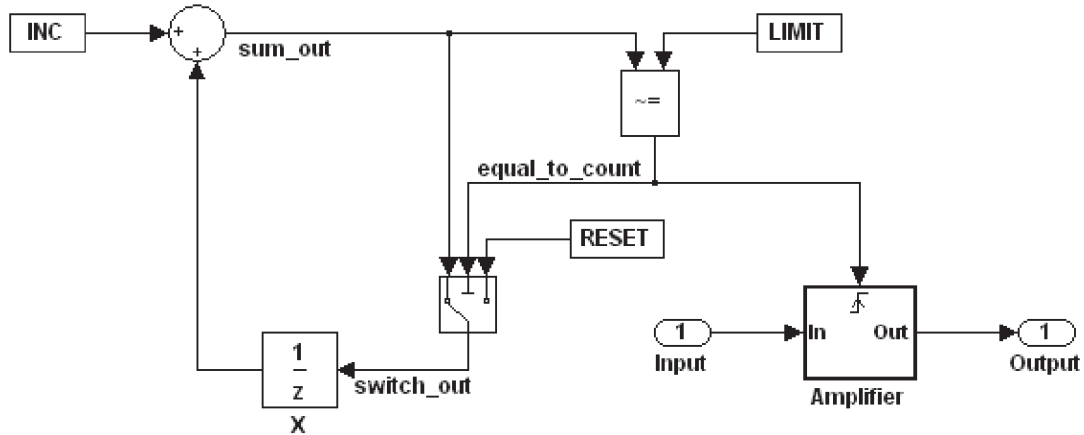


Figure 17. Case study Simulink model

blank links of code. The part of code that was related with the specific model was 847 lines long and was the subject of Ren's case study. The Echo approach was applied to verify full functional correctness of the C program against the original model with the following two augments:

- **Code Translation.** The C program was translated to SPARK Ada and annotated with SPARK annotations, for which the current instantiation of Echo could use. This was done partly by tools and partly by manual intervention. Although this translation was inconvenient, it did not detract from the major goal for this case study, preliminary assessment of feasibility and utility. Direct Ada code synthesis tool for Simulink was also present but we didn't have access when the case study was conducted.
- **Specification Derivation.** A prototype tool that automatically derives PVS specification from Simulink models was developed by Ren. The derived PVS specification for

the case study model was 59 lines long and was used as the original specification in the Echo verification process.

With the derived PVS specification and translated SPARK Ada code, the Echo approach was applied and verified the code implements the specification.

An example property of the model, “the amplifier is triggered for every LIMIT number of execution iterations”, was also defined and stated in PVS, and subsequently proved directly over the Echo extracted PVS specification without invoking the implication proof for full functional correctness verification.

For both the full functional verification and the property proof during the case study, all proofs remained manageable and were completed by a single person in a short amount of time. This showed feasibility of applying Echo to verification of synthesized code in MBD. Again, as stated for Echo’s general applicability, it can verify the functional correctness of the synthesized software. Aspects such as concurrency and real-time attributes are not included.

Chapter 15. Related Work

There is an extensive amount of research on formal verification in the literature. I summarize the most relevant work to the Echo approach in this chapter.

15.1. General Verification Approaches

Light-weight program analyses [22] are often used to find bugs in or gain confidence about programs. Compared to more complete formal verification, their expressive power is limited and no formal proof of compliance is produced. Heavier-weight techniques like the B method [2] are more suited to full formal verification, but they intertwine code production and verification. Using the B method requires a B specification and then enforces a lock-step code production approach on developers.

A more general technique is traditional Floyd-Hoare verification [28]. Unfortunately, it requires generation and proof of many detailed lemmas and theorems. It is very hard to automate and requires significant time and skill to complete. Annotations and verification condition generation, such as that employed by the SPARK Ada toolset [7], is used in practice. However, the annotations used by SPARK Ada (and other similar techniques) are generally too close to the abstraction level of the program to encode higher-level specification properties. Thus, the Echo approach uses verification condition generation as an intermediate step.

Other techniques are available for the properties that I do not address with the Echo approach. Model checking techniques [37], for example, have been quite successful at verifying hardware, protocols and temporal properties; they complement the Echo

approach in such areas. While model checking can generate proofs that the software model adheres to the specification, it does not prove that the software model is faithful to the original program. More recent model extraction [37], aims to address this problem and mechanically extracts a system model from the source code so that model checking can be applied. However, model extraction does not produce a full assurance argument since model checking is not targeted at full functional correctness.

15.2. More Related Approaches

Tudor et al. have developed a largely automatic verification process called witnessing analysis for code automatically generated from Simulink [74]. Simulink is a tool used to specify control laws that can automatically generate Ada source code that implements the control law specification. The Ada code generator for Simulink is not trusted to ultra-dependable levels, however, and so verification of the Ada code is necessary in critical systems. Tudor et al. used a toolset with manual assistance to produce a formal specification in Z from Simulink, compared it with a Z specification extracted from the Ada code generated by Simulink, and then produced and proved verification conditions for the compliance argument between the two. The Echo approach is similar to theirs, but it looks at verification in a more general way. I characterize classes of languages to which Echo can apply, and define a verification process that is otherwise language-independent. Furthermore, by incorporating the annotations, some of the implementation details can be abstracted away so that the specification extraction process can more easily capture only the properties that are relevant to the verification. Also, the reverse synthesis technique can help reduce the verification complexity introduced by program optimizations.

Andronick et al. developed an approach to verification of various security properties of imperative source code embedded on smart cards [5]. Similar to reverse synthesis, they proved a C source program against supplementary annotations and generated a high-level formal model of the annotated C program that was used to verify certain global security properties. The proof of the annotated C program and the validation of the model generation were done formally using the Caduceus tool. Other than the language difference, their approach focuses on proving the security properties on top of the generated model, while the Echo approach allows showing not only properties on the source program as one of its extended usage, but also broad compliance with the original specification from which the system was built by introducing the implication proof. Also the reverse synthesis process in Echo incorporates verification refactoring to reduce the complexity of proof involved. Their approach, however, shows the potential to expand the applicability of the Echo approach to more languages as long as proper tools are available.

Heitmeyer et al. also developed a similar approach for verifying a system's high-level security properties [34]. They partition the code and construct a compact security model from the code containing only information needed to reason about the security properties of interest. The security properties are then represented and proved formally in terms of the model. The approach was developed to support a common criteria evaluation of the separation kernel of an embedded software system. Again, their approach is focused on verifying security properties only, whereas the Echo approach incorporates annotation and refactoring to increase abstraction and reduce complexity, and is aimed at general functionality of the target software.

15.3. Hardware Verification

Engineers developing digital circuits face the verification challenge also. Digital circuits that are relatively simple are verified by testing. Increasingly, however, circuits have become so complex that testing is extremely time consuming and relatively ineffective, and so formal verification has been pursued.

Various techniques have been developed for the verification of digital hardware, up to and including complete microprocessors. The FM9001, for example, was specified and its gate-level implementation verified using a variety of formal techniques [41]. Especially relevant to the proposed research is the work of Hoskote and Abraham [38, 39]. In this work, formal verification is applied in a comprehensive and practical way to digital circuits. Of particular interest is the use of an automatic extraction process in which a low-level specification of a circuit is examined and a high-level specification created. This is done primarily by developing an abstract model of the low-level specification doing so by exploiting various characteristics of digital circuits. The process is human guided, in part, because a key aspect of the approach is to eliminate data registers from consideration. Another relevant research is Kuehlmann and Baumgartner's transformation-based verification [9, 46] for sequential verification of circuit-based designs. Their work uses structural transformation that relocates registers in a circuit-based design representation without changing its actual input-output behavior, to increase the capacity of symbolic state traversal, and thus improve reachability analysis and the verification of temporal properties. These techniques are restricted forms of the reverse synthesis technique in the Echo approach, which adopts similar idea to transform the target being verified, but for software.

15.4. Reverse Engineering

Ward et al. have developed a method called inverse engineering and an associated tool for reverse engineering, by which a high-level specification can be retrieved from low-level source code of a program by a process of semantic-preserving program transformations [76, 78]. In their approach, they translated a program into their internal WSL language, applied a series of program transformations to it under user guidance, until it was sufficiently abstract that it could be translated it back in to a specification. Each transformation was proved to produce a semantically equivalent result based on a theory of program refinement and transformation [77]. Similar approaches are also adopted by many others in the reverse engineering domain. Chung et al., for example, also retrieved high-level specifications from the source code by abstractions [16].

These approaches are very similar to the reverse synthesis process in Echo. However, the criteria are not the same. Their goal is to support software maintenance, to make the unstructured or poorly engineered source code amenable to further analysis. Thus they don't have a specific target form to transform it into. The reverse synthesis process in Echo aims to aid verification, specifically to reverse the complexities caused by program optimization and to match the structure of the original specification so that the implication theorem can be proved. Moreover, by incorporating the intermediate annotations, the Echo approach can more easily capture the properties relevant to verification while still abstracting implementation details. These techniques do, however, show the feasibility of approaches similar to reverse synthesis.

15.5. Automatic Code Generation

Automated translation, or code generation [79], of a formal specification to an implementation provides an alternative to verification. This approach constructs an implementation automatically from the specification using formal translation rules. Compliance of the implementation with its specification is implied by the translation rules, as long as the rules preserve specification semantics. If the translation rules are correct and the translator implements them correctly, it guarantees that the behavior of the implementation is consistent with the formal specification.

Automatic code generation is gaining increasing prominence under the name model-based development. Tools are built such as Simulink [66] and the SCADE Suite [65]. However, its success at present is primarily confined to narrow domains such as control systems. For most safety-critical systems, it is very difficult to automatically generate a well-structured implementation from a formal specification. Furthermore, automatic code generation sometimes yields an implementation lacking properties, such as efficiency. This lack leads to the same problem as verification based on refinement, any change to the generated code (e.g. to improve efficiency) invalidates the verification argument. Finally, even if automatic code generation is possible, the generator itself needs to be verified, and the state of the art is not sufficient to produce a verified code generator without significant research.

15.6. Other Related Work

Paul et al. [58] are developing an approach to the determination of how refactorings affect the verifiability of a program. Their focus is object-oriented design, and the goal is to see whether a syntactic change can make more properties amenable to analysis.

Smith et al. [67] have developed an infrastructure for verifying block ciphers, including AES, and they have verified AES implementations in Java byte code. They noted different representations between the specification and the implementation, and provided transformation functions between the two. Their work, however, takes advantage of many properties of block ciphers and is tied specifically to such verification.

Klein et al. demonstrated that full functional formal verification is practical for large systems by verifying the seL4 microkernel from an abstract specification down to its 8700 lines C implementation [42]. With proof by parts, the Echo approach is more widely applicable and does not impose restrictions on the development process.

Chapter 16. Conclusion

In this chapter, I conclude the work and summarize the contributions, limitations, and future work on the Echo approach.

16.1. Conclusion

In this thesis, I have defined the Echo approach, a verification technique based upon the use of an intermediate point of abstraction between a high-level formal specification and its concrete implementation. This intermediate point is a low-level specification documented by annotated source code. The verification approach shows that the source code correctly implements the annotations and that the annotated source code implies the high-level specification.

I have introduced the new technique of reverse synthesis that mechanically creates a high-level specification from the source program documented with the low-level specification. A crucial component of reverse synthesis is the application of semantics-preserving refactoring transformations to reduce verification complexity.

I have also introduced a proof structure in which the major proof is carried out by matching static specification structures, and organizing the proof as the conjunction of a series of lemmas about the specification structure, in order for the proof to scale.

Human insight guides much of the process in Echo, but the analysis and thus the verification is either automatic or machine-checkable. In effect, the verification proceeds in a direction opposite to that of traditional verification approaches, exploiting automated reasoning and program transformation to increase the practicality of verification. It dove-

tails directly with traditional development processes and artifacts. Many existing development methods can continue to be used, yet formal verification and all its benefits can be applied.

My goals with Echo are to give developers greater freedom to use existing software development methods in creating software systems, and to exploit available verification tools and techniques to create a practical and well-controlled verification process. I claim that the Echo approach is a practical yet comprehensive approach to formal verification of functional correctness, and makes such verification readily available, applicable, cost effective, and useful to the community that needs it.

16.2. Contributions

The claim made in the preceding section are supported by the following specific contributions:

- A new approach to verification in which analysis is partitioned by adopting a low-level specification as an intermediate point. Each partitioned proof operates on different abstraction level and can be tackled with separate specialized techniques to alleviate the proof difficulty. The major proof step is pushed to the abstract level to reduce the verification effort.
- A new technique, reverse synthesis, and effective reverse synthesis algorithms by which semantic-preserving transformations are performed to reduce verification complexity, and high-level specifications are extracted from low-level specifications and implementations mechanically to support the verification. Reverse synthesis bridges the gap between the partitioned proofs and enables developers to continue to use exist-

ing software development methods, i.e., they are not limited solely to tools and processes that support verification.

- A crucial component of reverse synthesis, verification refactoring, by which complexity-reducing but semantics-preserving refactoring transformations are applied to the source program to facilitate both proofs. In general, it is easier to transform the program than to transform the proof. Thus, transformations facilitate verification by reducing the complexity of the source program and thereby the proof obligation. Verification refactoring deals with many of the issues that limit the applicability of formal verification. In particular, it overcomes the problem of unworkably large verification conditions and frees developers from the rigid development process necessary for refinement.
- Heuristics, templates, and complexity metrics to guide the selection and application of transformations in the verification refactoring process, so that one can determine which transformations can be selected to reduce complexity and when the refactored program is likely to be amenable to proof.
- A scalable proof structure, proof by parts, in which the major proof is carried out by matching static specification and implementation structures. The proof can then be organized as the conjunction of a series of lemmas about the matched structure. By setting up a different lemma for each distinct element and proving each lemma independently, it facilitates the proof to scale for verification of large systems.
- A controlled verification process that integrates a number of powerful existing notations, tools and processes to exploit existing verification capabilities, that links the gap

of existing techniques by author developed tools and processes, that automates the process to the extent possible.

- Extended usages that combines with other types of analysis such as run-time checks in the synergistic analysis framework to allow richer properties to be constructed in whole-system assurance arguments, that can benefit verification of synthesized code in model-based development, and that allows safety/security properties be established without carrying out full functional verification.
- Case studies that assessed and evaluated applicability and practicability of Echo by verifying three specimen systems in the safety and security domain.

16.3. Limitations

Although the Echo approach provides certain benefits over existing techniques, it is in no way a verification “silver bullet”. As with any formal verification technique, it requires the use of formal languages, various analytic tools including a theorem-proving system, and considerable skill on the part of the developer. One specific additional responsibility placed on the developer is to annotate the source code with pre- and post-condition documentation.

Also the use of annotations, though not uncommon in modern software development, introduces a source of defects over and above those that might be present in the program.

The Echo approach only targets functional correctness verification. Real-time properties are not included. Especially when applying verification refactoring, the subject program is changed and no verification of real-time properties is possible.

When floating-point arithmetic is present in the subject program, the Echo approach can check the functionality as if it is real-valued arithmetic. However it cannot check whether the adopted floating-point arithmetic is adequately precise or not. It has to accept the implementation language's approximation on the target platform.

Finally, the successful use of the Echo approach relies upon the use of notations with formal semantic definitions and suitable verification tools. Defects in the tools used, can also break the verification argument.

16.4. Future Work

The time and resources available in a single doctoral program is limited. More enhancement of the Echo approach is left to future work, which contains but is not limited to:

- Expand the languages and notations supported by Echo;
- Introduce and prove more transformations that facilitates verification refactoring;
- Add machine support to automatically generate annotations to the extent possible;
- Add machine support to automatically infer applicable verification refactoring transformations and locate the spot to apply them;
- Certify the specification extraction process and tool;
- Encapsulate all tools and processes involved in a single console that checks and controls the verification steps in Echo to provide ease of use;
- Integrate with techniques that check other types of properties, such as model checking, to provide more comprehensive assurance argument.

