Formal Verification of Large Software Systems

Xiang Yin
University of Virginia
Charlottesville, VA, USA
xyin@cs.virginia.edu

John Knight
University of Virginia
Charlottesville, VA, USA
knight@cs.virginia.edu

Abstract

We introduce a scalable proof structure to facilitate formal verification of large software systems. In our approach, we mechanically synthesize an abstract specification from the software implementation, match its static operational structure to that of the original specification, and organize the proof as the conjunction of a series of lemmas about the specification structure. By setting up a different lemma for each distinct element and proving each lemma independently, we obtain the important benefit that the proof scales easily for large systems. We present details of the approach and an illustration of its application on a challenge problem from the security domain.

1 Introduction

Formal verification of software continues to be a desirable goal, although such verification remains the exception rather than the rule. By formal verification, we mean verification of the functionality of the software in the sense of Floyd and Hoare [7]. Many challenges arise in formal verification [10], and these challenges are compounded by the complexity that arises with the increasing size of the software of interest.

In this paper, we present an approach to the formal verification of large software systems. The approach operates within a previously developed formal verification framework named Echo [13, 14, 15]. Our goal with the approach to proof was to develop a technique that would be: (a) *relevant*, i.e., applicable to programs that could benefit from formal verification; (b) *scalable*; i.e., be applicable to programs larger than those currently considered suitable for formal verification; (c) *accessible*, i.e., could be used by engineers who are competent but not necessarily expert in using formal methods; and (d) *efficient*, i.e, could be completed in an amount of time that would be acceptable in modern development environments.

We do not define "large" in a quantitative way. Rather, we claim that the approach we have developed can be applied successfully to software systems that are larger than those successfully verified in previous work, and we illustrate the approach in a case study of a system that is several thousand source lines long.

2 The Echo Approach

Details of Echo are available elsewhere [14, 15], and we present here only a brief summary. Echo verification is based on a process called *reverse synthesis* in which a high-level, abstract specification (referred to as the *extracted* specification) is synthesized mechanically from a combination of the software source code and a low-level, detailed specification of the software. Reverse synthesis includes verification refactoring in which semantics-preserving transformations are applied to the software to facilitate verification.

Formal verification in Echo involves two proofs: (1) the *implementation* proof, a proof that the source code implements the low-level specification correctly; and (2) the *implication* proof, a proof that the extracted specification implies the original system specification from which the software was built.

The basic Echo approach imposes no restrictions on how software is built except that development has

to start with a formal system specification, and developers have to create the low-level specification documenting the source code. Our current instantiation of Echo uses: (1) PVS [12] to document the system specification and the extracted specification; (2) the SPARK subset of Ada [3] for the source program; and (3) the SPARK Ada annotation language to document the low-level specification. The implementation proof is discharged using the SPARK Ada tools. The implication proof is the focus of this paper, and that proof is facilitated greatly by the implementation proof; most loop invariants, for example, are not present.

3 Structural Matching Hypothesis

We refer to our approach to formal verification of large software systems as *proof by parts*. The heart of proof by parts is the *structural matching hypothesis*. We hypothesize that many systems of interest have the property that the high-level structure of a specification is retained, at least partially, in the implementation. Ideally, a specification should be as free as possible of implementation detail. However, the more precise a specification becomes, the more design information it tends to include, especially structural design information. While an implementation need not mimic the specification structure, in practice an implementation will often be similar in structure to the specification from which it was built because: (a) repeating the structural design effort is a waste of resources; and (b) the implementation is more maintainable if it reflects the structure of the specification.

The hypothesis tends to hold for model-based specifications that specify desired system operations using pre- and post-conditions on a defined state. The operations reflect what the customer wants and the implementation structure would mostly retain those operations explicitly.

4 Proof Support

Proof Structure. The proof of interest is the implication proof. Given implementation I and specification S, this proof has to establish the implication $I \Rightarrow S$. Specifically, we prove the implication weakens the pre-condition and decreases non-determinism from the specification:

$$(pre(S) \Rightarrow pre(I)\ ) \wedge\ (post(I) \Rightarrow post(S))$$

In order to establish this proof for large software systems, the proof structure involved must be scalable. In proof by parts, we rely upon the structural matching hypothesis, and we match the static operational structure of the extracted specification created by reverse synthesis to the original specification. We then organize the proof as a series of lemmas about the specification structure, i.e., proof by parts.

