

# Echo: A Practical Approach to Formal Verification

Elisabeth Strunk  
University of Virginia  
151 Engineer's Way  
Charlottesville, VA 22904-4740  
+1 434.982.2225  
strunk@cs.virginia.edu

Xiang Yin  
University of Virginia  
151 Engineer's Way  
Charlottesville, VA 22904-4740  
+1 434.982.2225  
xyin@cs.virginia.edu

John Knight  
University of Virginia  
151 Engineer's Way  
Charlottesville, VA 22904-4740  
+1 434.982.2216  
knight@cs.virginia.edu

## ABSTRACT

Safe operation is crucial to safety-critical systems, and formally verified implementations are desirable. Traditional formal verification approaches follow the Floyd-Hoare style, setting up a direct compliance argument between an abstract formal specification and a concrete implementation. Such approaches require proofs of large numbers of verification conditions. The process of generating and proving verification conditions can require significant skill and can be time-consuming.

In this paper, we introduce a general formal verification approach that closely models the Floyd-Hoare pattern, yet avoids the tedious direct compliance proof between the formal specification and the implementation. The approach moves the major proof step to a point between two abstract specifications.

Our preliminary design takes PVS as our specification language and SPARK Ada as our implementation language. We first use a human-guided refinement to manually generate Ada code along with appropriate SPARK annotations from a PVS specification. We then verify the annotations' compliance with the specification by (1) mechanically extracting a PVS specification from them, and (2) proving that properties of the generated specification imply all of the properties of the original. We rely on the existing SPARK toolset to verify the Ada code against the SPARK annotations. The process is largely automatic or computer-aided.

A case study using a hypothetical avionics system provides a preliminary indication that this approach is feasible and practical.

## Categories and Subject Descriptors

D.2.4 Software/Program Verification *Correctness proofs, formal methods.*

## General Terms

Reliability, verification.

## Keywords

Formal specification, formal verification.

## 1. INTRODUCTION

For safety-critical systems, such as fly-by-wire flight-control systems in aircraft, the risk of unsafe operation must be reduced to an acceptable level. One element of risk reduction is ensuring that the system's software is both specified and implemented correctly. Formal specification is the most effective tool for developing precise specifications. It permits extensive analysis and in some cases even animation of a specification so as to engender confidence in the correctness of the specification.

When it comes to verification of an implementation, the difficulties are considerable. Testing alone is not sufficient because it is infeasible to conduct the number of test cases required to establish a high level of confidence in anything but the simplest system [3]. Automatic code generation from the specification can, in principle, assure that an implementation correctly implements a specification, but its applicability is limited. It remains difficult to automatically generate a well-structured and efficient implementation for most safety-critical systems. Existing approaches to formal verification tend to be tedious and difficult to use. In this paper, we introduce a new verification technique that provides a practical approach to showing that a high-level language program implements its specification.

Generally speaking, a formal specification is the "expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy" [4]. It abstracts unnecessary implementation details to give a high-level mathematical model of the system. Formal rules can be used to mathematically prove that the system behaviors satisfy the formal specification. This process is formal verification.

Theoretically formal verification can be done using Floyd-Hoare verification. This approach sets up a compliance argument between the abstract formal specification and the concrete implementation. The argument involves proving a theorem that the execution of the implemented sequence of operations, starting in a state defined by the precondition, implies the postcondition. Establishing a proof of the theorem requires a statement-by-statement analysis of the program, a process that involves the creation of numerous mathematical statements called verification conditions (VCs). Although such formal proof is ideally suited to the development of safety-critical systems, its adoption has been limited. The process of generating VCs requires extensive effort and, once they are generated, the proof of the VCs requires significant skill and can be time-consuming, especially when the specification has abstracted away significant amounts of detail.

In this paper, we introduce a general verification approach which closely models the Floyd-Hoare pattern, yet avoids the tedious direct compliance proof between the formal specification and the implementation. In our prototype design, we set up and prove an equivalence argument from the original specification to certain SPARK annotations [1] through a process that involves:

- refinement from the formal specification to SPARK Ada annotations;
- extraction of a second specification from the SPARK annotations; and
- proving that the second specification implies the first.

Finally, we use the existing SPARK toolset (examiner, simplifier, and proof checker) to verify the Ada implementation against the SPARK annotations. A case study provides preliminary results that suggest this approach is practical and can be largely automated or computer-aided.

The remainder of this paper is organized as follows. Section 2 overviews existing approaches for assuring implementation properties, and the rationale behind our approach. Section 3 describes our approach in an abstract way, and Section 4 details the approach for the case of PVS and SPARK Ada. A case study and its results are shown in Section 5. Limitations of our approach and future work in this direction are discussed in Sections 6 and 7. We conclude in Section 8.

## 2. ASSURING IMPLEMENTATION PROPERTIES

In this section, we discuss different ways of assuring an implementation’s correctness with respect to its specification, including traditional forms of formal verification and automatic code generation. Then we introduce the underlying rationale of our approach. We claim that our approach is practical and greatly mitigates the difficulty with direct compliance approaches.

