

Echo: A Practical Approach to Formal Verification

Elisabeth A. 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 C. 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 formal verification of implementations is a desirable means to increase confidence in safety. 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. Creation of both the conditions and their proofs can be difficult and 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 approach verifies SPARK Ada implementations against PVS specifications. 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. We present an example of the approach using a hypothetical avionics system.

Categories and Subject Descriptors

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

General Terms

Reliability, verification.

Keywords

Formal specification, formal verification.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FMICS'05, September 5–6, 2005, Lisbon, Portugal.

Copyright 2005 ACM 1-59593-148-1/05/0009...\$5.00.

1. INTRODUCTION

For safety-critical systems, such as fly-by-wire flight-control systems in aircraft, the risk of unsafe operation must not exceed acceptable levels. One significant element of risk reduction is ensuring that the system's software is both specified and implemented correctly. Formal specification is the most effective technology for developing precise statements of requirements. It permits extensive analysis of a specification so as to engender confidence in the validity of the specification. The results of the analysis might not apply to the implemented system, however, if the system implementation is not verified to comply with its specification.

Implementation verification presents considerable difficulties in complex software systems. 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 software systems [3]. Automatic code generation from a specification can ensure that a program correctly implements a specification, but its applicability is limited: automatic generation of a well-structured and efficient implementation remains difficult for most complex systems. Furthermore, the code generators themselves are not always verified. Finally, existing approaches to formal verification tend to be tedious and difficult to use. In this paper, we introduce a practical verification technique that uses the analysis capability of a theorem prover to avoid the difficulties of current approaches.

The current state of the art in formal verification involves application of the Floyd-Hoare approach. This technique sets up a compliance argument between an abstract formal specification and a 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 since the process of generating and proving VCs requires extensive effort. The proof of the VCs requires significant skill, 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. We then instantiate the approach with a process in which we set up and prove an equivalence argument from a

formal specification written in PVS [7] to SPARK annotations [1], declarative properties that hold over Ada source code. Our process consists of:

- refinement from the PVS specification to SPARK annotations;
- extraction of a second PVS specification from the SPARK annotations; and
- proof that the properties of the second specification imply the properties of 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. Section 3 describes our approach in an abstract way, and Section 4 instantiates the approach using PVS and SPARK Ada. An example is presented in Section 5. Limitations of our approach and future work in this direction are discussed in Section 6. We conclude in Section 7.

2. CURRENT VERIFICATION TECHNIQUES

2.1 Traditional Approaches to Verification

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

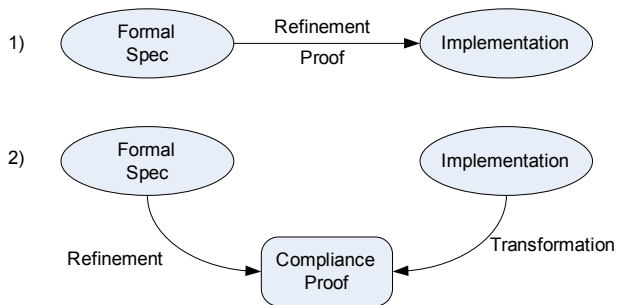


Figure 1. Traditional Approaches to Formal Verification

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 compliance. 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 involves some form of direct compliance proof between the specification and the

implementation, requiring generation and proof of many detailed lemmas and theorems. These approaches are often hard to automate, and they require significant time and skill to complete.

2.2 Automatic Code Generation

Automated translation, or code generation [10], of a formal specification to an implementation provides an alternative to verification for assuring an implementation's correctness with respect to its specification. This approach constructs an implementation automatically from the specification using formal translation rules. Compliance of the implementation with its specification is implied by the translation rules, as long as the rules preserve specification semantics. If the translation rules are correct, it guarantees that the behavior of the implementation is consistent with the formal specification.

