To appear in the APA Newsletter on Philosophy and Computers, Fall 2011 (No. 1, Vol. 11).

When Formal Systems Kill: Computer Ethics and Formal Methods

Darren Abramson[*] Lee Pike[†]

February 5, 2012

"Beware of bugs in the above code; I have only proved it correct, not tried it."

—Donald Knuth

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

—Edsger Dijkstra

Abstract

Computers are different from all other artifacts in that they are automatic formal systems. Since computers are automatic formal systems, techniques called formal methods can be used to help ensure their safety. First, we call upon practitioners of computer ethics to deliberate over when the application of formal methods to computing systems is a moral obligation. To support this deliberation, we provide a primer of the subfield of computer science called formal methods for non-specialists. Second, we give a few arguments in favor of bringing discussions of formal methods into the fold of computer ethics.

1 Introduction

1.1 Our Goals

Consider the following general ethical question: 'what moral obligations do software engineers and programmers have to apply formal methods to the fruits of their labour'? A subdiscipline of computer science called *formal methods* aims to dramatically increase the quality of software by mathematically *proving* programs correct as opposed to merely testing them.

This paper attempts to do two things. First, it is a call to arms to computer ethicists to answer this general question. Second, it argues that answering

[*]Dalhousie University. email: `da@dal.ca`
[†]Galois, Inc. email: `leepike@galois.com`

the question of how much formal methods is a natural and important part of computer ethics. We show that safety for computational artifacts is different in kind from conventional artifacts. In doing so, we provide a primer that will enable practitioners of computer ethics to better understand what formal verification is.

Some might argue, in considering the first of our two goals, that it is trivially obvious that the question of how much expense should be taken to ensure the safety of software, especially when morally significant losses are at risk, is worthy of study by computer ethics. We are not merely making this claim. Indeed, this paper is motivated by the following pragmatic conundrum: Given that there are unique, expensive methods to ensure safety of computer systems, how come there is little philosophical discussion among computer ethicists regarding their use? We speculate the reason is a lack of widespread understanding that ensuring safety for computers is different from ensuring safety for other artifacts. The central purpose of this paper is *not dialectical but practical*: we hope to broaden the discipline of computer ethics, and by doing so, change social expectations of the safety of computer systems.

We take the view that doing applied ethics is not merely an intellectual pursuit, but also has as its end to change human practices. Just as bioethics demanded, and procured, considerations of autonomy in doctor/patient settings[1] we hope that computer ethics will demand, and procure, considerations of formal methods in the production and consumption of software.

1.2 Scope and Outline

In this paper, we advocate for the normative implications of the automatic formal system property of computers. To ground our investigation, we mostly restrict ourselves to *safety-critical* computer applications. These applications are ones in which failures may lead to the loss of human life. Such uses include aircraft control systems, automotive subsystems, computer systems for piloted spacecraft, and computerized medical devices. There is no reason, though, that in discussion of the question 'how often should formal methods be applied to software' computer ethicists must limit themselves to such applications.

In Section 3, we overview the field of formal methods, while in Section 4, we particularly focus on obstacles – both technological and cultural – for the field of formal methods, and we attempt to summarize the attitudes within computer science on the subject. Knowing what formal methods is, and explaining computer scientists' attitudes on its application, are prerequisites for an informed productive discussion of how often it should be applied.

Next we argue that consideration of formal methods belongs in computer ethics on general grounds of what justifies computer ethics. To more concretely motivate the place of formal methods within the discipline of computer ethics, in Section 5 we describe both how it informs and is informed by a debate central

[1]The result of this deliberation directly affected the nature of medicine in the West. See, for example, [Emanuel and Emanuel, 1992].

to computer ethics – the free software debate. Concluding remarks are given in Section 6.

2 Computers and Computer Ethics

2.1 Automatic Formal Systems

Before we explain what formal methods are, it is useful first to say something about what defines an object as a computer. It is the defining characteristic of computers that makes ensuring safety different for them than other objects. It is worth noting that this defining characteristic of computers has been well studied by cognitive scientists but not computer ethicists.

Cognitive scientists, by and large, agree that what makes computational systems distinct is that they are instances of, to use John Haugeland's term, *automatic formal systems* (AFS) [Haugeland, 1989, Fodor, 1990, Haugeland, 1997, 8-9]. An AFS is a concrete system that satisfies the following three properties:

1. *Token manipulation*: computers manipulate symbolic tokens according to formal rules (like games or logics).

2. *Digitalization*: computers have exact, repeatable results, as opposed to continuous systems (e.g., billiards or the weather).

3. *Finite "playability"*: no computations take infinite time or require an "oracle".

Therefore, on this characterization, a computer realizes a formal system. A formal system can be described mathematically by a logic. Therefore, if we hold constant certain issues of fabrication and usage, we can, by applying a mathematical proof, *guarantee* that certain properties will hold of a computational system. Notice that we are only claiming a narrow implication: AFSes, implementing a sort of mathematical object, can in principle be subject to proof concerning certain abstract properties. This is not to say, of course, that there is a method such that for any abstract property and any formal system, the method can prove or disprove that the property holds of that system. To claim otherwise would be in violation of Church's Theorem, since assessing first order validity is an instance of determining abstract properties for a formal system.