### 2.1 Traditional Approaches to Verification

Traditional approaches to formal verification take one of the two forms shown in Figure 1:

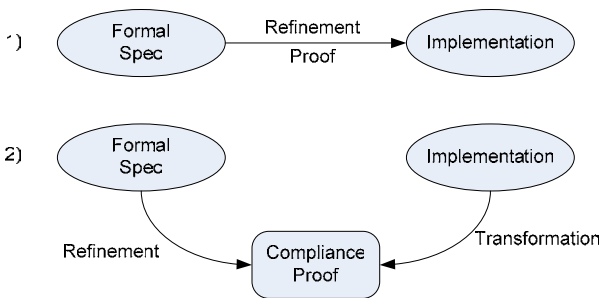


Figure 1. Traditional Approaches to Formal Verification

1) Starting with the formal specification, a sequence of refinements is applied. Each refinement adds implementation details to work towards an executable implementation. For each refinement, a proof that the refinement preserves application

properties is established. This process of refinement continues until a fully verified implementation is reached.

2) The formal specification and the implementation are combined to form a theorem of equivalence. A proof of the theorem is created by concurrently refining the specification as described above and abstracting detail from the implementation using the semantic definitions of implementation language elements. Again, this process continues until a compliance proof can be created.

Each of the two approaches requires some form of direct compliance proof between the specification and the implementation. This approach to verification requires generation and proof of many detailed lemmas and theorems. The approach is often hard to automate and requires significant time and skill to complete.

### 2.2 Automatic Code Generation

Automated translation, or code generation [10], of a formal specification to implementation code provides an alternative to verification for assuring an implementation’s correctness against its specification. This approach constructs an implementation automatically from the specification using formal translation rules. Compliance between the implementation and the specification is implied by the translation rules, as long as the rules preserve specification semantics. If the translation rules are correct, it guarantees that the behavior of the implementation is consistent with the formal specification and thus significantly reduces the cost of compliance proof in conventional verification.

Automatic code generation is gaining increasing prominence under the name *model-based development*. However, its success at present is primarily confined to control systems. For most safety-critical systems, it is very difficult to automatically generate a well-structured and efficient implementation from the formal specification. The verification approach described here applies to systems whose specifications cannot be implemented automatically.

### 2.3 The Echo Approach to Verification

Our goal is to avoid the difficulty of developing a direct compliance proof between an abstract specification and a concrete implementation. An outline of our approach is shown in Figure 2.

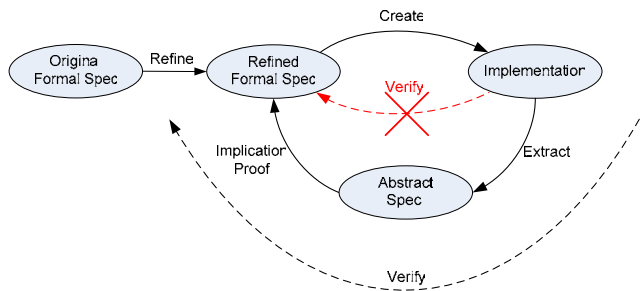


Figure 2. The Echo Verification Approach

The first step in our approach is to refine the original formal specification to remove any semantics that cannot be implemented directly, such as real-number arithmetic and infinite sets. Domain experts must then review and validate the

specification refinement to ensure that any semantic changes are appropriate for the application. This is necessary for any formal specification refinement approach, because there is no implementation that can behave in accordance with the original specification. Our approach ensures that these decisions are made at the specification level instead of allowing them to be made by a programmer, who might not understand application domain behavior.

The next step in our approach is to implement the refined formal specification in a manual but rigorous way. Then, instead of directly verifying the implementation's compliance with the specification, we extract an abstract specification from the implementation automatically. We then prove that the resulting extracted specification implies the original refined one. In this way, we prove that the implementation complies with the original specification but move the major proof step to a point between two abstract specification models. If the two specifications are of similar structure, the implication argument is relatively simple to set up and can be largely proved mechanically.

Tudor *et al.* have developed a mathematically-based and largely automatic verification process called *witnessing analysis* for code automatically generated from Simulink [9]. Simulink is a tool used to specify control laws that can automatically generate Ada source code that implements the control law specification. The Ada code generator for Simulink is not trusted to ultradependable levels, however, and so verification of the Ada code is necessary in critical systems. Tudor *et al.* used a toolset with manual assistance to produce a formal specification in Z from Simulink, compared it with a Z specification extracted from the Ada code generated by Simulink, and then produced and proved verification conditions for the compliance argument between the two. Our approach is similar to theirs, but we look at verification in a more general way. We characterize classes of languages to which our work can apply, and define a verification process that is otherwise language-independent.

### 3. VERIFICATION PROCESS

In this section, we present a more detailed model of our verification approach. We begin by characterizing general specification and implementation languages that could be used in our process. We then explain the general approach we take to verification.

#### 3.1 Language and Tool Requirements

Our work assumes several languages and tools with particular characteristics.

1. An abstract *specification language*. That the specification language is abstract is not a requirement *per se*; in general, our verification process would work if the specification language and implementation language were the same, with translation rules that each statement be left unchanged. The requirement for abstraction is aimed at allowing implementation detail to be left out of the specification, so that our approach would be useful in the development process for a system.
2. A mechanical prover that can analyze inferences constructed using the specification language. This could be an automatic theorem prover, or a proof checker; the goal is to know what implication means by having a formal set of rules for showing it.