Automatic code generation is gaining increasing prominence under the name *model-based development*. However, its success at present is primarily confined to narrow domains such as control systems. For most safety-critical systems, it is very difficult to automatically generate a well-structured and efficient implementation from a formal specification. Also, verifying the translators is extremely difficult. Our verification approach applies to systems whose specifications cannot be implemented automatically using a verified translator.

2.3 Witnessing Analysis

Tudor *et al.* have developed a 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. THE ECHO APPROACH TO VERIFICATION

In this section, we present a more detailed model of our verification approach. We begin by characterizing the classes of specification and implementation languages that are used in our process. We then explain the steps that constitute our approach.

3.1 Language and Tool Requirements

Our work assumes five languages and tools with particular characteristics.

1. An abstract *specification language*. That the specification language is abstract is not a requirement *per se*; for example, our verification process would work if the specification language and implementation language were the same, with refinement/extraction 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. Property deduction can become impractical, however, because properties of the algorithms used to implement a specification can be observed, even if those properties are not required by the specification. Because the proof between the extracted specification and the original specification is of implication and not equivalence, the extra properties that stem from the algorithms do not in theory interfere with the approach. However, extra properties 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 to 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, and so use existing tools when available.

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 C# with Spec# [2] annotations.

3.2 Verification Process

Our verification approach consists of four major steps, shown in Figure 2:

1. An initial refinement of the original formal specification to restrict its semantics to those that can 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 restricted specification.

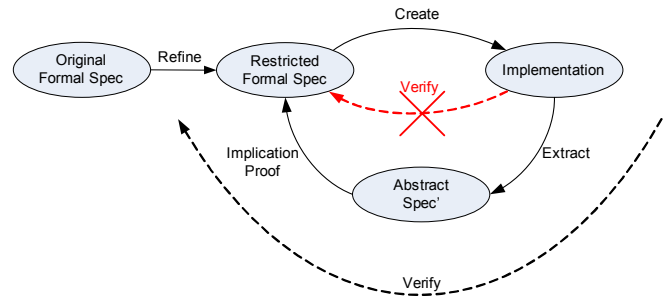


Figure 2. The Echo Verification Approach

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.

We now detail the steps, as shown in Figure 3.

Step 1. Initial Refinement

First we manually refine the original formal specification S into an implementable specification A by removing any unimplementable semantics. This step is necessary for any formal specification refinement approach: for any formal specification with unimplementable semantics, there is no possible implementation that will behave in accordance with the original specification. In traditional refinement approaches, adding limits on set length or precision for real numbers is viewed as adding implementation detail; but in truth, addition of such detail alters the semantics of the specification. 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. If the restricted specification is constructed by program developers, domain experts must then review and validate the restricted specification to ensure that any semantic changes are appropriate for the application.

Restricting the specification's semantics does not necessarily impact abstraction in reasoning about a specification. Decisions on arithmetic precision, for instance, could be documented with the declaration of the real number type. Any use of that number would then be defined to work with the given precision. We do not make any assumptions on where the restrictions are documented; the software developer may do what is appropriate in the context of a particular system.

Restricting the specification to a finite set of states enables a large number of results outside of traditional verification to be employed. Theoretically, the implementation's properties might be guaranteed by an exhaustive state space search. Significant practical barriers to the use of other technologies for verification