The main thesis of the work concerns the feasibility and practicality of the Echo approach, which was evaluated through a number of case studies for verification of three selected specimen systems. The case studies were conducted by the author of the Echo approach, and were restricted by time and resource constraints to a single developer. More evaluations of the Echo approach, e.g. efficacy and utility comparison with other verification techniques, application by different developers or to different systems and domains, are left for future work.

Bibliography

- [1] Abrial, J. R., *Formal methods in industry: achievements, problems, future*, Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, pp. 761-768, 2006.
- [2] Abrial, J. R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
- [3] Adacore, *GNAT Metric Tool*, <http://www.adacore.com>.
- [4] Anderson, T., and R. W. Witty, *Safe Programming*, BIT Vol.18, 1978, pp. 1-8.
- [5] Andronick, J., B. Chetali, and C. Paulin-Mohring, *Formal Verification of Security Properties of Smart Card Embedded Source Code*, In: Fitzgerald, J., Hayes, I. J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 302-317. Springer-Verlag, 2005.
- [6] Banach, R. and M. Poppleton, *Retrenchment: An Engineering Variation on Refinement*, Proceedings of B-98, Bert (ed.), LNCS 1393, 129-147, Springer, 1998.
- [7] Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.
- [8] Barnes, J. and R. Chapman, *Engineering the Tokeneer Enclave Protection Software*, International Symposium on Secure Software Engineering (ISSSE'06), IEEE, 2006.
- [9] Baumgartner, J., A. Kuehlmann, and J. Abraham, *Property checking via structural analysis*, Computer-Aided Verification, July 2002.
- [10] Bensalem, S., P. Caspi, C. P. Vigouroux, and C. D. Canovas, *A methodology for proving control systems with Lustre and PVS*, Seventh Working Conference on Dependable Computing for Critical Applications (DCCA7), San Jose, January 1999.
- [11] Boldo, S. and J. Filliatre, *Formal Verification of Floating-Point Programs*, Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH '07), IEEE, Washington, DC, USA, 2007.
- [12] Bowen, J. P. and V. Stavridou, *Safety-Critical Systems, Formal Methods and Standards*, IEE/BCS Software Engineering Journal, 8(4):189-209, July 1993.
- [13] Bravenboer, M., K. T. Kalleberg, R. Vermaas, and E. Visser, *Stratego/XT 0.16. A Language and Toolset for Program Transformation*, Science of Computer Programming, 2007.

- [14] Burdy, L., Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, *An overview of JML tools and applications*, International Journal on Software Tools for Technology Transfer, 7(3):212-232 (2005).
- [15] Butler, R and G. Finelli, *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*, IEEE Transactions on Software Engineering, Vol. 19, No 1, January 1993.
- [16] Chung, B. and G. C. Gannod, *Abstraction of Formal Specifications from Program Code*, In: IEEE 3rd International Conference on Tools for Artificial Intelligence, pp. 125-128 (1991).
- [17] Cooper, D., and J. Barnes, *Tokeneer ID station: EAL5 demonstrator: Summary report*, August 2008.
- [18] Cousot, P. and R. Cousot, *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation*, Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming, pp. 269-295 (1992).
- [19] Croxford, M. and R. Chapman, *Correctness by construction: A manifesto for high-integrity software*, CrossTalk, The Journal of Defense Software Engineering, 2005, pp. 5-8.
- [20] Daemen, J. and V. Rijmen, *AES Proposal: Rijndael. AES Algorithm Submission*, 1999.
- [21] Das, M., *Formal Specifications on Industrial Strength Code: From Myth to Reality*, Computer-Aided Verification 2006, Seattle WA (August 2006).
- [22] Das, M., S. Lerner, and M. Seigle, *ESP: path-sensitive program verification in polynomial time*, Programming Languages, Design and Implementation, pp. 57-68 (2002).
- [23] Dijkstra, E. W., *Guarded commands, nondeterminacy and formal derivation of programs*, Communications of the ACM, 18(8):453-457, August 1975.
- [24] Elder, M. C., *Specification of User Interfaces for Safety-Critical Systems*, M.S. Thesis, Department of Computer Science, University of Virginia, August 1995.
- [25] Evans, D. and D. Larochelle, *Improving Security Using Extensible Lightweight Static Analysis*, IEEE Software, Jan/Feb 2002.
- [26] FIPS PUB 197, *Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, November 2001.
- [27] Flanagan, C. and K. Leno, *Houdini, an annotation assistant for ESC/Java*, Formal Methods Europe, Berlin, Germany (2001).

- [28] Floyd, R. W., *Assigning meanings to programs*, in Schwartz, J.T. (ed.), *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19* (American Mathematical Society), Providence, pp. 19-32, 1967.
- [29] Gagnon, E. M. and L. J. Hendren, *SableCC, an Object-Oriented Compiler Framework*, TOOLS (26), pp. 140-154, IEEE Computer Society, 1998.
- [30] Graydon P., and J. Knight, *Software Process Synthesis in Assurance Based Development of Dependable Systems*, Proceedings of the 8th European Dependable Computing Conference (EDCC), Valencia, Spain, 2010.
- [31] Graydon, P., J. Knight, and X. Yin, *Practical Limits on Software Dependability: A Case Study*, Proceedings of the 15th International Conference on Reliable Software Technologies (Ada-Europe), Valencia, Spain, 2010.
- [32] Hall, A. and R. Chapman, *Correctness by Construction: Developing a Commercial Secure System*, IEEE Software, 19(1), pp. 18-25, 2002.
- [33] Hayhurst, K. J., D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, *A Practical Tutorial on Modified Condition/Decision Coverage*, NASA/TM-2001-210876, NASA Langley Research Center, Hampton, Virginia, May 2001.
- [34] Heitmeyer, C., M. Archer, E. Leonard, and J. McLean, *Applying Formal Methods to a Certifiably Secure Software System*, IEEE Transaction on Software Engineering, Vol.34, No.1, 2008.
- [35] Henzinger, T., R. Jhala, R. Majumdar, G. Nacula, G. Sutre, and W. Weimer, *Temporal-Safety Proofs for Systems Code*, Lecture Notes in Computer Science, Volume 2404, Jan 2002, Pages 526 - 538.
- [36] Hoare, C. A. R., *An axiomatic basis for computer programming*, Communications of the ACM, 12(10):576–585, October 1969.
- [37] Holzmann, G. J., *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004.
- [38] Hoskote, Y., *Formal Techniques for Verification of Synchronous Sequential Circuits*, Ph.D. Dissertation, The University of Texas at Austin, December 1995.
- [39] Hoskote, Y., J. Abraham, D. Fussell and J. Moondanos, *Automatic Verification of Implementations of Large Circuits Against HDL Specifications*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 16, March 1997, pp. 217-228.
- [40] Howden, W. E., *Reliability of the Path Analysis Testing Strategy*, IEEE Transactions on Software Engineering 2(3):208-215, 1976.

- [41] Hunt, W. and B. Brock, *A Formal HDL and Its Use in the FM9001 Verification*, in “Mechanized Reasoning and Hardware Design”, C. Hoare and M. Gordon, eds., Prentice Hall, 35-47, 1992. First published in “Philosophical Transactions of the Royal Society of London”, Series A, Vol. 339, 1992.
- [42] Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, *seL4: formal verification of an OS kernel*, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, October, 2009.
- [43] Knight, J. C., and P. E. Ammann, *An Experimental Evaluation of Simple Methods For Seeding Program Errors*, ICSE-8: Eighth International Conference on Software Engineering, London, UK, 1985.
- [44] Knight, J. C., C. L. DeJong, M. S. Gobble, and L. G. Nakano, *Why Are Formal Methods Not Used More Widely?*, Fourth NASA Langley Formal Methods Workshop, Hampton, VA, September 1997.
- [45] Kataoka, Y., M. Ernst, W. Griswold, and D. Notkin, *Automated support for program refactoring using invariants*, International Conference on Software Maintenance, pp. 736-743 (2001).
- [46] Kuehlmann, A. and J. Baumgartner, *Transformation-based Verification using generalized retiming*, Computer-Aided Verification, July 2001.
- [47] Leavens, G. T. and Y. Cheon, *Design by Contract with JML* (Draft), <http://jmlspecs.org>, 2006.
- [48] Lerner, S., T. Millstein, E. Rice and C. Chambers, *Automated soundness proofs for dataflow analyses and transformations via local rules*, Principles of Programming Languages, pp. 364-377 (2005).
- [49] Linger, R. C., H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.
- [50] Liskov, B. and J. Wing, *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, 16(6):1811--1841, November 1994.
- [51] MathWorks Inc., *Real-Time Workshop*, <http://www.mathworks.com/products/rtw/>.
- [52] MathWorks Inc., *Generating code using the Real-Time Workshop product*, Real-Time Workshop Demos, Published with MATLAB 7.8, 2009.
- [53] Meyer B., *Applying "Design by Contract"*, IEEE Computer, 25(10), pp. 40-51, 1992.

- [54] Myers, W., *Can software for the strategic defense initiative ever be error-free?*, IEEE Computer, November 1986, 19, (11).
- [55] National Institute of Standards Technology, *The Common Criteria Evaluation and Validation Scheme*, <http://niap.nist.gov/cc-scheme/index.html>.
- [56] National Institute of Standards Technology, *Cryptographic Toolkit*, <http://csrc.nist.gov/CryptoToolkit/>.
- [57] Owre, S., N. Shankar, and J. M. Rushby, *PVS: A Prototype Verification System*, CADE 11, Saratoga Springs, NY, June 1992.
- [58] Paul, J., N. Kuzmina, R. Gamboa, and J. Caldwell, *Toward a Formal Evaluation of Refactorings*, Proceedings of The Sixth NASA Langley Formal Methods Workshop, 2008.
- [59] Potter, B., J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, Prentice Hall: London, 1996.
- [60] Praxis, *The SPARK Ravenscar Profile*, <http://praxis-his.com>, 2006.
- [61] Rahul, S. and G. Necula, *Proof Optimization Using Lemma Extraction*, Technical Report UCB/CSD-01-1143, University of California, Berkeley, May 2001.
- [62] Robby, M., B. Dwyer, and J. Hatchliff, *Bogor: An extensible and highly-modular model checking framework*, Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 267–276, 2003.
- [63] Runeson, J., S. Nystrom, and J. Sjodin, *Optimizing code size through procedural abstraction*, Languages, Compilers and Tools for Embedded Systems, pp. 204-215 (2000).
- [64] Rustan, K., M. Leino, G. Nelson, and J. B. Saxe, *ESC/Java User's Manual*, Technical Note 2000-002, Compaq Systems Research Center, Palo Alto, CA, October 2000.
- [65] SCADE Suite, Esterel Technologies, <http://www.esterel-technologies.com/>.
- [66] Simulink - Simulation and Model-Based Design, <http://www.mathworks.com/products/simulink/>.
- [67] Smith, E. and D. Dill, *Formal Verification of Block Ciphers, A Case Study: The Advanced Encryption Standard (AES)*, Stanford University.
- [68] Spivey, J. M., *The Z Notation: A Reference Manual*, Prentice-Hall, 1992.
- [69] Storey, N., *Safety Critical Computer Systems*, Addison-Wesley, 1996.

- [70] Strunk, E. A., J. C. Knight, and M. A. Aiello, *Assured Reconfiguration of Fail-Stop Systems*, The International Conference on Dependable Systems and Networks, Yokohama, Japan, June 2005.
- [71] Strunk, E. A., X. Yin, and J. C. Knight, *Echo: A Practical Approach to Formal Verification*, Tenth International Workshop on Formal Methods for Industrial Critical Systems, co-located with ESEC/FSE '05, Lisbon, Portugal, September 2005.
- [72] Throckmorton, A., A. Untaroiu, P. Allaire, H. Wood, D. Lim, M. McCulloch, and D. Olsen, *Numerical design and experimental hydraulic testing of an axial flow ventricular assist device for infants and children*, American Society for Artificial Internal Organs (ASAIO) Journal, 53(6):754–761, November–December 2007.
- [73] Throckmorton, A. L., A. Untaroiu, D. S. Lim, H. G. Wood, and P. E. Allaire, *Fluid force predictions and experimental measurements for a magnetically levitated pediatric ventricular assist device*, Journal of Artificial Organs, 31(5):359–368, 2007.
- [74] Tudor, N., M. Adams, P. Clayton, and C. O'Halloran, *Auto-Coding/Auto-Proving Flight Control Software*, 23rd Digital Avionics Systems Conference, October 2004.
- [75] Untaroiu, A., H. Wood, and P. Allaire, *Implantable axial-flow blood pump for left ventricular support*, Proceedings of the 45th International ISA Biomedical Sciences Instrumentation Symposium, Copper Mountain, CO, April 2008.
- [76] Ward, M., *Abstracting a Specification from Code*, Journal of Software Maintenance: Research and Practice, Vol 5, pp. 101-122, 1993.
- [77] Ward, M., *Proving Program Refinements and Transformations*, DPhil Thesis, Oxford University, 1989.
- [78] Ward, M., *Reverse Engineering through Formal Transformation*, The Computer Journal, Vol 37, No 9, pp. 795-813, 1994.
- [79] Whalen, M. and M. Heimdahl, *An Approach to Automatic Code Generation for Safety-Critical Systems*, Proceedings of the 14th IEEE International Conference on Automated Software Engineering, October 1999.
- [80] Woodcock, J. and R. Banach, *The Verification Grand Challenge*, Journal of Universal Computer Science, Vol. 13, No. 5, 2007, pp. 661-668.
- [81] Xu, R., *Using Echo to Verify Implementation in Model-Driven Development*, Master's Thesis, University of Virginia, May 2010.
- [82] Yin, R., *Case Study Research: Design and Methods*, SAGE Publications: Thousand Oaks, 2003.

- [83] Yin, X., J. C. Knight, E. A. Nguyen, and W. Weimer, *Formal Verification By Reverse Synthesis*, SAFECOMP 2008: The 27th International Conference on Computer Safety, Reliability and Security Newcastle, UK, September 2008.
- [84] Yin, X., J. C. Knight, and W. Weimer, *Exploiting Refactoring in Formal Verification*, DSN 2009: The International Symposium on Dependable Systems and Networks Lisbon, Portugal, June 2009.
- [85] Yin, X. and J. C. Knight, *Formal Verification of Large Software Systems*, NFM 2010: Second NASA Formal Methods Symposium, Washington DC, April 2010.

