The Echo Approach to Formal Verification

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

ABSTRACT
In this research abstract, we propose Echo: a general formal verification approach that combines theorem proving, model checking, and code-level tools to show an implementation’s compliance with its formal specification. We believe that this approach is novel since the major proof step 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. We present our prototype design and implementation of the major components of the approach and we instantiate the approach to verify SPARK Ada implementations against PVS specifications. We conducted an initial experiment to determine the feasibility of the approach using a hypothetical avionics system.

Categories and Subject Descriptors
D.2.4 [Software Engineering]:Software/Program Verification – correctness proofs, formal methods, reliability.

General Terms
Reliability, verification.

Keywords
Formal specification, formal verification.

1. MOTIVATION
Safe operation is crucial to safety-critical systems, such as fly-by-wire flight-control systems in aircraft. These systems have to be correct, or, more precisely, the risk of being incorrect has to be reduced to an acceptable level. Testing alone is not sufficient because it is infeasible to conduct the number of test cases required to establish a very high level of confidence in software systems [5]. Thus, formal specification and a formally verified implementation are desirable, as they increase one’s confidence level in the software.

Traditional formal verification creates a direct compliance argument between an abstract formal specification and a concrete implementation. The argument strictly follows the Floyd-Hoare pattern and 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 such a theorem requires a statement-by-statement analysis of the program, which requires significant skill and is hard to automate, especially when the specification has abstracted away significant amounts of detail.

To mitigate these difficulties, we propose Echo: a general verification approach which closely models Floyd-Hoare verification, yet avoids the tedious direct compliance proof between a formal specification and its implementation. The approach is practical and can be largely automated. We have instantiated the preliminary approach with a process that verifies SPARK Ada [3] implementations against formal specifications written in PVS [10].

2. TRADITIONAL VERIFICATION AND ALTERNATIVES

2.1 Traditional Verification
By verification, we do not mean verifying programming-language level properties, e.g. absence of array overflows. Rather, we mean verifying that an implementation is a refinement of its specification or an implementation has certain high-level properties, e.g., temporal properties.

Traditional approaches to formal verification either work through a series of refinements along with proofs from the formal specification to reach a fully verified implementation, or simultaneously refine the specification and abstract the implementation to create a proof.

Both of these approaches involve a direct compliance proof between the abstract specification and the concrete implementation, requiring generation and proof of many detailed lemmas and theorems. These approaches are often hard to automate, and are widely considered to be impractical for large systems.

2.2 Automated Code Generation
Automated translation, or code generation [14], of a formal specification to an implementation provides an alternative to formal 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. If the translation rules are correct, the behavior of the implementation is guaranteed to be consistent with the formal specification. Standard verification, however, can still be required if: (1) no code generator for a specification language exists; (2) an existing code generator isn’t verified; or (3) some manual manipulation of the generated code is needed, which might often be the case in systems that must operate with tight
performance requirements. This is because the generated code is often not well-structured, not maintainable, and not efficient enough. However, any manual manipulation breaks the verification argument of the automated code generation.

3. ECHO VERIFICATION APPROACH

Here we outline the preliminary design of the Echo verification approach that we propose. The process consists of five major steps, shown in Figure 1:

1. An initial refinement of the original formal specification to restrict its semantics to those that can be implemented. Common sources of unimplementable semantics involve arbitrary-precision arithmetic and infinite sets. This step must be either performed or checked by a human, since it modifies the semantics of the specification.

2. A primary refinement from the restricted formal specification to an annotated executable implementation. This step is done either manually or with human guidance.

3. A code-level verification of the implementation against the implementation property annotations using existing automatic tools. This step usually follows the Floyd-Hoare style, but since we adopt existing tools to achieve this, it is not our main focus in the Echo process.

4. An automatic extraction of abstract properties from the implementation. The extraction process expresses the semantics of the implementation property annotations in the specification language, model checker language, or other forms, depending on what kind of property it is. The extracted algebraic properties will form an extracted specification as the input for a theorem prover, and the extracted temporal model will go into a model checker.