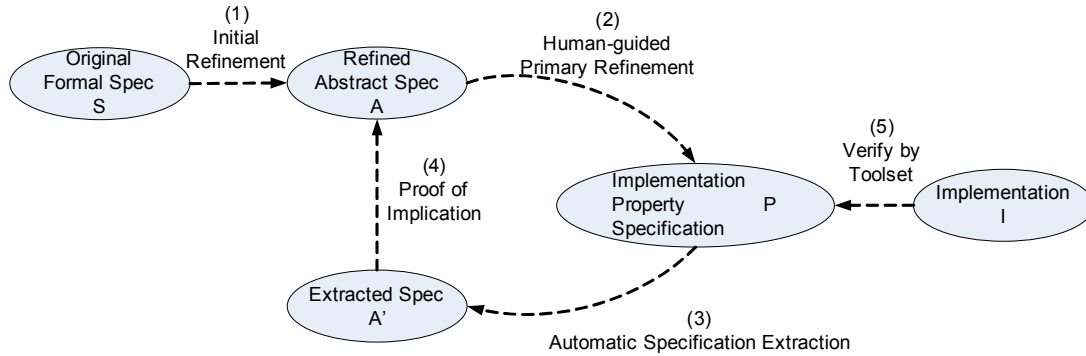


Figure 3. Refined Model for Our Verification Approach

remain, however. For example, while implementable arithmetic is 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 can provide guarantees on computed results in an abstract framework. In many cases, the approach is more practical for guarantees of precise semantics of complex systems than state-search-based approaches.

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 manually, although it can be automated to the extent possible given application characteristics. After this primary refinement, the automatic toolset can ensure that I complies with 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 created, 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 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 straightforward translation 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 hypothesize that, in most cases, all design information included in a specification will be retained in an implementation. Ideally, a specification contains only a statement of requirements

and no design at all. Constructing such a specification, however, is very difficult in practice because of the difficulty in expressing a formal model of requirements without first giving those requirements a particular structure. While an implementation does not have to retain the design included in a specification to be considered compliant, in practice it will generally be similar in structure because repeating the design effort is a waste of resources. If the implementation retains the design information, then the structure of the extracted specification will be similar to the structure of the restricted specification. An injective function from specification identifiers to implementation identifiers can be retained during the implementation stage, and the inverse (with the inverse's domain equal to the original function's image) used to create the extracted specification.

The above hypothesis is implicitly assumed in Floyd-Hoare verification, as well. Because the Floyd-Hoare approach requires a stepwise proof that a function implementation complies with the function's specification, it also implicitly requires that there is a direct correspondence from functions and variables in the specification to functions and variables in the implementation. Thus, we have not added assumptions, only evaluated existing ones in more detail.

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. In many cases, such as in traditional Floyd-Hoare verification, the form of the theorem will be straightforward. In other cases, however, different specification formalisms are suited to expressing different properties, and in order to represent the implication, one or both formalisms must be altered slightly to express comparable properties.

To explain the implication theorem we set up, we first briefly present a general model of specifications based on functions in higher-order logic. Computational state is represented as a variable that is input to and output from a function. The function's precondition is equal to the type restrictions on the input variables, and its postcondition is equal to the restrictions on the output variables. Such a representation fits well with our specific approach involving PVS. If a state-based language such as Z is used, the state schemas can be converted to types, and operations can be converted to functions from some instance of the type to another instance of the type. Input and output variables in a

schema can be accounted for using extra parameters in the function signature for input variables, and composite types to hold the altered state and output variables.

Converting operations to functions requires that operations be deterministic, and it might be the case that specifying deterministic operations overconstrains the system. To allow nondeterminism, we model the specified operation as a function from $(inputs, outputs) \rightarrow \{true, false\}$. In this model, the operation's result is a valid output given some set of inputs if the combination of the inputs and outputs maps to *true*.

Our model permits us to cleanly define properties of the specification as the set of properties of the specification's operations. Each operation has two types of properties:

1. It possesses restrictions on acceptable behavior, denoted by mapping some $(input, output)$ pairs to *false*.
2. It defines required behavior by mapping some $(input, output)$ pairs to *true*. This requirement prevents the implementation from trivially satisfying the specification by doing nothing.

For any input i , if for some output o there is a mapping from (i, o) to *true*, then the implementation must provide one of the possible outputs, given input i .

Because the design information captured in a specification *spec* (e.g., functional decomposition) is retained in the implementation *impl*, there exists an injective function *IFop* from the set *OPspec* of operations in the specification to the set *OPimpl* of operations in the implementation. Likewise, there exists an injective function *IFstate* from the set *STspec* of state variables in the specification to *STimpl* of state variables in the implementation. These mappings are shown in Figure 4 for reference. Showing that the properties of *spec* are implied by the properties of *impl* means showing that, for each operation op_spec in *spec*,

1. For any pair of specification input and output states in_st_spec and out_st_spec , if $op_spec(in_st_spec, out_st_spec) = false$, then $IFop(op_spec)(IFstate(in_st_spec), IFstate(out_st_spec)) = false$. This forces the implementation to possess the restrictions of the specification.