Each lemma is set up for declarative properties over a single distinctive element, e.g. type or operation, and is proved independently. The conjunction of all the lemmas then forms the whole implication theorem that the extracted specification implies the original specification. Since each lemma is over a different element of the system and is proved independently without reference to the whole system, proof by parts is expected to scale for large software systems.

Clearly, the construction of a proof in this way is only possible for a system for which the structural matching hypothesis holds for the *entire* implementation. Inevitably, this will rarely if ever be the case for real software systems. For systems in which the two structures do not match, we employ a technique within Echo referred to as *verification refactoring*, to restructure the implementation to match the specification structure.

Verification Refactoring. There are many reasons why the structural matching hypothesis will only apply partially to a particular system, e.g., optimizations introduced by programmers that complicate the program structure. Verification refactoring consists of selecting transformations that can be applied to the source program, proving they are semantics preserving, and applying them to the program before specification extraction. Details of verification refactoring are discussed in an earlier paper [15].

Echo's verification refactoring mechanism provides a set of semantics-preserving transformations that

can be applied to an implementation to facilitate verification in various ways. Several transformations help to reduce the length and complexity of the proof obligations to facilitate the implementation proof. Although the implementation proof is completed mostly by tools (the SPARK Ada tools in our current Echo instantiation), we have found that verification refactoring can be extremely beneficial for the implementation proof, in the limit making proof possible where the tools failed on the original program.

A second set of transformations restructures a program to align the structure of the extracted specification with the structure of the original specification, i.e., to make the matching hypothesis apply to more of the program.New transformations might be developed to accommodate a particular program. Even so, verification refactoring might not provide a match that will enable the implication proof in which case a different verification approach will be needed.

Matching Metric. We have defined a matching metric that summarizes the similarity of the structures of the original and the extracted specifications and thereby indicates the feasibility of proof by parts. The match ratio is defined as the percentage of key structural elements — data types, system states, tables, operations — in the original specification that have direct counterparts in the extracted specification. The match ratio does not necessarily imply the final difficulty of the proof, but the match ratio does provide an initial impression of the likelihood of successfully establishing the proof. In our Echo prototype, the metric is evaluated by visual inspection although significant tool support would be simple to implement.

Establishing the match ratio is fairly straightforward in many cases. Some of the matching can be determined from the symbols used, because the names used in the original specification are often carried through to the implementation and hence to the extracted specification.

5 Approach to Proof

The property that needs to be shown in the implication proof is implication not equivalence, hence the name. 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 behaviors in the original specification than the implementation. The basic definition of implication we use for this is that set out by Liskov and Wing known as behavioral subtyping [11].

The way in which the extracted specification is created influences the difficulty of the later proof. In the case where the implementation retains the structural information from the original specification, a simple way to begin proof by parts is to also retain the structure by directly translating elements of the implementation language, such as packages, data types, state/operation representations, pre-conditions, post-conditions, and invariants, into corresponding elements in the specification language. For each pair of matching elements, we establish an implication lemma that the element in the extracted specification implies the matching element from the original specification. The final proof is organized as the conjunction of a series of such lemmas. There are three types of implication lemmas that we discuss in the next three subsections.

5.1 Type lemmas

For each pair of matching types, we define a retrieval function from the extracted type to the original type. When trying to prove the relation between each pair, two possibilities arise:

Surjective Retrieval Function. If the retrieval function can be proved to be surjective, the extracted type is a refinement of the original type, i.e., all properties contained in the types in the original specification are preserved. If the retrieval function can be proved to be a bijection, the two types are equivalent.

Non-surjective Retrieval Function. If the retrieval function is not surjective, then either: (a) there is a defect if the two types are intended to be matched; (b) certain values in the original specification can never arise; or (c) a design decision has been made to further limit the type in the implementation, i.e.,

to make the post-condition stronger. Upon review, if the user does not confirm that there is a defect or does not further refine the specification, we postpone the proof by transforming the types into subtype predicates on the same base type (e.g. integer). These extra predicates are added as conjuncts in function pre-conditions or post-conditions depending on where they appear, and they are checked when the later lemmas regarding those functions are established.