Overall, the AFS property has received little attention in regards to its normative implications. Recent texts and anthologies in computer ethics instead focus on the unprecedented abilities of computers to store, transfer, and analyze information [Johnson, 2001, Ermann and Shauf, 2003, Winston and Edelbach, 2006, Tavani, 2007]. We do not mean to dispute the factual claim that computers indeed have these unprecedented abilities that result in new ethical problems. Rather, we present the AFS property as being an additional property justifying the study of computer ethics, the thesis of which is central to the philosophical underpinnings of formal methods.

2.2 The Standard Justification

Let us consider AFS property with respect to a "standard justification" for computer ethics. In her influential textbook on computer ethics, Deborah Johnson argues, in answering the question, "Why computer ethics?" that the issues raised by computers are neither wholly new, nor wholly old.

> The ethical issues surrounding computer and information technology are not new in the sense that we have to create a new ethical theory or system. They call upon us to come to grips with new species. This means understanding the new issue in familiar moral terms, using traditionalist moral concepts. For the most part this is consistent with the traditionalist account because once connected to standard moral categories and concepts, the new issue can be addressed by extending familiar moral concepts to the new situation and drawing on our experience with other cases. However, the new species may not fit easily into standard categories and concepts: allowances for the special or new features of the new situation have to be taken into account. new species have special features and, as pointed out earlier, if we simply treat them as the same as other, familiar cases, we may fail to recognize how the new features change the situation in morally significant ways. [Johnson, 2001, 22]

Johnson introduces what she calls the 'genus-species account' of computer ethics to describe how we ought to address the ethical puzzles introduced by computing technology. It is dangerous, she argues, to think that we can understand our obligations by applying policies and arguments concerning older technologies and issues. Clearly, safety of manufactured products is a human concern with an existing history of substantial policies, laws, and arguments for responsibilities of various kinds. However, this genus of moral consideration, ensuring safety of human artifacts, cannot be applied to the species of AFSes in a traditional fashion.

Likewise, AFSes have a sort of complexity that is incommensurable with other artifacts (see Section 4.1). This new form of complexity is most obvious when we consider both the amount of testing needed to ensure that an artifact is safe and the certainty available from testing. On the one hand, testing traditional artifacts can only yield probabilistic measures of safety. On the other hand, the verification of AFSes, with respect to their formal specification and under environmental assumptions, can yield absolute certainty of safety.

3 What is *Formal Methods*?

Formal methods is[2] a subdiscipline of computer science. Formal methods takes the AFS property, described in Section 2.1, as an underpinning to its entire

[2]The phrase "formal methods" is used both to refer a field of practice (in which case the noun phrase is used in the singular) and as a description of the various methods used in the field (in which case the noun phrase is used in the plural).

enterprise. Formal methods are mathematical techniques that are used to prove that particular programs (or, alternatively, hardware units) are *correct* – that is, that software or hardware satisfy a mathematical description of its desired functionality. This abstract description of desired functionality is called the *specification* of the hardware or software.

Formal methods, used to construct computer systems that behave as intended, can be contrasted with empirically testing the computing system. As an analogy, formal methods is related to conventional bug testing for computer design in much the same way that computational fluid dynamics (CFD) is related to wind tunnel testing for aircraft design [Rushby, 1993].[3] The field of CFD is concerned with building mathematical models to investigate the aerodynamic properties of, say, an aircraft wing design or a boat's dynamics through water. CFD techniques allow dynamics to be analytically studied as opposed to building a physical model to test using a wind tunnel or other empirical experiment.

Similarly, manufacturers of computer systems employ engineers to be testers so that they can run their products through a "wind tunnel" (or "test harness," as it is called in software engineering). A test harness contains a large suite of inputs with the hope of covering both the full range of normal uses of the system as well as exceptional circumstances. As we have already described, through formal methods, one attempts to analytically analyze the system to prove properties about it for *any possible* inputs.

Broadly, three research directions are pursued in the field of formal methods:

1. *How to mathematically model formal systems and their environments.* Unlike other engineering artifacts that are mathematically-modeled (bridge stresses, aircraft aerodynamics, etc.), many concepts in computer science are brand new, and a research challenge is simply how to mathematically model these concepts. Such concepts of course include modeling software and hardware and the environments in which they operate.

