Calculated Trust

Automated Reasoning Tools and the Formal Verification of Software

Lucas Stephen Beeler
lsbeeler@umich.edu

1. Introduction—the Cost of Failure

On the night of January 26, 1991, a Soviet-built SS-1 "Scud" tactical ballistic missile was launched southward from Iraq, into neighboring Saudi Arabia. In what would become the deadliest single combat incident of the first Gulf War, the Scud penetrated the sophisticated Patriot missile defense network that U.S. forces had deployed to protect coalition ground installations from missile attack, and detonated over a barracks and supply facility at Dhahran. 97 soldiers and civilian support personnel were seriously injured; 28 lost their lives [1, 2, 3].

In an earlier war, in an earlier time, hardened underground shelter facilities might have been provided to ground personnel to afford them protection against air and missile attack. But this was the first war of last decade of the twentieth century, and it was believed that Patriot—an integrated, computer-controlled interceptor missile system of unprecedented capability—had rendered bunkers and earthworks obsolete. Patriot, after all, was among the most expensive and sophisticated defensive weapons systems ever deployed [1, 2]. When it was sent to Saudi Arabia in the Autumn of 1990, few doubted its ability to handily down any missile that the Iraqis possessed [2]. Indeed, the only missiles thought to be beyond Patriot's reach were giant, nuclear-tipped, intercontinental ballistic missiles or ICBMs—and these were deployed in appreciable numbers only by the Soviet Union and the United States. Against a third-world adversary like Iraq, Patriot was thought unstoppable.

But the Patriot system had one lurking weakness that would prove deadly: the programmers who coded its control software used a particular floating point number representation to record time values [1, 2, 3]. Over days of continuous operation, this representation gradually caused small errors to accumulate in the values of timing measurements. Ultimately, this error became so great that the missiles could no longer execute their flight programs reliably, rendering them impotent and unable to intercept their targets. This fault did not appear during testing because in no single test instance were the missiles ever left to sit idle, with their computers turned on, for a time long enough for it to emerge [1, 2].

This is not to say that the Patriot system was developed hastily or tested inadequately. Indeed, while other catastrophic examples of software failure might've been the products of a small group of people on a tight budget or only sparsely tested [4], the same could not be said of Patriot. Patriot was a flagship weapons system of the United States defense establishment, and it had undergone extensive evaluation and certification prior to its deployment [1, 2]. This is the ultimate lesson of Dhahran: that even with extensive testing, software remains—more so than almost any other modern engineering product—uniquely capable of spectacular and unpredictable failure. The exact reasons why this is so will be discussed later, but for now it suffices to say that if we are going to trust software to heal us when we are sick, to transport us safely from one place to another, to defend us from our enemies, and to do all of the other things that our modern, technological civilization relies on it to do, then we had better have some strong assurances that our trust isn't misplaced.

This paper is about one particular approach to building software systems of high integrity and reliability, called *automated formal verification*[1] (hereafter also *automated verification*, *formal verification*, or simply *verification*) Automated formal verification aims to use *automated reasoning tools* (hereafter also *automated tools*, *reasoning tools*, *theorem provers* or simply *tools*) to prove that computer programs are correct relative to some formal specification of their behavior. When properly applied, this paradigm of formal verification goes a long way toward enabling us to build software systems that we can trust. Such trust, however, comes at a price. The introduction of formal methods can dramatically complicate the software development process, even to the point of placing unjustifiable burdens on programmers and managers. So where do we draw the line? Which development projects are amenable to automated verification and which ones aren't? In this paper, we seek to answer this question by better understanding it. We discuss the reasons why software is more prone to failure than other engineering products and the mechanisms by which automated verification works to mitigate them. We discuss how the field of automated verification has evolved and some of the major systems, tools, and approaches in use today. Finally, we ask and answer: just what does the state-of-the art in automated verification buy us—and what does it cost?

2. Why is Building Reliable Software Hard?

In the previous section, I claimed that software was more predisposed to failure than perhaps any other modern engi-

(1) There is another methodology that is *automated* and attempts to *verify* the *formal* properties of software—a state-space exploration paradigm called *model checking*. For the purposes of this paper however, we ignore model checking and use the term "automated formal verification" only in the sense defined here.

neering product. This is a serious assertion. So we should ask: is it a reasonable one? And—if it is—why?

