Two Weak Links in the Formal Methods Chain

Carl A. Gunter Insup Lee Andre Scedrov*

August 3, 1997

There has been considerable recent progress on the development of tools supporting the use of formal methods, as well as a better appreciation of the kinds of problems that are particularly suited for formal methods. Together with the availability of faster computers, these developments have resulted in some significant achievements for formal methods. This has lead to a degree of optimism and enthusiasm about the prospects of using formal methods in the verification of important systems and protocols, especially for hardware and for high-assurance software with safety or security requirements. Attempts to use formal methods can lead to clearer thinking about problems and hence lead to the discovery of new insights or bugs, but it is also generally hoped that formal methods can *prove* that a system satisfies its requirements in a mathematically rigorous way. If this is the goal, there are two weak links in a chain of inferences that threaten to make it unachievable:

1. The specification may not be rigorous enough to support mathematical proof or may too incomplete to imply the desired properties.
2. The implementation may be too complicated or not mathematical enough to allow a proof that the specification is satisfied.

We believe that the conditions for meaningful applicability of formal methods to security protocols would be a good topic for discussion at the DIMACS Workshop on Design and Formal Verification of Security Protocols. As a starting point for such a discussion, this note elaborates on the nature of the two problems above and some of their possible remedies.

We illustrate the first of these two problems with war stories from the specification of programming languages and cryptographic protocols. It is sometimes thought that a language such as C is not sufficiently well-specified to support formal proofs. However, this is only a part-truth. A bigger problem is that the specification leaves so many possibilities open that proofs of properties from a corresponding formal specification would be quite difficult. For instance, the following correct C++ program

```
void main(){
  int a[] = {1,2,3};
  for (int i = 0; i < 3; i++)
    { a[i-1] = -1; cout << "i = " << i << ", "; }
}
```

*Author email addresses: {gunter,lee,scedrov}@cis.upenn.edu.

has the output `i = -1, i = -1, ...` for at least one machine, compiler, and program run, because its semantics depends on the runtime layout of the variables.[1] By contrast, it is easier to formulate properties for a more restrictive design, *if* there is a sufficiently rigorous specification. For instance, significant progress has been made in proving properties of the SML programming language [13, 14] based on its rigorous, rule-based operational semantics [11], but it is fundamentally difficult to prove properties about the semantics of a programming language because of its size and complexity. A more recent language, Java [5], has attempted to simplify the task by compiling the language into instructions for a standardized virtual machine [8], which is hoped to be simple and precise enough that security properties can be assured. However, this raises some questions about the link between the standard and any formal proof that may be given. Consider the following requirement from [8]:

> If an error occurs during verification, then an instance of [an exception] will be thrown at the point in the Java program that caused the class to be verified.

A formal specification would require a mathematical interpretation of terms like 'at the point in' and 'caused'. If, for the sake of argument, other parts of Java VM specification do not clarify this, there is a risk that a standard-conforming implementation might not attach the same meaning to these terms as the formal definition. Such an implementation may fail to have whatever property is being attributed to the standard based on the formal proof. The chain forged by the formal proof has been broken at the link between the standard specification and its formalization. Since the informal specification is the standard, correct implementations may fail to satisfy the desired property and the formal proof of correctness is irrelevant. The lesson, in short, is that formal methods cannot achieve their primary aim if they must be based on standards that are not precise enough to support formal proofs.

A related issue that arises in the specification of cryptographic protocols, described in [10], is that a formalization may compound fallacious informal assumptions. Indeed, a logic of authentication introduced in [1] provides a formal correctness proof of the Needham-Schroeder public key protocol, but the logic relies on the fallacious assumption that principals never divulge secrets and thus the formalism misses the man-in-the-middle attack [9]. Furthermore, the

[1] The off-by-one array indexing error clobbers the loop control variable, which happens to be stored one location past the lower end of the array.

error is compounded by the suggestion in [1] that nonces themselves be used authenticators in subsequent communication, in contrast to the original suggestion in [12] that authentication be supplied by public-key signatures. As Kreisel used to emphasize [6] in the context of mathematical logic, no amount of formalization can make up for the lack of *informal rigor*.

