Formal Verification By Specification Extraction[1]

Xiang Yin John C. Knight*
Dept. of Computer Science
University of Virginia
{xyin | knight}@cs.virginia.edu

Elisabeth A. Strunk
Software Systems Engrg. Dept.
The Aerospace Corporation
elisabeth.a.strunk@aero.org

Westley Weimer
Dept. of Computer Science
University of Virginia
weimer@cs.virginia.edu

*Contact author - contact information:

Department of Computer Science
University of Virginia
151 Engineer's Way, PO Box 400740
Charlottesville, VA 22904-4740

knight@cs.virginia.edu

+1 434.982.2216 (Voice)
+1 434.982.2214 (FAX)

Submission Category: Regular Paper

Submission track: DCCS

1. Cleared for submission but not for public release

Formal Verification By Specification Extraction

Xiang Yin John C. Knight
Dept. of Computer Science
University of Virginia
{*xyin* | *knight*}*@cs.virginia.edu*

Elisabeth A. Strunk
Software Systems Engrg. Dept.
The Aerospace Corporation
elisabeth.a.strunk@aero.org

Westley Weimer
Dept. of Computer Science
University of Virginia
weimer@cs.virginia.edu

Abstract

In this paper we describe Echo, a novel yet practical approach to the formal verification of implementations. Echo splits verification into two major parts. The first part verifies an implementation against a low-level specification. The second uses a technique called reverse synthesis to extract a high-level specification from the low-level specification. The extracted specification is proved to imply the original system specification. Much of Echo is automated, and it alleviates the verification burden by distributing it over separate tools and techniques. Reverse synthesis is achieved largely by mechanically applying a variety of structural transformations, including efficiency-reducing transformations, which can be viewed as optimizing the program for verification rather than for size or speed. We give a detailed example of Echo, verifying an implementation of the Advanced Encryption Standard (AES) against the official specification of AES.

1. Introduction

We present a novel approach to software verification called *Echo* [11]. Echo is designed to ensure that a software implementation satisfies its formal specification, without the incompleteness of testing or the time and cost of traditional formal verification. For efficiency, it uses a number of powerful existing notations, tools and processes. It also introduces a new approach to analysis in which verification burdens are distributed over separate techniques (with tool support) and in which high-level specifications are extracted from low-level specifications. Many existing software development methods can continue to be used, yet formal verification and all of its benefits can be applied.

In general, verification is the process of showing that one representation of a software artifact is equivalent to another. The most common instance of this is showing that an implementation in a high-level language matches its specification. Such verification provides confidence that mistakes in software development have been avoided or eliminated. However, because of its difficulty, it is often not completed—leaving a notable point of weakness in software assurance arguments.

In many cases, verification is undertaken by testing the developed software artifact against its specification. Testing, however, is not adequate for high levels of assurance [4]. Formal verification becomes an attractive alternative under such circumstances, and in some cases—such as at Evaluation Assurance Level 7 of the Common Criteria [8]—it is required. This paper uses *verification* to mean formal verification.

Proof-based verification is desirable but not commonly used. Many valid reasons have been put forward to explain this. One important problem is the difficulty of current techniques. Verification approaches are time-consuming and require high levels of experience. Linking the source code of a software artifact directly to a high-level specification is currently quite hard.

The Echo verification process involves two proofs: (1) that a low-level specification matches the code; and (2) that a high-level specification, extracted from the low-level specification in a mostly automatic manner, implies the original system specification. This extraction activity is referred to as *reverse synthesis*.

This reverse synthesis approach fills in a major gap in existing techniques. Fundamentally, verification is about reconciling two different views of the same software system and proving that the program implements the specification. Reverse synthesis extends the state of the art by allowing practitioners to approach the problem from both directions: an intermediate point between the specification and the program is selected, it is shown to both describe the program and also adhere to the specification. In the limit, where the intermediate description is taken to be the original low-level specification, the process mimics previous verification approaches. The flexibility of starting from an intermediate point allows Echo to dovetail with standard development processes. Echo makes it easier for developers to verify software systems by automatically extracting useful intermediate representations.

In this paper we describe the Echo approach, present a detailed example of its use: verifying an implementation of the Advanced Encryption Standard (AES) against the AES reference specification, and compare it to other approaches to formal verification.

2. The Echo Framework For Verification

In the development of Echo, we sought to address three goals. Most importantly, we wanted the approach to be completely sound, or to have limited, known points where it is not sound. Otherwise, formal verification would not be provably superior to testing.

The second goal was to allow developers the maximum freedom possible in building a system. We observe that showing *compliance* of an implementation with a specification should not necessitate a specific method for *constructing* the implementation: development decisions should be restricted as little as possible by the goal of verification. This is not the case currently with approaches such as the B method [1].

Our third goal was to use existing technology as much as possible. Many powerful notations and tools are available for mechanical analysis of software, and exploiting these notations and tools offers the opportunity to make progress more quickly. Also, existing notations and tools have addressed very difficult challenges. Thus, they both solve part of the problem and point in a positive technical direction.