Anecdotally, we encounter evidence of software's unique propensity to fail every time our word processor or our email client crashes. We've even become habituated to software failure. We accept having to terminate and restart our email client software several times during our workdays, where we'd never accept car engines requiring intermittent killing and cranking during our commutes, or dishwashers requiring intermittent stopping, unloading and reloading during their wash cycles.

What's more, beyond these anecdotal observations lie hard and objective facts. Articles, papers, and case studies are routinely published on the subject of the "software crisis" [3, 5, 6, 7, 8]. The very existence of this term is indicative, and it defines the present state of affairs in software engineering, where by some estimates more than half of all software projects initiated either don't finish on time, or don't finish within budget, or don't meet their stated design goals [5, 7, 8].

But why is this so? Why do software systems appear harder to build than other complex engineering artifacts? I think the answer lies in software's very nature. Software, we should remember, isn't really a traditional engineering product in any meaningful sense. Indeed, it's a different animal altogether.

Contrast civil engineering with software engineering. Consider a traditional product of civil engineering—a bridge. Let us say that we have a bridge *B*. Now, let us say although *B* is a bridge of wholly new design, that the design of *B* belongs to some established category in the design of bridges, such as cantilever designs or suspension designs, with which we are familiar. Moreover, let us say that we are familiar with the structure and behavior of the basic substrates of bridge construction—braided cables and trellises and concrete. With this knowledge, we have a fairly solid idea of how *B* will behave. This is because a bridge, as structural and functional entity, is a continuous system. The dynamic behavior of its parts are all governed by continuous mathematics like calculus and linear algebra and differential equations. In ordinary circumstances, under the governance of these mathematics, continuous systems behave—well—continuously, whereby relatively small changes in system inputs produce relatively small changes in system outputs. Consider an example: let us say that *B* is not in danger of collapsing until its suspension cables elongate to, say, 800mm from their rest lengths. If loading eight cars onto the span of *B* has caused its suspension cables to elongate to 4mm from their rest lengths, then placing a ninth car on it may cause them to elongate to 4.5 mm, or 5 mm, or even some manifestly non-linear value like 12 mm, but it is not likely to cause *B* to collapse.

No such weak assurance of predictability comforts the software engineer. Unlike the degree of cable elongation in response to its load, or the flow of air over the nacelle of jet engine in response to its groundspeed, or the changing electrical resistance of a wire in response to its temperature, software systems do not behave continuously. Civil engineers, aerospace engineers, and analog electrical engineers all enjoy an assurance of predictability because the cables and nacelles and wires that are the work products of their professions are physical entities governed by the continuous laws of the natural world. But software doesn't belong to the natural world at all. The work products of software engineering aren't even material objects but are instead *directives and programs that are recipes of management and control* [9]. The physical substrate over which software operates is computer hardware: processors, registers, caches and memories. While these substrates may be governed by the continuous mathematics of electromagnetism, the software engineer never sees the continuous nature of them. To him, computer hardware is a nothing more than palette of discrete states [10, 11]. And the job of the software engineer is to create a program of control that, given an arbitrary input state configuration, shepherds the hardware through various intermediate state configurations until the state of the system becomes one that corresponds to an acceptable output, at which point the program halts [9].

Unlike a suspension cable that can be exhaustively loaded until it breaks, the exhaustive testing of discrete systems like software is impossible. This is because of the exponential-combinational nature of large discrete systems. Indeed, the number of possible system states that can be instantiated in modern computer hardware easily dwarfs the number of atoms in the known universe [11]. This, at last, is why the design and construction of reliable computer software is so hard.

3. Motivating Automated Formal Verification

If we can't guarantee the reliable functioning of software through the traditional mode of exhaustive testing, then how can we? Could we attack the problem from a completely different direction—say, by seeking to prove mathematically that software will function as it should? This is the goal of automated formal verification. As mentioned in the previous section, a software program is a recipe of management and control whose job is to direct the evolution of the state of a discrete system from some input state configuration to some output state configuration appropriate to the input. And while we can't exhaustively test a software program, we can seek to prove that it is *correct*. This is to say, we can seek to show that, given any valid input configuration, our recipe of control will lead us inexorably and inescapably to its corresponding valid output configuration. To achieve this goal, the enterprise of automated formal verification relies on the established rules of symbolic logic [12].

