Formal Verification by Reverse Synthesis

Xiang Yin[1], John C. Knight[1], Elisabeth A. Nguyen[2], and Westley Weimer[1]

[1] University of Virginia
Department of Computer Science, Charlottesville, Virginia, U.S.A.
{xyin,knight,weimer}@cs.virginia.edu
[2] The Aerospace Corporation
Software Systems Engineering Department, Chantilly, Virginia, U.S.A.
elisabeth.a.nguyen@aero.org

Abstract. In this paper we describe a novel yet practical approach to the formal verification of implementations. Our approach splits verification into two major parts. The first part verifies an implementation against a low-level specification written using source-code annotations. The second extracts a high-level specification from the implementation with the low-level specification, and proves that it implies the original system specification from which the system was built. Semantics-preserving refactorings are applied to the implementation in both parts to reduce the complexity of the verification. Much of the approach is automated. It reduces the verification burden by distributing it over separate tools and techniques, and it addresses both functional correctness and high-level properties at separate levels. As an illustration, we give a detailed example by verifying an optimized implementation of the Advanced Encryption Standard (AES) against its official specification.

Keywords: Formal verification, formal methods, software dependability.

1 Introduction

In previous work, we introduced a novel approach to software verification called *Echo* [22]. In this paper we present details of a critical component of Echo, *reverse synthesis*, and we show how it is used in the overall verification process. We also present an evaluation in which we applied it to a non-trivial system.

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 [5]. Formal verification is an attractive alternative under such circumstances for systems in which safety and security are critical concerns. It provides confidence with mathematical rigor that many classes of errors in software development have been avoided or eliminated. In some cases—such as at Evaluation Assurance Level 7 of the Common Criteria [19]—it is required. Verification of functional correctness helps to avoid defects introduced in software development that manifest themselves as security vulnerabilities or safety hazards. We note that this complements the notion of proving that a system possesses certain specific safety or security properties.

Our approach is aimed at making formal verification of functional correctness more practical. It uses existing notations, tools and techniques, distributing the verification

M.D. Harrison and M.-A. Sujan (Eds.): SAFECOMP 2008, LNCS 5219, pp. 305– 319, 2008.
© Springer-Verlag Berlin Heidelberg 2008

burden over separate levels. At its core, a high-level specification is extracted from a low-level, detailed specification of a system. We refer to this activity as reverse synthesis. This low-level specification is shown to both describe the program and also adhere to the high-level specification. Thus, formal verification by reverse synthesis involves two proofs each of which is either generated automatically or mechanically checked. These proofs are: (1) a proof that the source code implements the low-level specification correctly; and (2) a proof that a high-level specification which is extracted from the low-level specification implies the original system specification. The two proofs can be tackled with separate specialized techniques.

In order to facilitate both proofs, a variety of *semantics-preserving transformations* are used to refactor the implementation. These refactorings reduce the complexity of verification caused by program refinements and optimizations that occur in practice. They are either effected or checked mechanically, and they are a crucial element of our verification approach because they can be used to simplify both of the proofs, in some cases making proofs feasible that otherwise would not be.

The introduction of a low-level specification as an intermediate point and the application of semantics-preserving refactorings allow our approach to dovetail with standard development processes more easily than existing approaches to formal verification. As a result, relatively few limitations are imposed on developers and many existing software engineering development methods can continue to be used, yet formal verification and all of its benefits can be applied.

In this paper, we begin by summarizing our approach to formal verification by reverse synthesis and then discuss the process and elements involved in detail. Next we present a detailed example of the use of reverse synthesis: verifying an implementation of the Advanced Encryption Standard (AES) against the official AES specification. Finally, we compare our approach to formal verification to other approaches.