The Echo verification technique [11], shown in Figure 1, is an approach to formal verification that meets our three goals. It consists of five major steps:

(1) Initial Refinement: A refinement of the original specification to restrict its semantics to those that can be implemented. We refer to the resulting specification as the *restricted specification*. This step is important because formal specifications often abstract away detail, such as bounds on arithmetic operations, that in practice forms part of a program's semantics. If those semantics are not present in the specification, an implementation that complies with the specification cannot be built. This activity is referred to in the literature as retrenchment and has been studied in some depth [2].

(2) Primary Refinement: A manual (perhaps with machine assistance) refinement from the restricted formal specification to an executable *implementation* with declarative property annotations that we refer to as the *low-level specification*. This is the primary software development step, and we wish to limit it the least so as to allow developers as much freedom as possible. Because this is a development step, it does not affect the soundness of the verification approach.

(3) Implementation Proof: A proof that the implementation implements the low-level specification (the declarative property annotations) correctly. This proof is automated to the extent possible, and our Echo prototype uses the SPARK Ada toolset [3]. The SPARK toolset is sound except for showing loop termination.

(4) Extraction: An automatic extraction via reverse synthesis (with human guidance) of an *abstract specification* from the low-level specification (the implementation's annotations). The extractions are mechanically checked and the guidance does not affect soundness.

(5) Implication Proof: A proof that the properties of the abstract specification imply the properties of the restricted specification. This proof is automated to the extent possible, and our prototype realization of Echo uses the PVS mechanical proof system [9]. PVS checks proofs—whether they are constructed manually or automatically—ensuring the soundness of this step.

The Echo verification argument is based on the third, fourth and fifth steps. Provided that: (1) the extraction is either automated or mechanically checked; (2) that the implication proof can be constructed; and (3) that the implementation can be shown to implement the annotations, we have a complete argument that the implementation behaves according to the specification. The refinement steps are automated to the extent possible and assist the subsequent steps.

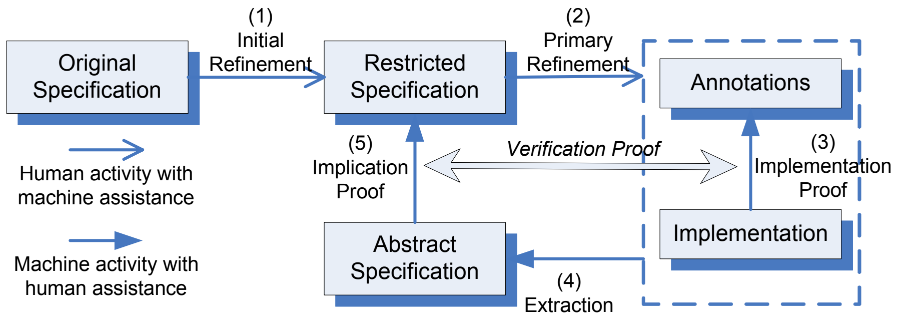

Figure 1. The Echo verification system

A crucial element of the overall Echo approach is the use of an annotated implementation. The annotations are more abstract than traditional programming languages, so they facilitate the refinement activity. Existing notation and proof systems, such as SPARK Ada, can verify the code against the annotations. Annotations of the kind we require have also been adopted by Microsoft in the development of both its Vista operating system and its Office tool suite [5].

The Echo approach addresses all of our goals for practical verification. The first goal is addressed by using sound automated tools. The second goal is addressed by showing compliance of an existing implementation, rather than designing an implementation to show compliance. The reverse synthesis step helps Echo verification to integrate more naturally with software development. The third goal is addressed both by using existing formal specification languages and programming languages, and also by targeting verification at the combination of a static code analyzer and a theorem prover. Thus, Echo need only fill in the gaps left by tools already available.

3. Specification Extraction

The extraction of the abstract specification from the annotated implementation is the central challenge that arises with the Echo concept. We discuss this process, which we call *reverse synthesis*, in depth in this section. The other four steps in Echo are established technology, and so we will not discuss them further.

Reverse synthesis constructs the abstract specification from the annotated implementation by an automatic or semi-automatic but mechanically-checked process. Reverse synthesis is the crux of our verification argument: we transform the annotated implementation, making it amenable to more powerful analysis than can be done with code annotations. Reverse synthesis is far from a simple task, and we know of no way in which the process can be automated completely and satisfactorily in the general case.

Echo proceeds by seeking partial solutions and combining them in a way that allows their application in a synergistic and guided manner. This is similar to a theorem-proving system in which some proof steps are taken automatically and others are applied under human control. Reverse synthesis operates similarly: a developer guides the extraction process, although we have not yet developed all of the desirable tool support.

Presently, reverse synthesis in Echo combines and exploits four basic techniques: (1) architectural and direct mapping; (2) library transformation; (3) model synthesis; and (4) efficiency reduction. For any particular program, combinations of techniques will be used, each contributing to the goal of successful reverse synthesis for that program. We examine each of these in turn in the remainder of this section.

3.1. Architecture and Direct Mapping

We claim that the architectural or high-level design information in a specification will usually be retained in the implementation. While an implementation need not mimic the design for compliance, in practice it will generally be similar in structure because repeating the design effort is a waste of resources. For example, consider a model-based specification, written in a language like *Z*, that specifies the desired operations using pre- and post-conditions on a defined state. The operations reflect what the customer wants, and the implementation architecture would almost certainly retain those operations explicitly.