Appendix A: Example Specification Extraction and Implication Theorem

A.1 Original Specification of MBCS

```

%%=====
%%                                     Target Configuration                                     %%
%%=====

vad_cntrl_conf: THEORY
BEGIN

value_type: TYPE = real

dac_value_type: TYPE = {i: int | i >= 0 AND i < 4096} CONTAINING 0
adc_value_type: TYPE = {i: int | i >= 0 AND i < 4096} CONTAINING 0

input_size: posnat = 4
output_size: posnat = 6
state_size: posnat = 16

time_type: TYPE = nat
coil_failure_type: TYPE = {f: int | f = 0 OR f = 1} CONTAINING 0
input_vector_type: TYPE = [below(input_size) -> adc_value_type]
output_vector_type: TYPE = [below(output_size) -> dac_value_type]

END vad_cntrl_conf

%%=====
%%                                     Math Utilities                                     %%
%%=====

vad_cntrl_vector[N: posnat]: THEORY
BEGIN
IMPORTING vad_cntrl_conf

vector: TYPE = [below(N) -> value_type];

+(a, b: vector): vector = LAMBDA (i: below(N)): a(i) + b(i)

% Sigma f(i), i from low to high
sigma(low, high: below(N), f: vector): RECURSIVE value_type =
  IF low > high THEN 0
  ELSIF low = high THEN f(low)
  ELSE f(low) + sigma(low+1, high, f)
  ENDIF
  MEASURE abs(high-low)

END vad_cntrl_vector

vad_cntrl_matrix[M: posnat, N: posnat]: THEORY
BEGIN
IMPORTING vad_cntrl_vector

matrix: TYPE = [below(M), below(N) -> value_type];

```

```

*(m: matrix, v: vector[N]): vector[M] =
  LAMBDA (i: below(M)): sigma(0, N-1, LAMBDA (j: below(N)): m(i,j)*v(j))

END vad_cntrl_matrix

%%=====%%
%%                               Control Software                               %%
%%=====%%

vad_cntrl_sw: THEORY
BEGIN
IMPORTING vad_cntrl_matrix

%%----- Types & Constants -----%%

u_vector_type: TYPE = vector[input_size]
c_vector_type: TYPE = vector[output_size]
x_vector_type: TYPE = vector[state_size]

hws_val_type: TYPE = {Op, NotOp}
hw_status_type: TYPE = [below(6) -> hws_val_type]
hws_AllOp: hw_status_type = LAMBDA (i: below(6)): Op

%% cv_value_type: TYPE = {
%%   cv_normal, cv_1u, cv_1v, cv_1w, cv_2u, cv_2v, cv_2w, cv_uu,
%%   cv_uv, cv_uw, cv_vu, cv_vv, cv_vw, cv_wu, cv_wv, cv_ww}

%% for prototype only:
cv_value_type: TYPE = {cv_normal, cv_1u}

cvf(hws: hw_status_type): cv_value_type =
  CASES hws(0) OF
    Op:    cv_normal,
    NotOp: cv_1u
  ENDCASES

A(cv: cv_value_type): matrix[state_size, state_size] =
  LAMBDA (i: below(state_size), j: below(state_size)): 0

B(cv: cv_value_type): matrix[state_size, input_size] =
  CASES cv OF
    cv_normal: LAMBDA (i: below(state_size), j: below(input_size)):
      IF i = j THEN 1
      ELSE 0
      ENDIF,
    cv_1u:    LAMBDA (i: below(state_size), j: below(input_size)): 0
  ENDCASES

D(cv: cv_value_type): matrix[output_size, state_size] =
  CASES cv OF
    cv_normal: LAMBDA (i: below(output_size), j: below(state_size)):
      COND
        i = 0 AND j = 0 -> 12,
        i = 1 AND j = 0 -> -6,

```

```

        i = 1 AND j = 1 -> 12,
        i = 2 AND j = 0 -> -6,
        i = 2 AND j = 1 -> -12,
        i = 3 AND j = 2 -> 12,
        i = 4 AND j = 2 -> -6,
        i = 4 AND j = 3 -> 12,
        i = 5 AND j = 2 -> -6,
        i = 5 AND j = 3 -> -12,
        ELSE
            -> 0
        ENDCOND,
    cv_1u:      LAMBDA (i: below(output_size), j: below(state_size)): 0
ENDCASES

E(cv: cv_value_type): matrix[output_size, input_size] =
CASES cv OF
    cv_normal: LAMBDA (i: below(output_size), j: below(input_size)):
        COND
            i = 0 AND j = 0 -> 16,
            i = 1 AND j = 0 -> -8,
            i = 1 AND j = 1 -> 16,
            i = 2 AND j = 0 -> -8,
            i = 2 AND j = 1 -> -16,
            i = 3 AND j = 2 -> 16,
            i = 4 AND j = 2 -> -8,
            i = 4 AND j = 3 -> 16,
            i = 5 AND j = 2 -> -8,
            i = 5 AND j = 3 -> -16,
            ELSE
                -> 0
            ENDCOND,
    cv_1u:      LAMBDA (i: below(output_size), j: below(input_size)): 0
ENDCASES

frame_period: time_type = 25600

%%----- System State -----%%

hardware_type: TYPE =
[# time_base:      time_type,
  coil_failure: coil_failure_type
  % QADC
  % DAC
  % ...
#]

internal_state_type: TYPE =
[# input:          input_vector_type,
  output:          output_vector_type,
  last_frame:      time_type,
  next_frame:      time_type,
  hw_status:       hw_status_type,
  last_hws:        hw_status_type,
  cv:              cv_value_type,
  u:               u_vector_type,
  c:               c_vector_type,

```

```

x:          x_vector_type #]

system_status_type: TYPE =
  [# hw: hardware_type,
   st: internal_state_type #]

%%----- Control Calculation -----%%

adc_scale:  value_type = 0.000625
adc_offset: value_type = -1
input_to_u(input: input_vector_type): u_vector_type =
  LAMBDA (i: below(input_size)):
    (input(i) * adc_scale) + adc_offset

dac_scale:  value_type = 1024
dac_offset: value_type = 2
c_to_output(c: c_vector_type): output_vector_type =
  LAMBDA (i: below(output_size)):
    LET output_i = (c(i) + dac_offset) * dac_scale IN
    COND
      output_i > 4095          -> 4095,
      output_i >= 0 AND output_i <= 4095 -> floor(output_i + 0.5),
      output_i < 0            -> 0
    ENDCOND

init(st: internal_state_type): internal_state_type =
  st WITH [ `hw_status := hws_AllOp,
            `last_hws := hws_AllOp,
            `cv := cvf(hws_AllOp),
            `u := LAMBDA (i: below(input_size)): 0,
            `c := LAMBDA (i: below(output_size)): 0,
            `x := LAMBDA (i: below(state_size)): 0]

convert_input(st: internal_state_type): internal_state_type =
  st WITH [ `u := input_to_u(st`input)]

compute_output(st: internal_state_type): internal_state_type =
  st WITH [ `c := D(st`cv) * st`x + E(st`cv) * st`u]

compute_new_state(st: internal_state_type): internal_state_type =
  st WITH [ `x := A(st`cv) * st`x + B(st`cv) * st`u]

convert_output(st: internal_state_type): internal_state_type =
  st WITH [ `output := c_to_output(st`c)]

calculate_control(st: internal_state_type): internal_state_type =
  IF st`hw_status = st`last_hws THEN
    convert_output(compute_new_state(compute_output(convert_input(st))))
  ELSE
    st WITH [ `last_hws := st`hw_status,
              `cv := cvf(st`hw_status),
              `c := LAMBDA (i: below(output_size)): 0,
              `x := LAMBDA (i: below(state_size)): 0,
              `output := c_to_output(LAMBDA (i: below(output_size)): 0)]

```

```

ENDIF

%%----- Hardware Interface -----%%

%% Hardware interface is not modeled here.

%% Assume hardware is correctly configured,
%% st`time_base, st`coil_failure, st`input are correctly read,
%% and st`output is correctly wrote.

%% Bogus interfaces only
%% to simulate the event sequence in the control frame

read_QADC(s: system_status_type): system_status_type
update_DAC(s: system_status_type): system_status_type

convert_status_post(s_: system_status_type): bool =
  IF s`hw`coil_failure = 0 THEN
    s`st = s_`st WITH [ `hw_status := hws_AllOp WITH [(0) := NotOp]]
  ELSE
    s`st = s_`st WITH [ `hw_status := hws_AllOp]
  ENDIF

convert_status(s_: system_status_type): system_status_type =
  choose({s: system_status_type | convert_status_post(s_, s)})

%%----- Cyclic Executive -----%%

frame_init_post(s_, s: system_status_type): bool =
  s`st = s_`st WITH [ `last_frame := s`hw`time_base,
    `next_frame := s`hw`time_base + frame_period]

frame_init(s_: system_status_type): system_status_type =
  choose({s: system_status_type | frame_init_post(s_, s)})

frame_sync_post(s_, s: system_status_type): bool =
%% block until next frame
  s`hw`time_base >= s_`st`next_frame AND
  s`st = s_`st WITH [ `last_frame := s_`st`next_frame,
    `next_frame := s_`st`next_frame + frame_period]

frame_sync(s_: system_status_type): system_status_type =
  choose({s: system_status_type | frame_sync_post(s_, s)})

calculate_control_post(s_, s: system_status_type): bool =
  s`st = calculate_control(s_`st)

calculate_control(s_: system_status_type): system_status_type =
  choose({s: system_status_type | calculate_control_post(s_, s)})

control_frame(s: system_status_type): system_status_type =

update_DAC(calculate_control(convert_status(read_QADC(frame_sync(s)))))

```

```
%% Main loop for control frame is not modeled here.
```

```
END vad_cntrl_sw
```


A.2 Extracted Specification of MBCS

```
Standard: THEORY
BEGIN
```

```
Integer: TYPE = {n: int | n >= -2147483648 AND n <= 2147483647}
Float: TYPE = {r: real | r >= -3.40282*10^38 AND r <= 3.40282*10^38}
```

```
END Standard
```

```
HW_Bit_Types: THEORY
BEGIN
```

```
IMPORTING Standard
```

```
HW_1Bit: TYPE = mod(2^1)
HW_2Bits: TYPE = mod(2^2)
HW_3Bits: TYPE = mod(2^3)
HW_4Bits: TYPE = mod(2^4)
HW_5Bits: TYPE = mod(2^5)
HW_6Bits: TYPE = mod(2^6)
HW_7Bits: TYPE = mod(2^7)
HW_8Bits: TYPE = mod(2^8)
HW_9Bits: TYPE = mod(2^9)
HW_10Bits: TYPE = mod(2^10)
HW_11Bits: TYPE = mod(2^11)
HW_12Bits: TYPE = mod(2^12)
HW_13Bits: TYPE = mod(2^13)
HW_14Bits: TYPE = mod(2^14)
HW_15Bits: TYPE = mod(2^15)
HW_16Bits: TYPE = mod(2^16)
HW_17Bits: TYPE = mod(2^17)
HW_18Bits: TYPE = mod(2^18)
HW_19Bits: TYPE = mod(2^19)
HW_20Bits: TYPE = mod(2^20)
HW_21Bits: TYPE = mod(2^21)
HW_22Bits: TYPE = mod(2^22)
HW_23Bits: TYPE = mod(2^23)
HW_24Bits: TYPE = mod(2^24)
HW_25Bits: TYPE = mod(2^25)
HW_26Bits: TYPE = mod(2^26)
HW_27Bits: TYPE = mod(2^27)
HW_28Bits: TYPE = mod(2^28)
HW_29Bits: TYPE = mod(2^29)
HW_30Bits: TYPE = mod(2^30)
HW_31Bits: TYPE = mod(2^31)
HW_32Bits: TYPE = mod(2^32)
```

```
END HW_Bit_Types
```

```

Control_Calculation: THEORY
BEGIN

IMPORTING Standard
IMPORTING HW_Bit_Types

Value_Type: TYPE = Float

HW_Status_Type: TYPE = HW_6Bits
ADC_Value_Type: TYPE = HW_12Bits
DAC_Value_Type: TYPE = HW_12Bits

Input_Index: TYPE = {n: int | n >= 0 AND n <= 3}
Output_Index: TYPE = {n: int | n >= 0 AND n <= 5}
State_Index: TYPE = {n: int | n >= 0 AND n <= 15}

Input_Vector_Type: TYPE = ARRAY [Input_Index -> ADC_Value_Type]
Output_Vector_Type: TYPE = ARRAY [Output_Index -> DAC_Value_Type]
State_Vector_Type: TYPE = ARRAY [State_Index -> Value_Type]

CV_Index: TYPE = {CV_Normal, CV_1U}
CVF_Type: TYPE = ARRAY [HW_Status_Type -> CV_Index]

Matrix_A_Row: TYPE = ARRAY [State_Index -> Value_Type]
Matrix_B_Row: TYPE = ARRAY [Input_Index -> Value_Type]
Matrix_D_Row: TYPE = ARRAY [State_Index -> Value_Type]
Matrix_E_Row: TYPE = ARRAY [Input_Index -> Value_Type]

Matrix_A: TYPE = ARRAY [State_Index -> Matrix_A_Row]
Matrix_B: TYPE = ARRAY [State_Index -> Matrix_B_Row]
Matrix_D: TYPE = ARRAY [Output_Index -> Matrix_D_Row]
Matrix_E: TYPE = ARRAY [Output_Index -> Matrix_E_Row]

A_Variants_Type: TYPE = ARRAY [CV_Index -> Matrix_A]
B_Variants_Type: TYPE = ARRAY [CV_Index -> Matrix_B]
D_Variants_Type: TYPE = ARRAY [CV_Index -> Matrix_D]
E_Variants_Type: TYPE = ARRAY [CV_Index -> Matrix_E]

U_Vector_Type: TYPE = ARRAY [Input_Index -> Value_Type]
C_Vector_Type: TYPE = ARRAY [Output_Index -> Value_Type]

A: A_Variants_Type = LAMBDA (i: CV_Index):
  CASES i OF
    CV_Normal: LAMBDA (j: State_Index):
      LAMBDA (k: State_Index): 0.0,
    CV_1U:      LAMBDA (j: State_Index):
      LAMBDA (k: State_Index): 0.0
  ENDCASES

B: B_Variants_Type = LAMBDA (i: CV_Index):
  CASES i OF
    CV_Normal: LAMBDA (j: State_Index):
      COND
        j = 0 -> LAMBDA (k: Input_Index):

```

```

                                COND
                                k = 0 -> 1.0,
                                ELSE -> 0.0
                                ENDCOND,
j = 1 -> LAMBDA (k: Input_Index):
                                COND
                                k = 1 -> 1.0,
                                ELSE -> 0.0
                                ENDCOND,
j = 2 -> LAMBDA (k: Input_Index):
                                COND
                                k = 2 -> 1.0,
                                ELSE -> 0.0
                                ENDCOND,
j = 3 -> LAMBDA (k: Input_Index):
                                COND
                                k = 3 -> 1.0,
                                ELSE -> 0.0
                                ENDCOND,
                                ELSE -> LAMBDA (k: Input_Index): 0.0
                                ENDCOND,
CV_1U:    LAMBDA (j: State_Index):
                                LAMBDA (k: Input_Index): 0.0
ENDCASES

D: D_Variants_Type = LAMBDA (i: CV_Index):
CASES i OF
  CV_Normal: LAMBDA (j: Output_Index):
    COND
      j = 0 -> LAMBDA (k: State_Index):
        COND
          k = 0 -> 12.0,
          ELSE -> 0.0
        ENDCOND,
      j = 1 -> LAMBDA (k: State_Index):
        COND
          k = 0 -> -6.0,
          k = 1 -> 12.0,
          ELSE -> 0.0
        ENDCOND,
      j = 2 -> LAMBDA (k: State_Index):
        COND
          k = 0 -> -6.0,
          k = 1 -> -12.0,
          ELSE -> 0.0
        ENDCOND,
      j = 3 -> LAMBDA (k: State_Index):
        COND
          k = 2 -> 12.0,
          ELSE -> 0.0
        ENDCOND,
      j = 4 -> LAMBDA (k: State_Index):
        COND
          k = 2 -> -6.0,

```

```

        k = 3 -> 12.0,
        ELSE -> 0.0
    ENDCOND,
j = 5 -> LAMBDA (k: State_Index):
    COND
        k = 2 -> -6.0,
        k = 3 -> -12.0,
        ELSE -> 0.0
    ENDCOND
    ENDCOND,
CV_1U:    LAMBDA (j: Output_Index):
        LAMBDA (k: State_Index): 0.0
ENDCASES

E: E_Variants_Type = LAMBDA (i: CV_Index):
CASES i OF
    CV_Normal: LAMBDA (j: Output_Index):
        COND
            j = 0 -> LAMBDA (k: Input_Index):
                COND
                    k = 0 -> 16.0000,
                    ELSE -> 0.0000
                ENDCOND,
            j = 1 -> LAMBDA (k: Input_Index):
                COND
                    k = 0 -> -8.0000,
                    k = 1 -> 16.0000,
                    ELSE -> 0.0000
                ENDCOND,
            j = 2 -> LAMBDA (k: Input_Index):
                COND
                    k = 0 -> -8.0000,
                    k = 1 -> -16.0000,
                    ELSE -> 0.0000
                ENDCOND,
            j = 3 -> LAMBDA (k: Input_Index):
                COND
                    k = 2 -> 16.0000,
                    ELSE -> 0.0000
                ENDCOND,
            j = 4 -> LAMBDA (k: Input_Index):
                COND
                    k = 2 -> -8.0000,
                    k = 3 -> 16.0000,
                    ELSE -> 0.0000
                ENDCOND,
            j = 5 -> LAMBDA (k: Input_Index):
                COND
                    k = 2 -> -8.0000,
                    k = 3 -> -16.0000,
                    ELSE -> 0.0000
                ENDCOND
        ENDCOND,
CV_1U:    LAMBDA (j: Output_Index):