   Regarding environmental assumptions, any formal verification makes implicit and explicit assumptions. For example, in proving some property about a program, one might implicitly assume that the computer on which the software executes is not physically destroyed, that no bugs in the hardware change the intended program semantics, that the process thread executing the code is not terminated, and so on. Likewise, some assumptions might be explicit. For example, one might state as a hypothesis that an integer input to a program takes no more than a fixed number of bytes to represent it in binary. Other kinds of environmental assumptions are probabilistic. For example, fault-tolerant systems found in commercial aircraft are designed to mask a certain kind of and number of faults. The kinds of and number of faults is characterized by a *maximum fault assumption* (MFA). The MFA should hold with sufficiently-high probability, but

[3]The authors thank Jeffrey M. Maddalon of the NASA Langley Research Center for pointing out that the analogy breaks down in a critical way. We describe the shortcoming in Section 4.1.2.

it is only probabilistic. However, supposing the MFA holds, the system should *provably* satisfy its specification. Thus, the correctness of the system is ultimately probabilistic even if its correctness under the MFA has been formally verified. *That formally verified system relies on possibly probabilistic hypotheses about the environment is a subtle point sometimes glossed over in the formal verification debate.*

In addition to modeling the environment, another challenge is to define and refine the abstraction of systems. For example, software models may be of the source-code semantics, an intermediate representation, a memory-aware model, or the machine code semantics. Hardware models may be of the microarchitecture, the register-transfer level, or of physical-layer protocols between components. System-level models may be of interacting subcomponents and interfaces at different levels of abstraction. Research is also devoted to modeling the connections between different levels of abstraction and aspects of systems and the software and hardware from which they are composed.

2. *How to mechanically check the correctness of mathematical proofs.* The output of this research is perhaps most visible in the development of *mechanical theorem-provers* [Wiedijk, 2006]. These software systems allow mathematics to be formulated[4] in a formal language, and the theorem-provers can check the correctness of proof scripts over the formulations that a user writes. Some theorem provers also provide a degree of automation to assist in the development of the proofs.

3. *How to automate mathematical proofs.* The focus of this research is on how to automatically generate proofs of correctness. The approach is limited by the state-explosion problem – even simple formal systems often contain an infeasible number of states to check whether some property holds of those states. However, techniques developed over the last two decades have made the approach feasible for systems with well over 10^{20} states [J.R. Burch et al., 1990]. Decision procedures for decidable logics is also an active area of research, allowing infinite-state systems to be automatically verified [de Moura et al., 2004, Lahiri and Seshia, 2004].

4 The History of the Debate

In Section 4.1, we discuss the apparent paradox of the mathematical modeling of formal systems lagging behind the success of mathematically modeling less abstract artifacts, such as bridges and aircraft, for example. We describe the

[4]We wish to emphasize the difference between formalization and formulation, particularly because the two notions are sometimes conflated in the field. *Formalization* is the act of expressing a concept mathematically; for example, Peano formalized arithmetic and Euclid formalized planar geometry. *Formulation* is the act of expressing mathematics in a formal language; for example, proofs can be formulated in the sequent calculus. (Steve Johnson of Indiana University made this distinction clear to the second author.)

status of ethical imperatives to use formal methods within computer science in Section 4.2. In Section 4.3, we very briefly describe the inroads that formal methods have made while focusing on what has motivated their use.

Taken together, these sections supports the pragmatic purpose of this paper, by providing the computer ethicist with sufficient background on the use, benefits, and detriments of formal methods. We particularly try to convey a "computer scientists"' perspective, realizing we make some simplifications and generalizations.

4.1 Bridges, Planes, and Programs

On the face of things, bridges and mathematics do not appear to be intimately connected, but civil engineers and physicists have figured out how to mathematically model bridges to determine analytically all sorts of characteristics such as the maximum load a bridge can withstand, the effects of strong winds and earthquakes, and so on. The problem of mathematically modeling a bridge is essentially solved. The same story holds for studying the aerodynamics of aircraft, as mentioned in the previous section.

A priori then, if a computer is by definition an implementation of a formal, mathematical system, one may find it surprising that our ability to mathematically model programs is inchoate relative to bridges or planes. Why is that?

Three obstacles prevent formal methods from being widely adopted: formal requirements specification, the complexity of proofs, and the size of software systems.

4.1.1 Requirements Specification

Software requirements, particularly in safety-critical systems, are notoriously difficult to get right [Lutz, 1993, Berry and Wing, 1985]. Indeed, the very idea of requirements being "right" may be incoherent. Requirements evolve as the needs and expectations of users evolve. Software gets used in ways that architects and developers would never have expected. Environmental interactions may produce unexpected results (e.g., a programmer for your web browser neglects to handle the case in which the user is navigating to a new web page and concurrently opens a new browser window, causing the application to crash).

While even formulating requirements is difficult, formalizing them is even more so. Stated informally, requirements often lack detail or omit corner-cases. For example, one might have the requirement that a hardware device multiply two numbers. Stating the requirement in a mathematical notation may lead the designers to consider the requirement in more detail: What happens if one of the numbers overfills a buffer? What if a buffer is modified during the computation? If the result overfills the result buffer, what should be returned? And so on.

On the other hand, formal methods may have its greatest payoff during the requirements engineering stage of a software project by forcing designers to overcomes these difficulties early, before software has been implemented [Berry and Wing, 1985].