Let's review some of the properties of symbolic logic by considering the following logical equation:

$$\begin{array}{l} \forall x,\ [\mathsf{DOG}(x) = \mathrm{TRUE}] \Rightarrow [\mathsf{BROWN}(x) = \mathrm{TRUE}] \\ \quad\ \mathsf{DOG}(Bud) = \mathrm{TRUE} \\ \hline \therefore\quad \mathsf{BROWN}(Bud) = \mathrm{TRUE} \end{array} \qquad \text{(eqn. 1)}$$

This is, of course, nothing more than an ornate and stylized way of expressing a venerable truism of grade-school logic puzzles: *if all dogs are brown and Bud is a dog, then Bud must be brown*. Now note that the fact that Bud is brown, given our two premises, is absolutely true, correct and incontrovertible. The reason why this statement must be so is that we have reached our conclusion from our premises by applying one of the

established rules of deductive logic [12]. In this particular case, the rule we've applied is *universal instantiation* [12], which is just a fancy name for the self-evident notion that if all things belonging to a certain class or category (like dogs) exhibit a certain property (like being brown) then any single, individual object belonging to that category (like Bud) must also have that property.

From the standpoint of the automated verification of computer programs, two details from this example are critical. The first critical detail is that there exist established rules of formal logic. Moreover, these rules are finite in number and have been well understood for centuries [12]. The second is that given a universe of premises, we can use the rules of formal logic to derive new conclusions which are absolutely correct.

Indeed, it is precisely these notions of finiteness and absoluteness that allow us to *automate* the process of formal reasoning and to use it to verify the correctness of computer programs. With automated formal verification, if we want to develop an application program of high integrity and reliability, we can do so by making some changes to our development process and introducing one critical new tool [13, 14].

Let's consider the changes to the development process first. Most current development processes begin with a requirements capture and elaboration stage [3], where software developers talk to project stakeholders and decide on a finite set of requirements to be met by the system under development. With the introduction of automated verification, the fundamental nature of the requirements capture and elaboration phase remains the same, but the process becomes far more rigorous and precise. Instead of generating at the end of the phase a textual requirements specification, we generate an absolute and unambiguous encoding of the system requirements specification, expressed not in a natural language like English or Hindi, but as a collection of inter-related logic equations. Above, we saw that we could encode unambiguous natural language statements like "if all dogs are brown" and "Bud is a dog" as logic equations. When developing a software product that will undergo formal verification, we do exactly the same thing to our requirements, encoding system specifications like "if the valuation is greater than 67% of client's net worth, then the transaction will not be executed immediately, but will queued for risk review" (as we might find in a financial trading application) as logic equations.

After we finish expressing our requirements in logic, we implement them by writing our program code, just as we would in a traditional development process. With the introduction of formal verification however, what happens after implementation changes dramatically. Instead of a rigorous testing phase, we replace testing—either wholly or partly—with a *verification phase*. The verification phase is made possible by a new development tool, the automated reasoning system, and it proceeds as follows.

First, observe that we must have implemented our program code in some language, and that this language must have precise semantics (within limits)—otherwise it would be impossible to translate programs written in the language into machine instructions. Put another way, the constructs of our programming language—all of the `if`s and `while`s and `for`s and arithmetic operators and semicolons—all of these constructs *do* something, and what they do can be logically defined.

We now have 3 distinct logical objects:

- a specification of what our system must do to be "correct"
- a primer describing the semantics of the language in which our program is implemented
- our program code itself

With these in hand, the task of automated verification can begin.

We carry out the task with an automated reasoning tool, a software program that is, at its essence, a mechanical mathematician [13, 14]. Products of logic, philosophy, and artificial intelligence, automated reasoning tools can—with a bit of human help and "books" of past results [13]—construct proofs of mathematical and logical statements (we should qualify this by saying *much of the time, but not always* and *subject to certain constraints and assumptions*, but we'll address these caveats later on). We verify our program code by asking the automated reasoning system to prove the statement: "given the semantics of my programming language and the program that I've written, does my program do exactly what its requirements say it should do, and no more."