```

```

                                LAMBDA (k: Input_Index): 0.0
ENDCASES

CVF: CVF_Type = LAMBDA (i: HW_Status_Type):
  COND
    i = 0 -> CV_Normal,
    i = 1 -> CV_Normal,
    i = 2 -> CV_Normal,
    i = 3 -> CV_Normal,
    i = 4 -> CV_Normal,
    i = 5 -> CV_Normal,
    i = 6 -> CV_Normal,
    i = 7 -> CV_Normal,
    i = 8 -> CV_Normal,
    i = 9 -> CV_Normal,
    i = 10 -> CV_Normal,
    i = 11 -> CV_Normal,
    i = 12 -> CV_Normal,
    i = 13 -> CV_Normal,
    i = 14 -> CV_Normal,
    i = 15 -> CV_Normal,
    i = 16 -> CV_Normal,
    i = 17 -> CV_Normal,
    i = 18 -> CV_Normal,
    i = 19 -> CV_Normal,
    i = 20 -> CV_Normal,
    i = 21 -> CV_Normal,
    i = 22 -> CV_Normal,
    i = 23 -> CV_Normal,
    i = 24 -> CV_Normal,
    i = 25 -> CV_Normal,
    i = 26 -> CV_Normal,
    i = 27 -> CV_Normal,
    i = 28 -> CV_Normal,
    i = 29 -> CV_Normal,
    i = 30 -> CV_Normal,
    i = 31 -> CV_Normal,
    i = 32 -> CV_1U,
    i = 33 -> CV_1U,
    i = 34 -> CV_1U,
    i = 35 -> CV_1U,
    i = 36 -> CV_1U,
    i = 37 -> CV_1U,
    i = 38 -> CV_1U,
    i = 39 -> CV_1U,
    i = 40 -> CV_1U,
    i = 41 -> CV_1U,
    i = 42 -> CV_1U,
    i = 43 -> CV_1U,
    i = 44 -> CV_1U,
    i = 45 -> CV_1U,
    i = 46 -> CV_1U,
    i = 47 -> CV_1U,
    i = 48 -> CV_1U,

```

```

        i = 49 -> CV_1U,
        i = 50 -> CV_1U,
        i = 51 -> CV_1U,
        i = 52 -> CV_1U,
        i = 53 -> CV_1U,
        i = 54 -> CV_1U,
        i = 55 -> CV_1U,
        i = 56 -> CV_1U,
        i = 57 -> CV_1U,
        i = 58 -> CV_1U,
        i = 59 -> CV_1U,
        i = 60 -> CV_1U,
        i = 61 -> CV_1U,
        i = 62 -> CV_1U,
        i = 63 -> CV_1U
    ENDCOND

Default_HWS: HW_Status_Type = 0
DAC_Scale: Value_Type = 1024.0
DAC_Offset: Value_Type = 2.0
ADC_Scale: Value_Type = 0.000625
ADC_Offset: Value_Type = -1.0

State: TYPE = [# C      : C_Vector_Type,
               CV       : CV_Index,
               Last_HWS : HW_Status_Type,
               U        : U_Vector_Type,
               Input     : Input_Vector_Type,
               HW_Status : HW_Status_Type,
               Output    : Output_Vector_Type,
               X         : State_Vector_Type #]

Init(st: State): State =
    st WITH [ `X := LAMBDA (i: State_Index): 0.0,
              `C := LAMBDA (i: Output_Index): 0.0,
              `CV := CVF(Default_HWS),
              `HW_Status := 0,
              `Last_HWS := Default_HWS,
              `U := LAMBDA (i: Input_Index): 0.0 ]

Convert_Input_pre(st: State): bool = TRUE
Convert_Input_post(st_: State, st: State): bool =
    st`C = st_`C AND
    st`CV = st_`CV AND
    st`Last_HWS = st_`Last_HWS AND
    st`Input = st_`Input AND
    st`HW_Status = st_`HW_Status AND
    st`Output = st_`Output AND
    st`X = st_`X AND
    FORALL (i: Input_Index):
        st`U(i) = st_`Input(i) * ADC_Scale + ADC_Offset
Convert_Input: FUNCTION [State -> State]
Convert_Input: LEMMA
    FORALL (st: State):

```

```

Convert_Input_pre(st) => Convert_Input_post(st, Convert_Input(st))

Compute_Output_pre(st: State): bool = TRUE
Compute_Output_post(st_: State, st: State): bool =
  st`CV = st_`CV AND
  st`Last_HWS = st_`Last_HWS AND
  st`U = st_`U AND
  st`Input = st_`Input AND
  st`HW_Status = st_`HW_Status AND
  st`Output = st_`Output AND
  st`X = st_`X AND
  FORALL (i: Output_Index): st`C(i) = D(st_`CV)(i)(0) * st_`X(0) +
    D(st_`CV)(i)(1) * st_`X(1) +
    D(st_`CV)(i)(2) * st_`X(2) +
    D(st_`CV)(i)(3) * st_`X(3) +
    D(st_`CV)(i)(4) * st_`X(4) +
    D(st_`CV)(i)(5) * st_`X(5) +
    D(st_`CV)(i)(6) * st_`X(6) +
    D(st_`CV)(i)(7) * st_`X(7) +
    D(st_`CV)(i)(8) * st_`X(8) +
    D(st_`CV)(i)(9) * st_`X(9) +
    D(st_`CV)(i)(10) * st_`X(10) +
    D(st_`CV)(i)(11) * st_`X(11) +
    D(st_`CV)(i)(12) * st_`X(12) +
    D(st_`CV)(i)(13) * st_`X(13) +
    D(st_`CV)(i)(14) * st_`X(14) +
    D(st_`CV)(i)(15) * st_`X(15) +
    E(st_`CV)(i)(0) * st_`U(0) +
    E(st_`CV)(i)(1) * st_`U(1) +
    E(st_`CV)(i)(2) * st_`U(2) +
    E(st_`CV)(i)(3) * st_`U(3)

Compute_Output: FUNCTION [State -> State]
Compute_Output: LEMMA
  FORALL (st: State):
    Compute_Output_pre(st) =>
      Compute_Output_post(st, Compute_Output(st))

Compute_New_State_pre(st: State): bool = TRUE
Compute_New_State_post(st_: State, st: State): bool =
  st`C = st_`C AND
  st`CV = st_`CV AND
  st`Last_HWS = st_`Last_HWS AND
  st`U = st_`U AND
  st`Input = st_`Input AND
  st`HW_Status = st_`HW_Status AND
  st`Output = st_`Output AND
  FORALL (i: State_Index): st`X(i) = A(st_`CV)(i)(0) * st_`X(0) +
    A(st_`CV)(i)(1) * st_`X(1) +
    A(st_`CV)(i)(2) * st_`X(2) +
    A(st_`CV)(i)(3) * st_`X(3) +
    A(st_`CV)(i)(4) * st_`X(4) +
    A(st_`CV)(i)(5) * st_`X(5) +
    A(st_`CV)(i)(6) * st_`X(6) +
    A(st_`CV)(i)(7) * st_`X(7) +

```

```

A(st_`CV)(i)(8) * st_`X(8) +
A(st_`CV)(i)(9) * st_`X(9) +
A(st_`CV)(i)(10) * st_`X(10) +
A(st_`CV)(i)(11) * st_`X(11) +
A(st_`CV)(i)(12) * st_`X(12) +
A(st_`CV)(i)(13) * st_`X(13) +
A(st_`CV)(i)(14) * st_`X(14) +
A(st_`CV)(i)(15) * st_`X(15) +
B(st_`CV)(i)(0) * st_`U(0) +
B(st_`CV)(i)(1) * st_`U(1) +
B(st_`CV)(i)(2) * st_`U(2) +
B(st_`CV)(i)(3) * st_`U(3)

Compute_New_State: FUNCTION [State -> State]
Compute_New_State: LEMMA
  FORALL (st: State):
    Compute_New_State_pre(st) =>
      Compute_New_State_post(st, Compute_New_State(st))

Convert_Output_pre(st: State): bool = TRUE
Convert_Output_post(st_: State, st: State): bool =
  st_`C = st_`C AND
  st_`CV = st_`CV AND
  st_`Last_HWS = st_`Last_HWS AND
  st_`U = st_`U AND
  st_`Input = st_`Input AND
  st_`HW_Status = st_`HW_Status AND
  st_`X = st_`X AND
  FORALL (i: Output_Index):
    ((st_`C(i) + DAC_Offset) * DAC_Scale > 4095 AND st_`Output(i) = 4095)
    OR
    ((st_`C(i) + DAC_Offset) * DAC_Scale < 0 AND st_`Output(i) = 0)
    OR
    ((st_`C(i) + DAC_Offset) * DAC_Scale <= 4095 AND
     (st_`C(i) + DAC_Offset) * DAC_Scale >= 0 AND
     st_`Output(i) = floor((st_`C(i) + DAC_Offset) * DAC_Scale) + 0.5)
Convert_Output: FUNCTION [State -> State]
Convert_Output: LEMMA
  FORALL (st: State):
    Convert_Output_pre(st) =>
      Convert_Output_post(st, Convert_Output(st))

Calculate_Control(st: State): State =
  LET st1 = IF st_`HW_Status = st_`Last_HWS THEN
    Compute_New_State(Compute_Output(Convert_Input(st)))
  ELSE
    st WITH [ `C := LAMBDA (i: Output_Index): 0.0,
              `CV := CVF(st_`HW_Status),
              `Last_HWS := st_`HW_Status,
              `U := LAMBDA (i: Input_Index): 0.0,
              `X := LAMBDA (i: State_Index): 0.0 ]
  ENDIF IN
  Convert_Output(st1)

END Control_Calculation

```



```

Hardware_Interface: THEORY
BEGIN

IMPORTING Standard
IMPORTING HW_Bit_Types
IMPORTING Control_Calculation

Time_Type: TYPE = mod(2^64)
Ticks_Per_uSec: Time_Type = 128
Ticks_Per_mSec: Time_Type = 1000 * Ticks_Per_uSec
Ticks_Per_Sec: Time_Type = 1000 * Ticks_Per_mSec

State: TYPE+

Get_Time_Base(st: State): Time_Type
Get_Coil_Failure(st: State): HW_1Bit

Configure_System_Clock(st: State): State

Configure_Balls(st: State): State

Configure_DSPI_Inputs(st: State): State

Configure_DSPI_C(st: State): State

Configure_DAC(st: State): State

Configure_QADC(st: State): State

Configure_Time_Base(st: State): State

Get_Time_pre(st: State): bool = TRUE
Get_Time_post(st_: State, Time: Time_Type, st: State): bool =
  Time = Get_Time_Base(st)
Get_Time(st: State): [# Time: Time_Type, st: State #]
Get_Time: LEMMA
  FORALL (st: State):
    Get_Time_pre(st) =>
      Get_Time_post(st, Get_Time(st)`Time, Get_Time(st)`st)

Trigger_And_Read_QADC(st: State): [# st: State,
                                     Input: Input_Vector_Type #]

Update_DAC_Outputs(Output: Output_Vector_Type, st: State): State

Get_HW_Status_pre(st: State): bool = TRUE
Get_HW_Status_post(st_: State,
                   st: State,
                   HW_Status: HW_Status_Type): bool =
  (Get_Coil_Failure(st) = 0 AND HW_Status = 32) OR
  (Get_Coil_Failure(st) = 1 AND HW_Status = 0)
Get_HW_Status(st: State): [# st: State, HW_Status: HW_Status_Type #]
Get_HW_Status: LEMMA
  FORALL (st: State):

```

```

    Get_HW_Status_pre(st) =>
        Get_HW_Status_post(st,
            Get_HW_Status(st)`st,
            Get_HW_Status(st)`HW_Status)

END Hardware_Interface

Logging: THEORY
BEGIN

IMPORTING Standard
IMPORTING Control_Calculation

State: TYPE+

Log_Control_Frame(HW_Status: HW_Status_Type,
    Input: Input_Vector_Type,
    Output: Output_Vector_Type,
    X: State_Vector_Type,
    st: State): State

END Logging

main_program: THEORY
BEGIN

IMPORTING Standard
IMPORTING Control_Calculation
IMPORTING Hardware_Interface
IMPORTING HW_Bit_Types
IMPORTING Logging

Frame_Period: Time_Type = 200 * Ticks_Per_uSec

State: TYPE = [# st_c:      Control_Calculation.State,
    st_h:      Hardware_Interface.State,
    st_l:      Logging.State,
    Next_Frame: Time_Type,
    Last_Frame: Time_Type #]

Initialize_Frame_pre(st: State): bool = TRUE
Initialize_Frame_post(st_: State, st: State): bool =
    st`st_c = st_`st_c AND
    st`st_l = st_`st_l AND
    st`Last_Frame = Get_Time_Base(st`st_h) AND
    st`Next_Frame = rem(2^64)(st`Last_Frame + Frame_Period)
Initialize_Frame: FUNCTION [State -> State]
Initialize_Frame: LEMMA
FORALL (st: State):
    Initialize_Frame_pre(st) =>
        Initialize_Frame_post(st, Initialize_Frame(st))

```

```

Block_Until_Next_Frame_pre(st: State): bool = TRUE
Block_Until_Next_Frame_post(st_: State, st: State): bool =
  st`st_c = st`st_c AND
  st`st_l = st`st_l AND
  Get_Time_Base(st`st_h) >= st`Next_Frame AND
  st`Last_Frame = st`Next_Frame AND
  st`Next_Frame = rem(2^64)(st`Last_Frame + Frame_Period)
Block_Until_Next_Frame: FUNCTION [State -> State]
Block_Until_Next_Frame: LEMMA
  FORALL (st: State):
    Block_Until_Next_Frame_pre(st) =>
      Block_Until_Next_Frame_post(st, Block_Until_Next_Frame(st))

%% manually modeled infinite loop in Cyclic_Executive

Trigger_And_Read_QADC(st: State): State =
  st WITH [ `st_h := Trigger_And_Read_QADC(st`st_h)`st,
            `st_c`Input := Trigger_And_Read_QADC(st`st_h)`Input ]

Get_HW_Status(st: State): State =
  st WITH [ `st_h := Get_HW_Status(st`st_h)`st,
            `st_c`HW_Status := Get_HW_Status(st`st_h)`HW_Status ]

Calculate_Control(st: State): State =
  st WITH [ `st_c := Calculate_Control(st`st_c) ]

Log_Control_Frame(st: State): State =
  st WITH [ `st_l := Log_Control_Frame(st`st_c`HW_Status, st`st_c`Input,
                                         st`st_c`Output, st`st_c`X, st`st_l) ]

Update_DAC_Outputs(st: State): State =
  st WITH [ `st_h := Update_DAC_Outputs(st`st_c`Output, st`st_h) ]

%% splitted from Cyclic_Executive
Configure(st: State): State =
  st WITH [ `st_h := Configure_Time_Base(
    Configure_QADC(
      Configure_DAC(
        Configure_DSPI_C(
          Configure_DSPI_Inputs(
            Configure_Balls(
              Configure_System_Clock(st`st_h)))))) ]

%% splitted from Cyclic_Executive
Frame(st: State): State =
  Update_DAC_Outputs(
    Log_Control_Frame(
      Calculate_Control(
        Get_HW_Status(
          Trigger_And_Read_QADC(st))))))

Cyclic_Executive_Loop(st: State, i: nat): RECURSIVE State =
  IF i = 0 THEN
    st

```

```

ELSE
  LET st1 = Frame(Block_Until_Next_Frame(st)) IN
  Cyclic_Executive_Loop(st1, i-1)
ENDIF
MEASURE i

%% execute for n cycles
Cyclic_Executive(st: State, n: nat): State =
  LET st1 = Initialize_Frame(Configure(st)) IN
  Cyclic_Executive_Loop(st1, n)

END main_program