4.1.2 Proof Complexity

Whereas the in CFD, a system is modeled with differential equations, programs are modeled using logic (propositional, first-order predicate, and second-order predicate logic are all used). A program can be modeled by determining the satisfiability of a logical formula modeling it – intuitively, a program is turned into a logical formula stating something of the form, "For all inputs, program P yields outputs with property X." The computational complexity of determining the satisfiability of even boolean formulas – the simplest of logics – is NP-complete, meaning it belongs to a mathematical class of "very hard" computational problems.

In mathematics, complexity is oftentimes managed by abstraction or problem-decomposition. While these techniques certainly pertain to software as well, their application is limited. Intuitively, the difficulty with decomposing software verification results from small local changes having global effects. For example, if a program changes a single "1" to a "0", the entire program could result in completely opposite behavior. De Millo, Lipton, and Perlis call this property the *discontinuity* property of software, as contrasted with the *continuous* functions reasoned about in Newtonian physics [Millo et al., 1979]. That said, well-designed software is built to be compositional so that to the greatest extent possible, errors are localized. For example, a software bug that causes an application to crash should not cause the operating system to crash. In continuous domains, composition is inherent and depends less on good design principles. For example, a small, localized change to, say, the shape of an airfoil will have relatively small, localized effects on the aerodynamics of the aircraft.

In contrast, physical systems, like those analyzed using CFD models, are inherently continuous. CFD models can be used to model a system's behavior under nominal conditions. To account for potentially anomalous behavior, a "safety factor" can be built in. For example, if an wing is expected to undergo x kilograms per square centimeter (kg/cm^2) under nominal conditions, it can be built to withstand $1.5x$ kg/cm^2, or some other safety factor. Due to the discontinuity of software, an analogous safety factor cannot be similarly computed.

4.1.3 Software System Size

The second reason that formal methods have not been more widely used is the size of software systems. For example, there is an estimated one billion lines of code on the new Airbus 380 airliner (not all of it is safety-critical, however) [Knight, 2002a]. Comparatively, the largest aircraft carriers in the world have on the order of one billion parts.

Together, the difficulty of mathematically modeling computers is more apparent – imagine if one missing bolt in an American aircraft carrier turned it into a Soviet submarine. That's software.

4.2 The "Computer Science Perspective"

Philosophers, logicians and computer scientists have written extensively on metaphysical underpinnings of formal methods and program verification [Barwise, 1989, Smith, 1985, James, 1988]; a nice annotated bibliography is available on-line [Rapaport, 2007]. Much of the debate has ranged over issues such as what it means to prove a program correct and what is the nature of a mathematical proof (i.e., is it sufficient for a machine to verify a proof or must a human do so?).

However, as far as the authors are aware, the normative implications of formal methods have largely been ignored by professional ethicists, and they have been mostly taken for granted amongst computer scientists. Of course, generalizing the viewpoint of all computer scientist professional or even all formal verification practitioners is impossible, but the normative questions rarely arise. When they do, there is a presupposition that they will further support program verification. The point might best be made by a quip heard by one of the authors at a recent formal verification conference: "It's true that no catastrophic commercial aircraft crash has been determined to be the result of faulty software, and that's a pity – program verification would be in greater demand if one had."

On the other hand, formal methods practitioners do not simply deliver diatribes against software and hardware developers. Formal verification practitioners are computer scientists themselves. They know that building correct software is difficult and that the most stringent program verification practices are not sufficiently mature to be used by non-experts in large-scale projects (indeed, skeptics often point out that few formal methods advocates verify programs they write themselves). Furthermore, industrial practitioners are employed by corporations producing the potentially-faulty software. Consider that although an airbag designer may genuinely believe in the potential airbags to save lives, how many (publicly) condemn their automobile manufacturer for not installing airbags in every make and model? Likewise, no verificationist would seriously claim that *all* software should be verified.

Perhaps a fair characterization of the formal verification community's position is something along the following lines:

> Program verification is difficult. Our job is to figure out ways to reduce the difficulty so that the practice is feasible for industrial-scale endeavors. For security-critical and safety-critical systems, program verification is a "best practice" that should be employed, but we acknowledge the trade-offs between assurance of correctness and cost. Just as a car with anti-lock brakes is safer than one without, some individuals decide the trade-off of greater safety is not worth the additional cost. Our job is to make the practice more feasible and to advocate for formal methods, but only in a few circumstances would we claim that program verification is a moral imperative.

Let us expand the last point a bit regarding the moral obligation to practice program verification. Continuing our comparison to best practices in automobile manufacture, the infamous Ford Pinto case seemed to be a clear-cut moral issue. The Ford Motor Company calculated the cost of settling lawsuits due to a known unsafe design against the cost of recalling the vehicle [Dowie, 1977]. In comparison, malicious software development and deployment has not yet had similar high-profile cases of maliciousness. Indeed, faulty software has relatively rarely been deemed to be the result of gross incompetence, given the acknowledged complexity of software (see Section 4.3) and difficulty of formal verification. Indeed, we state the following *formal methods dilemma*:

> If formal methods is a best practice of software engineering, then an engineer who does not employ it is either negligent or incompetent. But formal methods is beyond the capability of typical software engineers (otherwise, why do we need formal methods experts and researchers?) or is too time-intensive to employ, so it cannot be considered to be a best practice today.