3. An *implementation language*. This will most likely be a high-level programming language, but could be any language from which an executable program can be created with assurance that the program exhibits the behavior specified by the language constructs.

4. A language to specify declarative properties of the implementation language constructs. We refer to this as the *property language*. Again, the ability to specify declarative properties is not strictly a requirement; properties of high-level language programs can be deduced. This can become impractical, however, because properties of the algorithms used to implement a specification can be observed, even if those properties were not required by the specification itself. Because the proof between the extracted specification and the original specification is over implication and not equivalence, the extra properties that stem from the algorithms do not in theory interfere with the approach. However, they are likely to make the specification extraction and implication proof significantly harder since those processes must account for unnecessary detail. Thus, we assume that the properties which specific algorithms are meant to achieve can be stated in a declarative way.

5. An automatic toolset that can ensure that the implementation complies with the declared implementation properties. Again, this is not strictly necessary, but on a small scale Floyd-Hoare analysis can be practical to complete mechanically. We concentrate on mechanizing our approach as much as possible.

Languages satisfying our requirements exist. Examples for the specification language are PVS with its incorporated theorem prover and Z with the ProofPower tool [6]. For implementation languages, examples are SPARK Ada with its annotations, Java with JML [5] annotations, and Spec# [2] from Microsoft.

#### 3.2 Verification Steps

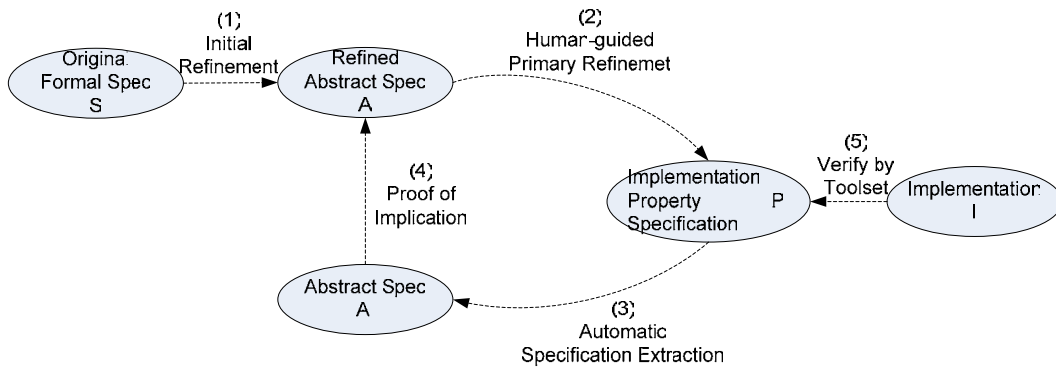
As we have described in Section 2, our verification approach consists of four major steps:

1. An initial refinement of the original formal specification to remove any semantics that cannot be implemented. We refer to this specification as the *restricted formal specification*.
2. A manual, primary refinement from the restricted formal specification to an annotated executable implementation.
3. An automatic extraction of an abstract specification from the implementation.
4. A proof that the properties of the extracted specification imply the properties of the original specification.

The Echo verification argument is based on steps 3 and 4. Provided that step 3 is either automated or mechanically checked, and provided that the proof in step 4 can be constructed, we have a complete argument that the implementation behaves according to the specification. The goal with steps 1 and 2 is to automate them to the extent possible and to use them to assist steps 3 and 4.

##### *Step 1. Creation of the Restricted Specification*

First we manually refine the original formal specification  $\mathcal{S}$  into an implementable specification  $\mathcal{A}$  by removing any unimplementable semantics such as real-number arithmetic and infinite or arbitrarily large sets. This step is often overlooked in



**Figure 3. Refined Model for Our Verification Approach**

formal specification; specification languages specify arithmetic and implementers are left to choose necessary precision and maximum possible set sizes. If formal verification is to be used in a system, the specification must have these elements removed because otherwise the implementation cannot exhibit behavior that is specified. For example, a function that divides some arbitrary integer by another and returns a real number result will not comply with the specification if its inputs are 1 and 3, respectively, because the calculated result must have infinite precision in order to express the required real-number result. In many cases, default precision is sufficient; but this must be established in the context of the application domain, and so must be determined at the specification level.

Restricting the specification to a finite set of states enables a large number of results outside of traditional verification to be employed. Significant practical barriers to the use of other technologies for verification remain, however. For example, while arithmetic in this case must be finite, there are still very large numbers of potential values that numeric variables can take. The combinations of potential states of several numeric variables can quickly exceed the number of states that can be reasonably dealt with. Our approach still has considerable value in that it provides guarantees on computed results in an abstract framework.

#### Step 2. Creation of the Implementation

Next, we refine the restricted specification  $A$  into an implementation  $I$  along with appropriate implementation-level property specifications  $P$ . This step is accomplished by humans, although it can be automated to the extent possible given application characteristics. After this primary refinement, the automatic toolset can ensure the implementation  $I$  complies with the properties specified by  $P$ .

Since our work is targeted at systems that are not amenable to automatic code generation, this step must be performed manually. We are not especially concerned about the mathematical complexity of this step because human guidance is involved, and humans would have to construct an implementation even if they were not using our approach.

When the implementation is constructed, it is essential that the constructs created in the declarative property language not be of higher order than the abstract specification language. Generally (as with PVS and SPARK) the declarative language will be of equal to or lower order than the specification language, so this problem does not arise.