2 Formal Verification by Reverse Synthesis

A crucial element of our overall approach is the use of a low-level specification since it is the intermediate representation of the software upon which our proofs are based. The level that we define for this is an *annotated implementation*, i.e., an implementation supplemented with declarative property annotations such as preconditions, postconditions, and invariants. These annotations can be defined and inserted into the source code by the developers or partially generated directly from the code, to describe the desired behaviour of subprograms in the code. Existing annotation-and-proof systems [3, 16] can verify source code against such annotations mechanically, and in our prototype system we use SPARK Ada [3]. Although we have not done so, our approach could be used with languages other than our choice of SPARK Ada, and so this choice is not a fundamental limitation. Annotations and proofs of the kind we require have also been adopted by Microsoft in both Vista and Office [8].

As part of our Echo approach [22], we assume that the original specification from which the software was developed is complete and its semantics have been restricted to those that can be implemented, and we assume a reasonable development practice has been followed to create an executable implementation together with proper annotations. Then our verification approach, shown in Fig. 1, consists of the following steps:

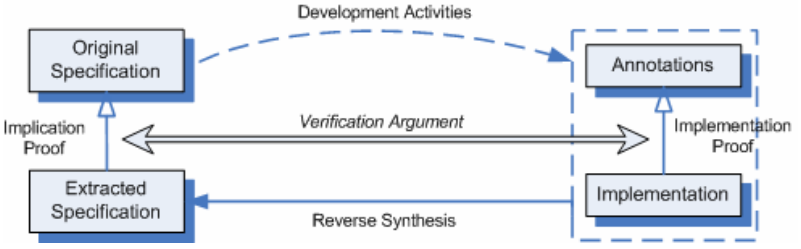

Fig. 1. Formal Verification by Reverse Synthesis

(1) Implementation Proof: A proof that the implementation implements the annotations correctly. Our prototype uses the SPARK Ada system [3] for this proof.

(2) Reverse Synthesis: A mechanical extraction (with human guidance) of a high-level abstract specification from the annotated implementation. Tools we have built perform this extraction, and the abstract specification is written in PVS.

(3) Implication Proof: A proof that the properties of the extracted specification imply the properties of the original specification. Our prototype uses the PVS system for this proof.

The implementation proof, reverse synthesis, and the implication proof are partly automated and partly mechanically checked. Thus, with this process we have a complete formal argument that the implementation behaves according to the specification.

This approach makes verification more practical. It does this in part by combining existing powerful techniques, in part by introducing reverse synthesis, and in part by allowing an engineer to work with an existing implementation rather than requiring that an implementation be designed to show compliance. Showing compliance of an implementation with a specification should not necessitate a specific method for constructing the implementation: development decisions should be minimally restricted by the goal of verification. This is not the case currently with refinement-based approaches such as the B method [1].

By exploiting existing notations and tools, the approach offers the opportunity to make progress more quickly since existing tools both solve part of the problem and point in a positive technical direction. Annotations are tightly coupled with the source code, thus are suitable to prove low-level functional correctness. High-level specification languages and are more expressive and are better at reasoning about high-level properties. Reverse synthesis provides a mechanical link between annotations and high-level specification proofs thereby filling in the gaps left by tools already available.

3 The Reverse Synthesis Process

Reverse synthesis, shown in Fig. 2, is composed of three phases: (1) *implementation* refactoring; (2) *implication* refactoring; and (3) specification extraction. The refactoring phases each transform the program being verified so as to preserve its semantics but to make the associated proof easier. Implementation refactoring assists the user in enhancing and completing the annotations of the source program and thereby

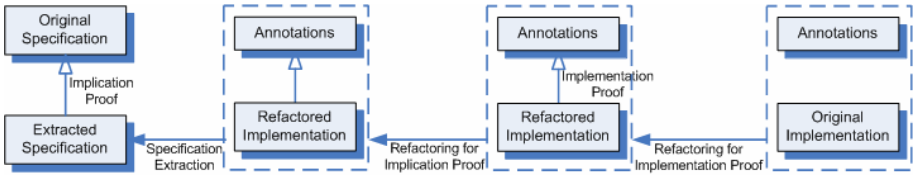

Fig. 2. Detailed Reverse Synthesis Process

facilitates the proof that the source code matches the annotations. Implication refactoring aids the specification extraction phase and reduces the effort in the proof that the extracted specification implies the original one. The specification extraction phase mechanically extracts a high-level abstract specification from the refactored annotated implementation.

