DEPENDABLE
COMPUTING

Reengineering of a Software Module using Formal Specification

A Microstandard

Version 1

Jonathan Rowanhill
A. Benjamin Hocking

December 31, 2021

1 Introduction

This document provides guidance for the assured reengineering of a software module using formal specification and source code synthesis.

1.1 Purpose

When the design and implementation of software in a system reaches a high level of maturity, then it is often possible to identify the software modules of high importance to the system's critical functions. It is often then that software engineers consider investing resources to reengineer these critical modules with improved focus on their **dependability**, including improving module *safety*, *security*, *reliability*, *availability*, and *maintainability* [1].

One option available to the engineer is the application of *formal methods.* Using formal methods, the engineer can prove dependability properties on reengineered software design and implementation.

This guide focuses on the application of a formal methods approach called *formal specification*. In this approach engineers rewrite their software module in a formal specification language. They then use proof assistance tools to prove desired properties of the specification. Then engineers can use careful synthesis approaches to implement the design in a software language. Using a software language supporting proof capabilities, such as Frama-C [2] or SPARK Pro (Ada) [2], they can then prove correctness properties for their implementation against their formal specification.

When performed correctly, the result is highly assured software, where specifically proven properties of the software are extremely likely to hold in the compiled executable.

Formal methods are therefore an extremely powerful means to reengineer more assured software modules. The challenge of applying formal methods, however, is three-fold:

- **Experience**: Applying formal specification and writing proofs is a skill and an art that takes practice, just like software programming.
- **Effort**: It can take considerable work to produce formal specifications and proofs, even for an experienced engineer. Software modules can take 900 or more hours to specify and prove.
- **Misapplication**: The correct use of formal methods is highly sensitive to input error. For example, one must carefully verify and validate the properties one wants to prove, otherwise one may end up with proof of the wrong (and potentially useless) software properties.

It is therefore possible to spend a great deal of effort on formally reengineering software while misapplying the technique. Subtle misapplication can result in misplaced trust in the reengineered software and can waste a significant amount of effort that could have been avoided if misapplication was prevented or detected at earlier stages of reengineering.

The purpose of this standard is to help those applying formal specification and implementation to reengineer a software module to:

- Assure that they are applying formal specification and implementation properly.
- And that the resulting reengineered module is highly assured to have the properties proven against its specification.

1.2 Format

This standard is a rationalized microstandard [4]. This means that the standard consists of four key components as follows:

- **Guidance**: Documents guiding the user through conduction of process and collection of evidence in conformance with this standard.
- **Evidence Templates**: Template documents for the evidence that users of this standard must populate in partial fulfillment of this standard.
- **Argument Templates**: The rationale for why a formally re-engineered module is assured if all evidence is produced.

Those that wish to comply with the standard will primarily be interested in the first three elements of the standard. Those that wish to understand the underlying reasoning behind the standard, contribute to, improve, or reuse parts of the standard, will be additionally interested in the argument templates.

Furthermore, those who wish to produce an explicit assurance case from this standard can do so by instantiating the argument templates.

1.3 Guidance Material

The remainder of this document is as follows. Section 2 presents the problem of formal reengineering in more detail. This includes a definition of the problem, examples of common pitfalls.

Section 3 outlines the basic rationale (argument) of this standard for assurance of assured formal reengineering.

Sections 4, 5, and 6 provide guidance on the core techniques applied by the standard, as driven by rationale: informal software reverse-engineering with service definitions, formal specification of requirements software, and synthesis of correct source code from formal specification.

Section 7 discusses the theory and practice of rationale microstandards in more detail, and introduces the advanced user to evaluating, editing, and authoring backing argument for the standard.

2 Overview: Reengineering a Software Module using Formal Specification

Assume that a set of software systems, **S**, share a common software source module **M** that provides important critical functionality as illustrated in Figure 1. In the figure, three systems are shown using module **M**. Each has a set of requirements that **M** is intended to satisfy.

The user's goal is to determine what these requirements are and then make a new version of the software module, called **M'** ("M prime") that is more highly assured to satisfy them through application of formal methods in the reengineering process.

Specifically, the standard supports the design of the replacement module **M'** through **formal specification** followed by careful synthesis of source code that is unlikely to violate the formal specification.

Following the standard results in the systems to the right in Figure 1. Module **M** is replaced with module **M'**, to, with reasonable assurance, satisfy the requirements of each respective system of which it is a component. Furthermore, the design of **M'** has been proven to satisfy a subset of

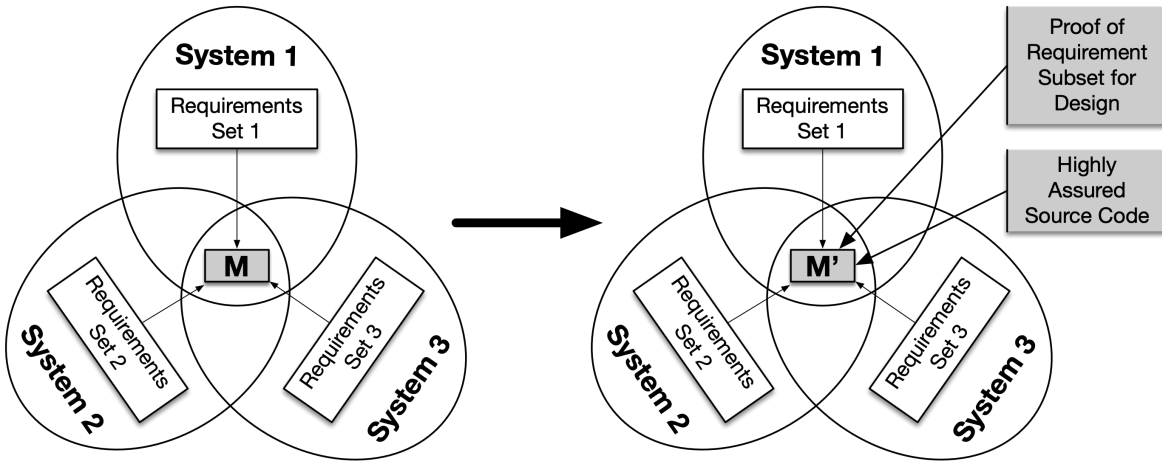

Figure 1. An illustration of the formal reengineering project.

those requirements, and the source code of **M'** has been carefully written, with source-level proofs and careful arguments, to be highly assured of satisfying the specification.

After application of this standard, a resulting software module **M'** should be such that it can replace **M** in the compilation and execution of the *target systems* of set S with

1) No loss of services in any system of S
2) Improved assurance of function and dependability.

For the purposes of this standard, a **software source module** is any componentized unit of traditional software source code with a clear delineating boundary. It could be a source module, file set, or class definition.

This standard is tailored to traditional software that is of static design. Data-driven-design software, *e.g.,* trained neural networks, are *not* the target of this standard. In contrast, a traditional software module defining a neuron on which such a neural net is built *would* be a good candidate. A typical appropriate module might be a message service implementation or class that stores and maintains complex data.

An applicable **software system** can be any software system described by source code and hardware. It can be a purely informatic system or a cyber-physical system.

The standard is written to explicitly assume that the software module to be replaced is a component of more than one system. The most useful modules to formally reengineer are those providing critical services that are subject to common re-use.

2.1 Reference Process

The reference process for module replacement consists of four key steps illustrated in Figure 2.

1) Determining software module requirements as ***interface contracts*** (functional and nonfunctional).
2) **Formal specification** of the software module data, functions, and requirements.
3) Verification of contracts against the formal specification using formal **proof** and other techniques like specification **execution** and **animation** [5] testing.
4) Development of correct software implementation of the formal specification, where correct is defined as (a) an **implementation that satisfies a formal specification**, to the extent possible, and (b) **satisfies all contracts** verified against the formal specification.

Each of these steps is designed to produce the formally reengineered module (**M'**) and evidence of assured assurance. Details of these steps appear in proceeding chapters.

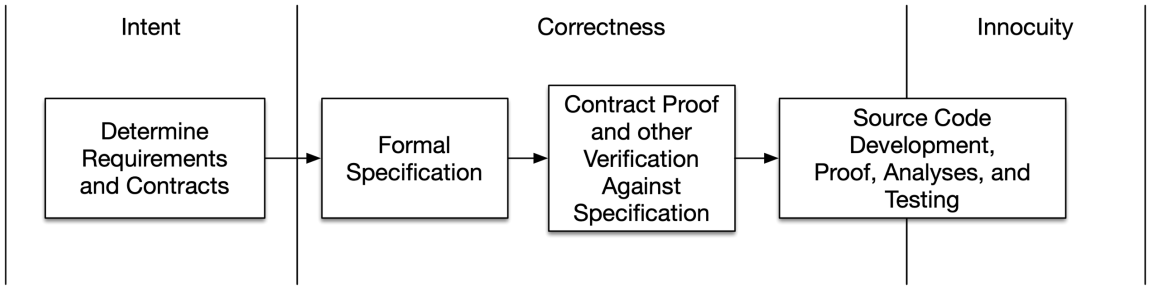

Figure 2. The formal reengineering reference process.

As this process is carried out, evidence will be collected to reasonably assure that the *intended* requirements (functional and nonfunctional) of the software module are understood, that the formal specification satisfies these requirements, showing deductive proof where possible, and that a source code implementation is a *correct implementation* of the specification, again using deductive proof where possible. Furthermore, one will be providing evidence that any behavior of the new implementation that is not intended by the replacement module does not impede the functional and nonfunctional requirements (innocuity extended).