To rephrase this property as one of implication from the extracted specification to the restricted specification, we take its contrapositive: For any pair of implementation input and output states in_st_impl and out_st_impl , $op_impl(in_st_impl, out_st_impl) = true \Rightarrow IFop^{-1}(op_impl)(IFstate^{-1}(in_st_impl), IFstate^{-1}(out_st_impl)) = true$.¹

¹ We use function inverse here as a shorthand for the inverse of the function whose range has been restricted to its image. We know that the original function is injective because mapping two specification operations/state elements to the same implementation operation/state element violates the hypothesis that all design information is retained. Because the original function is injective, we know that the restricted inverse exists.

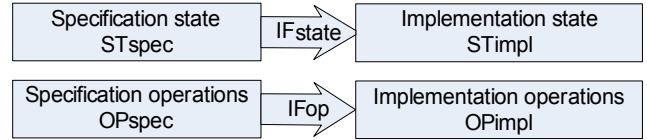


Figure 4. Functions from Specification Constructs to Implementation Constructs

2. For any specification input state in_st_spec for which there is some specification output state out_st_spec such that $op_spec(in_st_spec, out_st_spec) = true$, there exists some implementation output state out_st_impl such that $IFop(op_spec)(IFstate(in_st_spec), out_st_impl) = true$. This forces the implementation to exhibit the functional requirements of the system.

Again, to rephrase the implication in the appropriate direction, we take the contrapositive: if there exists an implementation input state in_st_impl where for all implementation output states out_st_impl , $op_impl(in_st_impl, out_st_impl) = false$, then for all specification output states out_st_spec , $IFop^{-1}(op_spec)(IFstate^{-1}(in_st_impl), out_st_spec) = false$.

Proofs of the implication properties can generally be much more straightforward than the general case presented here suggests. For instance, the functions from specification operations and state to implementation operations and state might not be captured explicitly; they can be implied by using the same names in both the specification and implementation. Also, there is no requirement that the function from input and output states to some Boolean value be enumerated. It can, instead, be represented as a predicate as would normally be done with an operation's postcondition.

Floyd-Hoare verification is one example of a proof technique within this more general framework. It involves constructing a proof of that the implementation operation has the properties of the specification operation by directly mapping specification state to implementation state, and manipulating the specification state based on implementation computation steps. Such a proof can, in many cases, be accomplished much more easily by allowing a mechanical checker to construct the very basic sets of proofs operationally, and allowing a theorem prover to show that sets of declarative properties over functions combine to imply the specification's properties. Our work facilitates the latter approach.

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; although, as mentioned above, the decision problem might be intractable in practice. We are working on a more specific characterization of the practicality of the approach.

Process Automation

The Echo 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 the 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 automatic 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]. When constructing the example discussed in Section 5, we were able to use the PVS system’s deduction procedures to automatically discharge most of the verification conditions in the proof.

Implementation Language: SPARK Ada

SPARK Ada is a high-level language designed for the development of dependable software [1]. The SPARK programming language comprises a kernel that is a subset of the Ada language. The SPARK subset excludes features that 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

SPARK provides a set of tools for verifying that a program exhibits the properties specified by 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].

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 Instantiation

In this section, we discuss the verification process steps in terms of the specific languages and tools we used. 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 arithmetic does not have to be refined directly in the specification. However, SPARK’s rules for arithmetic refinement must be approved by application domain experts. Infinite sets must also be restricted in this step; the restriction must involve human guidance or review.

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 means that the semantics of the resulting specification can be expressed using a finite-state machine. The resulting specification thus can be implemented in Ada.