We examine these reverse synthesis steps and proofs in turn in the remainder of this section. Implementation proof uses code-level tools such as static code analyzers, proof obligation generators, and proof checkers. This technology is well established, and we do not discuss it further.

3.1 Refactoring for Verification

Software implementations are often influenced by the need for efficiency in time or space. More complex algorithms are used to reduce executions times and data structures are sometimes chosen to reduce computation (and vice versa). Such implementation decisions tend to add considerably to a program's overall complexity. It is often easy to show that refactoring a program and reducing its efficiency does not change its computed function. Reducing efficiency can, however, reduce complexity and thereby facilitate verification. Hence instead of directly extracting a high-level specification from the annotated implementation and performing proofs on them, our approach first tries to refactor the implementation and reduce the complexity of proofs to the extent possible.

Refactoring for verification is the application of semantics-preserving transformations to the annotated implementation. The transformations modify the implementation in some way, and this usually simplifies the implementation, decreasing the implementation's efficiency. This is in sharp contrast to the usual role of semantics-preserving transformations where some form of improvement in efficiency is the goal. The standard approach is exemplified by the use of optimizing transformations in compilation.

We hypothesize 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. That is, given complex proof obligations for a program, it is easier to simplify the program than to simplify the logical terms directly. A loop and its unrolled form yield proof obligations that are equi-satisfiable, but those obligations have different structures and are not equally easy to verify. Refactoring in reverse synthesis, therefore, reduces complexity while leaving the program semantics unaltered, thereby assisting the proofs involved in the verification.

Refactoring for verification involves both computation and storage. Programs can be made more amenable to verification by adding redundant computation or storage,

by adding intermediate computation or storage, or by restructuring the program. 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.

Refactoring is based on the following four stages: (1) *identify candidate refactoring transformations*—since refactoring might address certain optimizations and refinements introduced during development, this usually needs guidance from developers to identify the occurrences of optimizations, although some can be found mechanically; (2) *determine the order to apply the transformations*—the order matters if there are dependencies among the transformations; (3) *prove the transformations are semantics-preserving*—all transformations should be proved to preserve the semantics and should not require the user to discharge complex proof obligations. In order to make the proofs reusable, we identify common refactoring transformations, characterize them into templates, and prove that they are semantics-preserving; and (4) *apply the transformations to the code*—all of the transformations should be applied mechanically to avoid introducing errors. In our prototype toolset, we adopt the Stratego [4] program transformation language and associated XT tools to achieve this.

Presently, refactoring for verification in our reverse synthesis approach has two phases, namely refactoring to facilitate the implementation proof and refactoring to facilitate the implication proof:

(1) Implementation refactoring: These transformations are intended to simplify the proof between the code and the annotations. The transformations are usually applied within subprograms and do not change the existing pre- and post-condition annotations for the subprograms. However, corresponding proof obligations for these annotations are likely to become much simpler to discharge. After the refactoring, the user also has the chance to enhance and complete the annotations for those elements that were otherwise obscured by the optimizations done in the original development process.

(2) Implication refactoring: These transformations are intended to aid the later specification extraction and to simplify the proof between the extracted specification and the original one. The transformations usually involve changes to the structure of the entire program with the goal of aligning the extracted specification and the original specification. This alignment simplifies the implication proof. Each transformation might involve several subprograms and the annotations usually need to be modified, although the modification can in many cases be done mechanically.

The two refactoring phases can be overlapped since some transformations may help both proofs. Neither one of them is strictly required. However, if they are applied, the resulting proof obligations are likely to be much simpler to discharge than in most traditional verification circumstances because the proof involves a transformation from a more-complex to a less-complex program. Refactoring for verification plays an important role in the whole process, and we detail an example of its application in Section 5.