5.2 State lemmas

State is the set of system variables used to monitor or control the system. State is defined over types, thus type lemmas can be used to facilitate proofs of state lemmas. As with type lemmas, we set up a retrieval function from the extracted state to the original state. For each pair, we prove the following two lemmas:

State Match. As with the type lemmas, we prove that the retrieval function is surjective to show refinement (or equivalence in the bijection case). If it cannot be proved, indicating certain values of the original state cannot be expressed by the extracted state, we again present it for user review. It is either a defect or, by definition, certain values of original state cannot arise.

State Initialization. For states that require initialization, the extracted specification will contain an initialization function. We prove that whenever a state is initialized in the extracted specification, the corresponding retrieved original state also satisfies the initialization constraints in the original specification.

5.3 Operation lemmas

System operations are usually defined as functions or procedures over the system state. When matching pairs of operations in the extracted specification and the original specification, we set up an implication lemma for each pair. The operation extracted from the implementation should have weaker pre-condition and stronger post-condition than the operation defined in the original specification. Specifically, we prove:

Applicability. The extracted operation has a weaker pre-condition than the original operation. For any state `st` upon which the operation operates, given `R` as the retrieval function for `st`, we prove:

```
FORALL st: Pre_org(R(st)) => Pre_ext(st)
```

Correctness. The extracted operation has a stronger post-condition than the original operation if applicable. Given any `st1` and `st2` as input and output for an operation `f`, `R` as the retrieval function for state, we prove:

```
FORALL st1, st2 | st2 = f(st1):
   Post_ext(st2) AND pre_org(R(st1)) => post_org(R(st2))
```

By reasoning over predicates such as the pre-conditions and post-conditions in the low-level specification, we avoid implementation details as much as possible when proving these implication lemmas.

5.4 Implication theorem

The conjunction of all the lemmas forms the implication theorem. All the resulting proof obligations need to be discharged automatically or interactively in a mechanical proof system. Since the extracted specification is expected to have a structure similar to the original specification, the proof usually does not require a great deal of human effort. Also, by setting up the lemmas operation by operation rather than property by property and proving each operation independently, the proof structure easily scales.

6 Proof Process

Our process for applying the proof in practice is shown in Figure 1. In most situations, we choose to proceed with verification refactoring first to increase the match-ratio metric until it becomes stabilized through transformations. There are other types of refactoring and corresponding metrics we evaluate to

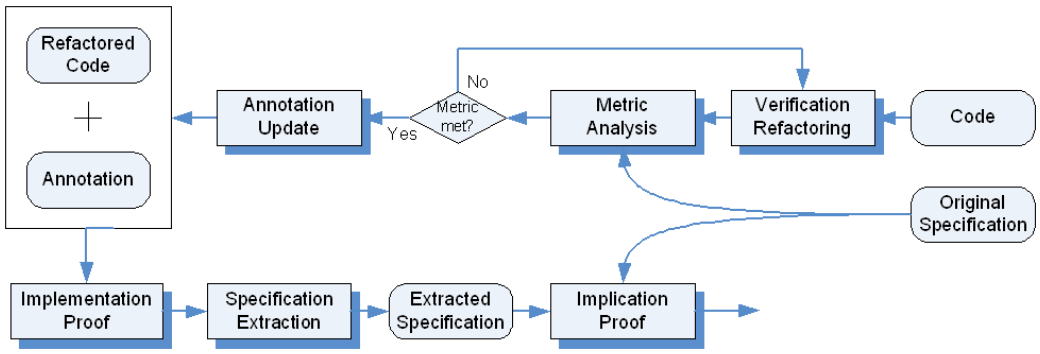

Figure. 1. The proof process.

facilitate the proof process, e.g., to reduce the size of the proof obligations generated. Details of this verification refactoring and metric analysis process and the benefits it brings to our proof have been discussed in our earlier paper [15].