#### Step 3. Specification Extraction

To argue that the implementation is correct, an abstract specification  $A'$  is extracted from the implementation properties  $P$  by an automatic or semi-automatic but mechanically-checked translator. We know that this process is theoretically possible since the implementation properties must be expressible by the specification language. A simple method for automating the process would be to write extraction rules for the implementation language's grammar. Such a strategy might be too simplistic, however, because the way in which the extracted specification is created influences the difficulty of the implication proof. We believe that capturing key design decisions during the primary refinement and using them to inform the specification extraction can greatly facilitate the implication proof.

#### Step 4. Implication Proof

The implication argument,  $A' \Rightarrow A$ , is shown by setting up and proving an implication theorem in the prover associated with the specification language. If the specification language allows first-order quantification, then whether the implication holds is undecidable. However, we have restricted the use of the specification language to be less expressive than first-order logic by requiring that all sets be of bounded cardinality. In theory, then, the problem is decidable.

If the extracted specification is so different from the original specification that equal or greater human effort is required to construct the proof of implication than would have been expended on traditional Floyd-Hoare analysis, then our work provides no benefit to software developers. However, as explained above, capturing information during the primary refinement can allow the extraction of a specification that is similar enough to the original specification so that the proof can be simplified. We plan to study what properties of specifications we can capture during primary refinement and specification extraction to construct an argument that in many situations, this will not be a problem.

The whole process is largely mechanical. The only activities that need substantial human intervention are the initial refinement, the primary refinement, and setting up the implication proof. In this way, if we have confidence in the prover associated with specification language, the automatic toolset associated with the implementation language, and the automatic specification extractor (or extraction checker, if automatic extraction is infeasible for a particular application), we will have confidence that the program complies with the original specification.

## 4. Verification of SPARK Ada against PVS

We have developed an instantiation of our general verification approach using SPARK Ada implementations and PVS specifications. Before going into the details, we give a short background introduction to PVS and SPARK Ada.

### 4.1 Languages and Tools Used

*Specification Language: PVS*

The PVS language is a higher-order logic specification language. It has been used for algorithm analysis in several critical system domains. No auto-code generator for PVS exists, however, so it is important that a PVS specification be verifiable if research results that depend on it are to transfer into practice.

*Prover: PVS System*

PVS is a system for creating specifications, constructing proofs, and checking proofs mechanically. Proofs can be constructed using a collection of primitive inference procedures and more complex deductive strategies that can be applied interactively [7]. The case study in Section 5 shows that the PVS system's powerful and highly mechanical deduction procedures enable us to automatically discharge most of the verification conditions in the proof and discharge the rest with some guidance.

*Implementation Language: SPARK Ada*

SPARK Ada is a high-level language designed for the development of software for high integrity or safety critical applications [1]. The programming language part of SPARK comprises a kernel that is a subset of the Ada language. It accepts a subset of Ada, not allowing features which create difficulties in proving that a program is correct (with respect to its annotations), e.g. GOTO statements or pointer variables.

For clarity, we refer to SPARK Ada implementations as Ada implementations, although we assume that only the SPARK subset of Ada is used.

*Property Language: SPARK Annotations*

SPARK Ada includes additional features inserted as annotations in the form of Ada comments. The annotations are used to specify intended properties of Ada subprograms (procedures and functions) in a declarative way.

*Automatic Verification Tools: The SPARK Toolset*

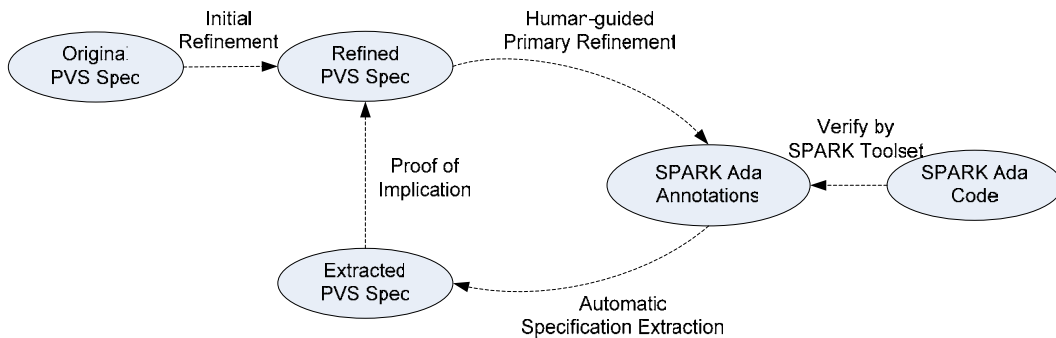


Figure 5. Major Steps in SPARK-PVS Verification

SPARK provides a set of tools for verifying that a program exhibits the properties specified in its annotations. The Examiner is used to generate the verification conditions (VCs) that would be created using traditional Floyd-Hoare analysis. It provides mechanisms to perform data flow analysis, information flow analysis, or formal proof of the annotations. The VCs generated by the Examiner are passed to the Simplifier. The Simplifier attempts to reduce each VC to `true`; if it is unable to do this for some VC, then it passes that VC to the Proof Checker for further interactive manipulation [1]. The overall SPARK verification procedure is shown in Figure 4.