3.2 Specification Extraction

The specification extraction step extracts an abstract specification from the refactored annotated implementation to be used in the proof of implication with the original specification. Presently, specification extraction exploits three basic techniques: (1) architectural and direct mapping; (2) component reuse; and (3) model synthesis, which are discussed in detail below. For any particular program, combinations of techniques will be used, each contributing to the goal of successful specification extraction for that program. We have developed a prototype toolset for specification extraction that handles architectural and direct mapping from SPARK Ada implementations to PVS specifications completely, along with minor elements of the other two techniques.

Specification extraction is automated or mechanically checked, which ensures the extracted high-level specification is a correct representation of the annotated implementation. However, to make the verification sound, we must also make sure that the extracted specification is complete, suitably abstract but not too abstract, so that we can construct and complete the implication proof. Since we extract the high-level specification mostly from the low-level annotations, it means we have to make sure the annotations in the source code describe the entire semantics. Presently we have no completely automated way to check this property, and we rely on human review and cross-check with the derivation relations between input/output variables to do this.

Architectural and Direct Mapping. We hypothesize that it is often the case that the architectural or high-level design information in a specification is retained in the implementation. While an implementation need not mimic the specification architecture, in practice it will often be similar in structure because repeating the architectural design effort is a waste of resources.

As an 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.

In a case where the implementation retains the architectural information from the original restricted specification, a simple way to begin the process of specification extraction is to directly translate elements of the annotated implementation language, such as packages, data types, state/operation representations, preconditions, postconditions, and invariants, into corresponding elements in the specification language. The extracted specification will be structurally similar to the restricted specification. Such a strategy is straightforward, but it does have considerable potential in our approach.

Component Reuse. Software reuse of both specification and code components is a common and growing practice. If a source-code component from a library is reused in a system to be verified and that component has a suitable formal specification, then that specification can be included easily in the extracted specification [24].

Model Synthesis. In some cases, specification extraction 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, we use a process called 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.

At present, our implementation of model synthesis relies on human insight. In future work, we plan to mechanize model synthesis by exploring ideas such as hypothesizing invariants in extended static checking [11] and obtaining partial models and invariants from iterative abstraction refinement and software model checking.

3.3 Implication Proof

The extracted specification needs to be matched to the original specification to complete the verification argument. The property that needs to be shown here is implication, not equivalence; by showing that the extracted specification implies the original specification, but not the converse, we allow the original specification to be nondeterministic, and allow more behaviours in the original specification than the implementation.

The implication argument is shown by matching the structures and components of these two specifications and setting up and proving an implication theorem using the prover associated with the specification language. The formal definition of implication we use for this is that set out by Liskov and Wing known as behavioral subtyping [18]. Behavioral subtyping was studied in the context of languages that permit inheritance, in order to define what it meant for a subtype to comply with the type constraints of a supertype. Intuitively, the requirement is similar in verification: we want to ensure that the function implementation complies with the constraints defined in its specification. While our instantiation is more general, not making assumptions on what is or is not required of a type system, the principles are the same.

Then, by implication, we mean that the types and functions in the extracted specification are subtypes of the matching types and functions in the original specification. More specifically, the extracted function specification (which represents the implementation) should have a weaker precondition and a stronger postcondition than the original function specification:

$$Pre_{original} \Rightarrow Pre_{extracted} \quad \wedge \quad Post_{extracted} \Rightarrow Post_{original}$$

To set up the theorem, we need human guidance to match elements such as variables and functions between the two specifications, but in many cases they can be suggested automatically. The resulting proof obligations need to be discharged automatically or interactively in a mechanical proof system. When the extracted specification shows structure similarity to the original one, the proof usually does not require considerable human efforts as will be illustrated in Section 5. Also, by setting up the implication proof theorem function by function, not property by property, we can easily locate the error if the implication theorem fails to be proved, since it must be inside the structure or component that cannot be proved.

4 An Example Application

In this section we present an example of applying formal verification by reverse synthesis to a small but important application. This example illustrates the various aspects of the approach and provides some preliminary evaluation. A comprehensive evaluation and development of industrial-strength support tools is relegated to future work.