```

A.3 Implication Theorem of MBCS

```

vad_cntrl_proof: THEORY
BEGIN
IMPORTING org@vad_cntrl_sw
IMPORTING ext@main_program

%% ===== types ===== %%

%% value_type

value_type_ret(v: Value_Type): value_type = v

value_type_ret_inv(v: value_type): Value_Type = v

value_type_equiv: LEMMA
  left_inverse?(value_type_ret_inv, value_type_ret) AND
  right_inverse?(value_type_ret_inv, value_type_ret)

%% dac_value_type

dac_value_type_ret(d: DAC_Value_Type): dac_value_type = d

dac_value_type_ret_inv(d: dac_value_type): DAC_Value_Type = d

dac_value_type_equiv: LEMMA
  left_inverse?(dac_value_type_ret_inv, dac_value_type_ret) AND
  right_inverse?(dac_value_type_ret_inv, dac_value_type_ret)

%% adc_value_type

adc_value_type_ret(a: ADC_Value_Type): adc_value_type = a

adc_value_type_ret_inv(a: adc_value_type): ADC_Value_Type = a

adc_value_type_equiv: LEMMA
  left_inverse?(adc_value_type_ret_inv, adc_value_type_ret) AND
  right_inverse?(adc_value_type_ret_inv, adc_value_type_ret)

%% input_index

input_index_ret(i: Input_Index): below(input_size) = i

input_index_ret_inv(i: below(input_size)): Input_Index = i

input_index_equiv: LEMMA
  left_inverse?(input_index_ret_inv, input_index_ret) AND
  right_inverse?(input_index_ret_inv, input_index_ret)

%% output_index

output_index_ret(o: Output_Index): below(output_size) = o

```

```

output_index_ret_inv(o: below(output_size)): Output_Index = o

output_index_equiv: LEMMA
  left_inverse?(output_index_ret_inv, output_index_ret) AND
  right_inverse?(output_index_ret_inv, output_index_ret)

%% state_index

state_index_ret(s: State_Index): below(state_size) = s

state_index_ret_inv(s: below(state_size)): State_Index = s

state_index_equiv: LEMMA
  left_inverse?(state_index_ret_inv, state_index_ret) AND
  right_inverse?(state_index_ret_inv, state_index_ret)

%% time_type

time_type_ret(t: Time_Type): time_type = t

time_type_ret_inv(t: time_type): Time_Type = t

time_type_equiv: LEMMA
  left_inverse?(time_type_ret_inv, time_type_ret) AND
  right_inverse?(time_type_ret_inv, time_type_ret)

%% coil_failure_type

coil_failure_type_ret(c: HW_1Bit): coil_failure_type = c

coil_failure_type_ret_inv(c: coil_failure_type): HW_1Bit = c

coil_failure_type_equiv: LEMMA
  left_inverse?(coil_failure_type_ret_inv, coil_failure_type_ret) AND
  right_inverse?(coil_failure_type_ret_inv, coil_failure_type_ret)

%% input_vector_type

input_vector_type_ret(i: Input_Vector_Type): input_vector_type =
  LAMBDA (p: below(input_size)): adc_value_type_ret(i(p))

input_vector_type_ret_inv(i: input_vector_type): Input_Vector_Type =
  LAMBDA (p: Input_Index): adc_value_type_ret_inv(i(p))

input_vector_type_equiv: LEMMA
  left_inverse?(input_vector_type_ret_inv, input_vector_type_ret) AND
  right_inverse?(input_vector_type_ret_inv, input_vector_type_ret)

%% output_vector_type

output_vector_type_ret(o: Output_Vector_Type): output_vector_type =
  LAMBDA (i: below(output_size)): dac_value_type_ret(o(i))

output_vector_type_ret_inv(o: output_vector_type): Output_Vector_Type =

```

```

    LAMBDA (i: Output_Index): dac_value_type_ret_inv(o(i))

output_vector_type_equiv: LEMMA
  left_inverse?(output_vector_type_ret_inv, output_vector_type_ret) AND
  right_inverse?(output_vector_type_ret_inv, output_vector_type_ret)

%% u_vector_type

u_vector_type_ret(u: U_Vector_Type): u_vector_type =
  LAMBDA (i: below(input_size)): value_type_ret(u(i))

u_vector_type_ret_inv(u: u_vector_type): U_Vector_Type =
  LAMBDA (i: Input_Index): value_type_ret_inv(u(i))

u_vector_type_equiv: LEMMA
  left_inverse?(u_vector_type_ret_inv, u_vector_type_ret) AND
  right_inverse?(u_vector_type_ret_inv, u_vector_type_ret)

%% c_vector_type

c_vector_type_ret(c: C_Vector_Type): c_vector_type =
  LAMBDA (i: below(output_size)): value_type_ret(c(i))

c_vector_type_ret_inv(c: c_vector_type): C_Vector_Type =
  LAMBDA (i: Output_Index): value_type_ret_inv(c(i))

c_vector_type_equiv: LEMMA
  left_inverse?(c_vector_type_ret_inv, c_vector_type_ret) AND
  right_inverse?(c_vector_type_ret_inv, c_vector_type_ret)

%% x_vector_type

x_vector_type_ret(x: State_Vector_Type): x_vector_type =
  LAMBDA (i: below(state_size)): value_type_ret(x(i))

x_vector_type_ret_inv(x: x_vector_type): State_Vector_Type =
  LAMBDA (i: State_Index): value_type_ret_inv(x(i))

x_vector_type_equiv: LEMMA
  left_inverse?(x_vector_type_ret_inv, x_vector_type_ret) AND
  right_inverse?(x_vector_type_ret_inv, x_vector_type_ret)

%% hws_val_type

hws_val_type_ret(b: bit): hws_val_type = IF b THEN NotOp ELSE Op ENDIF

hws_val_type_ret_inv(h: hws_val_type): bit =
  CASES h OF
    Op:    FALSE,
    NotOp: TRUE
  ENDCASES

hws_val_type_equiv: LEMMA
  left_inverse?(hws_val_type_ret_inv, hws_val_type_ret) AND

```

```

    right_inverse?(hws_val_type_ret_inv, hws_val_type_ret)

%% hw_status_type

hw_status_type_ret(h: HW_Status_Type): hw_status_type =
  LET bv: bvec[6] = nat2bv(h) IN
  LAMBDA (i: below(6)): hws_val_type_ret(bv(5-i))

hw_status_type_ret_inv(h: hw_status_type): HW_Status_Type =
  LET bv: bvec[6] = LAMBDA (i: below(6)): hws_val_type_ret_inv(h(5-i)) IN
  bv2nat(bv)

hw_status_type_equiv: LEMMA
  left_inverse?(hw_status_type_ret_inv, hw_status_type_ret) AND
  right_inverse?(hw_status_type_ret_inv, hw_status_type_ret)

%% cv_value_type

cv_value_type_ret(c: CV_Index): cv_value_type =
  CASES c OF
    CV_Normal: cv_normal,
    CV_lU:      cv_lu
  ENDCASES

cv_value_type_ret_inv(c: cv_value_type): CV_Index =
  CASES c OF
    cv_normal: CV_Normal,
    cv_lu:      CV_lU
  ENDCASES

cv_value_type_equiv: LEMMA
  left_inverse?(cv_value_type_ret_inv, cv_value_type_ret) AND
  right_inverse?(cv_value_type_ret_inv, cv_value_type_ret)

%% ===== constants ===== %%

%% default_hws

default_hws_equiv: LEMMA
  hw_status_type_ret_inv(hws_AllOp) = Default_HWS

%% cvf

cvf_equiv: LEMMA
  FORALL (h: hw_status_type):
    cv_value_type_ret(CVF(hw_status_type_ret_inv(h))) = cvf(h)

%% A

A_equiv: LEMMA
  FORALL (c: CV_Index, i: State_Index, j: State_Index):
    ext@Control_Calculation.A(c)(i)(j) =
      org@vad_cntrl_sw.A(cv_value_type_ret(c))(i, j)

```



```

%% B

B_equiv: LEMMA
  FORALL (c: CV_Index, i: State_Index, j: Input_Index):
    ext@Control_Calculation.B(c)(i)(j) =
      org@vad_cntrl_sw.B(cv_value_type_ret(c))(i, j)

%% D

D_equiv: LEMMA
  FORALL (c: CV_Index, i: Output_Index, j: State_Index):
    ext@Control_Calculation.D(c)(i)(j) =
      org@vad_cntrl_sw.D(cv_value_type_ret(c))(i, j)

%% E

E_equiv: LEMMA
  FORALL (c: CV_Index, i: Output_Index, j: Input_Index):
    ext@Control_Calculation.E(c)(i)(j) =
      org@vad_cntrl_sw.E(cv_value_type_ret(c))(i, j)

%% frame_period

frame_period_equiv: LEMMA
  time_type_ret(Frame_Period) = frame_period

%% adc_scale

adc_scale_equiv: LEMMA
  value_type_ret(ADC_Scale) = adc_scale

%% adc_offset

adc_offset_equiv: LEMMA
  value_type_ret(ADC_Offset) = adc_offset

%% dac_scale

dac_scale_equiv: LEMMA
  value_type_ret(DAC_Scale) = dac_scale

%% dac_offset

dac_offset_equiv: LEMMA
  value_type_ret(DAC_Offset) = dac_offset

%% ===== state ===== %%

system_status_ret(s: main_program.State): system_status_type =
  (# `hw := (# `time_base := time_type_ret(Get_Time_Base(s`st_h)),
    `coil_failure :=
coil_failure_type_ret(Get_Coil_Failure(s`st_h))
    #),
    `st := (# `input := input_vector_type_ret(s`st_c`Input),

```

```

        `output := output_vector_type_ret(s`st_c`Output),
        `last_frame := time_type_ret(s`Last_Frame),
        `next_frame := time_type_ret(s`Next_Frame),
        `hw_status := hw_status_type_ret(s`st_c`HW_Status),
        `last_hws := hw_status_type_ret(s`st_c`Last_HWS),
        `cv := cv_value_type_ret(s`st_c`CV),
        `u := u_vector_type_ret(s`st_c`U),
        `c := c_vector_type_ret(s`st_c`C),
        `x := x_vector_type_ret(s`st_c`X)
    #)
#)

system_status_impl: LEMMA surjective?(system_status_ret)

%% ===== operations ===== %%

%% init

init_impl: LEMMA
  FORALL (s_, s: main_program.State):
    s`st_c = Init(s`st_c) AND
    s`st_l = s`st_l AND
    s`Last_Frame = s`Last_Frame AND
    s`Next_Frame = s`Next_Frame
    =>
    system_status_ret(s)`st = init(system_status_ret(s)`st)

%% convert_input

convert_input_impl: LEMMA
  FORALL (s_, s: main_program.State):
    Convert_Input_post(s`st_c, s`st_c) AND
    s`st_l = s`st_l AND
    s`Last_Frame = s`Last_Frame AND
    s`Next_Frame = s`Next_Frame
    =>
    system_status_ret(s)`st = convert_input(system_status_ret(s)`st)

%% compute_output

compute_output_impl: LEMMA
  FORALL (s_, s: main_program.State):
    Compute_Output_post(s`st_c, s`st_c) AND
    s`st_l = s`st_l AND
    s`Last_Frame = s`Last_Frame AND
    s`Next_Frame = s`Next_Frame
    =>
    system_status_ret(s)`st = compute_output(system_status_ret(s)`st)

%% compute_new_state

compute_new_state_impl: LEMMA
  FORALL (s_, s: main_program.State):
    Compute_New_State_post(s`st_c, s`st_c) AND

```

```

s`st_l = s_`st_l AND
s`Last_Frame = s_`Last_Frame AND
s`Next_Frame = s_`Next_Frame
=>
system_status_ret(s)`st =
compute_new_state(system_status_ret(s_)`st)

%% convert_output

convert_output_impl: LEMMA
FORALL (s_, s: main_program.State):
  Convert_Output_post(s_`st_c, s`st_c) AND
  s`st_l = s_`st_l AND
  s`Last_Frame = s_`Last_Frame AND
  s`Next_Frame = s_`Next_Frame
=>
system_status_ret(s)`st = convert_output(system_status_ret(s_)`st)

%% calculate_control

calculate_control_impl: LEMMA
FORALL (s_, s: main_program.State):
  s`st_c = Calculate_Control(s_`st_c) AND
  s`st_l = s_`st_l AND
  s`Last_Frame = s_`Last_Frame AND
  s`Next_Frame = s_`Next_Frame
=>
system_status_ret(s)`st =
calculate_control(system_status_ret(s_)`st)

%% convert_status

convert_status_impl: LEMMA
FORALL (s_, s: main_program.State):
  Get_HW_Status_post(s_`st_h, s`st_h, s`st_c`HW_Status) AND
  s`st_c`C = s_`st_c`C AND
  s`st_c`CV = s_`st_c`CV AND
  s`st_c`Last_HWS = s_`st_c`Last_HWS AND
  s`st_c`U = s_`st_c`U AND
  s`st_c`Input = s_`st_c`Input AND
  s`st_c`Output = s_`st_c`Output AND
  s`st_c`X = s_`st_c`X AND
  s`st_l = s_`st_l AND
  s`Last_Frame = s_`Last_Frame AND
  s`Next_Frame = s_`Next_Frame
=>
convert_status_post(system_status_ret(s_), system_status_ret(s))

%% frame_init

frame_init_impl: LEMMA
FORALL (s_, s: main_program.State):
  Initialize_Frame_post(s_, s)
=>

```

```

    frame_init_post(system_status_ret(s_), system_status_ret(s))

%% frame_sync

frame_sync_impl: LEMMA
  FORALL (s_, s: main_program.State):
    Block_Until_Next_Frame_post(s_, s)
    =>
    frame_sync_post(system_status_ret(s_), system_status_ret(s))

END vad_cntrl_proof

```

Appendix B: Example Verification Refactoring

B.1 PVS Theorems for Example Transformations

B.1.1 Loop Rerolling

Following is the PVS theorem that is used for proving the semantics-preserving nature of loop rerolling transformation.

```
% loop_body(1)
% loop_body(2)          for i from 1 to n
% .                      loop
% .                      loop_body(i)
% .                      end loop
% loop_body(n)

loop_rerolling[n: posint] : THEORY
BEGIN

  id: NONEMPTY_TYPE
  val: TYPE = int
  state: TYPE = [id -> val]

  % statement: TYPE = [state -> state]
  % seq_of_statements: TYPE = {s: finseq[statement] | length(s) > 0}
  seq_of_statements: TYPE = [state -> state]
  predicate: TYPE = pred[state]

  loop_index: TYPE = {i: posint | i <= n}
  loop_body: [loop_index -> seq_of_statements]
  loop_body_pre: [loop_index -> predicate]
  loop_body_post: [loop_index -> predicate]

  pre: predicate
  post: predicate

  % {pre(i)} loop_body(i) {post(i)}
  % pre = pre(1), post(i) = pre(i+1), post(n) = post

  unrolled_body: AXIOM
    FORALL(i: loop_index, st: state):
      loop_body_pre(i)(st) => loop_body_post(i)(loop_body(i)(st))

  unrolled_pred_1: AXIOM
    pre = loop_body_pre(1)

  unrolled_pred_i: AXIOM
    FORALL (i: {j: loop_index | j <= n-1}):
      loop_body_post(i) = loop_body_pre(i+1)

  unrolled_pred_n: AXIOM
    loop_body_post(n) = post

  forloop(i: loop_index, st: state): RECURSIVE state =
```

```

    IF i >= n THEN loop_body(n) (st)
    ELSE forloop(i+1, loop_body(i) (st))
    ENDIF
MEASURE n-i;

forloop_pred_i: LEMMA
  FORALL(i: loop_index, st: state):
    loop_body_pre(i) (st) => loop_body_post(n) (forloop(i, st))

rerolled_loop: seq_of_statements =
  LAMBDA (st: state): forloop(1, st)

rerolled_equiv: THEOREM
  FORALL(st: state):
    pre(st) => post(rerolled_loop(st))

END loop_rerolling

```

B.1.2 Lift-if

Following is another example PVS theorem that is used for proving the semantics-preserving nature of lift-if transformation. Informally, given condition c and statement sequences s_0 , s_1 , s_2 and s_3 , the two transformations in the commented PVS below preserve the semantics provided that the statement sequence s_0 has no side effect on condition c . Such transformation moves statement blocks into conditional statements and can help to simplify execution paths and to reveal certain properties. Note the reverse (that moves statement blocks out of conditional statements) is also a valid semantics-preserving transformation that may help verification under certain circumstances.

```

% provided s0 has no side effect on c

% s0
% if (c) then
%   s1
% else
%   s2
% endif
%
% if (c) then
%   s1
% else
%   s2
% endif
% s3

% if (c) then
%   s0; s1
% else
%   s0; s2
% endif
%
% if (c) then
%   s1; s3
% else
%   s2; s3
% endif

```

```

lift_if : THEORY
BEGIN

  id: NONEMPTY_TYPE
  val: TYPE = int
  state: TYPE = [id -> val]

  % statement: TYPE = [state -> state]
  % seq_of_statements: TYPE = {s: finseq[statement] | length(s) > 0}
  seq_of_statements: TYPE = [state -> state]
  predicate: TYPE = pred[state]

  s0, s1, s2, s3: seq_of_statements
  condition: predicate

  s0_condition: AXIOM
  FORALL(st: state):
    condition(st) = condition(s0(st))

  lift_if_equiv1: THEOREM
  FORALL(st: state):
    LET st1: state = s0(st) IN
    IF condition(st1) THEN
      s1(st1)
    ELSE
      s2(st1)
    ENDIF
    =
    IF condition(st) THEN
      s1(s0(st))
    ELSE
      s2(s0(st))
    ENDIF

  lift_if_equiv2: THEOREM
  FORALL(st: state):
    s3(IF condition(st) THEN
      s1(st)
    ELSE
      s2(st)
    ENDIF)
    =
    IF condition(st) THEN
      s3(s1(st))
    ELSE
      s3(s2(st))
    ENDIF

END lift_if

```


B.2 Example Transformation of AES

I only use the encrypt function in AES source program, and only present some of the intermediate versions, as a demonstration of how verification refactoring affects the code.