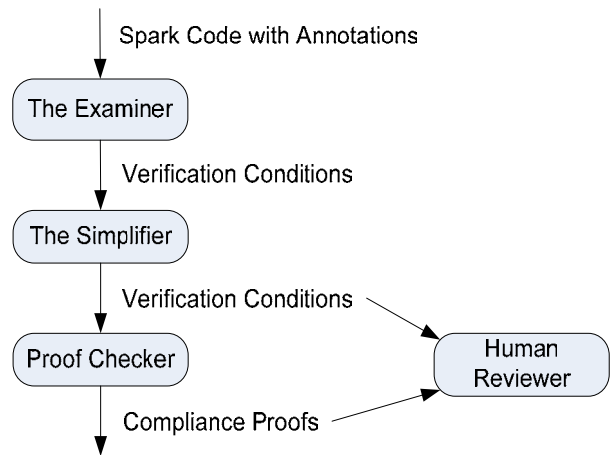


Figure 4. SPARK Proof Process

SPARK and SPARK Ada are interchangeable names, but we use SPARK when speaking strictly of the annotations and tools. We use SPARK Ada when discussing the composite of implementation, annotations, and/or tools.

### 4.2 Verification Process Implementation

The major steps here are: the human-guided initial and primary refinements, the automatic extraction of the specification, and the implication proof. Although there is one more step to verify the Ada code's compliance with its SPARK annotations, we do not focus on it since it is achieved by the SPARK toolset. The

detailed steps are shown in Figure 5 and elaborated below.

### Step 1. Initial Refinement

This step refines the PVS specification to exclude any unimplementable semantics. The SPARK toolset refines real-number arithmetic to finite arithmetic, so that change does not have to be made directly in the specification. However, SPARK’s rules for doing this must be approved by application domain experts. Infinite sets must also be refined in this step and should involve human guidance or review.

An important aspect of the initial refinement step is constraining the expressivity of the specification language. PVS is a higher-order logic, and so it is possible to specify in PVS computations for which no algorithm exists. Restricting sets to be finite, however, means that the semantics of the resulting specification can be expressed using a finite-state machine. Because even finite integer arithmetic has extremely large numbers of states, however, a finite-state machine is not necessarily the best formalism to use in writing the specification.

### Step 2. Primary Refinement

The goal of the primary refinement is to find a mapping  $\tau$ :

$$\tau: \text{PVS Specification} \rightarrow \text{SPARK annotations}$$

Since this mapping is done by humans, we are not concerned with the semantics of PVS specifications or the underlying reasoning in the refinement. We do care about the metrics and rules for the refinement, however, since the primary refinement should be structured in a way that facilitates the automatic specification extraction and the implication proof as much as possible.

A desirable refinement will maintain the structure and modularity of the original PVS specification. Ideally, the underlying semantic models will be identical. For example, a definition in PVS:

```
state: TYPE = [# a: int, b: int #]
foo(st: state) : state = st WITH [a = 1]
```

might be refined into SPARK Ada as:

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

```
procedure foo(st: in out state);
--# derives st from st;
--# post st.a = 1 and st.b = st~.b;
```

We have designed a prototype set of rules to guide the refinement. The rules we used for our case study in Section 4 can be captured in five categories: (1) system structure; (2) non-function type definitions; (3) non-function variable/constant declarations; (4) function definitions/declarations; and (5) other elements.

#### Category 1. System structure

We identify appropriate blocks in PVS such as theories or instantiations of abstract data types, and refine them into SPARK Ada packages. Structuring the implementation in this way enables the extracted specification to reflect the initial PVS structure.

importing relations among theories can be refined into “with” or “--# inherit” clauses in SPARK Ada.

#### Category 2. Non-function type definitions

Basic types such as integers, real numbers, enumeration types, record types, and also their subtypes are directly refined into the corresponding type representations in SPARK Ada. Types that do not have direct representations in SPARK Ada, such as uninterpreted types, parameterized types, set types, tuple types, and some other composite types, should be refined into appropriate forms by human decision with knowledge of their application-specific semantics. For instance, sets can be refined into array types or enumeration types, and tuple types can be refined into record types.

#### Category 3. Non-function variable and constant declarations

Variable and constant declarations can be refined directly into declarations in SPARK Ada. We put them into appropriate SPARK Ada packages, based on their location in the PVS specification. Variables can be refined into own variables for SPARK Ada packages with “--# own” annotations.

#### Category 4. Function declarations and definitions

Function declarations and definitions are the most difficult to encode in SPARK Ada. Since PVS is a higher-order logic language, a function in PVS can be arbitrarily complex. The most appropriate refinement of a single PVS function might be a normal function/procedure, an array, or any other type in SPARK Ada. Humans must decide which refinement is needed.

For PVS functions that are refined into SPARK Ada functions or procedures, any input/output should either be mapped to the input/output in the corresponding SPARK Ada function/procedure or appear in the “--# global” annotation of that function/procedure. We should add “--# derives” annotations for procedures to indicate the relationships among outputs and inputs. Type restrictions on the input variables are preconditions and should be refined into “--# pre” annotations. Type restrictions on the output variables (mostly the function bodies) are postconditions and should be refined into “--# post” annotations for procedures or “--# return” annotations for functions in SPARK.