2.2 Iterative, Parallel Development

The assurance reasoning backing this microstandard maintains a linear causal order, as illustrated in Figure 2, between activities and their impact on assurance. However, it is not recommended that activities be carried out in a waterfall approach.

It can be useful to write a formal specification assuming that one will repeatedly go back analyzing formal contracts and requirements. This is because it is difficult to detect all under-specification and behavior errors (resulting from bugs in the current software (design faults)) in an initial pass of analyzing a target software module. Often, attempting to respecify the software will point out where gaps and errors in behavior for the module must be repaired.

Furthermore, once a formal specification has been written, it can be useful to implement the specification before proving too many properties. There are two reasons for this:

1. **Proof Is Expensive**: Proving even simple properties on complex specification can be an order of magnitude (or more) costly in time and effort to complete than in authoring the specification itself.
2. **Implementation is Tricky**: Languages such as PVS are purely functional. And while many mathematical operations will appear to map cleanly between specification and source code, this is not always the case. The result can be that it is very difficult to implement a formal specification "correctly" (as a refinement). One must approach the problem carefully.

Combined, these circumstances mean that the engineer must carefully design their specification to be reasonably provable and implementable. Finding the right design and balance between the two is difficult and an artform. As a result, the engineer should be prepared to experiment and iterative on specification and implementation design. To that end, it is recommended that engineers specify software and plan implementation and proof strategies before commencing with each, then commit to some of each to see if those plans are working. Iteration is likely.

Finally, one is likely to discover during construction of a plan for specification and implementation that changes to desired behavior of the target module, acceptable to the systems of interest, will better suit specification and implementation.

The net result is that the actual process of development may look something like what is shown in Figure 3. There are two key loops that interact, one where the desired behavior of the software module is codified, and another where its specification and implementation are created and verified. This forms an interacting validation and verification cycle.

The assurance arguments backing this standard are not concerned with changes resulting from these iterations. They are only concerned with the product being of sufficient strength in that validation and verification. Therefore, they might lead the reader to assume that a waterfall model of development is sufficient. It likely is not. However, the structure of the arguments does support reasoning over the main concerns of design for specification and implementation. A reader familiar with these arguments and the required evidence might be better prepared to make decisions regarding their structure given tradeoffs in prove-ability and implementability. But in the end, practice of the art will be required.

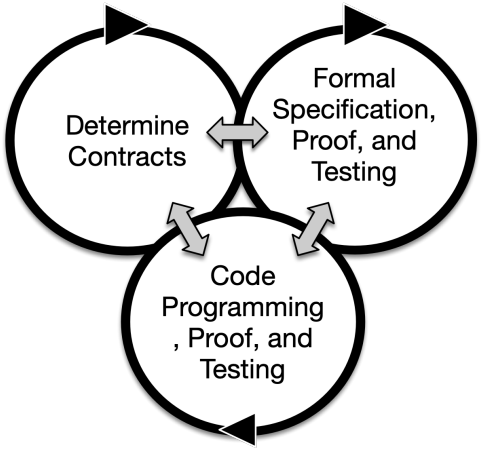

Figure 3. Iterative, parallel development of interface contracts, formal specification and its verification, and source code and its verification.

2.3 Required Technologies

This standard utilizes formal specification and source code proving. This requires two tools:

- A formal specification and proof support system. We have utilized PVS [6] [7] for example applications. Alternatives include Coq [8] [9], and Isabelle [12].
- A formal source code property language and proof generation system. We have utilized SPARK Pro Ada [3] for example applications. Alternatives include subsets of C with ACSL (ANSI/ISO C Specification Language), *e.g.*, FRAMA-C. [2].

The use of a specific tool is not required to use this standard, however some of the terminology of the standard might be currently focused on PVS and SPARK Pro Ada. Our goal will be to remove this over-specificity in later drafts.

2.4 Formalization for Deduction and Better Induction

The intermediate design step of formalization can help reduce design and implementation errors by removing faults in design and iteration process. Practically, strong design form allows strong deduction for proof of critical software properties. This is what people typically think of when they apply formal methods.

In addition, a useful form can also lend itself to better inductive reasoning and verification by other means. For example, developing strong, correct interface contracts on the original software module can reduce unit test design faults by supplying clearer interface contracts against which to test correctly and completely.

2.5 Common Pitfalls

There are many verification and validation failures that can occur in the reference process. This microstandard attempts to show how the above reference process, with sufficient reasoning about

how desired properties are achieved, is sufficient to avoid these points of potential fault. Below is a brief outline of faults that can occur in carrying out the process.

- Validation Failure: Understanding the Wrong Requirements of the Original Model
  - Not understanding all the systems that utilize the module
  - Not understanding all the functional interactions between the module and any given system
  - Not understanding the nonfunctional requirements supported by the module for a given system.
- Verification Failure:
  - Building an incorrect formal specification against the requirements
    - Failure to design the specification against all of the module requirements for all systems of interest that use it.
    - Failure to prove critical requirements
      - Failure to set up the specification to be provable
      - Failure to specify in a logically consistent manner
      - Failure to correctly state the requirements in a form for proof
      - Failure to complete proofs
      - Use of erroneous axioms
      - Circular proof
    - Failure to reasonably verify remaining (unproven requirements)
    - Failure to design the specification to have provable properties
    - Failure to successfully argue
  - Building an incorrect source code implementation against the formal specification
    - Failing to capture all semantics of a given simple-seeming function
    - Failing to map purely functional, recursive specification into a correct imperative program.
    - Failure to account for retrenchment from ideal values (e.g. real values) to computable values (e.g. floating-point values).
      - Failure to account for underflow and overflow
      - Failure to account for numerical instability
      - Failure to account for precision-induced branch divergence
  - Failure to check requirement satisfaction against the implementation by other means

The arguments developed for this microstandard are designed to assure correct and complete reimplementation of the software module desired by stakeholders in place of an original software module and its potential remaining design faults.

3 Standard Structure

3.1 Dependencies

This microstandard relies on one other microstandard, which in turn relies on two argument templates separately available, as illustrated in Figure 4. Those components are as follows:

- **Microstandard for Informal Reverse Engineering of a Software Module**: A standard that attempts to assert reasonable belief in the correctness and completeness of functional and nonfunctional interface contracts of a target software module situated in multiple systems of interest.
- **Argument template for quality requirements**: A simple argument template arguing

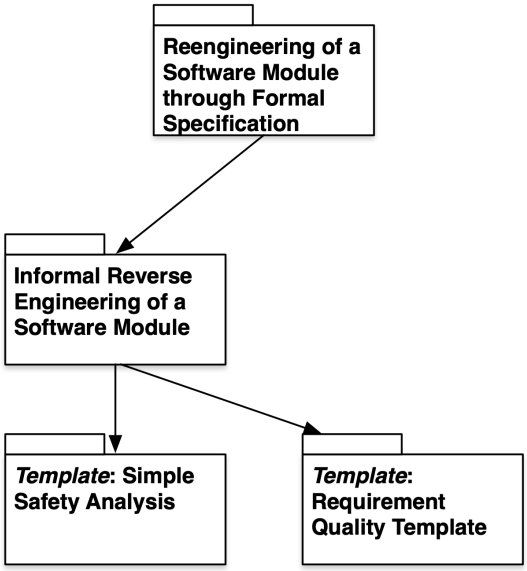

Figure 4. The assembly of microstandards and templates.

that a set of requirements have the attributes of a quality requirement set.
- **Argument template for simple safety analysis**: A simple argument template for asserting that safety requirements of a service are reasonably correct and complete. This is not a comprehensive safety requirement assurance approach and could be replaced by a more comprehensive safety microstandard.

4 Argument Model Format

The rationale of the microstandard is captured in the form of assurance arguments that accompany this guidance document. Arguments are modeled in Goal Structuring Notation 3 (GSN3) [10].

This standard applies GSN3 standard template notation. We utilize an open circle on edges to represent optional applicability of an argument edge, and a closed circle to represent enumerative multiplicity of an edge for each contextually relative instance of a set. See the GSN3 standard for details.

In addition, GSN3 templates are extended with a formal parameter syntax. Any text of the form "{-NAME-}" is a formal parameter with name "NAME" that should be text replaced with the appropriate actual value. For example, "{-System-}" should be replaced with the name of a system that is context appropriate.

5 Issue Tracking

Throughout this reference process, the user will be required to track and resolve issues. Each section of the process will call out issues that are of particular concern for a given activity.

The user is expected to log all issues and track their resolution as is common to any other modern software engineering activity.

6 Concepts

6.1 Overarching Properties

The microstandard relies on a generalization of the Overarching Properties. The Overarching Properties [11] have been created to attempt to identify more fundamental properties that are sufficient to show that a system is safe. The goal is to identify properties in such a way as to not rely on the many implicit assumptions of systems tied to traditional verification and validation ontology. This is critical, for example, as we apply and assure new technologies for which existing assumptions of when and where behavior is fixed or known are set aside. It is also useful as we try to develop property-oriented thinking independent of verification and validation processes.