B.2.1 Original AES Encrypt Function

Following is the original AES encrypt function as it was translated from the ANSI C implementation:

```

procedure aesEncrypt(rk: in key_schedule; Nr: in Integer; pt: in block;
ct: out block)
is
    s0, s1, s2, s3, t0, t1, t2, t3: word;
    b0, b1, b2, b3: byte;
begin
    s0 := CombineWord(pt( 0), pt( 1), pt( 2), pt( 3)) xor rk(0);
    s1 := CombineWord(pt( 4), pt( 5), pt( 6), pt( 7)) xor rk(1);
    s2 := CombineWord(pt( 8), pt( 9), pt(10), pt(11)) xor rk(2);
    s3 := CombineWord(pt(12), pt(13), pt(14), pt(15)) xor rk(3);

    t0 := Te0(byte(Shift_Right(s0, 24))) xor Te1(byte(Shift_Right(s1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
Te3(byte(s3 and 16#ff#)) xor rk( 4);
    t1 := Te0(byte(Shift_Right(s1, 24))) xor Te1(byte(Shift_Right(s2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
Te3(byte(s0 and 16#ff#)) xor rk( 5);
    t2 := Te0(byte(Shift_Right(s2, 24))) xor Te1(byte(Shift_Right(s3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
Te3(byte(s1 and 16#ff#)) xor rk( 6);
    t3 := Te0(byte(Shift_Right(s3, 24))) xor Te1(byte(Shift_Right(s0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
Te3(byte(s2 and 16#ff#)) xor rk( 7);

    s0 := Te0(byte(Shift_Right(t0, 24))) xor Te1(byte(Shift_Right(t1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t2, 8) and 16#ff#)) xor
Te3(byte(t3 and 16#ff#)) xor rk( 8);
    s1 := Te0(byte(Shift_Right(t1, 24))) xor Te1(byte(Shift_Right(t2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t3, 8) and 16#ff#)) xor
Te3(byte(t0 and 16#ff#)) xor rk( 9);
    s2 := Te0(byte(Shift_Right(t2, 24))) xor Te1(byte(Shift_Right(t3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t0, 8) and 16#ff#)) xor
Te3(byte(t1 and 16#ff#)) xor rk(10);
    s3 := Te0(byte(Shift_Right(t3, 24))) xor Te1(byte(Shift_Right(t0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t1, 8) and 16#ff#)) xor
Te3(byte(t2 and 16#ff#)) xor rk(11);

```

```

    t0 := Te0(byte(Shift_Right(s0, 24))) xor Te1(byte(Shift_Right(s1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
Te3(byte(s3 and 16#ff#)) xor rk(12);
    t1 := Te0(byte(Shift_Right(s1, 24))) xor Te1(byte(Shift_Right(s2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
Te3(byte(s0 and 16#ff#)) xor rk(13);
    t2 := Te0(byte(Shift_Right(s2, 24))) xor Te1(byte(Shift_Right(s3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
Te3(byte(s1 and 16#ff#)) xor rk(14);
    t3 := Te0(byte(Shift_Right(s3, 24))) xor Te1(byte(Shift_Right(s0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
Te3(byte(s2 and 16#ff#)) xor rk(15);

    s0 := Te0(byte(Shift_Right(t0, 24))) xor Te1(byte(Shift_Right(t1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t2, 8) and 16#ff#)) xor
Te3(byte(t3 and 16#ff#)) xor rk(16);
    s1 := Te0(byte(Shift_Right(t1, 24))) xor Te1(byte(Shift_Right(t2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t3, 8) and 16#ff#)) xor
Te3(byte(t0 and 16#ff#)) xor rk(17);
    s2 := Te0(byte(Shift_Right(t2, 24))) xor Te1(byte(Shift_Right(t3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t0, 8) and 16#ff#)) xor
Te3(byte(t1 and 16#ff#)) xor rk(18);
    s3 := Te0(byte(Shift_Right(t3, 24))) xor Te1(byte(Shift_Right(t0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t1, 8) and 16#ff#)) xor
Te3(byte(t2 and 16#ff#)) xor rk(19);

    t0 := Te0(byte(Shift_Right(s0, 24))) xor Te1(byte(Shift_Right(s1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
Te3(byte(s3 and 16#ff#)) xor rk(20);
    t1 := Te0(byte(Shift_Right(s1, 24))) xor Te1(byte(Shift_Right(s2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
Te3(byte(s0 and 16#ff#)) xor rk(21);
    t2 := Te0(byte(Shift_Right(s2, 24))) xor Te1(byte(Shift_Right(s3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
Te3(byte(s1 and 16#ff#)) xor rk(22);
    t3 := Te0(byte(Shift_Right(s3, 24))) xor Te1(byte(Shift_Right(s0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
Te3(byte(s2 and 16#ff#)) xor rk(23);

    s0 := Te0(byte(Shift_Right(t0, 24))) xor Te1(byte(Shift_Right(t1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t2, 8) and 16#ff#)) xor
Te3(byte(t3 and 16#ff#)) xor rk(24);
    s1 := Te0(byte(Shift_Right(t1, 24))) xor Te1(byte(Shift_Right(t2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t3, 8) and 16#ff#)) xor
Te3(byte(t0 and 16#ff#)) xor rk(25);
    s2 := Te0(byte(Shift_Right(t2, 24))) xor Te1(byte(Shift_Right(t3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t0, 8) and 16#ff#)) xor
Te3(byte(t1 and 16#ff#)) xor rk(26);
    s3 := Te0(byte(Shift_Right(t3, 24))) xor Te1(byte(Shift_Right(t0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t1, 8) and 16#ff#)) xor
Te3(byte(t2 and 16#ff#)) xor rk(27);

```

```

    t0 := Te0(byte(Shift_Right(s0, 24))) xor Te1(byte(Shift_Right(s1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
Te3(byte(s3 and 16#ff#)) xor rk(28);
    t1 := Te0(byte(Shift_Right(s1, 24))) xor Te1(byte(Shift_Right(s2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
Te3(byte(s0 and 16#ff#)) xor rk(29);
    t2 := Te0(byte(Shift_Right(s2, 24))) xor Te1(byte(Shift_Right(s3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
Te3(byte(s1 and 16#ff#)) xor rk(30);
    t3 := Te0(byte(Shift_Right(s3, 24))) xor Te1(byte(Shift_Right(s0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
Te3(byte(s2 and 16#ff#)) xor rk(31);

    s0 := Te0(byte(Shift_Right(t0, 24))) xor Te1(byte(Shift_Right(t1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t2, 8) and 16#ff#)) xor
Te3(byte(t3 and 16#ff#)) xor rk(32);
    s1 := Te0(byte(Shift_Right(t1, 24))) xor Te1(byte(Shift_Right(t2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t3, 8) and 16#ff#)) xor
Te3(byte(t0 and 16#ff#)) xor rk(33);
    s2 := Te0(byte(Shift_Right(t2, 24))) xor Te1(byte(Shift_Right(t3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t0, 8) and 16#ff#)) xor
Te3(byte(t1 and 16#ff#)) xor rk(34);
    s3 := Te0(byte(Shift_Right(t3, 24))) xor Te1(byte(Shift_Right(t0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t1, 8) and 16#ff#)) xor
Te3(byte(t2 and 16#ff#)) xor rk(35);

    t0 := Te0(byte(Shift_Right(s0, 24))) xor Te1(byte(Shift_Right(s1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
Te3(byte(s3 and 16#ff#)) xor rk(36);
    t1 := Te0(byte(Shift_Right(s1, 24))) xor Te1(byte(Shift_Right(s2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
Te3(byte(s0 and 16#ff#)) xor rk(37);
    t2 := Te0(byte(Shift_Right(s2, 24))) xor Te1(byte(Shift_Right(s3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
Te3(byte(s1 and 16#ff#)) xor rk(38);
    t3 := Te0(byte(Shift_Right(s3, 24))) xor Te1(byte(Shift_Right(s0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
Te3(byte(s2 and 16#ff#)) xor rk(39);

    if (Nr > 10) then
        s0 := Te0(byte(Shift_Right(t0, 24))) xor Te1(byte(Shift_Right(t1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t2, 8) and 16#ff#)) xor
Te3(byte(t3 and 16#ff#)) xor rk(40);
        s1 := Te0(byte(Shift_Right(t1, 24))) xor Te1(byte(Shift_Right(t2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t3, 8) and 16#ff#)) xor
Te3(byte(t0 and 16#ff#)) xor rk(41);
        s2 := Te0(byte(Shift_Right(t2, 24))) xor Te1(byte(Shift_Right(t3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t0, 8) and 16#ff#)) xor
Te3(byte(t1 and 16#ff#)) xor rk(42);
        s3 := Te0(byte(Shift_Right(t3, 24))) xor Te1(byte(Shift_Right(t0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(t1, 8) and 16#ff#)) xor
Te3(byte(t2 and 16#ff#)) xor rk(43);

```

```

    t0 := Te0(byte(Shift_Right(s0, 24))) xor Tel(byte(Shift_Right(s1, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
Te3(byte(s3 and 16#ff#)) xor rk(44);
    t1 := Te0(byte(Shift_Right(s1, 24))) xor Tel(byte(Shift_Right(s2, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
Te3(byte(s0 and 16#ff#)) xor rk(45);
    t2 := Te0(byte(Shift_Right(s2, 24))) xor Tel(byte(Shift_Right(s3, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
Te3(byte(s1 and 16#ff#)) xor rk(46);
    t3 := Te0(byte(Shift_Right(s3, 24))) xor Tel(byte(Shift_Right(s0, 16)
and 16#ff#)) xor Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
Te3(byte(s2 and 16#ff#)) xor rk(47);

    if (Nr > 12) then
        s0 := Te0(byte(Shift_Right(t0, 24))) xor Tel(byte(Shift_Right(t1,
16) and 16#ff#)) xor Te2(byte(Shift_Right(t2, 8) and 16#ff#)) xor
Te3(byte(t3 and 16#ff#)) xor rk(48);
        s1 := Te0(byte(Shift_Right(t1, 24))) xor Tel(byte(Shift_Right(t2,
16) and 16#ff#)) xor Te2(byte(Shift_Right(t3, 8) and 16#ff#)) xor
Te3(byte(t0 and 16#ff#)) xor rk(49);
        s2 := Te0(byte(Shift_Right(t2, 24))) xor Tel(byte(Shift_Right(t3,
16) and 16#ff#)) xor Te2(byte(Shift_Right(t0, 8) and 16#ff#)) xor
Te3(byte(t1 and 16#ff#)) xor rk(50);
        s3 := Te0(byte(Shift_Right(t3, 24))) xor Tel(byte(Shift_Right(t0,
16) and 16#ff#)) xor Te2(byte(Shift_Right(t1, 8) and 16#ff#)) xor
Te3(byte(t2 and 16#ff#)) xor rk(51);

        t0 := Te0(byte(Shift_Right(s0, 24))) xor Tel(byte(Shift_Right(s1,
16) and 16#ff#)) xor Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
Te3(byte(s3 and 16#ff#)) xor rk(52);
        t1 := Te0(byte(Shift_Right(s1, 24))) xor Tel(byte(Shift_Right(s2,
16) and 16#ff#)) xor Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
Te3(byte(s0 and 16#ff#)) xor rk(53);
        t2 := Te0(byte(Shift_Right(s2, 24))) xor Tel(byte(Shift_Right(s3,
16) and 16#ff#)) xor Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
Te3(byte(s1 and 16#ff#)) xor rk(54);
        t3 := Te0(byte(Shift_Right(s3, 24))) xor Tel(byte(Shift_Right(s0,
16) and 16#ff#)) xor Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
Te3(byte(s2 and 16#ff#)) xor rk(55);
        end if;
    end if;

    s0 := (Te4(byte(Shift_Right(t0,24)
)) and 16#ff000000#) xor
(Te4(byte(Shift_Right(t1,16) and 16#ff#)) and 16#00ff0000#) xor
(Te4(byte(Shift_Right(t2, 8) and 16#ff#)) and 16#0000ff00#) xor
(Te4(byte(
t3 and 16#ff#)) and 16#000000ff#) xor
rk(4*Nr );
    s1 := (Te4(byte(Shift_Right(t1,24)
)) and 16#ff000000#) xor
(Te4(byte(Shift_Right(t2,16) and 16#ff#)) and 16#00ff0000#) xor
(Te4(byte(Shift_Right(t3, 8) and 16#ff#)) and 16#0000ff00#) xor
(Te4(byte(
t0 and 16#ff#)) and 16#000000ff#) xor
rk(4*Nr+1);
    s2 := (Te4(byte(Shift_Right(t2,24)
)) and 16#ff000000#) xor
(Te4(byte(Shift_Right(t3,16) and 16#ff#)) and 16#00ff0000#) xor

```

```

        (Te4(byte(Shift_Right(t0, 8) and 16#ff#)) and 16#0000ff00#) xor
        (Te4(byte(
                    t1          and 16#ff#)) and 16#000000ff#) xor
rk(4*Nr+2);
s3 := (Te4(byte(Shift_Right(t3,24)
                    )) and 16#ff000000#) xor
        (Te4(byte(Shift_Right(t0,16) and 16#ff#)) and 16#00ff0000#) xor
        (Te4(byte(Shift_Right(t1, 8) and 16#ff#)) and 16#0000ff00#) xor
        (Te4(byte(
                    t2          and 16#ff#)) and 16#000000ff#) xor
rk(4*Nr+3);

ct := block'(others => 0);
SplitWord(s0, b0, b1, b2, b3);
ct( 0) := b0; ct( 1) := b1; ct( 2) := b2; ct( 3) := b3;
SplitWord(s1, b0, b1, b2, b3);
ct( 4) := b0; ct( 5) := b1; ct( 6) := b2; ct( 7) := b3;
SplitWord(s2, b0, b1, b2, b3);
ct( 8) := b0; ct( 9) := b1; ct(10) := b2; ct(11) := b3;
SplitWord(s3, b0, b1, b2, b3);
ct(12) := b0; ct(13) := b1; ct(14) := b2; ct(15) := b3;
end aesEncrypt;