Recall that one of our goals 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 technique. The approach we followed was to apply the technique to an important yet publicly-available system written entirely by others. Clearly, the system's development was not constrained by our verification requirements.

For this assessment, we used an implementation of the Advanced Encryption Standard (AES) [10]. We employed the following two artifacts: (1) the Federal Information Processing Standard (FIPS) specification of the AES [10] that specifies the AES algorithm, a symmetric iterated block cipher, mostly in natural language, with mathematical descriptions of some algorithmic elements; (2) a publicly available implementation written in ANSI C that contains various optimizations such as loop unrolling and function inlining. We assume that these artifacts were created by a traditional software development process, and that the developers took no actions that would make formal verification infeasible or very difficult.

We supplemented these artifacts as necessary to apply the reverse synthesis process. We translated the official FIPS specification into a formal specification in PVS. We formalized all the behaviors and constraints described in the FIPS specification in PVS and included them in the formal specification (as the original specification from Fig. 1). In practice, a formal specification might be produced by developers, making this type of translation unnecessary. We translated the ANSI C implementation into SPARK Ada and added annotations for pre- and post-conditions of functions (the annotated implementation from Fig. 1). Again, in practice an annotated implementation might be produced by developers, making this type of translation also unnecessary.

With these artifacts developed, we applied our reverse synthesis approach to formally verify the functional correctness of the SPARK Ada implementation with respect to the PVS specification. The details of the verification are described in the next section.

5 Verification of the AES Implementation

To verify the AES implementation, we applied refactoring and performed a series of complexity-reducing, semantics-preserving transformations using Stratego/XT tools. A proof that the code—with applied refactoring—adheres to its annotations was completed using the SPARK toolset with some straightforward human intervention. A PVS specification was derived from the refactored annotated implementation using our automatic specification extraction tool. The implication proof between the extracted specification and the original one was then established using the PVS theorem prover with some straightforward human intervention. Fig. 3 shows the detailed tool configuration we set up and the process we followed to conduct this case study. In all cases we included and verified only functions related to encryption and decryption; we

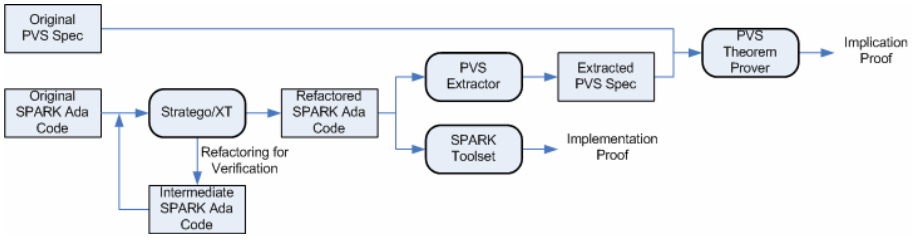

Fig. 3. Tool Configuration for AES Verification

did not describe or verify functions related to key expansion, or any of the NIST APIs. The relevant PVS specification contains 335 lines of functional specification, excluding lemmas and theorems that are required to prove its correctness. The relevant SPARK Ada code we are verifying has 733 lines of function declarations (including lookup tables), and 584 lines of function definitions excluding comments and annotations.

5.1 The Refactoring Process

According to the original AES documentation [7], the following four major optimizations had been applied to create the implementation: (1) loop unrolling; (2) word packing; (3) table lookup; and (4) function inlining. Table lookup and function inlining were dependent since the table entries encoded part of the defined functions. For each of the optimizations we identified, we developed a template defining the refactoring transformation so that they could be reused in other programs. We then characterized them and proved them to be semantics-preserving using PVS. Finally, we applied the transformations mechanically using Stratego. Besides the four major transformations, we also effected several minor transformations including adjusting intermediate variables, removing redundant statements, and aggregating data assignments. These transformations helped match the code to the transformation templates and clean up the code after the transformations. Each was proved to be semantics-preserving.