4.3 The Story So Far

Not surprisingly, advocates for formal methods are largely computer scientists, and even less surprisingly, the most ardent advocates are formal methods researchers and practitioners. The most significant inroads of formal methods into industrial design have come by way of economic motivation rather than ethical considerations. Perhaps the singularly most famous instance is Intel's "Pentium FDIV bug", which was a subtle hardware bug found in 1994 [Halfhill, 1995]. The bug eventually led to Intel's replacing defective chips, costing the company some half-billion dollars. Subsequently, Intel and other hardware companies began to augment their testing staff with formal methods experts.

In general, bugs have been less costly for software companies since software can be patched whereas hardware can only be replaced.[5] Nevertheless, more recently, software companies, such as Microsoft, also employ formal methods engineers to help improve code quality.[6] The motivation of a software company is increased reliability (e.g., reducing the likelihood of the "blue screen of death") and increased security.

Outside of the mainstream, safety-critical and security-critical computer developers have long advocated for – if not relied upon – formal verification [Neumann, 1996]. Safety-critical software, for example, includes automated flight-control software developed for commercial aircraft [Knight, 2002b]. Security-critical computers includes encryption devices [Pike et al., 2006]. Safety-critical and security-critical devices are usually designed to be as simple as possible with well-defined fixed functionality; see, for example, the L4 Microkernel[7] Project [Elphinstone et al., 2007]. Their simplicity begins to make formal verification feasible.

[5]Sometimes, software can be modified to mask faults in hardware.

[6]For example, the *Software Reliability Research* group at Microsoft Research (`http://research.microsoft.com/srr/`) builds tools and invents new techniques to assist Microsoft developers to build more robust systems.

[7]A *microkernel* is a lightweight (and usually highly-robust) operating system.

Today, safety-critical and security-critical computer systems are becoming more pervasive. For example, next-generation automobiles may have "autopilot" functionality [Baleani et al., 2003]. Software and robotics are used in medical devices [Jetley et al., 2006]. Many of us rely on security protections of on-line banking, shopping, and so on. The pervasiveness is coupled with more complexity and increased functionality.

Amongst formal methods practitioners, there has been a conventional wisdom that lawsuits will soon be a prime motivator. The idea has been that liability lawsuits will be brought against hardware or software companies for losses (such as financial loss, security, life, etc.), and when it is shown that best practices in the field include formal methods, companies not employing them will be deemed negligent. The problem is, this has been the conventional wisdom for more than twenty years!

Such lawsuits have not materialized, despite estimates that faulty software and abandoned software development projects costing the U.S. economy at least $5 billion per year [Charette, 2005]. Why they have not – when, for example, extravagant liability lawsuits have been brought against companies in most other economic sectors – is an interesting question itself. However, bringing the discussion of formal methods to a wider audience might alter expectations by the public for the software they use.

5 Formal Methods and Intellectual Property

In this section, we further motivate the importance of considering formal methods within computer science. In Section 1, we began to motivate the ethical consideration of formal methods by describing the AFS property and examining it within the context of the "genus-species" justification for computer science. Here, we explore a topic that is squarely within the domain of computer ethics – free software – and argue that the consideration of formal methods informs the free software debate. First, we examine arguments in favor of the imperative that all software be free, in a very specific sense. We show that these arguments do not establish the desired conclusion. Then we argue that consideration of formal methods strengthens the position of free software advocates.

5.1 Stallman's Freedom Manifesto

To begin, let us briefly review the free sofwate debate. Computer ethics routinely treats the issue of whether intellectual ownership of software is ethically permissible, forbidden, or optional. In an influential series of documents, Richard M. Stallman has advocated the second position: he claims that allowing an individual or corporation to own software is unethical. A central thesis of Stallman's *GNU Manifesto*[8] is that free software is better justified than proprietary software [Stallman, 1985]. We will briefly explain what this means, how Stallman

[8]'GNU' is a "recursive" acronym that stands for "GNU's not UNIX," where UNIX is a famous operating system standard.

argues for this claim in very broad terms, and then describe the importance of formal methods for the issue of the freedom of software.

First, we must understand what Stallman means by free software. In an endnote to the *Manifesto*, he describes an ambiguity later resolved in another article, *The Free Software Definition.*[9] Stallman does not mean that individuals ought to have free access to physical instances of software (or, as he puts it in the definition, “You should think of free as in free speech, not as in free beer.”). Rather, Stallman goes on to argue in the same article that individuals should have the following four freedoms with respect to software they acquire. We quote these freedoms below:

Freedom 0 The freedom to run the program, for any purpose.

Freedom 1 The freedom to study how the program works, and adapt it to your needs. Access to the source code is a precondition for this.

Freedom 2 The freedom to redistribute copies so you can help your neighbor.

Freedom 3 The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. Notice that access to the source code is a precondition for this.

Stallman's *Manifesto* is directed at encouraging support for the creation of a fully-featured operating system under these four freedoms. Moreover, he gives some arguments in favor that all software should be free in these senses. In justifying the GNU project, Stallman appears to argue that his idea for a software project follows from defensible, general principles for how software should be created.