We generalize Overarching Properties because in inductive argument form, we have found that the three overarching properties are applicable to a wide array of functional and nonfunctional properties of systems for which we have argued. The reader is referred to documentation on overarching properties for their standard definitions. This standard's generalized definitions are as follows:

A system satisfies a property class P (e.g. a quality attribute such as safety) if and only if

- **Intent**: The defined intended behavior of the system is correct and complete with respect to the desired behavior of the system with respect to property class P.
- **Correctness**: The actual behavior of the system is correct and complete with respect to the defined intended behavior of the system under foreseeable operating conditions.
- **Innocuity**: Any actual behavior of the system that is not defined intended behavior of the system does not contribute to failure property class P (in general and as specified in defined intended behavior) of the system under foreseeable operating conditions.

Intent can be seen as mostly the traditional validation problem reduced to idealized correct and complete system definition. The concept of 'validation at the end of the V' is submerged within the concept of that being a final test of the defined intent against observables by stakeholders.

Correctness is essentially classic verification against specification, where top-level correctness is induced by correctly and completely implementing the defined intent for a system.

Innocuity is often confused with "do-no-harm" or preventing extraneous behavior, but it is neither of those things. The primary concerns of innocuity are under-comprehension of desired behavior and extraneous behavior. The unknown unknowns of desire, *i.e.,* stakeholders not knowing what they want until they don't have it is an important component of innocuity. The second form of innocuity results from systems that can exhibit unintended behaviors. This can

arise from use of existing components, misconfiguration, etc. But it will always arise merely as a product of their being side effects in any real-world engineered system.

For this microstandard, we are concerned with the nature of assuring intent, correctness, and innocuity in the context of replacing an existing software module with a new version that has been re-engineered using formal specification, and so we limit its scope to the direct concerns arising therefrom.

6.2 Behavior Contracts

A contract is a set of *requirements*, both *functional* and *non-functional*, that state the guarantees of functionality and data of module **M** given guarantees of the surrounding systems **S**, along with preconditions that are assumptions that can be made by the module **M** and systems **S**.

Consider two example contracts. An example contract is presented in <<Figure Z>> to the left. The first contract, C1, states that given an input parameter of the form provided by A1, the resulting function of method F1 will be G1, with safety constraint S1. This is an example of an Assume/Guarantee contract.

F1 is an *interface point* of the module M. An interface point is any distinct design location on the interface of M where constraints between M and any systems of S must be enforced. C1 is the contract for interface point F1. Constraints G1 and S1 are *guarantees* that must be provided by an implementation of F1 for module M, given that the *assumption* A1 is satisfied by the systems S.

Another contract, C2, is presented for the data structure *interface point* on the example module. This is data that can be directly read or written to by external system elements in S, and therefore, the contracts C2 and C3 must be satisfied on read and write operations from systems of S.

6.2.1 Requirements

There are three classes of requirements under consideration in this standard are:

1) **Functional Requirements**: The functional behavior required of the prime software module.
2) **Nonfunctional Requirements**: Requirements about how the required functions are performed.
    a. **Dependability Requirements**: Non-functional requirements capturing required safety, security, reliability, availability, and maintainability of the prime software module.
    b. **Design Requirements:** Non-functional requirements capturing restrictions on the design and implementation of the prime software module. These can include restrictions on the source programming language and its features, as well as restrictions in software design or in required preservation of existing software features from the original software module.

The strategy applied by the argument is to apply a generalization of Overarching Properties. There are three overarching properties. They were developed to express fundamental properties of system safety. In our case, we utilize them more generally as follows:

6.2.2 Completeness and Correctness

The user of the standard must show that for their target module M and systems S that defined contracts are a complete and correct representation of the *desired behavior* of module M for **attributes of interest**.

In the ontology of Overarching Properties [11], correctness and completeness mean that the contracts the user defines for module M captures all the *desired behavior* for module M in systems S.

Note the emphasis on the word "desired" in the above definition. The actual behavior of the current implementation of module M is likely to contain software bugs (design faults). It is common to discover such bugs while analyzing the software module to develop its contracts.

Therefore, it is important that the user log all potential bugs as issues and resolve them with system stakeholders. Sometimes, it will be advantageous to keep them. But many times, they can be corrected in the contracts or reengineering of the module. In the former case, the correct contract defines the
"buggy" behavior. In the latter, it defines the desired behavior for the reengineered module.

The previous definition of completeness and correctness refers to attributes of interest. For this standard, *attributes of interest* are

1. Functionality: Functional Requirements of M
2. Dependability: Non-functional dependability requirements of M, as defined by John Knight in Fundamentals of Dependability for Engineers [1] as the overlapping quality attributes of
    a. Safety
    b. Security
    c. Reliability, and
    d. Maintainability

The standard does not require that a particular set of these attributes be covered, only that the set of such quality attributes that is covered be carefully stated by the user in their claim about the reengineered system.

6.2.3 Where Behavior Contracts are Attached to Module M

We have written that the contracts defined for M are "on" the *interface* of module M. The interface of module M is any location of the module's source code, or any grouped data (primitive or data structure), where the module and the surrounding systems may interact.

6.3 Functional Specification

There are many formal methods. This microstandard is for the case where a formal specification language will be used to formally specify a functional model of the target module. Formal specification languages tend to either be defined for mathematical analysis (functional specification), or modeling analysis (architecture languages such as AADL with annexes). It is the former that applies to this standard.

Tools in the functional specification domain include PVS, Coq, and Isabell, as well as others. The process, arguments, and evidence of this standard apply to all of these tools. However, this standard is currently oriented towards the PVS (Proof Verification System) language and tools. Future versions might be better generalized.

There are three key parts to these tools.

- **Specification Language**: A language consisting of two parts:
  - A language within which to define algorithms as mathematically defined functions or sequences of operations. PVS defines all specification through functions and relations over formally defined types.
  - A language to define potential properties of specified functions and algorithms as mathematical relationships. This language usually includes existential and universal quantification. n PVS, these asserted properties are written as *lemmas* which the analyst then sets out to prove using PVS tools.
- **Proof Assistant**: A system to organize and keep track of steps and state in desired proofs. In PVS, an interactive theorem prover is used by the analyst to prove lemmas. The Prover finds all type conditions (automatically constructed lemmas that are required for proof of explicitly stated lemmas), and then manages the state of proofs for those type conditions and lemmas as the analyst attempts to construct them.
- **Proof Techniques**: A collection of techniques to make deductive inferences from given information in (1) the functional specification and its declared types, and (2) conclusions drawn so far from the current state of the proof. Most techniques are the types of steps a human would perform in a typical proof, such re-distributing terms in an equation, or seeking an example case for an existence clause.

In PVS, the user 1) creates a specification for the target module, 2) states lemmas that imply or support implication of the target module's interface contracts, 3) attempts to prove that those lemmas, and any type conditions discovered by PVS, hold over the specification. The tools manage the proof structure for the analyst and provide techniques to try to make steps forward in the proof, but the core work of designing specification and proving properties are performed by the analyst/engineer.

Readers are strongly encouraged to study read the guides and tutorial for PVS [7] if they are new to developing formal operational specifications.

6.4 An Assurance-Driven Specification and Implementation Strategy

Because provability and implementation of a standard are central to the assurance of a reengineered target module, the arguments of this standard contain an inherent strategy towards the design of a formal specification and its implementation.

Formal reengineering of a module using formal, functional specification and proof is difficult for two main reasons:

3. **Proof Is Expensive**: Proving even simple properties on complex specification can be an order of magnitude (or more) costly in time and effort to complete than in authoring the specification itself.

4. **Implementation is Tricky**: Languages such as PVS are purely functional. And while many mathematical operations will appear to map cleanly between specification and source code, this is not always the case. The result can be that it is very difficult to implement a formal specification "correctly" (as a refinement). One must approach the problem carefully.

As such, the formal specifications one writes can have a profound impact on the cost of proof and the extent to which a sufficiently correct implementation can be constructed. The core "correctness" reasoning of this microstandard focuses on

- **Specification Verification**: Why the specification one has constructed is sufficiently correct (via proof and testing) relative to the interface contracts.
- **Implementation Verification**: Why the implementation is correct with respect to the specification (implementation verification).
- **Behavior Verification**: Why the implementation is correct with respect to the interface contracts.

While this standard cannot 'solve' these problems for the engineer, it is designed to help the engineer produce a viably verified specification and implementation. The most critical point is illustrated in the figure below.

The standard encourages the engineer to consider their functional specification as consisting of separate pieces for which larger properties hold in assembly by construction. Each piece is designed to be best proven and implemented independent of the other pieces. Each piece of the standard takes into consideration two key elements and a series of sub-techniques as follows:

- **Specification Verifiability**: The ability to imply contract guarantees or other useful properties on part of a specification.
- **Specification Implementability**: The ability to implement a part of the specification so that the properties proven and tested on the specification remain satisfied on the implementation without *requiring* retesting on the implementation, where possible. (Retesting on the implementation is still recommended).

Our guidance does not focus on structure for specification verifiability. It does require specific strategies for source code implementability. These are discussed below, but first we must introduce the concept of retrenchment.

6.4.1 Refinement

The best form of a source code implementation of a formal specification would be a correct and complete *refinement*. An implementation is a correct and complete refinement of an implementation if all functions of the implementation are exactly those of the specification. In such cases, one achieves transitive verity. That is, if an implementation is a correct and complete refinement of a formal specification, then any properties that hold for the formal specification hold for the source code implementation. Thus, if one proved an interface contract clause on the formal specification, it would be guaranteed to hold on the source code. Therefore, it would hold in execution on a correct operating system on correct hardware.