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

If the implementation retains the architectural information, then the extracted abstract specification will be structurally similar to the restricted specification. An injective function from specification identifiers to implementation identifiers can be retained during the primary refinement, and its inverse (with a domain equal to the original function's image) used when creating the extracted specification. A simple way to automate reverse synthesis is to directly map elements of the annotated implementation language into corresponding elements in the specification language.

3.2. Library Transformation

Similar structures are often present in multiple implementations because of common specification components or algorithm reuse. This commonality suggests that a library of specification fragments could be created by an organization using the Echo process to facilitate the reverse synthesis.

One approach to generating such a library is related to lemma extraction [10]. This technique, typically used to compress machine-generated proofs, involves automatically locating repeated bits of reasoning or proof structure. For example, many software

```
L:for i from 1 to #perms_of_cities loop
      permutation := next_permutation;
      total := 0;
      for j from 2 to #cities_in_perm loop
            total := total + distance(j-1, j);
            if total > minimum then exit L;
      end for;
      minimum := total;
      route   := permutation;
end for;

return(route, minimum);
```

(a)

```
for i from 1 to #perms_of_cities loop
      permutations[i] := next_permutation;
end for;

for i from 1 to #perms_of_cities loop
      total := 0;
      for j from 2 to #cities_in_perm loop
            total := total + distance(j-1, j);
      end for;
      totals[i] := total;
end for;

minimum := totals[1]; loc := 1;
for i from 2 to #perms_of_cities loop
      if minimum > totals[i] then
            loc     := i;
            minimum := totals[i]
end for;

return(permutations[loc], totals[loc])
```

(b)

Figure 2. Efficiency Reduction for Traveling Salesman Problem

systems making use of the same scientific sensor (or similar peripheral) may all have to demonstrate that they adhere to the timing requirements for accessing it.

We have developed some simple example transformations in an embryonic library, and we expect the library to expand as experience with Echo grows.

3.3. Model Synthesis

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

Model synthesis can exploit work on guessing invariants for extended static checking [12]. Such systems typically infer loop invariants and preconditions in automatic verification settings. Echo could guide a human in the creation of a higher-level model by considering a number of model templates and presenting the one that satisfies the greatest fraction of the correspondence proof obligations. It is also possible to obtain partial models and invariants from iterative abstraction refinement and software model checking [6].

3.4. Efficiency Reduction

Software implementations are often influenced by the need for efficiency in time or space. This adds considerably to the program's overall complexity. It is often easy to show that reducing a program's efficiency does not change its computed function. Reducing efficiency can, however, reduce complexity.

Efficiency reduction is based on the idea that semantics-preserving transformations are easier to carry out, understand and prove correct at the level of the *program* than at the level of the *proof system*. A loop and its unrolled form yield proof obligations that are equi-satisfiable, but those proof obligations have different structures and are not equally easy to verify.

Efficiency reduction can focus on computation or storage. Programs are simplified either by adding *redundant* computation or storage, or by adding *intermediate* computation or storage. Examples of adding redundant computation include moving computations out of conditionals, changing a loop that computes several things into a sequence of single-purpose loops, increasing loop bounds to a convenient limit, and replacing iteration with recursion. Retaining values after their initial computation so that they can be used in other (possibly redundant or intermediate) computations is an example of adding redundant storage.

As an example of efficiency reduction, consider solving the Travelling Salesman's Problem. Figure 2 shows the pseudocode for two approaches. A reasonably efficient approach (Figure 2a) is to compute a permutation of the cities, add distances until either the total exceeds a prior minimum or replaces the prior minimum, and then go on to the next permutation until all have been examined. A much less efficient approach (Figure 2b) is to compute all the permutations, retain them, and then compute the distances for all permutations, retaining them also. Finally the distances are scanned and the smallest determined. This