For example, he writes, “GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.”[Stallman, 1985, 155]

Stallman also addresses independent considerations of utility. For example, consider the last two paragraphs of the same article: Stallman insinuates a utopian future based on sharing of information. Last, Stallman appeals to the historical justification used by Western societies for introducing protections on intellectual property. He claims first that copyright law has been introduced to benefit society, and not merely innovating individuals. Then, he writes,

> The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually. . . (ibid., 159)

[9]Available at http://www.gnu.org/philosophy/free-sw.html.

Let us assume, with Stallman, for the time being, that copyright law has only been, and only ever is, justified on the basis of benefit to society and not the individual whose intellectual property might be protected. Furthermore, let us suppose also that there are clear personal benefits to open source software. Nevertheless, it has not been widely agreed that the facts concerning software Stallman cites do not establish his conclusion that, on balance, enforcement of copyright for software harms society. After all, it is not clear that, for example, Apple is on balance harming society by releasing only compiled code for its operating system to the public.

To be clear, it is true that open source software can be easily improved and exchanged by the public at large than closed source software can. However, there are possible incentives *for society* in having closed source software. As we have been stressing, some software projects are extremely large and expensive. By enforcing copyright for software, the public can ensure that companies that take great risk in creating a piece of software can have a reasonable expectation of recouping their expenses. Perhaps the benefit of having Apple invest considerable resources into its operating system, OS X, with the expectation of profit outweighs the harms of my not being able to (legally) copy and alter it. It is clear, then, that the questions of if and how much of software ought to be open source remains itself open. In the next section we show that new arguments concerning formal methods can be applied to this issue, and in a way that may tip the balance for certain uses of software.

5.2 Formal Methods and Free Software

Consider the following claim: the specification and verification of software depends on publicly scrutable proofs and to be accomplished effectively, it must be done in an open intellectual community. De Millo, Lipton, and Perlis have famously argued that mathematical proof is a *social process*, meaning that the value of a proof is to convince other mathematicians of the truth of the proved fact, and indeed, to say that a fact is proved is to say that the community has internalized the truth of the fact, not that some informal logical derivation has been published (however, they claim this does *not* hold for formal verification specifically; we address that claim shortly) [Millo et al., 1979].

Couched in terms of Stallman's freedoms (Section 5.1), the motivation for open proofs mostly falls under Freedom 1 (freedom to study how the proof works) and Freedom 3 (freedom to improve the proof). Presently, open, independently-verifiable results are not ubiquitous in formal methods specifically or computer science more generally. Indeed, Denning argues that computer science is not perceived as – and in some ways, does not act as – a science [Denning, 2005]. For example, Denning cites a study showing that 50% of the published models and hypotheses in computer science go untested [Tichy, 1998].

A number of reasons exist for the current state of affairs. Some reasons include (1) computer science being a young field, (2) much research occurs in industrial labs of for-profit companies, which sometimes do not release internal intellectual property validating the published research, and (3) there is a lack of data on which to validate models and approaches (in formal verification, data are programs and designs, many of which are closed-source). The full set of causes are surely complex and are a combination of economic, cultural, and political reasons.

Regardless of the reasons for why results are not publicized more broadly, the reasons for why they *should* be publicized are not different than the "four-freedoms justification" for why software should be open. For a concrete anecdotal reason, consider the following: A theory of a class of distributed protocols was developed, verified, and subsequently published [Rushby, 1999]. One of us read the published paper but was slightly unclear regarding a few of the details – the theory was only informally presented in the published paper. Fortunately, the original specifications were made publicly available, and we able to examine them. In doing so, we found that three out of four of the fundamental system assumptions (i.e., axioms in the formal theory) not only failed to model the domain of discourse but were in fact inconsistent. We were able to quickly correct the theory – the fundamental insights were correct – and publish the corrections [Pike, 2006].

If the original specifications and proofs had not been available, the inconsistencies would have remained. Nevertheless, merely making them public was also not enough: in this instance, the original paper had been cited two or three dozen times, and it had been reviewed multiple times. None of those citing the paper or reviewing it caught the inconsistencies (indeed, we had cited the paper in earlier work without inspecting the proofs themselves). With open software, there is the presupposition or hope that when it gets used, bugs will be caught and corrected. Perhaps it is the same with formal proofs (we believe we were the first to "use" the published proofs in this particular instance).

De Millo, Lipton, and Perlis contradict this claim [Millo et al., 1979]. Essentially, they argue that unlike pure mathematics, formal proofs about software, even if they are free, will never garner the interest required to bring the value of independent review like for pure mathematics.

> Verifications are long and involved but shallow; that's what's wrong with them. ... Nobody is going to run into a friends' office with a program verification. Nobody is going to sketch a verification on a paper napkin. Nobody is going to buttonhole a colleague into listening to a verification. Nobody is going to read it.