```

B.2.2 AES Encrypt Function after Loop Rerolling

Following is the AES encrypt function after loop rerolling was applied:

```

procedure aesEncrypt(rk: in key_schedule; Nr: in Integer; pt: in block;
ct: out block)
is
    s0, s1, s2, s3, t0, t1, t2, t3: word;
    b0, b1, b2, b3: byte;
begin
    s0 := CombineWord(pt( 0), pt( 1), pt( 2), pt( 3)) xor rk(0);
    s1 := CombineWord(pt( 4), pt( 5), pt( 6), pt( 7)) xor rk(1);
    s2 := CombineWord(pt( 8), pt( 9), pt(10), pt(11)) xor rk(2);
    s3 := CombineWord(pt(12), pt(13), pt(14), pt(15)) xor rk(3);

    for r in Natural range 1 .. Nr-1 loop
        t0 := Te0(byte(Shift_Right(s0,24)
                    )) xor
            Te1(byte(Shift_Right(s1,16) and 16#ff#)) xor
            Te2(byte(Shift_Right(s2, 8) and 16#ff#)) xor
            Te3(byte(
                    s3          and 16#ff#)) xor
            rk(4*r );
        t1 := Te0(byte(Shift_Right(s1,24)
                    )) xor
            Te1(byte(Shift_Right(s2,16) and 16#ff#)) xor
            Te2(byte(Shift_Right(s3, 8) and 16#ff#)) xor
            Te3(byte(
                    s0          and 16#ff#)) xor
            rk(4*r+1);
        t2 := Te0(byte(Shift_Right(s2,24)
                    )) xor
            Te1(byte(Shift_Right(s3,16) and 16#ff#)) xor
            Te2(byte(Shift_Right(s0, 8) and 16#ff#)) xor
            Te3(byte(
                    s1          and 16#ff#)) xor
            rk(4*r+2);
        t3 := Te0(byte(Shift_Right(s3,24)
                    )) xor
            Te1(byte(Shift_Right(s0,16) and 16#ff#)) xor

```

```

        Te2(byte(Shift_Right(s1, 8) and 16#ff#)) xor
        Te3(byte(
                s2          and 16#ff#)) xor
        rk(4*r+3);
    s0 := t0; s1 := t1; s2 := t2; s3 := t3;
end loop;

t0 := (Te4(byte(Shift_Right(s0,24)
                )) and 16#ff000000#) xor
      (Te4(byte(Shift_Right(s1,16) and 16#ff#)) and 16#00ff0000#) xor
      (Te4(byte(Shift_Right(s2, 8) and 16#ff#)) and 16#0000ff00#) xor
      (Te4(byte(
                s3          and 16#ff#)) and 16#000000ff#) xor
      rk(4*Nr );
t1 := (Te4(byte(Shift_Right(s1,24)
                )) and 16#ff000000#) xor
      (Te4(byte(Shift_Right(s2,16) and 16#ff#)) and 16#00ff0000#) xor
      (Te4(byte(Shift_Right(s3, 8) and 16#ff#)) and 16#0000ff00#) xor
      (Te4(byte(
                s0          and 16#ff#)) and 16#000000ff#) xor
      rk(4*Nr+1);
t2 := (Te4(byte(Shift_Right(s2,24)
                )) and 16#ff000000#) xor
      (Te4(byte(Shift_Right(s3,16) and 16#ff#)) and 16#00ff0000#) xor
      (Te4(byte(Shift_Right(s0, 8) and 16#ff#)) and 16#0000ff00#) xor
      (Te4(byte(
                s1          and 16#ff#)) and 16#000000ff#) xor
      rk(4*Nr+2);
t3 := (Te4(byte(Shift_Right(s3,24)
                )) and 16#ff000000#) xor
      (Te4(byte(Shift_Right(s0,16) and 16#ff#)) and 16#00ff0000#) xor
      (Te4(byte(Shift_Right(s1, 8) and 16#ff#)) and 16#0000ff00#) xor
      (Te4(byte(
                s2          and 16#ff#)) and 16#000000ff#) xor
      rk(4*Nr+3);
s0 := t0; s1 := t1; s2 := t2; s3 := t3;

ct := block'(others => 0);
SplitWord(s0, b0, b1, b2, b3);
ct( 0) := b0; ct( 1) := b1; ct( 2) := b2; ct( 3) := b3;
SplitWord(s1, b0, b1, b2, b3);
ct( 4) := b0; ct( 5) := b1; ct( 6) := b2; ct( 7) := b3;
SplitWord(s2, b0, b1, b2, b3);
ct( 8) := b0; ct( 9) := b1; ct(10) := b2; ct(11) := b3;
SplitWord(s3, b0, b1, b2, b3);
ct(12) := b0; ct(13) := b1; ct(14) := b2; ct(15) := b3;
end aesEncrypt;

```

B.2.3 Final AES Encrypt Function

Following is the final refactored AES encrypt function with optimizations of word packing, table lookup, and function inlining were all reversed:

```

procedure aesEncrypt(rk: in key_schedule; Nr: in Integer; pt: in block;
ct: out block)
is
    st: state;
begin
    st := AddRoundKey(Block2State(pt), rk(0), rk(1), rk(2), rk(3));

    for r in Natural range 1 .. Nr-1 loop

```

```
    st := AddRoundKey(MixColumns(ShiftRows(SubBytes(st))), rk(4*r),  
rk(4*r+1), rk(4*r+2), rk(4*r+3));  
    end loop;  
  
    st := AddRoundKey(ShiftRows(SubBytes(st)), rk(4*Nr), rk(4*Nr+1),  
rk(4*Nr+2), rk(4*Nr+3));  
  
    ct := State2Block(st);  
end aesEncrypt;
```

Appendix C: Example Direct Specification Extraction from Code

C.1 Specification Extraction of Tokeneer

The Tokeneer source program are not completed annotated with precondition and postcondition annotations. However, the program is mainly about state transitions (and to avoid insecure states) with little calculations. Augmenting the program with complete annotation is feasible but not helpful in abstracting out implementation details. Specification extraction was thus performed directly on top of the code. Following is an example of the extracted specification corresponding to the `UserEntry` package:

```
UserEntry: THEORY
BEGIN

IMPORTING AuditLog
IMPORTING Bio
IMPORTING CertificateStore
IMPORTING Clock
IMPORTING ConfigData
IMPORTING Display
IMPORTING Door
IMPORTING KeyStore
IMPORTING Latch
IMPORTING Stats
IMPORTING UserToken

prf_UserEntryUnlockDoor: Boolean

%% Traceto: FD.UserEntry.UserHasDeparted
UserHasDeparted(st: State): Boolean =
  st`UserEntry_Status > Quiescent and not UserToken.IsPresent(st)

%% Traceto: FD.UserEntry.UserTokenTorn
UserTokenTorn(TheStats: Stats_T, st: State): [# TheStats: Stats_T,
                                              st: State #] =
  LET st1 = AuditLog.AddElementToLog(UserTokenRemoved, Warning,
                                     UserToken.ExtractUser(st), NoDescription, st) IN
  LET st2 = Display.SetValue(Welcome, st1) IN
  LET st3 = st2 WITH [`UserEntry_Status := Quiescent] IN
  (# `TheStats := Stats.AddFailedEntry(TheStats),
   `st := UserToken.Clear(st3) #)

%% Traceto: FD.UserEntry.TISReadUserToken
%% Traceto: FD.UserEntry.BioCheckNotRequired
%% Traceto: FD.UserEntry.BioCheckRequired
%% Traceto: FD.UserEntry.ValidateUserEntryFail
%% Traceto: FD.UserEntry.UserTokenTorn
ValidateUserToken(TheStats: Stats_T, st: State): [# TheStats: Stats_T,
```

```

                                                                    st: State #] =
IF NOT UserToken.IsPresent(st) THEN
    UserTokenTorn(TheStats, st)
ELSE
    LET AuthCertOK = UserToken.ReadAndCheckAuthCert(st) `AuthCertOK,
        st1 = UserToken.ReadAndCheckAuthCert(st) `st IN
    IF AuthCertOK THEN
        LET st2 = AuditLog.AddElementToLog(UserTokenPresent, Information,
            UserToken.ExtractUser(st1), NoDescription, st1) IN
        LET st3 = AuditLog.AddElementToLog(AuthCertValid, Information,
            UserToken.ExtractUser(st2), NoDescription, st2) IN
        LET st4 = Display.SetValue(Wait, st3) IN
        LET st5 = st4 WITH [`UserEntry_Status := WaitingEntry] IN
        (# `TheStats := TheStats,
            `st := st5 #)
    ELSE
        LET Description = UserToken.ReadAndCheck(st1) `Description,
            TokenOK = UserToken.ReadAndCheck(st1) `TokenOK,
            st2 = UserToken.ReadAndCheck(st1) `st IN
        IF TokenOK THEN
            LET st3 = AuditLog.AddElementToLog(UserTokenPresent, Information,
                UserToken.ExtractUser(st2), NoDescription, st2) IN
            LET st4 = AuditLog.AddElementToLog(AuthCertInvalid, Information,
                UserToken.ExtractUser(st3), NoDescription, st3) IN
            LET st5 = Display.SetValue(InsertFinger, st4) IN
            LET st6 = st5 WITH [`UserEntry_Status := WaitingFinger,
                `FingerTimeout := Clock.AddDuration(
                    Clock.TheCurrentTime(st5),
                    ConfigData.TheFingerWaitDuration(st5))] IN
            LET st7 = Bio.Flush(st6) IN
            (# `TheStats := TheStats,
                `st := st7 #)
        ELSE
            LET st3 = AuditLog.AddElementToLog(UserTokenPresent, Information,
                UserToken.ExtractUser(st2), NoDescription, st2) IN
            LET st4 = AuditLog.AddElementToLog(UserTokenInvalid, Warning,
                UserToken.ExtractUser(st3), NoDescription, st3) IN
            LET st5 = Display.SetValue(RemoveToken, st4) IN
            LET st6 = st5 WITH[`UserEntry_Status :=
                WaitingRemoveTokenFail] IN
            (# `TheStats := TheStats,
                `st := st6 #)
        ENDIF
    ENDIF
ENDIF
ENDIF

%% Traceto: FD.UserEntry.ReadFingerOK
%% Traceto: FD.UserEntry.NoFinger
%% Traceto: FD.UserEntry.FingerTimeout
%% Traceto: FD.UserEntry.UserTokenTorn
ReadFinger_State_pre(st: State): bool =
    st `UserEntry_Status = WaitingFinger
ReadFinger(TheStats: Stats_T,
    st: (ReadFinger_State_pre)): [# TheStats: Stats_T,

```

```

                                st: State #] =
IF NOT UserToken.IsPresent(st) THEN
    UserTokenTorn(TheStats, st)
ELSE
    IF Clock.GreaterThan(Clock.TheCurrentTime(st), st`FingerTimeout)
THEN
        LET st1 = AuditLog.AddElementToLog(FingerTimeout, Warning,
                                UserToken.ExtractUser(st), NoDescription, st) IN
        LET st2 = Display.SetValue(RemoveToken, st1) IN
        LET st3 = st2 WITH [`UserEntry_Status := WaitingRemoveTokenFail] IN
        (# `TheStats := TheStats,
        `st := st3 #)
    ELSE
        LET FingerPresence = Bio.Poll(st) IN
        IF FingerPresence = Present THEN
            LET st1 = AuditLog.AddElementToLog(FingerDetected, Information,
                                UserToken.ExtractUser(st), NoDescription, st) IN
            LET st2 = Display.SetValue(Wait, st1) IN
            LET st3 = st2 WITH [`UserEntry_Status := GotFinger] IN
            (# `TheStats := TheStats,
            `st := st3 #)
        ELSE
            (# `TheStats := TheStats,
            `st := st #)
        ENDIF
    ENDIF
ENDIF
ENDIF
ENDIF

```

AchievedFARDescription(AchievedFAR: FarT): DescriptionT

```

%% Traceto: FD.UserEntry.ValidateFingerOK
%% Traceto: FD.UserEntry.ValidateFingerFail
%% Traceto: FD.UserEntry.UserTokenTorn
ValidateFinger(TheStats: Stats_T, st: State): [# TheStats: Stats_T,
                                                st: State #] =

    IF NOT UserToken.IsPresent(st) THEN
        UserTokenTorn(TheStats, st)
    ELSE
        LET TheTemplate = UserToken.GetIandATemplate(st) IN
        LET MaxFar =
            IF TheTemplate`RequiredMaxFAR < ConfigData.TheSystemMaxFar(st)
            THEN TheTemplate`RequiredMaxFAR
            ELSE ConfigData.TheSystemMaxFar(st)
            ENDIF IN
        LET MatchResult = Bio.Verify(TheTemplate, MaxFar, st)`MatchResult,
        AchievedFAR = Bio.Verify(TheTemplate, MaxFar, st)`AchievedFAR,
        st1 = Bio.Verify(TheTemplate, MaxFar, st)`st IN
        LET st2 = Bio.Flush(st1) IN
        IF MatchResult = Match THEN
            LET st3 = AuditLog.AddElementToLog(FingerMatched, Information,
                                UserToken.ExtractUser(st2),
                                AchievedFARDescription(AchievedFAR), st2) IN
            LET st4 = Display.SetValue(Wait, st3) IN
            LET st5 = st4 WITH [`UserEntry_Status := WaitingUpdateToken] IN

```

```

        (# `TheStats := Stats.AddSuccessfulBio(TheStats),
        `st := st5 #)
ELSE
    LET st3 = AuditLog.AddElementToLog(FingerNotMatched, Warning,
        UserToken.ExtractUser(st2),
        AchievedFARDescription(AchievedFAR), st2) IN
    LET st4 = Display.SetValue(RemoveToken, st3) IN
    LET st5 = st4 WITH [`UserEntry_Status := WaitingRemoveTokenFail] IN
    (# `TheStats := Stats.AddFailedBio(TheStats),
    `st := st5 #)
ENDIF
ENDIF

%% Traceto: FD.UserEntry.UpdateUserTokenNotRequired
%% Traceto: FD.UserEntry.WriteUserTokenOK
%% Traceto: FD.UserEntry.WriteUserTokenFail
%% Traceto: FD.UserEntry.UserTokenTorn
UpdateToken(TheStats: Stats_T, st: (KeyStore.PrivateKeyPresent)):
    [# TheStats: Stats_T,
    st: State #] =
    IF NOT UserToken.IsPresent(st) THEN
        UserTokenTorn(TheStats, st)
    ELSE
        LET UpdateOK = UserToken.AddAuthCert(st) `Success,
            st1 = UserToken.AddAuthCert(st) `st IN
        LET st2 = st1 WITH [`UserEntry_Status := WaitingEntry] IN
        LET UpdateOK1 = IF UpdateOK
            THEN UserToken.UpdateAuthCert(st2) `Success
            ELSE FALSE
            ENDIF,
            st3 = IF UpdateOK
                THEN UserToken.UpdateAuthCert(st2) `st
                ELSE st2
            ENDIF IN
        IF UpdateOK1 THEN
            LET st4 = AuditLog.AddElementToLog(AuthCertWritten, Information,
                UserToken.ExtractUser(st3), NoDescription, st3) IN
            LET st5 = Display.SetValue(Wait, st4) IN
            LET st6 = CertificateStore.UpdateStore(st5) IN
            (# `TheStats := TheStats,
            `st := st6 #)
        ELSE
            LET st4 = AuditLog.AddElementToLog(AuthCertWriteFailed, Warning,
                UserToken.ExtractUser(st3), NoDescription, st3) IN
            LET st5 = Display.SetValue(TokenUpdateFailed, st4) IN
            (# `TheStats := TheStats,
            `st := st5 #)
        ENDIF
    ENDIF
ENDIF

%% Traceto: FD.UserEntry.EntryOK
%% Traceto: FD.UserEntry.EntryNotAllowedC
%% Traceto: FD.UserEntry.UserTokenTorn
ValidateEntry(TheStats: Stats_T, st: State): [# TheStats: Stats_T,