IF-THEN-ELSE, CASES, and other expressions in PVS are also refined into Boolean expressions in SPARK. For example:

```
st = IF a > 0 THEN st1 ELSE st2 ENDIF
```

is refined into:

```
--# post
--#   (a > 0 AND st = st1) OR
--#   (a <= 0 AND st = st2);
```

PVS predicates are PVS functions in which the range is of type Boolean, and so PVS predicates are refined into Boolean functions in SPARK.

Quantification is limited in SPARK Ada, so in some cases quantified elements must be enumerated in order to refine a quantified expression. We know that this can be done since the initial refinement step bounded set size.

#### Category 5. Other elements

There are other elements such as formula declarations that must be refined. For instance, by writing a statement `FOR ALL a, b: int, p(foo(a, b))` where `p` is a predicate and `foo` is a function, we are quantifying over the function `foo`'s input types to specify its postcondition. Thus the content in predicate `p` must be refined under “`--# post`” or “`--# return`” annotations for `foo`'s representation in SPARK.

While in axiomatic languages axioms are used to specify behavior, in PVS they are generally used to assert a condition that will be true based on the characteristics of the system's operating environment. In the former case, they should be integrated into the main specification structure so that it is clear where they belong in the implementation. In the latter case they need not be included in the implementation property specification unless needed to show machine-enforced properties; the implementation can assume the axioms and is not obligated to enforce them.

Finally, in some situations a faithful refinement may be inappropriate or unnecessary. For example, the case study we present in Section 5 possesses the following abstract definition in the PVS specification:

```
data_id: NONEMPTY_TYPE
data_value: TYPE
data_state: TYPE = [data_id -> data_value]
```

The following example instantiation types:

```
disabled, off, engaged, ah_only: data_value

mode_status: set[data_value] =
  {v: data_value | v = disabled OR
   v = off OR v = engaged}
```

can be mapped to an enumeration type in Ada without changing the underlying semantics:

```
type mode_status is
  (disabled, off, engaged);
```

These refinement decisions arise because PVS is not able to express the types in the case study elegantly. In such cases, the mapping should be recorded for use in the specification extraction. Such decisions must be inspected with great care, however, because the primary refinement must be undone in the specification extraction. Unsound refinements at this stage could thus be masked, leaving a small but important weakness in the formal verification argument. We plan to study criteria for sound refinements in the future, so that it is possible to tell whether the extraction process might mask a defect in the primary refinement.

### Step 3. Specification Extraction

The specification extraction step seeks a mapping  $\tau$  from SPARK annotations to a PVS specification:

$$\tau: \text{SPARK annotations} \rightarrow \text{PVS Specification}$$

It is a mapping from first-order logic to higher-order logic, so it does not have the problem of expressive power that might exist in the primary refinement. The specification extraction is currently done by humans with a set of rules similar to those used for the primary refinement. We plan to formalize the rules and automate the process in the future. This way, implementation errors cannot be masked during the specification extraction as long as the

formal extraction rules are correct. In this section, we briefly describe our current specification extraction rules.

A PVS theory is extracted from each SPARK Ada package, with `importing` statements extracted from the `inherit` statements. Type definitions and variable/constant declarations that were trivially refined into SPARK Ada during the primary refinement can be trivially extracted back into PVS. Complex refinements can be reversed as described above.

A function is extracted from each subprogram in SPARK Ada. Type restrictions over input types are extracted from “`--# pre`” annotations, and PVS function bodies are extracted from “`--# post`” or “`--# return`” annotations. Enumeration performed in the primary refinement can be reversed to universal quantification over the type in this stage. Showing soundness of the reversal requires a simple check that the enumeration included all elements of the type.

As an example of the specification extraction, take:

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

procedure foo(st: in out state);
--# derives st from st;
--# pre st.a = 0;
--# post st.a = 1 and st.b = st~.b;
```

We extract:

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

foo(st: {s: state | s`a = 0}) : state =
  st WITH [`a = 1]
```

### Step 4. Implication Proof

A proof that the PVS specification resulting from the automatic specification extraction has all the properties of the original specification is done by writing an implication theorem in PVS and mechanically or interactively proving it in the PVS system.

If, by our effort in the above refinement and extraction, we can make the two PVS specifications as similar as possible in structure and size, the implication theorem can be easily set up in PVS. An example theorem fragment for a predicate in the original specification would be:

```
implication: THEOREM
  FORALL (st: state) :
    p_extracted(st) => p_original(st)
```

The implication theorem can be interactively proved by the PVS theorem prover, and we can thus finish the implication proof.

If the two PVS specifications are not of identical structure, we can still set up implication theorems since they should still possess the same semantics. We plan to investigate the degree of complexity the structural mismatch adds to the proof process, and how we can ensure that the implication theorems capture all of the semantics of the restricted specification.

## 5. CASE STUDY

As an illustration of our approach, we present a case study based on a simple autopilot example taken from the reconfiguration architecture of Strunk *et al.* [8]. In this case study, we have created and verified the implementation of representative functionality and the reconfiguration interface for an aircraft autopilot. The reconfiguration interface is designed to allow the autopilot to be reconfigured under a variety of circumstances. The autopilot implements different specifications for the different modes.