5. A proof that the extracted specification complies with the properties of the restricted specification using a theorem prover, and a proof that certain temporal properties hold in the extracted temporal model using a model checker.

The Echo verification argument is based on steps 4 and 5. Provided that step 4 is either automated or mechanically checked, and provided that the proof in step 5 can be constructed, we have a complete argument that the implementation behaves according to the specification and preserves necessary temporal properties. This argument is based on the proof between two abstract specification models, which mitigates the tedious direct proof required between the specification and implementation in traditional verification approaches. Steps 1 and 2 here aim to facilitate the extraction and the proof to the extent possible.

The process is largely mechanical. The only activities that need substantial human intervention are the initial refinement, the primary refinement, and setting up the proof in a theorem prover.

4. DESIGN AND IMPLEMENTATION

In this section, we elaborate the details of the prototype design and implementation of the Echo process and sketch its instantiation using PVS and SPARK Ada.

4.1 Languages and Tools

We have instantiated the Echo approach to verify SPARK Ada implementations against PVS specifications. PVS is a higher-order logic specification language associated with a theorem prover. SPARK Ada is a subset of the Ada language with annotations to specify intended properties of subprograms. It includes a toolset for verifying that a program exhibits the properties specified by its annotations. Other languages could be used provided a mechanical prover is available that can analyze inferences constructed using the specification language, and an automatic toolset is available to ensure that the implementation complies with the declared implementation properties. Detailed language and tool requirements are already discussed elsewhere [13].

We are also investigating the use of B [1] or Z [11] as the specification language since both of them are used widely in the community. We are considering the possibility of mechanical translation from each of them into PVS at or before the initial refinement to take advantage of the power of the PVS theorem prover at the later proof step. Such mechanical translators do exist, i.e. PBS [9]. There are also other choices for implementation languages such as Java with JML [7] annotations, and C# with Spec# [4] annotations.

4.2 Step 1: Initial Refinement

The initial refinement is intended to create an implementable specification. Removal of any unimplementable semantics is necessary for any formal specification refinement approach: there is no possible implementation that will behave in accordance with a formal specification with unimplementable

![Figure 1. The Echo approach to formal verification](image-url)
4.3 Step 2: Primary Refinement
In this step, we refine the restricted specification into an implementation along with appropriate implementation-level property specifications. This step is accomplished manually or automated to the extent possible given application characteristics.

Our PVS-SPARK process involves manual construction of an Ada implementation and SPARK annotations from a PVS specification. However, the way in which this primary refinement is undertaken does affect the ease of the later property extraction and proof. It is preferable for the resulting implementation to keep the structure and modularity of the specification as much as possible. We are investigating the impact of this goal to provide better guidance for the primary refinement.

4.4 Step 3: Code-level Toolset Verification
We use existing automatic code-level toolsets to verify that the implementation complies with the declared implementation properties. 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.

Our PVS-SPARK instantiation adopts the SPARK toolset (examiner, simplifier, and proof checker) to show that the SPARK Ada code is consistent with its annotations. Although the toolset behaves— and can only behave—according to a set of predefined rules so that there might be verification conditions left unproved that are obvious to a human [3], the small proportion of human intervention is currently acceptable to SPARK Ada users.

4.5 Step 4: Property Extraction
This is the key step of the Echo approach. The automatic extractor is the main component that automates the whole process. In our design, the automatic extractor should be responsible for distinguishing the different things it extracts from the implementation and dispatching them to different provers. For instance, algebraic properties go to the theorem prover, temporal models go to the model checker, and so on. The use of a theorem prover is our focus here since integration of model checking into the Echo system is incomplete.

We hypothesize that, in most cases, all design information included in a specification will be retained in an implementation. Furthermore, since the specification language will usually be more expressive than the implementation property language, the structure and modularity of the implementation can often be easily maintained through the property extraction. The later proof will be greatly eased if this is the case, and the above hypothesis is implicitly assumed in traditional verification, as well: traditional approaches require a stepwise proof that a function implementation complies with the function’s specification and implicitly require that there be 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.