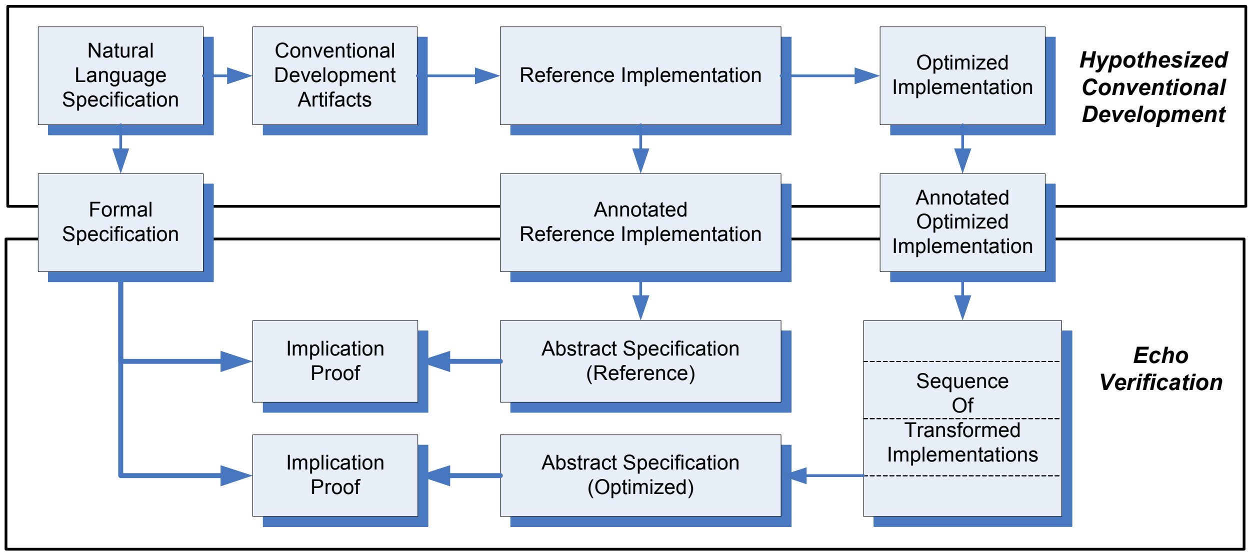
Natural Language Specification
Conventional Development Artifacts
Reference Implementation
Optimized Implementation
Hypothesized Conventional Development
Formal Specification
Annotated Reference Implementation
Annotated Optimized Implementation
Implication Proof
Abstract Specification (Reference)
Sequence Of Transformed Implementations
Echo Verification
Implication Proof
Abstract Specification (Optimized)

Figure 3. Application of Echo to an example application

approach, although vastly less efficient in terms of both space and time, is much simpler to verify.

Before we can rely on efficiency reduction in this case, we need to prove that the less efficient version is semantically equivalent to the original. This proof is carried out by showing that the same permutations are examined in the same order, and that the distance calculations are equivalent. This is proved by induction.

Efficiency reduction in Echo involves selecting a number of transformations, proving that they are semantics-preserving (in an outside system, e.g., [16]), applying them to the original program, and using their output for subsequent verification. The resulting proof obligation is likely *much* simpler to discharge than in most traditional circumstances because the proof involves a transformation from a more-complex to a less-complex program. Efficiency reduction is a key part of Echo's specification extraction step, and we detail an example of its application in Section 5.

3.5. Integration of Techniques

The four techniques described above all contribute to the goal of efficient specification extraction. These techniques can be combined in straightforward ways, but, just as in mechanical theorem proving, many details must be managed, multiple artifacts must be constructed, and the composition of the techniques must be checked for completeness and accuracy. We have developed a prototype toolset for reverse synthesis that handles direct translation from SPARK Ada implementations to PVS specifications completely, along with minor elements of the other techniques.

4. An Example Application

One of our goals for Echo was to allow developers the maximum freedom possible in building a system. We sought a way to assess our success in meeting this goal as well as the utility of the overall Echo technique. The approach we followed was to apply Echo to an important yet publicly-available system written entirely by others. Clearly, the system's development was not constrained by the Echo technique's requirements.

For this assessment, we used an implementation of the Advanced Encryption Standard (AES) [13]. We employed three artifacts: (1) the Federal Information Processing Standard specification of the AES [14]; (2) a reference implementation; and (3) an optimized implementation derived from the reference implementation. The specification is in natural language with mathematical descriptions of some algorithmic elements. Both implementations are written in C.

We assume that these artifacts were created by a traditional software development process, and that the developers took no actions that would make Echo infeasible or very difficult. We supplemented these artifacts as necessary and applied the Echo process. The artifacts that were used or created as part of this assessment together with their relationships are shown in Figure 3. Typical software-development artifacts are in the upper part of the figure and those specific to Echo in the lower part. Those that cross the boundary are discussed below.

We used the following system artifacts:

Natural language and PVS specifications. The original specification [14] specifies the AES algorithm, a symmetric iterated block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. The number of rounds employed is a function of the key lengths. Each encryption round is composed of four critical subroutines, SubBytes, ShiftRows, MixColumns, and AddRoundKey, while each decryption round is composed of the reverse of these steps. It also specifies a key expansion routine used to generate a series of round keys from the cipher key. The basic unit used in the specification is a byte (8 bits).

The specification available was mostly in natural language. We translated it into a formal specification in PVS (the *restricted specification* from Figure 1). We formalized all the behaviors and constraints described in the original specification in PVS and included them in the restricted specification. The essential requirement of a cipher is that it be hard to break. However, here we are only interested in verifying the functional correctness of the algorithm and any implementation derived from it, especially since testing seems infeasible because the number of possible inputs is at least 2^{128}. Therefore we formalized a key correctness property of the algorithm that decryption undoes encryption given the same key:

$$\forall(Block, Key)$$
$$Decrypt(Encrypt(Block, Key), Key) = Block$$

We proved this property using the PVS theorem prover at the specification level.

In practice a formal specification might be produced by developers, making this translation unnecessary. We thus show the formal specification as crossing the boundary between the two groups of artifacts.

Reference implementation. We adopted Vincent Rijmen's ANSI C reference code version 2.2 as the reference implementation because the code is written for clarity instead of efficiency. We assumed that a straightforward design process was followed. We also assumed that the creation of an unoptimized preliminary implementation would be likely in typical software development. We treated the reference implementation as this initial implementation.