After applying verification refactoring, we have a version of the implementation from which a specification can be extracted that shows structural similarity to the original specification. We then: (a) update and complete the low-level specification (documented as annotations) that might have become erroneous during the refactoring process (e.g. by splitting a procedure); and (b) prove that the code conforms to the low-level specification, i.e., create the implementation proof. This technology is well established, and we do not discuss it further here.

From the refactored code and the associated low-level specification, we extract a high-level specification, i.e., the extracted specification, using custom tools. We then establish the three types of implication lemma and the implication theorem following the approach discussed in section 5. Finally, we prove that the extracted specification implies the original specification.

6.1 Specification Extraction

In this section we demonstrate how we synthesize the extracted specification from the implementation.

Extraction from annotation. For functions that are annotated with proved pre- and post-condition annotations, the annotations provide a level of abstraction. This is helpful for programs that contain a lot of computation. For instance, when one specifies a function, the property that one cares about would be correctness of the output. The actual algorithm used is not important. If the function is annotated and the annotation is proved using code-level tools, we extract the specification from the annotations and leave out the unrelated implementation details. The proof of the annotations by the code-level tools can be introduced as a lemma that is proved outside the specification proof system. An example code fragment that is written in SPARK Ada and the extracted specification in PVS in which the details of the procedure body are omitted and an additional lemma introduced is shown in Figure 2. The additional lemma indicates that the post-condition will be met if the pre-condition is met. The lemma is marked as proved outside and can be used directly in subsequent proofs.

Direct extraction from code. The expressive power of the SPARK Ada annotations with which the low-level specification is documented is limited. Certain properties either: (a) cannot be expressed by the annotation language; (b) are expressible but not in a straightforward way; or (c) are expressible but are not helpful in abstracting out implementation details. In such situations we extract the specification directly from the source code. An example of both code and extracted specification fragments, again in SPARK Ada and PVS, is shown in Figure 3.

Skeleton extraction. A lightweight version of specification extraction is used to facilitate the metric analysis. When verification refactoring is used, we extract a skeleton specification from the transformed code.

```
type state is
   record
      a: Integer;
      b: Integer;
   end record;
procedure foo(st: in out state)
--# derives st from st;
--# pre   st.a = 0;
--# post  st = st~[a => 1];
is
begin
  -- procedure body
  …
end foo;
```

```
state: TYPE = [# a: int, b: int #]
foo_pre(st: state): bool = (st`a = 0)
foo_post(st_, st: state): bool =
   (st = st_ WITH [`a := 1])
foo(st: state): state
foo: LEMMA FORALL (st: state):
   foo_pre(st) => foo_post(st, foo(st))
```

Figure. 2. SPARK Ada code fragment and associated PVS fragment derived from annotations.

We refer to this specification as a skeleton because it is obtained using solely the type and function declarations and contains none of the detail from the annotations or the function definitions. The skeleton specification, however, does reflect the structure of the extracted specification. We can then compare the structure of the skeleton extracted specification with that of the original specification and evaluate the match-ratio metric to determine whether further refactoring is needed.

Component Reuse & Model Synthesis. 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.

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.

6.2 Implication Proof

The final step to complete the verification argument is the implication proof that the extracted specification implies the original specification. The implication argument is established by matching the structures and components of these two specifications and setting up and proving the implication theorem as discussed in section 5.

6.3 Justification of Correctness

The verification argument in Echo is as follows: (a) the implementation proof establishes that the code implements the low-level specification (the annotations); (b) the transformations involved in verification refactoring preserve semantics; (c) the specification extraction is automated or mechanically checked; (d) the implication theorem is proved; and (e) the combination of (a) through (d) provides a complete argument that the implementation behaves according to its specification.

Any defect in the code that could cause the implementation not to behave according to the specifica-

```
procedure foo(st: in out state)
is
begin
   foo1(st);
   st.a = 1;
   foo2(st);
end foo;
```

```
foo(st: state): state =
   LET st1 = foo1(st) IN
   LET st2 = st1 WITH [`a := 1] IN
   LET st3 = foo2(st2) IN
      st3
```

Figure. 3. SPARK Ada code fragment and associated PVS fragment derived from source code.

tion will be exposed in either the implementation proof or the implication proof. Any inconsistency between the code and the annotations will be detected by the code-level tools in the implementation proof. An inconsistency could arise because of a defect in either or both. If both are defective but the annotations match the defective code, it will not be detected by the implementation proof. However the annotations will not be consistent with the high-level specification in this case and so will be caught in the implication proof.