Let us consider now the other weak link in the chain, between the implementation and its specification. For example, it has been mentioned that SML has a precise specification, but is there any SML compiler implementation that has been formally proved to satisfy it? The answer is no, although progress on language specification and compiler verification has brought the possibility of doing this closer. The link between the rigorous specification and the implementation is fragile: we cannot attribute more mathematical precision to the compiler than the strength of this connection justifies. How serious is this problem? Fortunately it is possible to rely on two things for getting around it. First, the specification can be made closer to the implementation, thereby making the assurance of conformity stronger. Second, the specification can be used to generate test data that act like little theorems proved by running the implementation of the system. The first approach can rely on a layering of specifications; this may be more feasible in some circumstances (such as security protocols) than it is with general-purpose programming languages. In the second case we surrender the claim that a full *proof* of correctness has been performed.

Let us consider a way of using the specification to generate test data where there is no proof but where the level of assurance is at least quantifiable. Of course, if the test data covers all possible cases, this testing is as assuring as formal proof; unfortunately it is not usually feasible to do this much testing. A more relaxed condition is to show that the test data provides probabilistically high assurance. For instance, one approach [15] of this kind uses a deterministic finite-state Mealy machine as a specification and assumes that the implementation is a black box for which only input and output is observable. With the single assumption that the black box has no more states than the specification machine there is a polynomial-time randomized algorithm for determining the equivalence of the specification and its implementation.

Another approach is to generate tests from requirements specification and use them to validate design specifications as well as implementations. Recently, we have developed a framework for testing timing constraints of real-time systems as well as testing coverage criteria [2]. Here, tests are automatically derived from requirements specifications of minimum and maximum allowable delays between input/output events in the execution of a system. The test derivation scheme uses a graphical formalism for timing requirements and the real-time process algebra Algebra of Communicating Shared Resources (ACSR) [7] for representing tests and specifications. The advantage of this testing based approach is that it can be used to validate a design specification which has too many states for exhaustive exploration of the state space. Another advantage is that this approach produces tests as an artifact of the specification validation process. These tests which are used to validate the specification can also be used with minimal additional effort to validate the actual system implementation. This technique has been used to validate specifications which have to many states for complete state space exploration; e.g., the Sunshine ATM switch [3] and the Philips Audio Control Protocol [4]. We believe that a similar approach can be used to validate the design specifications and implementations of a security protocol.

References

[1] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[2] Dunca Clarke. *Testing Real-Time Constraints*. PhD thesis, Department of Computer and Information Science, The University of Pennsylvania, Philadelphia, PA 19104, Dec 1996.

[3] Duncan Clarke and Insup Lee. The Modular Specification and Analysis of an ATM Switching System Using a Process Algebra. In *Proc. International Test Conference*, Oct 1996.

[4] Duncan Clarke and Insup Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *IEEE Real-Time Technology and Applications*, pages 112–124, 1997.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[6] G. Kreisel. Informal rigour and completeness proofs. In *Problems in the Philosophy of Mathematics*, pages 138–186. North Holland, 1967.

[7] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994.

[8] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[9] G. Lowe. An attack on the Needham-Schroeder public key protocol. *Information Processing Letters*, 56:131–133, 1995.

[10] C.A. Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proc. of ESORICS*. Springer Verlag, 1996.

[11] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[12] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[13] D. Syme. Reasoning with the formal definition of Standard ML in HOL. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 43–60. Springer-Verlag, February 1994.

[14] Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, 1996.

[15] M. Yannakakis and D. Lee. Testing finite state machines: Fault detection. *Journal of Computer and System Sciences*, 50, 1995.