As explained above, Echo uses a low-level specification both for reverse synthesis and for static analysis of the program. Our instantiation of Echo uses SPARK Ada for defining the annotations that constitute this specification, so we translated the reference implementation into SPARK Ada and added annotations for pre- and post-conditions of functions (creating an *annotated implementation* from Figure 1). A proof that the code adheres to the annotations was completed using the SPARK toolset with some straightforward human intervention.

Had the implementation been in a language that supported annotations, or had the development process required annotations (a realistic possibility, as noted above), then adding annotations would not have been needed. Because the annotations can be part of Echo, we show these two artifacts crossing the boundary between the typical artifacts and the Echo artifacts in Figure 3.

The reference implementation and the restricted specification adhere to the *architecture mapping* hypothesis from Section 3.1: the general structure and key functions of the implementation mirror those of the specification.

Optimized implementation. A typical development process can include a step in which a preliminary implementation is refined to meet a goal such as performance, and we assumed that this is what happened here. For AES, the optimized ANSI C code version 3.0 was derived by its developers from the original reference code using various optimizations such as loop unrolling and function in-lining. It is more efficient and, thus, more like code that might be used in practice. However, it is also more difficult to verify. After we translated the optimized version into SPARK Ada (creating a second *annotated implementation* from Figure 1), the off-the-shelf SPARK toolset could not even generate verification conditions. Instead it quickly exhausted heap space and stopped, presumably because the generated proof obligations were too large and complex. Our attempt to annotate the code to ameliorate the problem resulted in a post-condition that we felt was too obscure and complex. Developers should not have to use something similar on a routine basis. To verify the functional correctness of this optimized implementation, we applied efficiency reductions and performed various structure transformations (described in the next section). A proof that the code—with applied efficiency reductions—adheres to its annotations was completed using the SPARK toolset with some straightforward human intervention.

Transformed implementations. Reverse synthesis involves the application of a variety of techniques that support verification (see Section 3). The techniques are sequentially applied to an implementation until the implementation is in a form that yields an extracted specification for which an implication proof can be established. These transformations are the key to verifying the optimized implementation. We present the details of the reverse synthesis for the optimized implementation in the next section.

	Transformation (Cumulative)	LOC (Code)	Diff LOC (Code)	LOC (API & tables)	Variables (Local & Formal)	Functions (Defined)
AES1	Original	316	-	733	47	6
AES2	undo loop unrolling	201	301	733	49	6
AES3	undo word packing	144	268	755	26	2
AES4	undo table lookups	156	141	121	29	3
AES5	undo func inlining	191	260	149	28	12

Table 1: AES versions transformed via efficiency reduction

Abstract specifications. A specification was derived from the reference implementation and from the optimized implementation with reduced efficiency (i.e., two *abstract specifications* from Figure 1) using our automatic toolset. Each abstract specification was used to form an implication theorem with the original formal specification. The proofs of implication were then established with the PVS theorem proving system, with some straightforward human intervention.

We verified the reference implementation very easily since the architecture mapping hypothesis was satisfied. The extraction was completed automatically by our prototype toolset. The verification of the optimized implementation is described in the next section.

5. Verifying the Optimized Implementation

Starting from the development artifacts described in Section 4, we used Echo to verify the optimized AES implementation with respect to its original specification. To verify the optimized version of AES, we started with the fully-optimized, unannotated version and applied a series of four efficiency-reducing, semantics-preserving transformations. We then annotated the final version and proved it using PVS.

5.1. The AES Versions

Table 1 lists details of the versions of the AES code used in verification. AES1 is the original optimized code and each subsequent version represents the application of an efficiency-reducing transformation. In all cases we include and verify only functions related to encryption and decryption; we do not describe or verify function bodies related to key expansion, or any NIST APIs. The transformation column lists the efficiency reduction used to obtain a version from the previous version. All transformations are cumulative: for example, AES3 has been changed with respect to loop unrolling and word packing. The details of the transformations are given below. The first two lines-of-code columns measure the lines of SPARK Ada code associated with the function bodies. The Diff column counts the number of changed lines with respect to the previous version (as reported by the `diff` tool): e.g., AES2 is 115 lines shorter than AES1, but there are 301 instances of line insertions, deletions or changes between those two versions. The third lines-of-code column measures function prototypes and lookup tables (in SPARK Ada .ads files). Differences between successive versions represent removing unused lookup tables and adding or removing function prototypes. The Variables column counts the total number of variables (local, formal, and loop induction) in the code. The Functions column counts the total number of functions and procedures defined in the code.

5.2. The First Transformation: Loops

We now describe the efficiency reduction transformations we applied, both in general and with particular regard to their role in this example. The point of efficiency reduction is that it is easier to perform and verify transformations in the code domain than in the proof domain. That is, given complex proof obligations for a program, it is easier to simplify the program than to simplify the logical terms directly.

The first transformation we applied was to undo loop unrolling in AES1. Undoing loop unrolling involved locating the repeated code, pulling it into a for-loop, and changing literal references to use the new loop induction variable. As shown in Table 1, this transformation introduced two new loop induction variables and dramatically shrank the code size. After the transformation, loop invariants could be annotated to help the verification; we did this at a later step.