Where possible one should design parts of a specification to be refine-able. But this is often not achievable. In many cases, a formal specification will apply values that cannot be fully represented on a finite computer. For example, real values in a formal specification make proof of properties on a specification tenable, but real values cannot be accurately represented by a computer. Instead, a real value is approximately represented on a computer within a finite range of allowed values by IEEE floating point values. As a result, not only can implementations of a specification develop accumulating error, but they can also undergo branch divergence, resulting in radically different behavior from a specification on the same inputs.

As a result, we need techniques and models to consider what makes an implementation of a formal specification "good enough". First, we need a standard way to talk about the mapping of values from specifications to implementations.

6.4.2 Retrenchment Mapping

Retrenchment is a set of mathematically defined rules mapping values from a specification to its implementation [13]. For example, a rule mapping Boolean values is typically a bijection that is "one-to-one" and "onto". That is, a TRUE or FALSE value in specification translates to its corresponding TRUE or FALSE value in implementation. In contract, rules mapping real values in specification to IEEE floating-point values in implementation must map an infinite range in specification to a finite range on a computer, and uncountably infinite values in specification to countably limited, discrete values on a computer.

Retrenchment rules are important for assurance of implementation because they provide a formal mapping from formal specification to an implementation language. The more formal the implementation language, the stronger the claims we can make about careful implementations. This standard encourages the use of source code languages that support proof against source code, such as SPARK Pro Ada [3] and FRAMA C [2].

Having a retrenchment mapping, one can then reason about techniques to assure maximum transitive verity exists between a formal specification and its implementation. This standard presents four techniques in two categories, each of which applies different argument reasoning.

6.4.3 Limited Refinement Techniques

In these approaches, one develops a formal specification such that rules mapping the values and operators of the formal specification to a computable source code implementation do not cause the behavior of the implementation to deviate from that of the specification.

Three models for this are supported by the standard and backed by reasoning branches of its assurance argument.

6.4.3.1 Discrete Value Retrenchment to a Limited Range

If a specification uses only discrete value types, then showing that input range limitations in source code are acceptable and prevent overflow is usually sufficient to build an implementation as a correct refinement. This is the most desire-able outcome as a correct and complete implementation in source code is often straightforward and the resulting transitive verity of the implementation therefore very strong. Wherever possible, one should produce important parts of

specification for which this can hold, so long as the contribution of its proven properties to ensuring interface contracts outweighs the cost to provability of properties in other components.

6.4.3.2 Nondeterministic Specification Using Interval-Arithmetic

If a specification is written as a non-deterministic operational specification [14], then some extra, then properties explicitly proven to hold on the specification will hold on a correct and complete implementation.

The trade-off of this approach is that it can be more difficult to prove properties on a nondeterministic specification than on a deterministic specification. In particular because of potential exponential combinations of cases under complex non-deterministic branching logic. The benefit of this approach is that it allows limited refinement even when specification use finite uncountable scalar types, such as real values.

A nondeterministic specification does not use infinite uncountable value types. Instead, it replaces them with a type for which it is assumed there is uncertainty in value and computation. All lemmas against such values can only be stated as closed intervals defining bound uncertainty. One can then analyze the specification under retrenchment rules to show that all such defined bounds are satisfied by an implementation under the allowed range of input state.

At the time this specification was written, the mapping from real values in specification to IEEE floating-point values in source code is the only true concern in this space. For the PVS specification language, a tool has been developed to produce sound range bounds on error in IEEE floating point values resulting from computation as compared with real values. This tool, PRECiSA [15], can be applied to show that bound ranges defined in lemmas for a nondeterministic specification are all satisfied in a proper implementation under well-defined retrenchment rules. If this condition is met, then one can build a correct and complete source code implementation for which all currently proven lemmas hold.

6.4.3.3 Functional Replacement

In some cases, a PVS function might not be specified by an operational expression (an expression on sub-functions). In such cases of declaratively defining a function, one cannot create an ‘operational replica’ of the specification in source code.

In other cases, a PVS function might be specified as a function expression, but there might be another implementation of the same function in source code against that one can prove is correct and complete against the specification.

In either case, one can try to show that one has built a function implementation in Spark ADA or FRAMA C, and then prove that it satisfies the formal semantics of the function as specified in PVS.

6.4.4 Reasoned Approximation Techniques

When it is not possible to show that an implementation is a limited refinement with given retrenchment rules, it is often possible to instead show that specific properties hold for a given implementation, and that those properties are sufficient to maintain properties proven against the specification.

6.4.4.1 Numerical Stability Analysis

A common technique applying reasoned approximation is numerical analysis. For example, many algorithms exist for which one can show numerical stability. Given an iterative (or recursive) specified function, one can often show that floating-point implementation's output error is bound with respect to input error.

One can show that the resulting implementation's numerical inaccuracy satisfies a particular guarantee of a software module's interface.

The advantage of this approach is its simplicity and maturity. The difficulty of this approach is that it must be carefully argued for each contract guarantee. Careful construction of a specification often produces valid results with reasonable effort. However, many specifications for general software modules will produce complex branch divergence scenarios under which such analysis techniques will become too burdensome.

6.5 Partition, Solve, and Compose

The goal of the engineer is to partition their target module into component sub-specifications where one of the above techniques will be effective. In addition, they must partition to make interface contract guarantees as strongly provable through analytic proof and testing as possible. As explained earlier, this is an art. But the microstandard deliberately supports reasoning over the total product through the partitioning of the specification and application of the above approaches to each part as best suits it.

7 Evidence and Other Standards

The following chapters details the reference process for re-engineering a software module using formal specification.

Evidence for conformance to the standard will be described in these chapters. All evidence has an associated solution node within the assurance argument backing the standard. Some of the evidence for the reference process will be source code files, formal specification, proof files, *etc*. Other parts of evidence can be stored in the "evidence-fre.xlsx" spreadsheet. The template for this spreadsheet is provided with the standard.

Some parts of the reference process will rely on other microstandards. These will be referenced directly at appropriate points in the reference process.

8 Process: Identification

The first step in reengineering is to correctly identify the elements of software to be reengineered and the context of systems within which that reengineering should take place. Keep in mind that we will need to identify both functional and nonfunctional behaviors of the target modules in the target systems, and this will require careful study of those systems, not just where other systems touch the target module interface.

8.1 Module Identification

The first step in reengineering a module is to clearly define the target module. In some cases, the module might be defined by source files. In others, it might consist of certain source code within files. In either case, the delineated between what is in the module and not in the module must be clearly defined.

Module identification should be defined in the "evidence-fre.xlsx" workbook on the "module" spreadsheet

The fields to be filled out by the user are:

- **Module Definition**: The user should define the module to replace with a definition that will clearly distinguish the module from any other source code of target systems. The reengineering team should review the module definition and reach agreement.
- **Module Sign-Off**: Sign-off from the team that they agree on the module definition as being clear and sufficient, and that they agree with the module definition.

8.2 Systems of Interest Definition

The team will need to analyze the interface of the software module relative to a set of systems of interest. This will require that we have a clear definition of the set of systems of interest. The systems of interest will be the only systems for which the interface contracts are correct and complete.

The current standard does not allow for defining abstract systems or classes of systems. The process requires specific systems against which the module is reengineered. However, one might choose systems that are 'sufficiently representative', functionally and non-functionally, of use of the target module. The difficulty in such a use will arise when one attempts to derive sufficiently contextual nonfunctional requirements against what might be different future system services.

Systems of interest should be collected in the "evidence-fre.xlsx" workbook on the "systems" tab. Each system should be identified with a clear definition and engineering team sign-off on the relevant system having an appropriate definition.

Sign-off should also be presented that no other systems are of interest.

8.3 Stakeholder Identification

For each system of interest in the system set, a set of stakeholder for the system should be identified. Each stakeholder should sign-off that they accept their role as stakeholder in the reengineering effort. Stakeholders will be expected to help confirm that the target module intent is correct for the target module reengineering.

Stakeholders will sign on the "stakeholder agreement" column of the "systems" tab in the "evidence-fre.xlsx" workbook.

9 Process: Defining Correct and Complete Interface Contracts

The next goal of the standard is to construct a rigorous definition of the target module's intended behavior. The goal of the standard is to develop a model while showing that the model is sufficiently complete and correct with respect to the intended behavior of the reengineered software module. The metamodel chosen for this modeling is that of natural language, assume/guarantee contracts covering the interface of the target module.

9.1 The Basic Contract Interface Development Process

There are four steps in the process of defining interface contracts.

1. Using analysis techniques and tools to determine the actual behavior of the current version of the target module.
2. Resolving conflicts between various analysis techniques to arrive at a logically consistent consensus on a single set of interface contracts.
3. Determining which behaviors of the current module version are erroneous and defining interface contracts to correct for faults in the current module, then solidifying the intended behavior of the software module through correction of these errors with defined correct behavior for the target module.
4. Determining correct and complete desired behavior of the target system for its next version relative to the intended behavior of the current version.

These steps can be iterated upon to arrive at a satisfactory final set of captured desired behavior of the software module based on any issues discovered. Issue recording and the resolution of issues will be part of collected evidence.

9.2 Step 1: Analyzing the Module in Systems

There are many engineering techniques one can use to determine and record the behavior of the target module in systems of interest. The arguments of the standard enumerate some potential techniques that might be included in the standard. At this time, only one is supported.

This standard currently supports use of the microstandard for informal reverse engineering of interface contracts from system source code. The reader is referred to that standard to complete the requisite work.