We must make sure that the extraction process does not change any semantics of the properties. In order to do this, we have to identify the structural match between the specification language and implementation language that preserves necessary semantics. Again, this must also be done in traditional verification, otherwise we cannot prove that an implementation is a refinement of its specification.

Our PVS-SPARK process extracts an abstract PVS specification from the SPARK declarations, definitions, annotations, and proof rules. We have implemented a prototype of a mechanical extractor that emits PVS structures corresponding to SPARK Ada structures and annotations. The extraction rules are not currently complete, but the completed parts of the process are successful in extracting the PVS specification that can be proved with the original specification in our initial experiment. As an example, consider the following SPARK Ada code:

```ada
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];

The extractor will generate the following PVS:

state: TYPE = [\ a: int, b: int \]  foostate st : state = st WITH ['a = 1]
```

Here a record type and a procedure in SPARK Ada are extracted into a record type and a function in PVS.

We are also adding the capability to the extractor to automatically extract temporal models from the SPARK tasks into Promela as input for the SPIN model checker [6]. The model checker can then be used to verify developed or inferred temporal properties, e.g., freedom from deadlock, that need to be checked in real-time concurrent systems.

4.6 Step 5: Proof
In order to make our verification argument sound and complete, we focus mostly on the proof that the extracted properties (algebraic properties) comply with the original properties in the specification. This is shown by setting up and proving an implication theorem in the theorem prover associated with the specification language. By implication, we mean that the extracted properties imply the original properties in the specification. 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. These alterations must be formalized and must
developed incrementally, thus simplifying the work. For the refinement and the later implication proof will be implementation. The big win here is that the tailoring needed verification, we have developed the specification in PVS and a salesman problem. By applying the idea of incremental theorems in seconds using the PVS theorem prover. We are able to set up and prove all of the implication processes for the autopilot example, the automatic extractor. After running the primary refinement was done manually and the property extraction was done by several iterations, we will arrive at the whole cycle over. After several iterations, we will arrive at the complete system specification and a corresponding, verified implementation. The big win here is that the tailoring needed for the refinement and the later implication proof will be developed incrementally, thus simplifying the work.

4.7 Incremental Verification
Another advantage Echo gives us is that it enables us to build verification arguments incrementally. In other words, we can start from building a basic specification, do the refinement to implementation annotations and code, verify from the annotations to the code, extract the synthetic specification, and develop the implication proof. Then we can extend the original specification a little, to include more functionality, and run the whole cycle over. Another advantage that it has is the way that the proof theorem is set up. We can easily locate the error if the implication theorem fails to be proved, since it must be inside the structure that cannot be proved.

5. EVALUATION AND EXPERIMENT
We will evaluate the Echo verification approach by applying it to real industrial-level applications and compare it with other verification techniques. As an initial experiment, we have applied it to a hypothetical avionics system constructed using the reconfiguration architecture of Strunk et al. [12]. In this experiment, we have created and verified the implementation of representative functionality and the reconfiguration interface for a simple autopilot. The primary refinement step was done manually and the property extraction was done by the automatic extractor. After running the primary refinement and property extraction processes for the autopilot example, we were able to set up and prove all of the implication theorems in seconds using the PVS theorem prover. We are also running through an experiment using the traveling salesman problem. By applying the idea of incremental verification, we have developed the specification in PVS and a brute-force implementation in SPARK. Although the complete verification hasn’t been finished, the incremental approach does give us promising results.

6. EXPECTED CONTRIBUTION
We propose Echo, a general formal verification approach. Echo avoids or mitigates the difficulty of the direct compliance proof of a concrete implementation against an abstract formal specification in traditional Floyd-Hoare verification by moving the major proof step into two abstract specification models. We plan to incorporate the capabilities of theorem proving and model checking to flexibly verify different properties of implementation. We will automate the process to the extent possible using available verification and theorem proving tools. The results of our initial experiment indicate the feasibility of this approach. We would like to identify the class of languages to which it can be applied and to make it applicable to real, industrial-level applications.

7. REFERENCES