Our anecdotal story above contradicts this claim. More generally, one of the present authors works in industry at a company that does formal verifications, and he is routinely buttonholed into listening to verifications. To be sure, he is not buttonholed into listening to the details of an entire verification but often key parts of a proof or new insights and techniques.

More generally, De Millo, Lipton, and Perlis make a few errors in their conclusions about the utility of open verifications. First, free and open verifications are important even if nobody manually reads them. Automated proof checkers exist today (De Millo et al.'s paper was written in 1979) so that one can check the correctness of a verification without reviewing it manually, but this still requires that the proofs be available. Second, more abstract concepts in computer science (e.g., the distributed protocol verification mentioned above) *are* short and simple enough to be manually reviewed. Third, "chicken-egg" phenomena may exist: nobody reviews free and open verifications, because there are no free and open verifications! Nearly 30 years ago, would anyone have predicted the success of free software? Would we have predicted the thousands of software develops who contribute to open-source software projects?

Finally, just like in free software, open verifications would allow society to share the considerable burden of formal methods. Linux is a full-featured free operating system built mostly by volunteers (indeed, you can buy computers today running Linux from general-market dealers like Wal-Mart). Linux competes favorably with proprietary extremely well-funded operating systems, like Windows. Linux's success is possible due to massively-distributed efforts. Similar efforts in verification can have an analogous effect. Indeed, this is already happening: when a researcher develops a new formal verification tool, one convincing way to demonstrate its effectiveness is to find bugs in free software (like Linux) using it (see, for example, work by Dawson Engler et al. [Engler et al., 2000]).

Let us reemphasize our main point of this section: the topic of free software is an example of a central issue in computer ethics, and the ethical issues of formal methods both inform and are informed by the free software debate. While we have presented specific arguments for why formal verifications should be free, we only wish to convince the reader that the debate over their freedom is important and belongs in computer ethics, too.

6 Conclusion

Why have formal methods *not* entered into the discourse in computer ethics in a central way already? Is it because of the contrast between computers and other things that can harm human beings? For example, there are widely-accepted moral obligations that go along with automobile manufacturing. They have to be recalled if a safety-critical defect is found. Cigarette manufacturers have had multi-million-dollar liability suits levied against them. They are held accountable for a product they knew to be dangerous. To be sure, the causes are not completely ethical; for example, an automobile manufacture has strong economic motivations for building safe vehicles. Still, the difference between the computer industry and other industries appears to be one of kind, not degree.

Why has the public perception of computers been different? Why may a software manufacturer attach a "non-warranty" to software, to which you, the reader, have likely assented upon initially booting a new computer you have purchase? Why have there been no multi-million dollar liability suits against software manufacturers? Are current practices in software engineering the best practices? Does there exist a moral imperative to increase the assurance of software via formal methods, even if there may be no immediate economical gain in doing so?

Recall the developments in bioethics from the late 1960s to the present. The health care system, as a result of an interdisciplinary critique by philosophers, lawyers, and computer scientists, moved from widespread paternalism to a recognition of the autonomy of the patient. A dialogue outside of the computing profession can contribute to raising of public awareness, which itself is necessary for computing professionals to satisfy the imperative to prove safety, where and when it is a justifiable imperative.

We don't take the foregoing considerations to provide an airtight case for the claim that all software ought to be open, or that there is indeed an obligation to use formal methods. Rather, we hope to have shown that considerations of formal methods properly belong to computer ethics. To summarize, formal methods belongs computer ethics for three reasons: first, on the account given of what the purpose and nature of computer ethics are, formal methods belongs there. Second, canonical content of computer ethics is connected to issues related to formal methods. Finally, normative questions ought to be answered within a dialogue that includes the wider community of computer *users*, not just computer producers, and there is evidence that practitioners of normative theory bear some responsibility for ensuring more global deliberation on issues of professional responsibility.

In closing, let us summarize our goals in this paper, which included arguing for the following:

1. Computers, considered as *automated formal systems*, can be made safe in a way which other artifacts cannot.

2. There are open philosophical problems in the applied ethics of formal methods. The computer ethics community must help address these problems.

In furthering these goals, we

1. described formal methods in enough detail for computer ethicists to begin ethics projects about the subject, and

2. showed that formal methods belongs to computer ethics for both general reasons and consideration of important canonical issues within computer ethics.

We opened this article with two quotes from two of the world's most influential computer scientists. The quote from Knuth is slightly tongue-in-cheek, but the point to be drawn from it is that proof can never replace testing a program on real data in the intended environment. Dijksta's quote is meant quite seriously to explain that correctness cannot be proved via testing. Taken together, the quotes delightfully expose the work required to provide an ethics of formal methods.

References

[Baleani et al., 2003] Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A., Peri, M., and Pezzini, S. (2003). Fault-tolerant platforms for automotive safety-critical applications. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 170–177.

[Barwise, 1989] Barwise, J. (1989). Mathematical proofs of computer system correctness. *Notices of the AMS*, 36:844–851.

[Berry and Wing, 1985] Berry, D. M. and Wing, J. M. (1985). Specification and prototyping: Some thoughts on why they are successful. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Vol.2*, pages 117–128. LNCS.