7 The Tokeneer Case Study

In order to evaluate our proof structure approach, we sought a non-trivial application that was built by others, was several thousand lines long, and was in a domain requiring high assurance. The target system we chose is part of a hypothetical system called *Tokeneer* that was defined by the National Security Agency (NSA) as a challenge problem for security researchers [4]. Our interest was in functional verification of the Tokeneer software, and so we applied the Echo proof approach to the core functions of Tokeneer.

7.1. Tokeneer Project Description

Tokeneer is a large system that provides protection to secure information held on a network of workstations situated in a physically secure enclave. The system has many components including an enrolment station, an authorization station, security resources such as certificate and authentication authorities, an administration console, and an ID station. We used the core part of the Tokeneer ID Station (TIS) in this research. TIS is a stand-alone entity responsible for performing biometric verification of the user and controlling human access to the enclave. To perform this task, the TIS asks the individual desiring access to the enclave to present an electronic token in the form of a card to a reader. The TIS then examines the biometric information contained in the user's token and a fingerprint scan read from the user. If a successful identification is made and the user has sufficient clearance, the TIS writes an authorization onto the user's token and releases the lock on the enclave door to allow the user access to the enclave.

Much of the complexity of the TIS derives from dealing with all eventualities. A wide variety of failures are possible that must be handled properly. Since Tokeneer is a security-critical system, crucial security properties such as unlocking only with a valid token and within an allowed time, and keeping consistent audit records need to be assured with high levels of confidence.

A fairly complete,high quality implementation of major parts of the TIS has been built by Praxis High Integrity Systems. The Praxis implementation includes a requirement analysis document, a formal specification written in Z (117 pages), a detailed design, a source program written in SPARK Ada (9939 lines of non-comment, non-annotation code), and associated proofs. Since our tools operate with PVS, we translated the Z specification of the TIS into PVS. The final specification in PVS is 2336 lines long.

7.2. Echo Proof of Tokeneer

The Praxis implementation of Tokeneer was developed using Correctness by Construction [6] with the goal of demonstrating rigorous and cost effective development. High-level security properties were established by documenting the properties using SPARK Ada annotations, including them in the code, and then proving them using the SPARK Ada tools. Our proof is of the functionality of the implementation as defined by the original, high-level specification. Given our proof of functionality, high-level security properties can be established by stating the properties as theorems and proving them against the high-level specification.

Turning now to the proof itself, upon review, we found that the TIS source program structure resembled the specification structure very closely, i.e., the structural matching hypothesis held. Almost all states and operations in the specification have direct counterparts in the source program, e.g., the `UnlockDoor` operation defined for system internal operations in the specification:

```
UnlockDoor(dla_i, dla_o: DoorLatchAlarm, c: Config): bool =
   dla_o`latchTimeout = dla_i`currentTime + c`latchUnlockDuration AND
   dla_o`alarmTimeout = dla_i`currentTime
                          + c`latchUnlockDuration + c`alarmSilentDuration AND
   dla_o`currentTime = dla_i`currentTime AND
   dla_o`currentDoor = dla_i`currentDoor
```

is implemented by a corresponding procedure with the same name in the Door package in the source code:

```
procedure UnlockDoor is
   LatchTimeout : Clock.TimeT;
begin
   LatchTimeout := Clock.AddDuration(
                       TheTime     => Clock.TheCurrentTime,
                       TheDuration => ConfigData.TheLatchUnlockDuration);
   Latch.SetTimeout(Time => LatchTimeout);
   AlarmTimeout := Clock.AddDuration(
                       TheTime     => LatchTimeout,
                       TheDuration => ConfigData.TheAlarmSilentDuration);
   Latch.UpdateInternalLatch;
   UpdateDoorAlarm;
end UnlockDoor;
```