The PVS specification of the autopilot consists of five theories. Three of them define the abstract data types for the system state that are used in applications other than the autopilot. The other two theories define the functional specifications for the two modes of the autopilot.

Each of the functional specifications was refined manually into a corresponding package in Ada along with appropriate SPARK annotations. The annotations were then translated back into PVS theories to produce the extracted specification. Both the refinement and the extraction were done according to the rules discussed above, and the extracted PVS specification was of similar structure to the original one. Implication theorems were written, and proofs of the theorems were constructed and checked using the PVS system.

In the case study, we did not undertake the initial refinement step in the Echo approach, although the specifications do contain some infinite arithmetic using `integer` and `real` types. For this case study, we were not concerned about the upper limits of the integers or the maximum precision of the real numbers, and so we used SPARK Ada's representations and approximations.

To illustrate Echo in more detail, we now present the verification of the autopilot's `halt` function in depth. This function is part of the reconfiguration interface that allows the autopilot to be stopped prior to beginning execution of an alternate specification. For clarity, the relations of the system state, the autopilot application, and the `halt` function are shown in Figure 6.

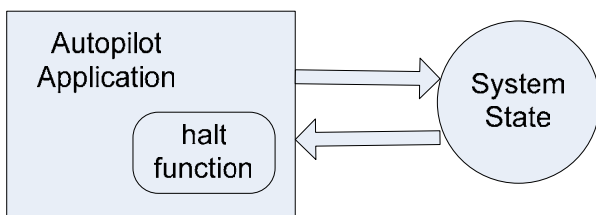


Figure 6: Autopilot Case Study

### 5.1 The `halt` Function

The `halt` function is a function within the autopilot module (which is, essentially, the autopilot application):

```

ap_module: module_spec =
  (# ...
   `halt := ap_halt,
   ...
  #)
  
```

The function involves the modification of certain variables in the system state. It is defined as follows:

```

ap_halt: func(ap_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
   f := (LAMBDA (st: data_state) : st WITH
        [ `alt_hold :=
          IF st(alt_hold) = engaged
            THEN off
            ELSE st(alt_hold)
          ENDIF,
          `hdg_hold :=
          IF st(hdg_hold) = engaged
            THEN off
            ELSE st(hdg_hold)
          ENDIF] )
  #)
  
```

Here `func` is a record type that represents a function. It contains a predicate for the function precondition and a function for the actual function body (which is its postcondition). `Data_state` is a function that maps each system variable to some possible data value.

### 5.2 Primary Refinement

Prior to refinement of the `halt` function, some application-specific decisions were made. For example, the `func` type can be mapped directly to a function/procedure, not a record type. `Data_state` is mapped to a record type, not a function. These decisions were recorded for later reversal. With these decisions in place, the autopilot module was refined into an Ada package with SPARK annotations, within which the `halt` function is as follows:

```

procedure apm_halt;
--# global in out system.state;
--# derives system.state.cur from system.state;
--# post (system.state.cur_alt =
--#       system.state~.cur_alt) and
--#       (system.state.tgt_alt =
--#       system.state~.tgt_alt) and
--#       (system.state.cur_hdg =
--#       system.state~.cur_hdg) and
--#       (system.state.tgt_hdg =
--#       system.state~.tgt_hdg) and
--#       (system.state.left_aileron =
--#       system.state~.left_aileron) and
--#       (system.state.right_aileron =
--#       system.state~.right_aileron) and
--#       (system.state.rudder =
--#       system.state~.rudder) and
--#       (system.state.elevator =
--#       system.state~.elevator) and
--#       (system.state.ap_fcs_cmd =
--#       system.state~.ap_fcs_cmd) and
--#       ((system.state~.alt_hold =
--#         system.engaged and
--#         system.state.alt_hold =
--#         system.off) or
--#       (system.state~.alt_hold /=
--#         system.engaged and
--#         system.state.alt_hold =
--#         system.state~.alt_hold)) and
--#       ((system.state~.hdg_hold =
--#         system.engaged and
--#         system.state.hdg_hold =
  
```

```
--#      system.off) or
--#      (system.state~.hdg_hold /=
--#      system.engaged and
--#      system.state.hdg_hold =
--#      system.state~.hdg_hold));
```

Here `system.state` is an instantiation of the record type refined from `data_state`. Note that we did not define it as a direct input/output for the function, but as a global variable. The postcondition in the “`--# post`” annotation restricts change to only two fields of `system.state`, while all other fields must keep their values inside this function. This is a case for which all the fields within the record must be enumerated.

The detailed SPARK Ada code is not shown here since Echo uses the existing SPARK toolset to show the compliance of the code against the annotations.