Table 1 lists details of the versions of the AES code used in verification. AES1 is the original, optimized code and each subsequent version is the result of applying a refactoring transformation. The rightmost two columns in Table 1 present the sizes of SPARK Ada code associated with function definitions and declarations (including lookup tables) respectively. We used bytes instead of lines of code to more precisely denote the size of the code since our tool does not generate proper line breaks for intermediate refactored code.

Table 1. AES versions transformed via refactoring for verification

	Transformation	Definitions (bytes)	Declarations & Tables (bytes)
AES1	Original	25,415	41,924
AES2	undo loop unrolling	8,561	41,924
AES3	undo word packing	7,180	103,389
AES4	undo table lookups	8,036	7,545
AES5	undo func inlining	8,620	8,128

5.2 The Refactoring Transformations

Reversing Loop Unrolling. The first transformation we applied was to undo loop unrolling in AES1. Undoing loop unrolling involved locating the repeated code, redefining it as a for-loop, and changing literal references to use the new loop induction variable. This transformation introduced two new loop induction variables and dramatically shrank the code size as shown in Table 1 since vast amount of repeated code were removed. After the transformation, loop invariants could be annotated to facilitate the verification. This transformation assisted the implementation proof, because by introducing new loop invariants and removing replicated loop bodies, it substantially reduced the states involved in the proof.

Loop unrolling is a well-known compiler transformation, and it might seem unusual for it to have been applied explicitly at the source code level in AES. However, it is not specific to AES, because not all compilers unroll loops and because manual unrolling is still a widespread practice (e.g. to expose concurrency). With further tool support, both identifying unrolled loops and verifying the reversing transformation can be done automatically (e.g., [17]). Here we manually identified two unrolled loops, but selecting the transformation spots, performing the transformation, and proving the preservation of the semantics were all machine checked using Stratego and PVS.

Reversing Word Packing. The second transformation involved undoing a word-packing representation optimization. The AES standard describes encryption in terms of bytes, but the original 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 used 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. As a result, the function definitions shrank slightly while the lookup tables expanded considerably. This is because the tables were originally composed with 32-bit words but were composed of four-byte arrays after undoing word packing. This transformation assisted the implication proof since the code and the specification used the same basic type to refer to data after it and were thus easier to verify.

Data structure transformations 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., [15]), we assume that this step is manually guided. We let the user indicate the links between the old and new representations or provide a type transformer. Once the types and the operations on the types have been selected, the behavioral equivalences of the representations are checked mechanically using PVS. Then transformation spots are selected and the code is transformed mechanically by Stratego.

Reversing Table Lookup. The third transformation replaced table lookups with explicit computations. A major optimization in the AES 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 [7], and AES4 replaced references to

these tables with inlined instances of the appropriate computations using Stratego. As a result, all tables were removed causing a dramatic code-size reduction as shown in Table 1. This transformation supported the next one (reversing function inlining), because some inlined functions were encoded in the tables. It also made the implication proof easier since the specification was phrased in terms of the computations, not the tables.

This transformation can be viewed as a general form of property substitution. The original implementation maintains the invariant `Table[i] = computation(i);` the transformation replaces reads of `Table[i]` with instances 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 pre-computed tables were applied. In general, once a human has identified a table and the computation, the transformation can be checked mechanically by going through all the table entries and comparing them with corresponding computations. Selecting sites and performing the transformation can be done automatically and was in our example.

Reversing Function Inlining. The final transformation we applied was to undo function inlining. After the above transformations, inlined functions continued to obscure events that are explicitly required by the specification. Reversing such inlining aided both the implementation proof and the implication proof. By finding cloned code fragments, it removed replicated or similar proof obligations in the implementation proof. By reversing the inlining, it aligned the code structure with the specification structure so that the implication proof was easier to be constructed. In this example, we identified and factored nine specified functions, each of which was quite small. After undoing inlining, the verbose function-definition syntax actually increased the source code size shown in Table 1, 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 [20] 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. We prove it is semantics-preserving and perform the transforming mechanically using PVS and Stratego respectively.