We did not include the interface functions in our verification, and after removing them and performing a skeleton extraction, we found the match ratio to be 74.7%. Almost all states and operations in the specification have direct counterparts in the source program. The match ratio is not 100% (or at least close to it) because refinements carried out during the development added several operations that were not defined in the specification. We concluded that the structures of the specification and the source program were sufficiently similar that we could extract the necessary specification effectively and easily without performing verification refactoring. The final extracted PVS specification is 5622 lines long. As an example, the extracted specification using direct extraction from code for the above `UnlockDoor` procedure is:

```
UnlockDoor(st: State): State =
   LET LatchTimeout = Clock.AddDuration(TheCurrentTime(st),
                                        TheLatchUnlockDuration(st)) IN
   LET st1 = SetTimeout(LatchTimeout, st) IN
   LET st2 = st1 WITH [`AlarmTimeout := Clock.AddDuration(LatchTimeout,
                                        TheAlarmSilentDuration(st))] IN
   LET st3 = UpdateInternalLatch(st2) IN
   UpdateDoorAlarm(st3)
```

Following extraction of the specification, we performed both the implementation proof and the implication proof. For the implementation proof, we used the SPARK Ada toolset to prove functional behaviors of those subprograms that had been documented with pre- and post-condition annotations. The SPARK Examiner generates verification conditions (VCs) that must be proved true to demonstrate that the code does indeed meet its specified post-conditions, within the context of its pre-conditions. The Examiner also generates VCs that must be satisfied to ensure freedom from run-time exceptions. Altogether there were over 2600 VCs generated of which 95% were automatically discharged by the toolset itself. The remaining 5% required human intervention, and were covered in the documents from Praxis' proof.

The implication proof was established by matching the components of the extracted specification with those of the original specification. Identifying the matching in the case study was straightforward, and in most situations could be suggested automatically by pairing up types and functions with the same name as showed by the above example. For each matching pair, we created an implication lemma and altogether there were just over 300 such lemmas. Typechecking of the implication theorem resulted in 250 Type Correctness Conditions (TCCs) in the PVS theorem prover, a majority of which were discharged automatically by the theorem prover itself. In 90% of the cases, the PVS theorem prover could not prove the implication lemmas completely automatically. However, the human guidance required was straightforward due to the

tight correlation between the original specification and source code, typically including expansion of function definitions, introduction of type predicates, or application of extensionality. Each lemma that was not automatically discharged was interactively proved in the PVS theorem prover by a single person with moderate knowledge about PVS, and thus the implication theorem was discharged. The total typechecking and proof scripts running time in PVS is less than 30 minutes.

In section 1 we listed four goals for our proof approach. Drawing firm conclusions about our goals based on a single case study is impossible, but from our experience of the Tokeneer proof (and also earlier, smaller proofs [15]), we conclude the following general indications about our goals:

Relevant. The approach is relevant in so far as Tokeneer is typical of security-critical systems. Tokeneer was established by the NSA as a challenge problem because it is "typical" of the systems they require. The approach can be applied in other domains provided any necessary domain-specific axioms are available in PVS. They might have to be developed by domain experts. We anticipate that verification refactoring will expand the set of programs to which the technique can be applied.

Scalable. The approach scales to programs of several thousand lines provided the structural matching hypothesis holds. There is no obvious limit on the size of programs that could successfully be proved.

Accessible. The approach is mostly automated and relatively easy to understand, and should be accessible to practicing engineers who are generally familiar with formal methods. The Tokeneer proof was completed by a single individual with good knowledge of the technology in use but no special training.

Efficient. Detailed measurements could not be made of the resources used for the Tokeneer proof, but we observe that the resources required were well within the scope of a typical software development.

A valuable benefit of proof by parts is that location of implementation defects becomes easier. In general, a defect is usually located inside the component for which the proof fails for the corresponding lemma. During the proof of Tokeneer, we found several mismatches between the source program and the specification, but later found that they were changes documented by Praxis.

8 Related Work

Traditional Floyd-Hoare verification [7] requires generation and proof of significant amount of detailed lemmas and theorems. It is very hard to automate and requires significant time and skill to complete. Refinement based proof like the B method [1] intertwine code production and verification. Using the B method requires a B specification and then enforces a lock-step code production approach on developers.

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.

Results in reverse engineering include retrieval of high-level specifications from an implementation by semantics-preserving transformations [5, 16]. These approaches are similar to our extraction and refactoring techniques, but the goal is to facilitate analysis of poorly-quality code, not to aid verification. Our approach captures the properties relevant to verification while still abstracting implementation details.