Loop unrolling is not specific to AES, and this sort of loop condensing efficiency reduction would enjoy wide applicability. With further tool support, an Echo library or a menu of common transformations could feature loop condensing. Both selecting potential spots for loop condensing and also verifying the transformation can be done automatically (e.g., [16]). In this example we selected the two loop sites manually, but the transformation is mechanically checkable. In essence, the checking of this Echo transformation is a conceptual dual of certifying optimizing compilers that might introduce loop unrolling (e.g., [7]).

5.3. The Second Transformation: Packing

The second transformation involved undoing a word-packing representation optimization. The AES standard describes encryption in terms of bytes, but the optimized implementation packs the bytes into 32-bit words to utilize efficient word-level operations. AES1 and AES2 include utility functions to split and combine 32-bit words; the bytes inside a word are referenced by bit shifting. In AES3, we replaced references to 32-bit words by arrays of four bytes. Thus splitting, combining, and references to bytes use native array operations. Specialized procedures for manipulating packed data were removed, but every line of code that referenced packed data had to be updated to use the new representation. After this transformation the code and the specification used the same basic type to refer to data and were thus easier to verify.

Packed data structures and efficient representations are also not specific to AES. While there has been some work toward automatically locating likely spots for such transformations (e.g., [17]), we assume that this step is manually guided, for example by the user providing a type transformer or otherwise indicating the links between the old and new representations. In this example, the packed types were indicated manually. Once the types and transformation spots have been selected, the behavioral equivalence of the representations can be checked mechanically, and their uses can be transformed and verified mechanically.

5.4. The Third Transformation: Tables

The third transformation replaced table lookups with explicit computations. A major optimization in the AES1 implementation was combining different cryptographic transformations into a single set of table lookups. The tables contain pre-computed outputs and thus reduce the run-time computation. The properties of those tables have been documented [15], and can be used as an input from the primary refinement. AES4 replaced references to these table values with inlined instances of the appropriate value computations.

This transformation can be viewed as a general notion of property substitution. The optimized implementation maintains the invariant that `Table[i] = computation(i);` the transformation replaces reads to `Table[i]` with instances of `computation(i)`. This transformation makes the code easier to verify because the specification is phrased in terms of `computation(i)`. Reasonable sites for such a transformation cannot, in general, be selected automatically, but the number of computations so described in the specification is limited, and the conventional software development artifacts may well record why and where such precomputed table optimizations were applied. In this example the sites were chosen manually. Once a human has identified the table and the computation, verifying the transformation can be done mechanically.

5.5. The Fourth Transformation: Inlining

The final transformation we applied was to undo function inlining. After undoing the table lookups, the different subroutines were still joined together, obscuring events that are explicitly required by the specification. Reversing such inlining makes the code easier to verify by aligning the code structure with the specification structure (see Section 3.1). In this example, we identified and factored nine specified functions, each of which was quite small. The verbose function-definition syntax meant that undoing inlining actually increased the source code line count, but the conceptual complexity was reduced.

Inlining functions is certainly not specific to AES. Finding places to undo function inlining is known in the compiler literature as *procedural abstraction* [18] and is used when optimizing for code size. Finding appropriate sites for this transformation can thus be done automatically, or it can be guided based on the specification structure. Once the sites have been identified, the transformation can be verified mechanically.

5.6. Annotation and Verification

The final program version, AES5, was then annotated, producing the annotated optimized implementation. The compliance of the code to the annotation was proved using the SPARK toolset. It completely discharged 136 out of 144 verification conditions, and the remaining ones needed very little human intervention. An abstract specification was automatically extracted, and an implication proof relating that extracted specification to the formal restricted specification was constructed (see Figure 3). All resulting obligations were automatically discharged by the PVS theorem prover in seconds. More than half of the implication theorems could be discharged by a simple `(grind)` command. Others could be discharged by applying a sequence of proof commands and lemmas that demanded little human insight. These proofs, combined with the proofs that the transformations were correct (which we assume), provide a formal assurance guarantee that the original optimized AES implementation adheres to the restricted specification.

We also tried annotating AES1 *without* applying the reduction transformations. We were able to extract an abstract specification from it and set up the implication proof with the restricted specification. However, that approach yielded a significant increase in the complexity of discharging the resulting proof obligations. The mechanically-checkable code-level transformations are preferable to such complex proof work.

5.7. Efficiency Reduction Summary

In summary, we applied a sequence of four semantics-preserving efficiency-reduction transformations to the optimized implementation to enable us to complete the Echo process. For this example we applied all of the transformations manually. Each transformation was mechanically checkable. Some required human guidance, but most could be suggested automatically. The transformations replaced optimizations for space or speed with optimizations for ease of verification. We claim that efficiency reduction transformations are just as easy to understand as traditional optimizations; in general they are the inverses of traditional optimizations. Such transformations can be guided by developers who are not theorem-proving experts, making efficiency reduction and thus Echo a practical part of industrial software development.

We are unaware of any other formal verification processes that could lead to such a result. We compare Echo to other techniques in the next section.