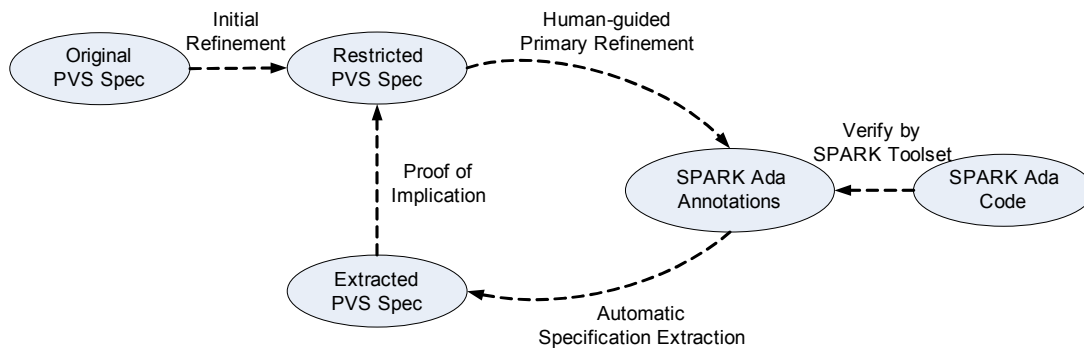


Figure 5. Major Steps in SPARK-PVS Verification

Step 2. Primary Refinement

The goal of the primary refinement is to find a mapping from the PVS specification to a SPARK Ada program. As explained above, since this mapping is done manually, we are not concerned with the semantics of PVS specifications or the underlying reasoning in the refinement. We do care about the rules for the refinement, however, because the primary refinement impacts the difficulty of automatic specification extraction and implication proof. Ideally, the underlying semantic models of the restricted specification and the extracted specification 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 = st~[a => 1];
```

Here a record type definition and a function declaration in PVS are refined into a record type and a procedure in SPARK Ada.

We have designed a prototype set of rules to guide the refinement. These rules can be captured in five categories: (1) software structure; (2) non-function type definitions; (3) non-function variable/constant declarations; (4) function definitions/declarations; and (5) other elements.

Category 1. Software structure

We identify appropriate blocks in PVS such as theories or instantiations of abstract data types, and refine them into SPARK Ada packages. Relations among PVS blocks, such as importing relations among theories, can be refined into package inheritances and compilation orders in SPARK Ada. Scoping and visibility can also be maintained. Structuring the implementation in this way enables the extracted specification to reflect the initial PVS structure.

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 or parameterized types, are manually refined into appropriate forms according to their application semantics. For instance, sets can be refined into array types or enumeration types.

Category 3. Non-function variable and constant declarations

Since type definitions are refined into types in SPARK Ada, variable and constant declarations can then be refined directly into declarations in SPARK Ada. We put them into appropriate SPARK Ada packages, based on their location in the PVS specification.

Category 4. Function declarations and definitions

PVS function declarations and definitions are the most difficult PVS elements 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. Developers must decide which refinement is needed.

For PVS functions that are refined into SPARK Ada functions or procedures, any input/output either maps directly to an input/output in the corresponding SPARK Ada function/procedure, or appears as a global variable accessed by that function/procedure. Type restrictions on the input variables should be refined into precondition annotations. Type restrictions on the output variables (mostly the function bodies) should be refined into postcondition annotations in SPARK Ada.

PVS predicates are special PVS functions in which the range is of type Boolean, and so PVS predicates are refined into Boolean functions in SPARK Ada. IF and CASES statements are also refined into Boolean expressions in SPARK Ada. 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);
```

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 always 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 as postcondition annotations for foo's representation in SPARK Ada.

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. These situations can arise because of the difficulty of using a particular language to express some concept; for example, if a language does not permit declaration of enumerated types, a set of possible values for a type might be listed and stated to be disjoint. In these cases, semantics-preserving transformations can be used. The details of the

transformations must be recorded so that they can be reversed during the specification extraction phase.