When that work is completed, analysts will have a set of interface contracts that cover the target module. The contracts will define on the target module interface all properties of an interaction that can be assumed by the target module and all guarantees that must be satisfied by the target module.

The applied microstandard, if applied conscientiously, assures that the resulting interface contracts are correct and complete to the extent practicable. Correctness and completeness are important for the next steps of the reengineering process.

9.3 Step 2: Resolving Conflicts

As there may be many forms of analysis that can be applied to create interface contracts, it is possible that different conclusions will be arrived at using different techniques.

Multiple analyses might be sound but might eclipse one another's conclusions (This is not technically conflict.) For example, symbolic executive analysis might produce tighter bounds on invariants than can be achieved with informal static analysis.

Or one or more analyses might, despite best efforts, produce an erroneous result. For example, a study of code documentation might disagree with assertions of nonfunctional requirements inferred on the source code. Therefore, it is important to reconcile such differences before proceeding with reengineering.

The product of conflict resolution is a single set of interface contracts for the target module. Conflict resolution consists of determining where contract analyses are more specific than one another, as well as determining where errors in an analysis occur.

In performing conflict resolution, analysts should analyze all contracts for all interface points against one another to look for discrepancies. For each interface point contract, each identified potential conflict should be recorded. The standard includes a table for this in the "evidence-fre.xlsx" spreadsheet on the "Contract Conflict Resolution" worksheet. When all potential conflicts are identified, team members discuss them and arrive at consensus that the conflict exists or not. If it does, they sign off in the provided table column. Otherwise, "false" is written in the column. If a conflict exists, its cause is identified and recorded in the table. Then a resolution is decided upon and recorded. Team members sign-off in the table that they agree with the resolution chosen.

Once all contracts are compared and all potential conflicts resolved, then an intermediate form for the contracts is recorded.

In practice, many errors will be detected during various analyses. For this standard, it is recommended that those errors be recorded in the provided table discussed in the next section during these analyses.

9.4 Step 3: Identifying and Resolving Erroneous Behaviors

There are typically unresolved bugs in any deployed software module. It is quite common to find some of these bugs when carefully analyzing software for ree-enginering. Each "bug" is a design fault—resulting from either failure to understand intended module behavior or failure to implement the behavior correctly. Design faults result in errors in module behavior. These errors should be present in the resulting interface contracts of the actual behavior of the target module.

In this step of the reengineering process, analysts identify errors in codified behavior found in interface contracts, identify the design faults that cause the behavior error, account for the actual intended behavior, and correct the affected interface contracts to replace erroneous actual behavior with the intended behavior of the software module.

To show relative completeness of this review process, the microstandard reasons over an assertion of correctness in intent of each interface contract of the target software module. This evidence is recorded in the "evidence-fre.xlsx" spreadsheet in the "Contract Error Correction" worksheet.

9.4.1 Two Error Types: Underspecified Behavior and Erroneous Behavior

There are two contract error types with which we are concerned.

- **Underspecified Behavior in Contract**: An underspecified behavior in contract is one where what could be inferenced from available analyses is insufficient to describe the apparent intent of the module. This often occurs for nonfunctional requirements. For example, the intended real-time performance of function might be insufficiently constrained by an existing contract term that was derived from an algorithm analysis.
- **Erroneous Behavior in Contract**: An erroneous behavior is one where the actual behavior of the software module differs from what was intended for the software module. Erroneous actual behavior that is captured in interface contracts represents faulty interface contract and must be corrected to represent the intended behavior of the module, so that the reengineered module does not contain the erroneous behavior.

9.4.2 Detecting Contract Errors

Analysts will detect underspecified and erroneous behavior codified in contracts by understanding the general intent of the software from the perspective of stakeholders. This process can be improved by having a set of written requirements for the target software module to compare against actual module behavior. The current argument of the microstandard considers that one can writing contracts from an existing requirement set, but such a microstandard is not written at this time. If one has such requirements and can derive contracts, then one can compare them against actual behavior to identify errors in the contracts derived from software analyses.

However, we assume that most software modules of interest do not have explicitly recorded requirements. Therefore, error detection is "to the extent practicable". The evidence for this is collected in the "evidence-fre.xlsx" spreadsheet in the "Contract Error Correction" workbook.

Analysis should perform an analysis filling out the following columns in the workbook:

- **Interface Point Name**: Each interface point should be enumerated by a row in this column. This will help show that all contracts have been analyzed for potential errors.
- **Interface Point Sign-Off**: The team lead signs here when it is believed that the team has adequately considered potential underspecified behavior and erroneous behavior in the actual behavior contract for a given interface point.
- **Properties of Interest Sign-Off**: The team lead signs here when it is believed that all properties of interest (*e.g.* safety, functionality, performance, *etc.*) for the systems of interest have been taken into account in error review.
- **Interface Point Contract Errors:** Here, the team lists potential errors that exist at each interface point. Each potential error receives its own row in the table for the contract under analysis.
- **Error Type:** Each error should be categorized as either "underspecified behavior" or "erroneous behavior".
- **Affected Systems:** Each system of interest that is affected by the potential error should be listed.
- **Analyses Detecting:** As stated in the previous section, many errors in actual behavior of software (software bugs) will be found during software analyses, such as in informal reverse engineering or documentation review. This column indicates where errors were detected and analyzed. Bugs found in this enumerative review process can be recorded as

"error review". Other common sources of detection will be "informal reverse engineering" and "documentation review".

- **Error Agreement Sign-Off By Team**: The team should agree that the error exists, and sign, or should write "false" to indicate that it has been decided that the potential error is not an error or an error worth correcting in contracts.
- **Contract Error Correction**: If the error is agreed upon by the team, then this cell holds the changes to the contract that are required to repair the corresponding error, either by providing sufficient specification of behavior or by correcting the contract to represent intended behavior where actual behavior of the module is erroneous.
- **Error Correction Sign-Off**: The analysis team signs here that they are in agreement with the correction to the contract.

Having filled out these columns, the analysis team will have shown that they have significantly reviewed all contracts for potential errors, considering under-specification and erroneous specification in the contract for properties of interest.

In practice, many errors will be detected during various analyses. For this standard, it is recommended that those errors be recorded in the provided table throughout analyses.

9.4.3 Identifying Faults and Preparing Mitigation

It will be useful for the development of correct formal specification to have codified what the design faults were that result in erroneous actual behavior. Several columns of the worksheet used in this process are provided for that recording. Analysts should record:

- Fault: The identified fault (or faults) in the target module or surrounding system that result in the erroneous behavior of the software module.
- Fault Mitigation: Corrections to software that would mitigate the design fault.
- Fault Mitigation Sign-Off: Sign-off from the team that they agree with the chosen fault mitigation approach.

Recording this data when errors are discovered is useful to avoid re-analysis of the existing software later.

9.5 Step 4: Updating Behaviors

At this point in the process the analysts have a set of interface point contracts for the intended behavior of the target software module. Often at this time, experience and the extent of the analysis will reveal changes and amendments to the desired behavior of the target software module. In this part of the process, analysts and stakeholders can amend the intended behavior contracts with changes that will better meet the needs of stakeholders for the systems of interest.

This process consists of rendering the interface point contracts into a form digestible by the various stakeholders in the systems of interest, targeting the impact that the current intent has on the systems and allowing the stakeholders to reflect on potential changes. The analysis team can also propose recommended changes that align with stakeholder interests.

The goal of the microstandard is to assure that the resulting defined intent for the reengineered target module clearly captures desired change in behavior. The evidence for this concern is captured in the "evidence-fre.xlsx" spreadsheet on the "Contract Updates" worksheet.

This is a good time to consider any changes to the software module's intended behavior that can be effected without negatively impacting the system.

9.5.1 Preparing Accessible Presentations

First teams create and then vet the completeness of presentations against interface point contracts. They judge whether the presentation is at the right level of abstraction for the intended stakeholder audience.

9.5.2 Assuring Presentation Completeness

Next, they analyze whether a given presentation adequately covers each contract of each interface point for the properties of interest relevant to a given stakeholder audience at the appropriate level of abstraction.

9.5.3 Assessing Stakeholder Comprehension

It can be difficult to determine if stakeholders adequately understand the presented material. Question and answer sessions are relevant to clearing up confusion and learning the material. A quiz on the presentation, and stakeholders passing the quiz, is collected as evidence of comprehension to assure that stakeholders can consider the ramifications for their systems.

9.5.4 Change Discussion Guided by Enumerative Reasoning

The analysis team should lead a discussion where all properties of interest are considered for all contracts discussed, and all relevant changes and amendments to module behavior that might impact system behavior are discussed. All changes and amendments are recorded in the provided worksheet, and the analysis teams sign off on each requested change or indicate "false" when a change is rejected.

9.6 Final Interface Point Contracts

The final interface point contracts are written in the "IP Contracts" worksheet of the "evidence-fre.xlsx" spreadsheet. The format of this worksheet is discussed in the microstandard for informal reverse engineering of a software module into interface contracts. Readers are referred to that microstandard for information on the format of the contracts as evidence.

10 Process: Building a Correct and Complete Formal Specification

The next step of the formal reengineering process is to construct a formal specification of the target module against which formal methods can be applied. This standard covers development of functional formal specifications, such as are developed in PVS [7], Coq [9], or Isabelle [12].

The developer's goal will be to show that a specification is such that for each contract and its stated assumptions, each of its guarantees is satisfied by the formal specification.