5.3 Specification Extraction and Proofs

The final program version, AES5, contained 262 lines of function declarations and table, and 214 lines of function definitions, including 126 lines of annotations. Proof functions and rules are also provided in additional files to facilitate the proof of the annotations. Most of the annotations were simple postconditions that could be straightforwardly derived, while others were loop invariants. The compliance of the code to the annotations was proved using the SPARK toolset. It automatically discharged 93% of the verification conditions, and the remaining ones needed very little human guidance to be discharged.

Using our prototype tool, a PVS specification was then automatically extracted using architectural and direct mapping. The result contained 606 lines of PVS and showed great similarity in structure to the original specification. Thus an implication

proof relating that extracted specification to the original specification was easily constructed, and all resulting obligations were discharged in seconds using the PVS theorem prover. More than half of the implication proof obligations 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, provide a formal assurance guarantee that the AES implementation adheres to the specification.

To get an idea of how refactoring helped verification, we tried to verify the original implementation as it was before refactoring. However, 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. We then tried annotating and verifying AES1, the version with loops rerolled. The SPARK toolset generated more than 15M bytes of verification conditions which is around 30 times larger than the refactored version. It took approximately 2 hours on a dual 1.0 GHz UltraSparc IIIi with 2GB RAM for the tools to analyze the verification conditions, while on the same machine it only took minutes for the refactored version. Moreover, unlike the refactored version, the verification conditions that could not be automatically discharged here were mostly major postconditions, whose proof simulated traditional formal verification, and required significant human insight and efforts.

6 Related Work

Light-weight program analyses [9] are often used to find bugs in or gain confidence about programs. Compared to more complete formal verification, their expressive power is limited and no formal proof of compliance is produced. 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 general technique is traditional Floyd-Hoare verification [12]. Unfortunately, it requires generation and proof of many detailed lemmas and theorems. It is very hard to automate and requires significant time and skill to complete. Annotations and verification condition generation, such as that employed by the SPARK Ada toolset, is used in practice. However, the annotations used by SPARK Ada (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 in our approach.

Automated code generation from a formal specification to an implementation, using tools such as the SCADE Suite [21], provides an alternative to verification. This approach constructs an implementation automatically from the specification using formal translation rules. If the translation rules are correct, it offers the possibility of assuring that the behaviour of the implementation is consistent with the formal specification. However, for most safety-critical systems, it is very difficult to automatically generate a well-structured or efficient implementation from a formal specification. If the developer changes the generated code to refine its structure or increase efficiency, the verification argument is invalidated.

Other techniques are available for the properties that we do not address. Model checking techniques [14], for example, have been quite successful at verifying hardware, protocols and temporal properties; they complement our approach in such areas. While model checking can generate proofs that the software model adheres to the specification, it does not prove that the software model is faithful to the original program. More recent model extraction [14], aims to address this problem and mechanically extracts a system model from the source code so that model checking can be applied. However, model extraction does not produce a full assurance argument since model checking is not targeted at full functional correctness.

Related work in the reverse engineering domain retrieves high-level specifications from the source code by semantics-preserving transformations and abstractions [6, 23]. These approaches are similar to reverse synthesis, but the goal is to make poorly-engineered code amenable to further analyses and not to aid verification. Our approach, which incorporates intermediate annotations, can more easily capture the properties relevant to verification while still abstracting implementation details. These techniques, however, show the feasibility of approaches similar to reverse synthesis.

Andronick et al. developed an approach to verification of a smart card embedded operating system [2]. Similar to reverse synthesis, they proved a C source program against supplementary annotations and generated a high-level formal model of the annotated C program that was used to verify certain global security properties. Our approach incorporates refactoring and allows us to show broad compliance with the original specification from which the system was built.

Heitmeyer et al. developed a similar approach to ours for verifying a system's high-level security properties [13]. Their approach is focused on verifying security properties, whereas ours is aimed at general functionality.

7 Conclusion

We have defined a verification technique based upon the use of an intermediate point of abstraction between a high-level formal specification and its concrete implementation. This intermediate point is a low-level specification documented by annotated source code. Our verification approach shows that the source code correctly implements the annotations and that the annotated source code implies the high-level specification.