6. Evaluation

Echo is a formal verification process that makes it easier for developers to prove that programs adhere to specifications. There are many related approaches that offer guarantees or assurances about programs. The key strengths of Echo are its treatment and refinement of the specification, the link between the proof and the original program, and the use of efficiency reduction transformations to reduce the proof burden.

Traditional model checking [19] has been quite successful at verifying hardware and protocols, but is difficult to apply to software. Model checkers verify temporal logic properties, which are often not natural for expressing low-level annotations or high-level prose specifications. Worse, software systems must be abstracted to finite-state systems before being explored by model checkers. Traditional checkers do not support common features such as heap structures or call stacks. Finally, while model checking can prove that the model adheres to the specification, it does not prove that the model accurately describes the software system.

Software modeling languages [20] feature a better handling of data structures (e.g., linked lists) than traditional model checking, but often require heavy programmer annotations. They are typically used to verify high-level algorithms and protocols or to visualize important parts of the state space. The user must create or define the model, and no proof is offered that the model is faithful to the original system.

Software model checking [21] uses iterative abstraction refinement to convert programs into checkable finite-state models. It is better at handling program features like function calls or integer variables than traditional model checking. However, it is not automatic for verifying uses of heap data structures, properties that describe more than typestate, or arithmetic that cannot be handled by the underlying decision procedures (e.g., non-linear relations). Software model checking implicitly generates loop invariants and pre- and post-conditions and is only guaranteed to terminate in the presence of an oracle that finds relevant predicates. In practice, software model checking loops forever or generates infeasible counterexamples (i.e., produces false positives) on complicated systems. The abstraction is automatic for particular domains, but it is difficult for the user to annotate or guide the process when it fails. Finally, while software model checking can generate proofs that the software model adheres to the specification [6], it does not prove that the software model is faithful to the original program.

Light-weight program analyses [22] are often used to find bugs in or gain confidence about programs. Compared to more formal verification, their expressive power is limited: typically only typestate properties and syntactic patterns can be checked; arbitrary pre- and post-conditions cannot be validated. Program analyses for non-trivial properties yield either false positives or false negatives. Analyses with false negatives may fail to report violations and are unsuited for verification. False positives usually require that unchecked annotations be placed in the code. In either case, no formal proof of compliance is produced.

The previously-mentioned approaches either do not produce proofs, do not handle critical programs and specification properties, or operate on a simplified model that is not provably related to the original program. Such lightweight techniques are useful for finding bugs and understanding programs, but they are not suited for Echo's goal of full formal verification.

Heavier-weight techniques like the B method [1] are more suited to full formal verification, but they intertwine code production and verification. Using the B method requires a B specification and then enforces a lock-step code production approach on developers.

A more appropriate technique is Floyd-Hoare verification. Axiomatic semantics can be used to link programs to predicates and to describe the weakest preconditions under which postconditions are guaranteed; those preconditions are then the proof obligations. The approach's most important drawback is that it cannot be implemented directly: general weakest preconditions cannot be computed.

In practice, verification condition generation is used instead. The SPARK toolset makes use of this technique when doing its analysis. However, the annotations used by it (and other similar techniques) are generally too close to the abstraction level of the program to encode higher-level specification properties. Thus, we use verification condition generation as an intermediate step.

The remaining difficulty with verification condition generation is that crafting the appropriate proofs can be a large burden for humans. The size of the required proof can easily exceed the size of the program: as a typical example, a 12,644-byte utility program required a 49,804-byte proof just to certify its type safety [23]. Recall that in our example (see Section 5) direct verification condition generation could not feasibly be applied to the optimized implementation without Echo's efficiency reduction step.

One way to address this proof burden is to use an automated proof assistant [9]. Primary refinement is still required to coerce the specification, and verification condition generation is still required to create a logical formula linked to program correctness. Proof may be difficult if the high-level form of the program or of the proof obligation does not directly mirror the specification; the Echo process explicitly handles such matching. In general, however, we claim that many facets of manipulating the program in order to verify it are more easily carried out at the source code level instead of the proof-and-logic level. This is the motivation behind efficiency reduction: it can be easier to change the program and prove that the change is correct than it is to change the program's proof obligation.

Verification condition generation is known to be invariant under many common optimizations and changes (e.g., variable renaming, dead code elimination). One might thus imagine that any transformation or efficiency reduction applied to the program to make it easier to verify might just as easily be applied at the proof level. There are many transformations for which that is not the case. Consider these code fragments:

```
F1: x=0; x+=5; x+=5;
F2: x=0; i=0; while i<2 do x+=5; i++; end
```

Fragment F1 and fragment F2 are equivalent (assuming **i** is unused elsewhere); the transformation between them is an easily-proved loop unrolling. However, they have very different verification conditions. Shown below are their verification conditions with respect to the post-condition "**x=10**":

```
VC1: 0+5+5=10
VC2: 0=5*0 && 0<=2 && (Forall i, Forall
x. x=5*i && i<=2 ==> (i<2 ==> x+5 =
5*(i+1) && i+1<=2) && (i>=2 ==> x=10))
```

Efficiency reduction works because the size of a proof of (and thus the size of the human effort related to) VC2 is larger than the combined proofs of VC1 and the equivalence of F1 and F2. This is especially evident when the equivalence of F1 and F2 can be proved externally by an already-available tool or technique, as was the case for the four transformations used in our example (see Section 5).