There are three basic activities to be conducted in this endeavor as follows:

- Development of a formal specification for the target module.

- Development of a set of lemmas and tests against the formal specification that imply all guarantees of all interface contracts.
- Proof of all required lemmas
- Successful development and passing of all tests.

It is unlikely that developers will perform these steps linearly. More likely, a spiral and iterative model will be used as would be the case for software development. Regardless, there is a set of concrete evidence that this microstandard requires to be collected in the assurance of these four activities.

This evidence can be collected in the template "evidence-fre.xlsx" spreadsheet in the worksheet "Contract Verification on Spec". The generation of this evidence in each of the above activities is discussed in the sub-sections below.

10.1 Development of a Formal Specification

Of the four activities in formal specification, it is the development of formal specification itself for which direct assurance evidence need not be acquired. As such, this guide does not discuss formal specification of software in depth. The reason for this is because assurance depends upon properties inferred to hold against a formal specification, rather on the form of the specification itself. As such, this section is brief.

New applicants should refer to relevant guidance and tutorials in the construction of formal functional specifications in their chosen toolset (*e.g.,* PVS, Coq, etc.)

10.2 Development of Implying Lemmas and Tests

It is sometimes the case that one develops lemmas against a formal specification that will directly translate into the guarantees that must be proven against the formal specification. However, the general case is that a collection of lemmas and acceptable axioms imply the guarantees. In addition, limited resources often means that it is not possible to develop and prove lemmas to imply all guarantees of all interface contracts. Therefore, it is reasonable to choose guarantees to prove through implication of lemmas, and others to verify through a combination of lemma proofs and testing. In the degenerate case, pure testing can be used.

PVS, for example, supplies tools to animate and/or execute operationally specified parts of a a functional specification. An operational functional specification is on in which a function is defined as a series of operations, where each operation has a formal definition defined elsewhere. So long as a function consists only of other functions that are defined by other operational functions and operations, it is an operational function.

The job of the formalist is to devise formal specification and lemmas, so that a sufficiently convincing set of proofs against lemmas and tests against operational specification can be devised. This is an art and requires practice. For assurance, one must be able to show the natural language deductive logic by which each interface contract claim is implied by a series of lemmas and tested properties.

One must then show that each lemma is formalized as intended and adequately proven. Then one must show that each tested property is as intended and executed/animated on the intended input sets against specification.

The following fields of the "Contract Verification on Spec" worksheet should be filled out by the specification developers and analysts as they develop the specification and verify contracts against it:

- **Interface Point**: Each interface point of the target module should be listed out in a line of this field, copied directly from the latest IP Contracts.
- **Guarantee/Requirement**: Each guarantee of each contract should be listed on a separated for its respective contract.
- **Covered**: This is an automated field that will indicate to the user whether there is sufficient verification coverage of a given guarantee. It will indicate a check mark if there is guarantee implication logic that has been approved by the analysis team.
- **Implication Reasoning**: Here, the analysts should indicate or reference the natural language deductive logic from which it is concluded that a given guarantee is implied by a set of lemmas and tests against the formal specification.
- **Implication Reasoning Review Sign-Off:** Here the team lead indicates that the team have peer-reviewed the implication reasoning and find no flaws.

Rather than performing the above activity purely before the next activity, it can be valuable to first attempt to formally state lemmas and prescribe tests before reviewing implication logic, as one might have to try various approaches before finding a efficiently solvable set of lemmas and tests with sufficient implication of a targeted contract guarantee.

10.3 Proving Lemmas

Analysts must prove formalize and prove lemmas (and sub-lemmas relied upon) that are required for the implication reasoning developed in the previous activity. Tools in this space directly allow statement of lemmas and then support management of proof performed by the human analyst.

10.3.1 Validating Proof Structure in Lemmas, Axioms, Types, and Theorems

Because lemmas are written in a basic formal language that is highly expressive, it is often possible to introduce design faults into the statement of lemmas. One can also rely on axioms that are incorrect, introduce concepts that are consistent but trivially meaningless, or introduce circular logic. Therefore, it is important to vet all lemmas utilized for proof. Proof work is often very expensive. It is therefore advised that analysts validate their lemmas and supporting logical framework before conducting proof work.

Analysts should state lemmas in the specification language, and then determine what explicit axioms are relied upon for statement and proof of those lemmas. They should review their lemmas and axioms, declared types, and functional specification theorems to determine if there are any implicit axioms between them that invalidate intended property proofs. Implicit axioms are assumptions inferred from relationships that exist because of declaring multiple types, lemmas, axioms, and theorems. In many cases these inferences are insufficiently true and might need explicit proof.

Analysts should also look for common pitfalls in their design of lemmas, axioms, types, and theorems. In particular, the standard requests that developers look for circular logic between their applied lemmas, axioms, and declared types. There are likely many more design faults that can occur in proving lemmas, and they will be added to the standard as they are determined useful to explicitly review and mitigate.

The "Contract Verification on Spec" worksheet collects evidence for assurance of lemmas as follows:

- **Lemma**: An entry in the worksheet should be created for each lemma created for implication of a given guarantee. In addition, each lemma relied upon in turn by those lemmas should have an entry somewhere within the worksheet. In PVS, lemmas can be referred to by their unique names.
- **Lemma Intention Review Sign-Off:** Here, the team peer reviews each lemma as it is stated in the formal specification language and determines whether it accurately and sufficiently captures the intent of the property to prove as defined in the natural language implication of its supported contract guarantee(s). If the team is convinced of this, then the lead signs, otherwise, this should be left blank.
- **Applied Explicit Axioms**: Here all explicit axioms that are utilized by each lemma should be listed in a comma separated form.
- **Explicit Axiom Review Sign-Off**: All explicit axioms utilized by a lemma should be reviewed for appropriateness in the context of the formal specification. Any axioms that are not universally true in that context should not be agreed to. Teams may also choose to limit use of axioms in general. Signature in this field indicates that peers agree with the current list of used axioms and can find no other axioms relied upon.
- **Implicit Axioms**: Here analysts list out any implicit axioms that might exist between lemmas, explicit axioms, and declared types.
- **Implicit Axiom Sign-Off**: Peers should review for presence of implicit axioms and vet them for acceptability. Team lead sign-off indicates that sufficient peer review has been occurred to identify, vet, and mitigate as needed such axioms for the given lemma's purpose.
- **Circular Logic Sign-Off**: Peers should vet lemmas, theorems, axioms, and types for circular reasoning. They should look for such reasoning as well as address any potential occurrences. Lead sign-off is present if peers agree that no circular logic could be found with reasonable effort in the present version of the specification.

10.3.2 Proving Lemmas

Having created a satisfactory set of lemmas, support-lemmas, axioms, and types, one can rely on proof against the required lemmas. In practice, one will often iteratively attempt proof of lemmas before and during vetting for the above criteria. Regardless of order of performance, one must produce proof and vet this proof with peers. Obtaining proof for lemmas can be an expensive activity that in the general case requires considerable human problem-solving insight. From the assurance perspective, however, we are concerned only with the quality of the resulting proof in the context of the property verification problem being solved. This standard relies on the trustworthy nature of proof assistance engines to prevent mechanical proof errors. As a result, we

rely on two pieces of evidence in assurance of the resulting proof noted in the following evidence fields in the "Contract Verification on Spec" worksheet:

- **Proof File**: The file containing the proof that can be rerun for its corresponding lemma. This must be listed for each lemma used in implication of a guarantee or any applied support lemma.
- **Proof Review Sign-Off**: A final review of proof should be conducted by peers to understand lemmas and how they are proven, as well as to make a final review of potential inaccuracies in modeled properties, or production of design faults. Team lead sign-off in this field means that proof of the given lemma for the current version of the specification is accepted by peers.

Having produced the above evidence, one has validated the lemmas and supporting logical framework for one's proof and then conducted proof to a satisfactory level.

10.4 Testing

There are various tools and techniques for testing against formal operational specification. Execution for specific inputs is largely possible through operational emulation based on a formal specification language's basic operations and any other operations/functions derived from them. Other techniques can extend analysis to include specification that is not written operationally. It is beyond the goal of this microstandard to specify techniques as they vary by tool. Instead, the microstandard argues assurance of basic verification and validation of tests written against formal functional specification.

To this end, for each test implemented, peers should review the test for its accuracy in testing the intended property of a formal specification. They should also verify that it is sufficient in the set of inputs it is intended to test against, and they should record the test execution and whether the test is passing or failing against the current specification. These are maintained in the following fields of the "Contract Verification on Spec" worksheet.

- **Test**: Each test appied in implication of contract guarantees should be stated on its own line and indicated by a unique name.
- **Test Passes**: The value of this field should be TRUE, FALSE, or blank for the test passing, not passing consistently, or not having been conducted against the current version of the formal specification.
- **Test Log:** The location of an informative log for the actual test execution from which reviewers can discern test execution and result.
- **Test Review Sign-Off**: Peers should review that the test measures the intended property for the intended set of inputs. If they agree, then the lead signs in this field for the given test.

The above simple test verification and validation could be supplanted by a more concrete microstandard for standard programmed testing of software-like systems.

10.5 Conclusions

Having performed the above activities, assuming all required proofs are completed and all tests pass, then one has reasonable assurance that the proofs and tests are valid and verified, such that all contract guarantees are satisfied against the formal specification.