### 5.3 Specification Extraction

The next step is to extract the PVS specification. For the `halt` function, the extraction step produces:

```
apm_halt(st: data_state) : data_state =
  st WITH [ `alt_hold :=
    IF st `alt_hold = engaged
      THEN off
      ELSE st `alt_hold
    ENDIF,
  `hdg_hold :=
    IF st `hdg_hold = engaged
      THEN off
      ELSE st `hdg_hold
    ENDIF ]
```

Note that the extracted function is similar in structure to the original `halt` function. But here, the initial extraction of `data_state` results in a record type. We have to apply the reverse of the application-specific decisions before we can produce an implication theorem. After reversing the decisions for `func` and `data_state` that we discussed in the primary refinement, we get:

```
apm_halt_f(st: data_state) : data_state =
  st WITH [ `alt_hold :=
    IF st(alt_hold) = engaged
      THEN off
      ELSE st(alt_hold)
    ENDIF,
  `hdg_hold :=
    IF st(hdg_hold) = engaged
      THEN off
      ELSE st(hdg_hold)
    ENDIF ]
```

```
apm_halt_1 : func_type_1 =
(# pre := (LAMBDA (st: data_state) : true),
  f := apm_halt_f
#)
```

### 5.4 Implication Theorem

The last step is the implication proof. Since the extracted PVS specification is of similar structure to the original one, it is simple to write an implication theorem in PVS:

```
ap_module_equiv: THEOREM
```

```
FORALL (... st: data_state ...) :
  ...
  apm_halt_1 `f(st) =
    ap_module `halt `f(st) AND
  ...
```

This theorem actually results in an equivalence argument rather than an implication argument. All of the theorems in the case study show equivalence, which is sufficient since equivalence subsumes implication.

The statement regarding the equivalence of the `halt` function is quickly proved in the PVS theorem prover using only the (`grind`) command.

## 5.5 Verification Results

After running the above primary refinement and specification extraction processes for the whole autopilot example, we proved all of the equivalence theorems using the PVS system. Besides the formulas to be proved resulting from the equivalence theorems, 10 type-correctness conditions (TCCs) were generated. All the TCCs were discharged by the theorem prover within seconds, and all the formulas were proved in the theorem prover except one. After investigation, we found this unprovable formula was generated from a translation error in the primary refinement. When it was corrected, the whole process was conducted again. This time all TCCs were discharged and all formulas were proved in just a few seconds.

## 6. LIMITATIONS OF THE APPROACH

Although the preliminary results for the Echo approach are promising, there are still some theoretical and practical limitations. In particular, we note the following:

1. The primary refinement might lead to a state explosion when constructing SPARK annotations because of the expressive power of PVS. The SPARK annotations are first order whereas PVS is not. Enumerating all elements of several very large sets could significantly reduce the clarity of the implementation.
2. If the restricted PVS specification is of higher than second-order logic, it will not have the same structure as the extracted specification. In this case, the application-specific decisions recorded in the primary refinement would be extremely important in the implication proof. We need to be sure that applying the reverse of these decisions will not mask any possible errors in the refinement. Unfortunately, so far we have not found a good way to deal with this problem.
3. The SPARK examiner, simplifier and proof checker are totally automatic. They behave and can only behave according to a set of pre-defined rules. There might be verification conditions left unproved that are obvious to a human [1]. This means human intervention might still be needed in the verification of the Ada code against the SPARK annotations, although this might be just a small portion of the effort and is currently acceptable to SPARK Ada users.

## 7. FUTURE WORK

Our next step is to investigate the things that can be expressed by PVS but not SPARK Ada, complete the rule set for primary

refinement, and implement a totally automatic translator for the construction of the extracted specification.

We also plan to provide similar capabilities for other languages and formally characterize the class of specification and implementation languages to which our verification approach applies. Furthermore, we seek a generalization of our verification strategy that allows it to be easily ported to various languages in that class.

## 8. CONCLUSION

In this paper, we introduced a general formal verification approach that consists of the procedures of two refinement steps, specification extraction from the implementation, and implication proof. We believe that this approach is novel since the implication proof is carried out between two abstract specification models, thus avoiding or mitigating the difficulty of the direct compliance proof of a concrete implementation against an abstract formal specification in traditional Floyd-Hoare verification. A preliminary design is described, and preliminary results from a case study indicate that our approach is feasible.

## 9. ACKNOWLEDGMENTS

This work was sponsored, in part, by NASA under grant number NAG1-02103.

## 10. REFERENCES

- [1] Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.
- [2] Barnett, M., K. Rustan M. Leino, and W. Schulte, *The Spec# Programming System: An Overview*, In CASSIS 2004, LNCS vol. 3362, Springer, 2004.
- [3] Butler, R and G. Finnelli, *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*, IEEE Transactions on Software Engineering, Vol. 19, No 1, January 1993.
- [4] van Lamsweerde, A., *Formal Specification: a Roadmap*, Future of Software Engineering Conference (A. Finkelstein, Ed), ACM Press, 2000.
- [5] Leavens, G. T. and Y. Cheon, *Design by Contract with JML*, draft paper, Iowa State University, April 2005.
- [6] ProofPower, (<http://www.lemma-one.com/ProofPower/index/index.html>)
- [7] PVS Specification and Verification System, (<http://pvs.csl.sri.com/>)
- [8] Strunk, E. A., J. C. Knight, and M. A. Aiello, *Assured Reconfiguration of Fail-Stop Systems*, The International Conference on Dependable Systems and Networks, Yokohama, Japan, June 2005.
- [9] Tudor, N., M. Adams, P. Clayton, and C. O'Halloran, *Auto-Coding/Auto-Proving Flight Control Software*, 23rd Digital Avionics Systems Conference, October 2004.
- [10] Whalen, M. W. and Mats P. E. Heimdahl, *An Approach to Automatic Code Generation for Safety-Critical Systems*, Proceedings of the 14th IEEE International Conference on Automated Software Engineering, October 1999.