```

```

                                                                    st: State #] =
IF NOT UserToken.IsPresent(st) THEN
  UserTokenTorn(TheStats, st)
ELSE
  IF ConfigData.IsInEntryPeriod(st, UserToken.GetClass(st),
                                Clock.TheCurrentTime(st)) THEN
    LET st1 = AuditLog.AddElementToLog(EntryPermitted, Information,
                                       UserToken.ExtractUser(st), NoDescription, st) IN
    LET st2 = Display.SetValue(OpenDoor, st1) IN
    LET st3 = st2 WITH [`UserEntry_Status := WaitingRemoveTokenSuccess,
                        `TokenRemovalTimeout := Clock.AddDuration(
                            Clock.TheCurrentTime(st2),
                            ConfigData.TheTokenRemovalDuration(st2))] IN
    (# `TheStats := TheStats,
     `st := st3 #)
  ELSE
    LET st1 = AuditLog.AddElementToLog(EntryDenied, Warning,
                                       UserToken.ExtractUser(st), NoDescription, st) IN
    LET st2 = Display.SetValue(RemoveToken, st1) IN
    LET st3 = st2 WITH [`UserEntry_Status := WaitingRemoveTokenFail] IN
    (# `TheStats := TheStats,
     `st := st3 #)
  ENDIF
ENDIF
ENDIF

%% Traceto : FD.UserEntry.UnlockDoorOK
%% Traceto : FD.UserEntry.WaitingTokenRemoval
%% Traceto : FD.UserEntry.TokenRemovalTimeout
UnlockDoor_pre(TheStats: Stats_T, st: State): bool =
  st`UserEntry_Status = WaitingRemoveTokenSuccess AND
  ((Latch.IsLocked(st) AND
   Door.TheCurrentDoor(st) = Open AND
   Clock.GreaterThanOrEqual(Clock.TheCurrentTime(st),
                           Door.prf_alarmTimeout(st))) <=>
   Door.TheDoorAlarm(st) = Alarming)
UnlockDoor_post(TheStats_: Stats_T, st_: State,
                TheStats: Stats_T, st: State): bool =
  ((Latch.IsLocked(st) AND
   Door.TheCurrentDoor(st) = Open AND
   Clock.GreaterThanOrEqual(Clock.TheCurrentTime(st),
                           Door.prf_alarmTimeout(st))) <=>
   Door.TheDoorAlarm(st) = Alarming) AND
  ((Latch.IsLocked(st_) AND NOT Latch.IsLocked(st))
   <=> prf_UserEntryUnlockDoor)
UnlockDoor_State_pre(st: State): bool =
  st`UserEntry_Status = WaitingRemoveTokenSuccess
UnlockDoor(TheStats: Stats_T,
           st: (UnlockDoor_State_pre)): [# TheStats: Stats_T,
                                       st: State #] =

IF NOT UserToken.IsPresent(st) THEN
  LET st1 = Door.UnlockDoor(st) IN
  LET st2 = UserToken.Clear(st1) IN
  LET st3 = Display.SetValue(DoorUnlocked, st2) IN
  LET st4 = st3 WITH [`UserEntry_Status := Quiescent] IN

```

```

    (# `TheStats := Stats.AddSuccessfulEntry(TheStats),
      `st := st4 #)
ELSE
  IF Clock.GreaterThan(Clock.TheCurrentTime(st),
    st`TokenRemovalTimeout) THEN
    LET st1 = AuditLog.AddElementToLog(EntryTimeout, Warning,
      UserToken.ExtractUser(st), NoDescription, st) IN
    LET st2 = Display.SetValue(RemoveToken, st1) IN
    LET st3 = st2 WITH [`UserEntry_Status := WaitingRemoveTokenFail] IN
    (# `TheStats := TheStats,
      `st := st3 #)
  ELSE
    (# `TheStats := TheStats,
      `st := st #)
  ENDIF
ENDIF
UnlockDoor: LEMMA
FORALL (TheStats_: Stats_T, st_: State):
  UnlockDoor_pre(TheStats_, st_)
  => UnlockDoor_post(TheStats_, st_, result`TheStats, result`st)
  WHERE result = UnlockDoor(TheStats_, st_)

%% Traceto: FD.UserEntry.FailedAccessTokenRemoved
FailedAccessTokenRemoved(TheStats: Stats_T,
  st: State): [# TheStats: Stats_T,
  st: State #] =
  LET st1 = AuditLog.AddElementToLog(UserTokenRemoved, Information,
    UserToken.ExtractUser(st), NoDescription, st) IN
  LET st2 = Display.SetValue(Welcome, st1) IN
  LET st3 = st2 WITH [`UserEntry_Status := Quiescent] IN
  (# `TheStats := Stats.AddFailedEntry(TheStats),
    `st := UserToken.Clear(st3) #)

%% Traceto: FD.UserEntry.UserEntryInProgress
InProgress(st: State): Boolean =
  st`UserEntry_Status > Quiescent AND
  st`UserEntry_Status < WaitingRemoveTokenFail

%% traceto: FD.UserEntry.CurrentUserEntryActivityPossible
CurrentActivityPossible(st: State): Boolean =
  InProgress(st) OR UserHasDeparted(st)

%% traceto: FD.UserEntry.UserEntryCanStart
CanStart(st: State): Boolean =
  st`UserEntry_Status = Quiescent AND UserToken.IsPresent(st)

%% traceto: FD.Interface.DisplayPollUpdate
DisplayPollUpdate(st: State): State =
  IF Latch.IsLocked(st) THEN
    LET NewMsg = IF st`UserEntry_Status = WaitingRemoveTokenFail
      THEN RemoveToken
      ELSE Welcome
    ENDIF IN
    Display.SetValue(NewMsg, st)
  ELSE
    st
  ENDIF

```

```

ELSE
    st
ENDIF

%% traceto: FD.UserEntry.ProgressUserEntry
Progress_pre(TheStats: Stats_T, st: State): bool =
    KeyStore.PrivateKeyPresent(st) AND
    st`UserEntry_Status > Quiescent AND
    (st`UserEntry_Status = WaitingRemoveTokenFail
    => NOT UserToken.IsPresent(st)) AND
    ((Latch.IsLocked(st) AND
    Door.TheCurrentDoor(st) = Open AND
    Clock.GreaterThanOrEqualTo(Clock.TheCurrentTime(st),
    Door.prf_alarmTimeout(st))) <=>
    Door.TheDoorAlarm(st) = Alarming)
Progress_post(TheStats_: Stats_T, st_: State,
    TheStats: Stats_T, st: State): bool =
    ((Latch.IsLocked(st) AND
    Door.TheCurrentDoor(st) = Open AND
    Clock.GreaterThanOrEqualTo(Clock.TheCurrentTime(st),
    Door.prf_alarmTimeout(st))) <=>
    Door.TheDoorAlarm(st) = Alarming) AND
    ((Latch.IsLocked(st_) AND NOT Latch.IsLocked(st))
    <=> prf_UserEntryUnlockDoor)
Progress_State_pre(st: State): bool =
    KeyStore.PrivateKeyPresent(st) AND
    st`UserEntry_Status > Quiescent AND
    (st`UserEntry_Status = WaitingRemoveTokenFail => NOT
    UserToken.IsPresent(st))
Progress(TheStats:
    Stats_T, st: (Progress_State_pre)): [# TheStats: Stats_T,
    st: State #] =

    LET LocalStatus = st`UserEntry_Status IN
    CASES LocalStatus OF
        GotUserToken:          ValidateUserToken(TheStats, st),
        WaitingFinger:         ReadFinger(TheStats, st),
        GotFinger:             ValidateFinger(TheStats, st),
        WaitingUpdateToken:    UpdateToken(TheStats, st),
        WaitingEntry:          ValidateEntry(TheStats, st),
        WaitingRemoveTokenSuccess: UnlockDoor(TheStats, st),
        WaitingRemoveTokenFail:   FailedAccessTokenRemoved(TheStats, st)
    ENDCASES
Progress: LEMMA
    FORALL (TheStats_: Stats_T, st_: State):
        Progress_pre(TheStats_, st_)
        => Progress_post(TheStats_, st_, result`TheStats, result`st)
        WHERE result = Progress(TheStats_, st_)

%% traceto: FD.UserEntry.TISReadUserToken
StartEntry(st: State): State =
    LET st1 = Display.SetValue(Wait, st) IN
    st1 WITH [`UserEntry_Status := GotUserToken]

END UserEntry

```

Appendix D: Example Obligation Partition in Synergistic Analysis

D.1 Synergistic Analysis of Tokeneer Requirements

In order to demonstrate the synergistic analysis approach, my colleague Ben Taitelbaum and I conducted a small case study based on Tokeneer ID Station (TIS). A representative subset (two) of the TIS security requirements defined by the NSA were selected for study. We refined each requirement into a set of more detailed obligations that was partitioned into static and dynamic subsets. Echo could thus be applied to prove the static part of the obligations.

The natural language and PVS statements of the first requirement that we chose to use in this example, R_1 , are shown in Figure 18. The statements for the second, R_2 , are shown in Figure 19. R_1 states that the correct sequence of checking actions will be taken by the TIS, and R_2 states that the alarm will be raised if the system is deemed to be “insecure”. Each of these requirements is dealt with most conveniently by partitioning into a set of shorter, more manageable obligations that are, in fact, defined in the original NSA requirements. The results of this partitioning are shown in the respective figures for R_1 and R_2 . To illustrate synergistic assurance, for each of these obligations we present the choice we made (static or dynamic) for the associated assurance and the rationale for each choice.

R_1 is a requirement about the user authentication and entry process. The user has to insert a card into a reader for authentication and, depending on the user and his or her circumstances, also present a finger for fingerprint checking. The process is multi-staged, and is documented as a finite-state machine with transitions on eight internal states: `quiescent`, `gotUserToken`, `waitingFinger`, `gotFinger`, `waitingUpdateToken`, `waitingEntry`, `waitingRemoveTokenSuccess`, and `waitingRemoveTokenFail`. Based

on the transitions of these internal states, we partitioned R_1 into the obligations shown in Figure 18 and dealt with these obligations as follows:

TIS Requirement 1:

If the latch is unlocked by the TIS and the TIS is not in possession of an Admin Token, then the TIS must be in possession of a User Token. The User Token must either have a valid Authorization Certificate, or must have valid ID, Privilege, and I&A Certificates, together with a template that allows the TIS to successfully validate the user's fingerprint.

R1: THEOREM

```
FORALL (f: StateChange):
  (FORALL (st: SystemState):
    st`Latch = locked AND
    f(st)`Latch = unlocked AND
    st`adminTokenPresence = absent
    =>
    TokenWithValidAuth(st`currentUserToken) AND
    UserTokenWithOKAuthCert(st`currentUserToken, st`currentTime)
    OR
    ValidToken(st`currentUserToken) AND
    UserTokenOK(st`currentUserToken, st`currentTime) AND
    FingerOK(st`currentFinger))
```

Obligation 1:

The enclave door is only unlocked in the `waitingRemoveTokenSuccess` state.

Obligation 2:

In order to reach the state `waitingRemoveTokenSuccess`, the state transitions must either follow: (1) `gotUserToken` \square `waitingFinger` \square `gotFinger` \square `waitingUpdateToken` \square `waitingEntry` \square `waitingRemoveTokenSuccess`, or: (2) `gotUserToken` \square `waitingEntry` \square `waitingRemoveTokenSuccess`.

Obligation3:

In the first path: `ValidToken` and `UserTokenOK` are checked in the transition to `waitingFinger`, `FingerOK` is checked in the transition to `waitingUpdateToken`. In the second path: `TokenWithValidAuth` and `UserTokenWithOKAuthCert` are checked in the transition to `waitingEntry`.

Obligation 4:

User token is not changed during the transitions from `gotUserToken` to `waitingRemoveTokenSuccess`, and user fingerprint is not changed during the transitions from `gotFinger` to `waitingRemoveTokenSuccess`.

Figure 18. TIS requirement R_1 and associated obligations

R₁ - Obligation 1:

This obligation is concerned with actually opening the enclave door and is obviously crucial. It is, however, easy to check both statically and dynamically, and so we checked it statically, the preferred approach. The check was completed by translation of the obligation into simple function postconditions.

R₁ - Obligation 2:

This obligation is simple to understand, but to verify it statically involves a lot of detail. It is necessary to prove that the postcondition of every function that can change the state satisfies the obligation, and some functions define more than one state change. Function level granularity might not be sufficient unless we perform full verification. On the other hand, checking this obligation dynamically only requires checking the assignment of the state. Thus we discharged this obligation dynamically.

R₁ - Obligation 3:

This obligation is concerned with the conduct of the various authentication mechanisms (for example, checking the fingerprint) as state transitions occur. It is obviously critical that the right checks occur in the right way at the right place and at the right time. Since the checks are tied to state transitions and implemented as predicate functions, the required actions are quite easy to check statically. If an attempt were made to check them dynamically, then the checking function would essentially have to duplicate the existing predicate functions because the obligation is to actually confirm the truth of something like the fingerprint check not merely that it was conducted. Thus the dynamic check would have to fully invoke the fingerprint-checking hardware. The verification of the check would be as complex as the static verification of the implementation. The static verifica-

tion has to prove that the hardware is used correctly and that its results are used to influence the flow of control correctly. We chose the static route.

R₁ - Obligation 4:

This obligation is concerned with the possibility of an attacker trying to switch artifacts in mid-authentication. The whole authentication process relies on continuity of trust between states. If a switch occurs part way through the process, the credentials associated with one individual might be hijacked by another. That the implementation ensures this obligation is, therefore essential. The obligation could be checked either statically or dynamically with roughly equal effort, and we chose statically.

R₂ is a requirement about the timing of the TIS. Time is an important part of the overall security mechanism. For example, if the enclave door were left open for an inordinate amount of time, it might be possible for an attacker to gain access without being noticed because an authorized user had entered at some point in the past.

TIS Requirement 2:

An alarm will be raised whenever the door/latch is insecure. “Insecure” is defined to mean the latch is locked, the door is open, and too much time has passed since the last explicit request to lock the latch.

```
R2: THEOREM
  FORALL (st: SystemState):
    st`Latch = locked AND
    st`Door = open AND
    st`currentTime >= st`alarmTimeout
    =>
    st`Alarm = alarming
```

Obligation 1:

The value `alarmTimeout` complies with the specification.

Obligation 2:

The alarm will sound when the door/latch is insecure.

Figure 19. TIS requirement R₂ and associated obligations

R₂ - Obligation 1:

This obligation is concerned with ensuring that the actual time-out value in the system is the one that was specified. The concern is with the possibility of the time-out value being changed incorrectly by a software defect. The obligation maps to the postconditions of those functions modifying the value of `alarmTimeout`. Either static verification or a dynamic check could be used for this obligation, each with little effort, and we used static verification.

R₂ - Obligation 2:

This obligation is a real-time requirement that is difficult to deal with statically. To discharge this obligation with static methods requires determination of worst-case execution time bounds on various sections of the software, a complete proof of functionality, a proof that machine time is an adequate representation of real-world time, etc. By contrast, provided machine time is an adequate representation of real-world time, a dynamic check is simple to implement and to verify. A timer could be set to halt the system when it goes off.

In summary, these six obligations derive from realistic requirements for a system that has serious consequences of failure. The partitioning and the associated rationale suggests that splitting the proof burden between static and dynamic techniques is quite feasible and is potentially effective as a way of producing a useful level and form of assurance. The Echo approach was used to prove the static part of the obligations and was seamlessly integrated with the dynamic checks. This increases the expressive power of Echo verification to allow a more comprehensive whole-system assurance arguments to be constructed.