11 Reference Process: Building and Verifying a Correct and Complete Implementation from the Formal Specification

The high-level techniques for building an implementation from the formal specification was outlined earlier in this work. The analyst should consider their specification partitioned into sub-specifications. Each specification part is then implemented using a strategy dependent upon the properties of its specification.

There are two key activities to perform. The first is to create a correct and complete implementation of the formal specification. The second is to test, at minimum, those properties that could not be strongly implied by proof and testing and/or directly proven against the specification.

Practically, one should test all interface guarantees as there may still be uncaught errors in formalization, proof, specification testing, and/or implication reasoning.

Testing of source code is not the purview of this standard. The current arguments refer to a potential future microstandard for interface testing of a software module. The remainder of this chapter is concerned with the process of creating an implementation and collecting required assurance evidence.

11.1 Defining a Specification Partition

Engineers should have designed the formal specification so that if it is to be divided into sub-specifications to be implemented using different techniques, that division of the formal specification is clear. Evidence is to be collected for this partition. The worksheet “Specification Partition” in the “evidence-fre.xlsx” spreadsheet collects this evidence. The fields to collect are as follows:

- **Partition, Team Review Sign-Off**: This field should be signed by the team lead when the engineering/analysis team agrees that the defined partition of the specification is complete and correct over its defined parts.
- **Part**: Each part of the specification should be given a unique name and listed on a line of the table.
- **Implementation Assurance Approach:** The implementation assurance approach intended to be used for the specification part should be listed here and is one of the methods presented in Sections 6.4.3 and 6.4.4.
- **Included Procedures**: The procedures, functions, and methods of the formal specification to be included in this part should be stated in this field in a comma separated list.
- **Included Data Structures**: The persistent data of the formal specification to be included in this part of the specification should be stated in this field in a comma separated list.

- **Part, Team Review Sign-Off:** This field should be signed by the team lead when the engineering analysis team agrees what the part definition is sufficiently complete and correct with respect to what it was intended to contain.

11.2 Basic Retrenchment Modeling

A retrenchment model is a complete function mapping every possible value of a value type to a set of values from another value type. The former value type is one found in a specification, and the latter value type is one applied for its implementation. Retrenchment is used to formally define a mapping of infinite or uncountable data type values to finite and countable data type values in computing systems.

Some data types can be mapped as a retrenchment that is one-to-one and onto. For example, the Boolean data type has values "true" and "false" in specifications as well as in computing systems.

Other data types map as simply "onto" functions. For example, the retrenchment of integers of PVS to integers in a computer program is merely onto. This mapping depends on the number of bits with which an integer will be represented in a computer program. The integers of a PVS specification are unbound, but countably infinite. A typical retrenchment maps the integer number line to representable integers in a program such that a maximum and minimum representable value in the computer is understood, and values greater to or less than these values, respectively, "wrap around" in the retrenchment model.

Other data types in PVS are uncountably infinite. The most common case applied is the real number line. Modern software programs can used fixed point or floating-point representations to approximate a countable, finite set of reals within a limited range. The latter format includes complexities of normalization for small numbers and other factors, such that the relationship between real values and floating-point values is very complex. Retrenchment rules for this case must be very carefully defined, and typically observe the IEEE standard for floating-point implementation, as well as describing mapping in underflow and overflow from the entire range of reals to those representable.

11.3 Specification Mirroring

In many cases, one will want to mirror the expressions of a specified function as expressions and statements in a source code language. Mirroring is the intuitive concept that if there is a one-for-one match between the basic functional operations of the formal specification and the operations of the source code expressions and statements, then the resulting implementation will be closer to correct. For example, if a specification states "x + y", then the implementation should mirror the specification of "x + y" with the expression "x+y".

The one case in which one will not want to mirror an implementation is when one is not copying the functional operations of a specification but instead proving an equivalent algorithm. However, even in that case, one will want to show that the specification is part of the postcondition of the function's implementation.

Otherwise, one will attempt to mirror a specification part's functions and persistent data structures, as well as its utilized complex data types. Rules for mirroring from PVS specification to SPARK Pro Ada can be found in a provided reference [16].

The current version of the standard assumes mirroring from PVS specification language to SPARK Ada. When mirroring is conducted, the developer should review mirroring with peers and assert that mapping rules are followed. The development team provides mirroring assurance data in the "evidence-fre.xlsx" spreadsheet in the "mirrored functions", "mirrored type predicates", and the "mirrored functions" workbooks.

11.3.1 Assuring Mirrored Data Types

In mirroring data types, the developers should fill out the "Mirrored Data Types" worksheet with the following data fields:

- **Part**: The part of the specification, by name, in which data types might be declared. All parts should be enumerated in this table.
- **No Extraneous Data Types Review Sign-Off:** When the implementation for a given part of the specification is completed, the team should review that each data type declared in the implementation corresponds to a generating data type in the specification part. This assures no extraneous data types are present in the implementation.
- **All Data Types Mirrored Sign-Off**: The team reviews the specification and implementation and this data table and signs-off here when they believe that all specified data types in the specification part are implemented and enumerated in this data table.
- **Data Type Name**: The name of each data type declared in a PVS specification, including anonymous types that should be named by specification location.
- **Primitive Type Conversion Matches**: The engineers should review the implemented data type and mark "TRUE" when they are convinced that all primitive types in each data type correctly map from PVS to Spark Ada types.
- **Standard Structures Handled Correctly:** The engineers should assure that standard structure (name records) in PVS map to structures with corresponding field names and data types. Marking "TRUE" here is an indication that on revue by the team this is believed to hold for the specification/implementation part pair.
- **Anonymous Records Handled Correctly**: The engineers should follow mirroring convention to map PVS anonymous records into a SPARK Ada data type. The team reviews such types in PVS and assures that they are mirrored properly in the source code, indicating "TRUE" when believed to be the case by the team following review.
- **List Structures Handled Correctly:** The engineers should follow mirroring convention to map PVS list structures into an appropriate SPARK Ada list structure. The team reviews data types and marks "TRUE" for each data type for which list structures (including if there are none) are properly mirrored.

The above data are the basic checks utilized to indicate that through extensive peer review, it has been determined that mirroring of data types in a specification part appear correct. The above evidence is used to argue completeness, correctness, and non-extraneousness of data type implementation.

11.3.2 Assuring Mirrored Type Predicates

PVS specifications employ a data value type mechanism called type predicates. These are mapped to SPARK Ada through properties asserted on data types using the SPARK precondition/postcondition language.

All such predicates must be mapped. This standard assures their correct, complete implementation as proof conditions by requiring the following evidence found on the "Mirrored Type Predicates" worksheet of the "evidence-fre.xlsx" spreadsheet:

- **Part**: The part of the specification, by name, in which data type predicates might be declared. All parts should be enumerated in this table.
- **No Extraneous Types Review Sign-Off**: The team reviews all postconditions / preconditions of the implementation part and determines that none of them are extraneous representations of non-existent or non-applicable PVS type predicates. If not true or unknown for current versions then make blank.
- **All Predicate Types Mirrored Sign-Off:** The team reviews all predicate types found in the specification part and determines that all are mapped to all affected data types of the implementation.
- **Data Predicate Name:** The name of a given data predicate in the PVS specification part.
- **Predicate Encoded on Data Types:** The team lists
- **Team Review of Predicate Encoding Sign-Off**: The team reviews the listed data predicate to make sure that it is correctly encoded in preconditions/postconditions for all implementation affected data types. If agreed, the lead signs here. If not yet reviewed or agreed for the current versions of specification and implementation, this remains blank.

11.3.3 Assuring Mirrored Functions

Mirroring functions involves implementing a function, either by (a) mirroring its operations of its specification or (b) implementing a function believed to be equivalent, and then (c) encoding the specification as postconditions of the function implementation in SPAR Ada to be proven using the SPARK Ada toolset.

The fields to be filled out for assurance of function implementation are as follows:

- **Part**: The part of the specification, by name, in which functions might be declared.
- **No Extraneous Functions Review Sign-Off**: The team reviews the complete implementation and determines that there are no functions that do not implement a specified function (either directly or in support as an implementation sub-function). The team lead signs here if this is true for the current versions upon review. Should be blank otherwise.
- **Complete Functions Review Sign-Off**: The team reviews the implementation and shows that all specification functions of the specification part have an implementation in its implementation part. The team lead signs here if this is true for the current versions upon review. Should be blank otherwise.
- **Function**: Each function of the specification part is listed with an entry for this field.
- **Translation Team Review Sign-Off**: The engineering team reviews the translation of each function from specification to implementation, agreeing that all mirroring rules have

been followed for mapping from specification operations to implementation operations. The team lead signs here if this is true for the current versions upon review. Should be blank otherwise.

- **Function Team Review Sign-Off**: The engineering team reviews the semantics of the implementation for each function, signing off if they believe it is a correct implementation of the function. The team lead signs here if this is true for the current versions upon review. Should be blank otherwise.
- **Proof Conditions Team Review Sign-Off**: If a function implementation is to be proven correct, then the function specification must be captured as preconditions and postconditions of the SPARK Ada implementation of the function to be proven against it. The team of engineers and analysts review the preconditions and postconditions annotating each function implementation in Spark Ada, to assure that they imply each corresponding function's specification. The team lead signs here if this is true for the current versions upon review. Should be blank otherwise.

Given the above, engineers have provided evidence that translation rules have been followed, the implementation appears to be correct, and that its provability has been established through correct preconditions/postcondition encoding in Spark Ada (or another proof-supporting source code language). In addition, the team has vetted the implementation for missing or extraneous function implementation.