[Charette, 2005] Charette, R. N. (2005). Why software fails. *IEEE Spectrum Online*. Available at `http://www.spectrum.ieee.org/sep05/1685`.

[de Moura et al., 2004] de Moura, L., Owre, S., Ruess, H., Rushby, J., and Shankar, N. (2004). The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland. Springer-Verlag.

[Denning, 2005] Denning, P. J. (2005). Is computer science science? *Communications of the ACM*, 48(4):27–31. Available at `http://cs.gmu.edu/cne/pjd/PUBS/CACMcols/cacmApr05.pdf`.

[Dowie, 1977] Dowie, M. (1977). Pinto madness. *Mother Jones Magazine*. Available at `http://www.motherjones.com/news/feature/1977/09/dowie.html`.

[Elphinstone et al., 2007] Elphinstone, K., Klein, G., Derrin, P., Roscoe, T., and Heiser, G. (2007). Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HOTOS).*

[Emanuel and Emanuel, 1992] Emanuel, E. J. and Emanuel, L. L. (1992). Four models of the physician-patient relationship. *Journal of the American Model Association*, 267(16):2221–2226.

[Engler et al., 2000] Engler, D., Chelf, B., Chou, A., and Hallem, S. (2000). Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation.* USENIX Association.

[Ermann and Shauf, 2003] Ermann, M. D. and Shauf, M. S., editors (2003). *Computer, Ethics, and Society.* Oxford University Press, 3rd edition.

[Fodor, 1990] Fodor, J. A. (1990). The big idea: Can there be a science of mind? *Times Literary Supplement*.

[Halfhill, 1995] Halfhill, T. R. (1995). The truth behind the pentium bug. *BYTE Magazine*.

[Haugeland, 1989] Haugeland, J. (1989). *Artificial Intelligence: The Very Idea*. Bradford Books.

[Haugeland, 1997] Haugeland, J. (1997). *Mind Design II*, chapter What is Mind Design? MIT Press.

[James, 1988] James, H. F. (1988). Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063.

[Jetley et al., 2006] Jetley, R., Iyer, S. P., and Jones, P. L. (2006). A formal methods approach to medical device review. *Computer*, 39(4):61–67.

[Johnson, 2001] Johnson, D. G. (2001). *Computer Ethics*. Prentice Hall, 3rd edition.

[J.R. Burch et al., 1990] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang (1990). Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C. IEEE Computer Society Press.

[Knight, 2002a] Knight, J. C. (2002a). Software challenges in aviation systems. On-line. Available at `groups.inf.ed.ac.uk/safecomp/Download/safecomp2002/knight_safecomp2002.pdf`.

[Knight, 2002b] Knight, J. C. (2002b). Software challenges in aviation systems. In *SAFECOMP*, Lecture Notes in Computer Science, pages 106–112. Springer.

[Lahiri and Seshia, 2004] Lahiri, S. K. and Seshia, S. A. (2004). The UCLID decision procedure. In *CAV'04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 475–478.

[Lutz, 1993] Lutz, R. R. (1993). Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133.

[Millo et al., 1979] Millo, R. A. D., Lipton, R. J., and Perlis, A. J. (1979). Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280.

[Neumann, 1996] Neumann, P. G. (1996). Using formal methods to reduce risks. *Communications of the ACM*, 39(7):114.

[Pike, 2006] Pike, L. (2006). A note on inconsistent axioms in rushby's "systematic formal verification for fault-tolerant time-triggered algorithms". *IEEE Transactions on Software Engineering*, 32(5):347–348. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/time_triggered.html`.

[Pike et al., 2006] Pike, L., Shields, M., and Matthews, J. (2006). A verifying core for a cryptographic language compiler. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 1–10. ACM Press. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/acl2.html`.

[Rapaport, 2007] Rapaport, W. (2007). Can programs be verified? Available at `http://www.cse.buffalo.edu/~rapaport/510/canprogsbeverified.html`.

[Rushby, 1993] Rushby, J. (1993). Formal methods and digital systems validation for airborne systems. Technical Report CR–4551, NASA.

[Rushby, 1999] Rushby, J. (1999). Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660.

[Smith, 1985] Smith, B. C. (1985). The limits of correctness. *SIGCAS Comput. Soc.*, 14,15(1,2,3,4):18–26.

[Stallman, 1985] Stallman, R. M. (2003/1985). *The GNU Manifesto.* Cambridge University Press. Originally published by the Free Software Foundation.

[Tavani, 2007] Tavani, H. T. (2007). *Ethics and Technology: Ethical Issues in an Age of Information and Communication Technology.* Wiley, 2nd edition.

[Tichy, 1998] Tichy, W. F. (1998). Should computer scientists experiment more? *Computer*, 31(5):32–40.

[Wiedijk, 2006] Wiedijk, F., editor (2006). *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science.* Springer.

[Winston and Edelbach, 2006] Winston, M. E. and Edelbach, R. D., editors (2006). *Society, Ethics, and Technology.* Thomson Wadsworth, 3rd edition.