The Echo process makes use of verification conditions and proof assistants. Unlike techniques that work purely on simplified models, Echo explicitly deals with the varying levels of abstraction at which the specification and program annotations are presented. Unlike techniques that handle certain special cases (e.g., programs without heap data structures or specifications about type safety only), Echo recognizes that human effort will be required and provides a framework for minimally-guided verification of arbitrary programs and properties. Unlike techniques that constrain development, Echo allows developers a great deal of implementation freedom. Unlike techniques that focus entirely on proof assistants, Echo closely mirrors a natural software development process, allowing code that has been optimized for speed or space to be verified as if it were optimized for obvious correctness. Echo has been demonstrated to work on a non-trivial example and we are aware of no other technique that can make such a full-system verification assurance argument.

7. Conclusion

Echo is a novel, practical approach to formal software verification. Its goals are to give developers freedom and to use existing techniques, analyses and tools while providing a sound solution to the software verification problem. Echo finds a midpoint of abstraction between a formal specification and its concrete implementation. That midpoint, the extracted specification, is proved to describe the source code and to adhere to the high-level specification. The new technique of reverse synthesis creates the extracted specification; it includes applying efficiency-reducing transformations that make the code easier to verify. Efficiency reduction works because it is easier to transform the program than to transform the proof. Echo is largely automatic and all of its steps are machine-checkable. It dovetails directly with traditional development processes and artifacts. We evaluated Echo by verifying an AES implementation against its formal specification.

Acknowledgments

We thank Praxis High Integrity Systems for donating educational licenses for the SPARK Ada tools to us and for making available the complete grammar for SPARK Ada for use in our prototype reverse-synthesis tooset. This work was sponsored, in part, by NASA under grant number NAG1-02103.

References

[1] Abrial, J. R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.

[2] Banach, R. and M. Poppleton, *Retrenchment: An Engineering Variation on Refinement*, Proceedings of B-98, Bert (ed.), LNCS 1393, 129-147, Springer (1998)

[3] Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley (2003)

[4] 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)

[5] Das, M., *Formal Specifications on Industrial Strength Code: From Myth to Reality*, Computer-Aided Verification 2006, Seattle WA (August 2006)

[6] Henzinger, T., R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer: *Temporal-Safety Proofs for Systems Code*, Lecture Notes in Computer Science, Volume 2404, pp. 526 - 538 (Jan 2002)

[7] Leroy, X., *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*, The 33rd Symposium on Principles of Programming Languages, Charleston, SC, pp. 42-54 (2006)

[8] National Institute of Standards Technology, *The Common Criteria Evaluation and Validation Scheme*, (http://niap.nist.gov/cc-scheme/index.html)

[9] PVS Specification and Verification System, (http://pvs.csl.sri.com/)

[10] Rahul, S. and G. Necula, *Proof Optimization Using Lemma Extraction*. Technical Report UCB/CSD-01-1143, University of California, Berkeley (May 2001)

[11] Strunk, E. A., X. Yin, and J. C. Knight, *Echo: A Practical Approach to Formal Verification,* FMICS-05: Tenth International Workshop on Formal Methods for Industrial Critical Systems, Lisbon, Portugal (September 2005)

[12] Flanagan, C. and K. Lieno, *Houdini, an annotation assistant for ESC/Java*, Formal Methods Europe, Berlin, Germany (2001)

[13] AES page available via http://www.nist.gov/CryptoToolkit

[14] FIPS PUB 197, *Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, November 2001

[15] Daemen, J. and V. Rijmen, *AES Proposal: Rijndael*, AES Algorithm Submission, September 3, 1999

[16] Lerner, S. T. Millstein, E. Rice and C. Chambers, *Automated soundness proofs for dataflow analyses and transformations via local rules*, Principles of Programming Languages, pp. 364-377 (2005)

[17] Kataoka, Y., M. Ernst, W. Griswold and D. Notkin, *Automated support for program refactoring using invariants*, International Conference on Software Maintenance, pp. 736-743 (2001)

[18] Runeson, J., S. Nystrom and J. Sjodin, *Optimizing code size through procedural abstraction*, Languages, Compilers and Tools for Embedded Systems, pp. 204-215 (2000)

[19] Vardi, M. and P. Wolper, *An automata-theoretic approach to automatic program verification*, IEEE Symp. on Logic in Computer Science, pp. 322-331 (1986)

[20] Jackson, D, *Alloy: a lightweight object modelling notation*, ACM Trans. Softw. Eng. Methodol., 11(2), pp. 256-290 (2002)

[21] Ball, T. and S. Rajamani, *Automatically validating temporal safety properties of interfaces*, Lecture Notes in Computer Science, volume 2057, pp. 103-122 (2001)

[22] Das, M., S. Lerner and M. Seigle, *ESP: path-sensitive program verification in polynomial time*, Programming Languages, Design and Implementation, pp. 57-68 (2002)

[23] Necula, G. and S. Rahul, *Oracle-based checking of untrusted software*, Principles of Programming Languages pp. 142-154 (2001)