Transformations must be employed with great care, because the primary refinement will 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 from SPARK annotations to a PVS specification. The extraction is intended to be the reverse of the primary refinement from PVS to SPARK Ada. Since PVS is more expressive, the extracted specification can easily maintain the structure and modularity of the implementation.

The specification extraction is currently done manually 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` relations extracted from the inheritances and compilation order among SPARK Ada packages. 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 precondition annotations, and PVS function bodies are extracted from postcondition 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 specification extraction, consider:

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

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

We extract:

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

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

Here a record type definition and a function declaration in PVS are extracted from SPARK Ada definitions and annotations. Comparing it with the previous example of primary refinement, it can be seen that the specification extraction step faithfully reverses the primary refinement step.

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 constructed by writing an implication theorem in PVS and mechanically or interactively proving the theorem in the PVS system. Currently, we consider only deterministic specifications. If a specification is deterministic, then implication and equivalence are the same. We currently use equivalence rather than implication, but plan to study the use of implication in nondeterministic specifications in the future. We also plan to investigate the degree of complexity that potential structural mismatch can add to the implication proof process.

5. EXAMPLE

As an illustration of our approach, we present an example based on a simple autopilot constructed using the reconfiguration architecture of Strunk *et al.* [8]. In this example, we have created and verified the implementation of representative functionality and the reconfiguration interface for the autopilot. The reconfiguration interface is designed to allow the autopilot to be reconfigured to two different modes 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. Equivalence theorems were written, and proofs of the theorems were constructed and checked using the PVS system.

In the example, 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 application, 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.

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 was mapped directly to a function/procedure, not a record type. `Data_state` was mapped to a record type, not a function. These decisions were recorded for later reversal. Using these decisions, 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 from system.state;
--# post
--# ((system.state~.alt_hold =
--#   system.engaged and
--#   system.state~.hdg_hold =
--#     system.engaged) ->
--# (system.state =
--#   system.state~[alt_hold => system.off;
--#                 hdg_hold => system.off])
--# )
--# and
...
--# );

```

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 IF-THEN-ELSE expressions in PVS were refined into conjunctive and implicative forms in SPARK annotations.

The Ada code is not shown here since Echo uses the SPARK toolset to show the code’s compliance with its 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]

```

The extracted function is similar in structure to the original `halt` function. The initial extraction of `data_state`, however, results in a record type. We have to apply the reverse of the application-specific decision to convert `data_state` from a function type to a record type before we can produce an equivalence 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 Equivalence Theorem

The last step in our Echo example is the equivalence proof. Since the extracted PVS specification was similar in structure to the restricted specification, constructing an equivalence theorem in PVS was straightforward:

```

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

```

The statement regarding the equivalence of the `halt` function was 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.

We were able to elegantly construct the example's implementation in a way that closely matched the specification design. The example is a small one, however, and we plan to explore the degree to which this hypothesis applies to production software systems. If the hypothesis does not hold in some cases, we will attempt to characterize classes of systems or languages for which the hypothesis does not hold, and explore the reasons that it does not. While it is very desirable to give specifiers complete freedom to construct a specification in the way that best suits their system, it is more important that they be able to verify that an implementation behaves as desired. If the implementation does not behave as specified, then much of the effort invested in creating the specification is wasted. If a specification cannot be verified, the specification has to be changed.

6. CURRENT APPROACH LIMITATIONS

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, the extracted specification cannot have the same structure as the restricted specification. In this case, the application-specific decisions recorded in the primary refinement are extremely important in the implication proof. We have not yet defined a way to ensure that applying the reverse of these decisions will not mask any possible errors in the refinement.
3. The SPARK examiner, simplifier and proof checker are fully automatic. They behave and can only behave according to a set of predefined rules. There might be verification conditions left unproved that are obvious to a human [1]. This means manual 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.

We plan to address these limitations as part of future work.

7. 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 an example indicate that our approach is feasible.

8. ACKNOWLEDGMENTS

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

9. 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.