Andronick et al. developed an approach to verification of a smart card embedded operating system [2]. 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.

Heitmeyer et al. developed a similar approach to ours for verifying a system's high-level security properties [8]. Our approach is focused on general functionality rather than security properties.

Klein et al. demonstrated that full functional formal verification is practical for large systems by verifying the seL4 microkernel from an abstract specification down to its 8700 lines C implementation [9]. Proof by parts is more widely applicable and does not impose restrictions on the development process.

9 Conclusion

We have described proof by parts, an approach to the formal verification of large software systems, and demonstrated its use on a program that is several thousand lines long. The approach is largely automated and does not impose any significant restrictions on the original software development.

Proof by parts depends upon the specification and implementation sharing a common high-level structure. We hypothesize that many systems of interest have this property. Where the structures differ, we refactor the implementation to align the two structures if possible. If refactoring fails to align the structures, then proof by parts is infeasible.

We expect proof by parts to scale to programs larger than the one used in our case study with the resources required scaling roughly linearly with program size. For software for which our hypothesis holds, we are not aware of a limit on the size of programs that can be verified using our approach.

10 Acknowledgements

We are grateful to the NSA and to Praxis High Integrity Systems for making the Tokeneer artifacts available. Funded in part by NASA grants NAG-1-02103 & NAG-1-2290, and NSF grant CCR-0205447.

References

[1] Abrial, J.R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
[2] Andronick, J., B. Chetali, and C. Paulin-Mohring, "Formal Verification of Security Properties of Smart Card Embedded Source Code", In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 302-317. Springer-Verlag, 2005.
[3] Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.
[4] Barnes, J., R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett, "Engineering the Tokeneer Enclave Protection Software", In: Proceedings of IEEE International Symposium on Secure Software Engineering, 2006.
[5] Chung, B. and G.C. Gannod, "Abstraction of Formal Specifications from Program Code", *IEEE 3rd Int. Conference on Tools for Artificial Intelligence*, 1991, pp. 125-128.
[6] Croxford, M. and R. Chapman, "Correctness by construction: A manifesto for high-integrity software", CrossTalk, *The Journal of Defense Soft. Engr*, 2005, pp. 5-8.
[7] Floyd, R., "Assigning meanings to programs", Schwartz, J.T. (ed.), Mathematical Aspects of Computer Science, *Proceedings of Symposia in Applied Mathematics* 19 (American Mathematical Society), Providence, pp.19-32, 1967.
[8] Heitmeyer, C., M. Archer, E. Leonard, and J. McLean, "Applying Formal Methods to a Certifiably Secure Software System", *IEEE Transaction on Software Engineering*, Vol.34, No.1, 2008.
[9] Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel", Proceedings of the *ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky, Montana, USA, October, 2009.
[10] Knight, J.C., C.L. DeJong, M.S. Gibble, and L.G. Nakano, "Why Are Formal Methods Not Used More Widely?", *Fourth NASA Formal Methods Workshop*, Hampton, VA, 1997
[11] Liskov, B. and J. Wing, "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, 16(6):1811--1841, 1994.
[12] Owre, S., N. Shankar, and J. Rushby, "PVS: A Prototype Verification System", Proceedings of *CADE 11*, Saratoga Springs, NY, 1992.
[13] Strunk, E., X. Yin, and J. Knight, "Echo: A Practical Approach to Formal Verification", Proceedings of FMICS05, *10th International Workshop on Formal Methods for Industrial Critical Systems*, Lisbon, Portugal, 2005.
[14] Yin, X., J. Knight, E. Nguyen, and W. Weimer, "Formal Verification By Reverse Synthesis", Proceedings of the *27th SAFECOMP, Newcastle*, UK, September 2008.
[15] Yin, X., J. Knight, and W. Weimer, "Exploiting Refactoring in Formal Verification", *DSN 2009: The International Symposium on Dependable Systems and Networks*, Lisbon, Portugal (June 2009).
[16] Ward, M., "Reverse Engineering through Formal Transformation", *The Computer Journal*, 37(9):795-813, 1994.