11.4 Applying Implementation Strategies

Having defined a partition, the engineer/analysts now produce an implementation of each specification part. The implementation should be written in a language against which proof of correct implementation can be constructed, such as SPARK Ada.

One of two basic approaches is used. One either produces evidence that the implementation is practicably correct and complete, or one shows that the implementation is sufficiently correct for contract guarantees.

11.4.1 Pure Refinement

If a specification part utilizes only data types for which there is a one-to-one and onto mapping within the limited value range available to source code implementation, then one can perform a pure refinement of the specification.

All operational specification can be copied from equivalent operations and functions of the source language. If the specification provides a declarative definition of a function, its implementation can be derived from the declaration.

One shows that all data types utilized in the specification part are refinements of data types in the specification by being one-to-one and onto.

Under the "Retrenchment Models" worksheet of the "evidence-fre.xlsx" spreadsheet, one must fill out the following field:

- **Data Refinement**: Here one shows that a given data type of the specification has a one-to-one and onto retrenchment rule. This must be shown for each data type of the specification part for which Pure Refinement is to be assured.

If this field cannot be checked for a given specification data type, then the Pure Refinement assurance approach is not supported.

11.4.2 Refinement with Retrenchment Rules from Nondeterministic Specification

If a formal specification is built as a nondeterministic specification [14] using bounded interval arithmetic in declared lemmas and axioms, then one can assure refinement under retrenchment rules by determining the following retrenchment evidence for the "Retrenchment Models" worksheet for the specification part:

- **Sound Bounds Confirmed**: The team analyzes the specification part or implementation part using a sound bounds analyzer. This determines the amount of accumulating error for any given data value propagating through the implementation relative to values in the specification. Such error accumulates, for example due to retrenchment between real values in specification and floating-point values in implementation. For example, the PRECiSA tool for PVS can determine sound bounds on all values computed in specification on the reals. Once one computes sound bounds on error of values under retrenchment, one can determine whether lemmas and specification stated with closed intervals satisfy the computed bounds. If they do, then proven lemmas hold under the error in execution of the implementation (as the error will fall within the computed sound bounds). This field is indication that a sound bounds analysis has been conducted for error on values of a given type, and that declared closed bounds on error of the specification and lemmas and axioms are satisfied.
- **Analysis Data**: This field should point to the analysis tools and data from which sound bounds were concluded to satisfy the nondeterministic specification's declared error bounds on values.
- **Analysis Retrenchment Match:** The team must confirm that the retrenchment rules mapping data values from implementation to specification are the same as those assumed in any analyses. The team lead signs here if the engineering team agrees that this is the case.

If the above fields are affirmative for each of the data types utilized by a specification part, then then the axioms of the specification part are sound, and the lemmas proven for it are still verified for a correct implementation.

One can therefore conclude that contract guarantees implied by proof and testing on the specification will hold on the correct implementation source code.

11.4.3 Functional Implementation from Specification with Retrenchment Rules

If one wishes to prove that an implementation of a function satisfies its specification under retrenchment rules, then one must show that the PVS preconditions and postconditions defined on the SPARK Ada implementation imply the specification of the function. This data field was discussed under mirroring.

11.4.4 Sufficient Implementation

If one cannot show that an implementation is a pure refinement, or a limited refinement under retrenchment rules, or a correct functional implementation, then one must show, individually, for

each guarantee of each interface contract of a specification part, that either (a) the guarantee hold for the implementation because it holds for the specification, or (b) that the guarantee holds for the implementation solely dependent on implementation properties. The latter can be performed, in many cases, with testing. The former often requires showing specific properties of the implementation relative to the specification. As discussed in a previous section, numerical stability is a common property utilized in this regard.

Assuring sufficient properties for implementations for all relevant contract guarantees involves performing specialized analyses on the implementation and filling out evidence required in the "Sufficient Implementation" worksheet of the "evidence-fre.xlsx" spreadsheet. The following data records must be collected:

- **Part**: Each specification part with an implementation that is to be determined correct and complete through implementation properties should be listed in the worksheet under this column.
- **Interface Contract**: Each interface contract that must be assured for the given specification part must be listed in this column for the given specification part.
- **Guarantee:** Each guarantee of each listed contract must be assigned a line in the table.
- **Implication:** The implication by which a given guarantee holds as a result of properties of the implementation and specification must be stated in this field. Every guarantee must state a clear implication. The implication may be deductive or inductive. But inductive implication is an inherently weak assurance.
- **Property Name:** Each property of the implementation part for one or more implications must be listed in this column.
- **Supporting Evidence:** The supporting evidence for each property must be listed in this field. This can be analyses tools and their input and output files, reasoning, logic, or other analyses. Analyses might include timing analyses or numerical analysis on the implementation algorithm, for example.

The above evidence is used to conclude that sufficient reasoning about properties of the implementation, in conjunction with properties of the specification, implies that guarantees of enforced contracts are met.

It is often the case that properties required in implications will need to be determined for implementation parts other than those of a given specification part. While the argument and evidence is not well-organized for this, cross-referencing withing the above worksheet is sufficient.

It should be noted that the argument for this implementation approach is a template with generative elements. The argument itself can be expanded with reasoning generated from the above table should a full argument for the corresponding evidence be desired. For example, the argument template will generate a sub-argument for each contract, with sub-argument for each interface contract, and sub-argument for each guarantee, with analysis evidence for each required property of the implementation.

11.5 Assuring the Verification Conditions are Sufficiently Satisfied

Having implemented a specification part, one must show that the implementation of the specification part satisfies all required data type properties and preconditions and postconditions on function definitions, so that the implementation mapping from the specification is more likely to be correct.

SPARK Ada will automatically generate *verification conditions* (VCs) from data type information and annotating preconditions and postconditions. It will attempt to utilize automated formal methods to discharge these verification conditions.

Some verification conditions might not be automatically provable. These will be listed by the tool. To assure implementation, engineers must provide data about which VCs are not discharged. They must also determine that the undischarged VCs are in fact true for the implementation using various analyses and/or inductive reasoning.

Engineers should collect assurance evidence about VCs in the "VCs" worksheet of the "evidence-fre.xlsx" spreadsheet. The following data fields should be collected:

- **VC Satisfaction:** This field should indicate the file(s) where SPARK Ada (or other language tools) reports discharged and undischarged VCs for the present version of the implementation. It should be blank if that has not been done for the present version.
- **Undischarged VCs**: Each undischarged VC should be listed in this table.
- **Undischarged VC Sign-Off:** Each VC that is not discharged must be manually discharged by sign-off from the engineering/analysts team. The reasoning and analysis for discharge should be reviewed and then the team, in agreement, should sign off on discharge of each VC for which they are convinced sufficient proof of truth has been provided.
- **Outside VC Discharge Rationale**: Each VC that is not discharged must be analyzed by other tools. This field lists the tools and analyses that are utilized, including general inductive reasoning. All related documentation should be present here. If artifacts are not for the current version of the implementation, they should not be listed here.

This evidence is applied to argue that all VCs are either discharged by SPARK Ada, or the team believes they are discharged through adequate proof by other means.

11.6 Conclusions

Having performed the steps above, one has produced an implementation and the evidence to argue that it is a correct implementation of the formal specification part. This process should be performed on all specification parts.

Note that the above process sometimes involves evidence from multiple parts of an implementation, and therefore not all evidence can be provided until other implementation parts on which it depends (for example, cross-used data types) are complete.

12 References

1. Knight, John. "Fundamentals of Dependable Computing." CRC Innovations in Software Engineering and Software Development: Boca Raton, FL, USA (2012).
2. Frama-C. https://frama-c.com
3. SPARK Pro (Ada). https://www.adacore.com/sparkpro
4. Knight, John C., and Jonathan Rowanhill. "The indispensable role of rationale in safety standards." International Conference on Computer Safety, Reliability, and Security. Springer, Cham, 2016.
5. Crow, Judy, et al. *Evaluating, testing, and animating PVS specifications*. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.
6. Owre, Sam, John M. Rushby, and Natarajan Shankar. "PVS: A prototype verification system." International Conference on Automated Deduction. Springer, Berlin, Heidelberg, 1992.
7. Proof Verification System (PVS). https://pvs.csl.sri.com
8. Bertot, Yves. "A short presentation of Coq." *International Conference on Theorem Proving in Higher Order Logics*. Springer, Berlin, Heidelberg, 2008.
9. Coq. https://coq.inria.fr
10. Goal Structuring Notation. https://scsc.uk/gsn
11. Holloway, C. Michael. *Understanding the Overarching Properties*. National Aeronautics and Space Administration, Langley Research Center, 2019.
12. Isabelle. https://isabelle.in.tum.de
13. Banach, Richard, and Michael Poppleton. "Retrenchment, refinement, and simulation." *International Conference of B and Z Users*. Springer, Berlin, Heidelberg, 2000.
14. Rowanhill, Jonathan. “Nondeterministic Specification for Specification over the Reals.” Technical Report. Dependable Computing LLC. 2019.
15. Moscato, Mariano, et al. "Automatic estimation of verified floating-point round-off errors via static analysis." *International Conference on Computer Safety, Reliability, and Security*. Springer, Cham, 2017.
16. Hocking, Ashlie B., Jonathan C. Rowanhill, and Ben L. Di Vito. "An analysis of implementing PVS in SPARK Ada." *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*. IEEE, 2020.