We have introduced the new technique of reverse synthesis that mechanically creates a high-level specification from the low-level specification. A crucial component of reverse synthesis is the application of complexity-reducing but semantics-preserving refactoring transformations. In general, it is easier to transform the program than to transform the proof. Thus, transformations facilitate verification by reducing the complexity of the source program and thereby the proof obligation.

Human insight guides much of the process, but the analysis and thus the verification is either automatic or machine-checkable. It dovetails directly with traditional development processes and artifacts. We evaluated our approach by verifying an AES implementation against its formal specification.

Although our approach provides certain benefits over existing techniques, it is in no way a verification "silver bullet". As with any formal verification technique, it requires the use of formal languages, various analytic tools including a theorem-proving

system, and considerable skill on the part of the developer. One specific additional responsibility placed on the developer is to annotate the source code with pre- and post-condition documentation. Although the various elements we have incorporated are not often part of current practice, our approach can be conducted in a production setting with comparable resources to those used now but with substantially higher assurance.

Acknowledgments. We thank Praxis High Integrity Systems for their technical support. 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, Cambridge (1996)
2. Andronick, J., Chetali, B., Paulin-Mohring, C.: Formal Verification of Security Properties of Smart Card Embedded Source Code. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 302–317. Springer, Heidelberg (2005)
3. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley, Reading (2003)
4. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.16. A Language and Toolset for Program Transformation. Science of Computer Programming (2007)
5. Butler, R., Finnelli, G.: The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. IEEE Trans. on Software Engineering 19(1) (1993)
6. Chung, B., Gannod, G.C.: Abstraction of Formal Specifications from Program Code. In: IEEE 3rd Int. Conference on Tools for Artificial Intelligience, pp. 125–128 (1991)
7. Daemen, J., Rijmen, V.: AES Proposal: Rijndael. AES Algorithm Submission (1999)
8. Das, M.: Formal Specifications on Industrial Strength Code: From Myth to Reality. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144. Springer, Heidelberg (2006)
9. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. Programming Languages, Design and Implementation, pp. 57-68 (2002)
10. FIPS PUB 197, Advanced Encryption Standard. National Inst. of Standards & Tech. (2001)
11. Flanagan, C., Lieno, K.: Houdini, an annotation assistant for ESC/Java. Formal Methods Europe, Berlin, Germany (2001)
12. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19 (American Mathematical Society), Providence, pp. 19–32 (1967)
13. Heitmeyer, C.L., Archer, M.M., Leonard, E.I., McLean, J.D.: Applying Formal Methods to a Certifiably Secure Software System. IEEE Trans. on Soft. Eng. 34(1) (2008)
14. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
15. Kataoka, Y., Ernst, M., Griswold, W., Notkin, D.: Automated support for program refactoring using invariants. In: Int. Conference on Software Maintenance, pp. 736–743 (2001)
16. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
17. Lerner, S.T., Millstein, E.R., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. Princ. of Prog. Lang., 364–377 (2005)

18. Liskov, B., Wing, J.: A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems 16(6), 1811–1841 (1994)
19. National Institute of Standards and Technology, The Common Criteria Evaluation and Validation Scheme, http://niap.nist.gov/cc-scheme/index.html
20. Runeson, J., Nystrom, S., Sjodin, J.: Optimizing code size through procedural abstraction. Languages, Compilers and Tools for Embedded Systems, pp. 204–215 (2000)
21. SCADE Suite, Esterel Technologies, http://www.esterel-technologies.com/
22. Strunk, E.A., Yin, X., Knight, J.C.: Echo: A Practical Approach to Formal Verification. In: FMICS 2005, Lisbon, Portugal (2005)
23. Ward, M.: Reverse Engineering through Formal Transformation. The Computer Journal 37(9), 795–813 (1994)
24. Weide, B.W.: Component-Based Systems. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering. John Wiley and Sons, Chichester (2001)