Clearly, if the aforementioned statement can be proven true, then our program must be bulletproof: correct, perfect, and free from error. This is the promise of formal verification—provably perfect code. Alas, this isn't quite the reality. To be sure, Gödel showed more than a half-century ago that there exist statements which are true, but can never be proven so [17]. What's more, there are true statements which can be proven true, but not within the remaining lifetime of the universe [11]. And these are just the philosophical problems. Beyond lie many practical issues that, if not addressed carefully, may cause an otherwise well-structured verification endeavor to collapse. Among these are: how do we map our system requirements from the subtle and ambiguous domain of human language into the absolute universe of symbolic logic? And, how do we give meaning to the constructs of our programming language?

The remainder of this paper seeks to address these challenges, but first, we need to examine more closely the new pieces that formal verification adds to the software development puzzle. While we touched on some of them in this section, but we need to characterize them precisely, since we'll use them to frame the rest of our discussion.

4. The Pieces of the Puzzle

Introducing formal verification changes the software development puzzle dramatically. It adds new pieces, and forces us to reinterpret the roles traditional pieces play. So that we have a point of reference as we move forward, let's briefly review the traditional pieces and the interactions between them [3, 16]. Figure 1 below presents an overview of the traditional process. The view is abstract and stylized; it doesn't attempt to corre-

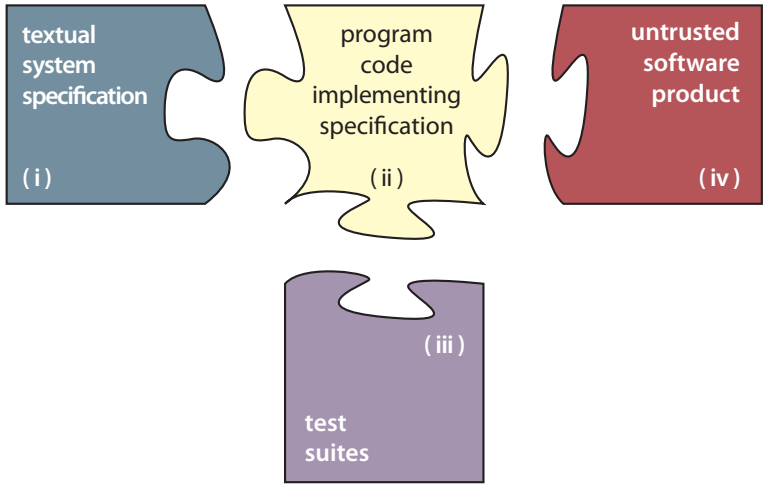

Figure 1. In this stylized view of a traditional software development process, we first capture stakeholders' requirements in a system specification (i). Then, we use this specification to develop program source code (ii) as well as a suite of test programs (iii) that check our program's behavior against the specification on a small number of input instances. Because we can only test over a relatively small number instances, the output of the traditional software development process is an *untrusted* software product (iv).

spond to any defined process model, like the IEEE Waterfall Model [15] or the Rational Unified Process [16]. We've taken this abstract view to capture the essence of the process, independent of any particular model.

Our traditional process begins with a requirements elaboration and capture phase [3, 15, 16], where project stakeholders enumerate exactly what the product under development should do and—to a lesser extent—how it should do what it does. The output of this phase is a textual requirements specification, written in a natural language like English. In Figure 1, this corresponds to (i). With this specification in hand, we move to write our program source code (ii). Our specification also allows us to develop test suites, ancillary programs that check whether our code correctly implements certain instances of our requirements. These test suites are coded to feed our program a number of common and boundary inputs and check that its output is in line with its specification. Testing corresponds to (iii). But testing is of limited power to verify correctness, as it only assures us that our program works over the small number of cases checked. Because of this, the output of the traditional process is an *untrusted* software product. We see this in (iv).

To support formal verification, the development process must be altered in substantial ways. Figure 2 presents a high-level overview of the new process. As in the traditional process, we begin with a requirements elaboration and capture phase that outputs a textual specification, as seen in (i). From this specification, we begin writing source code (iii), just as

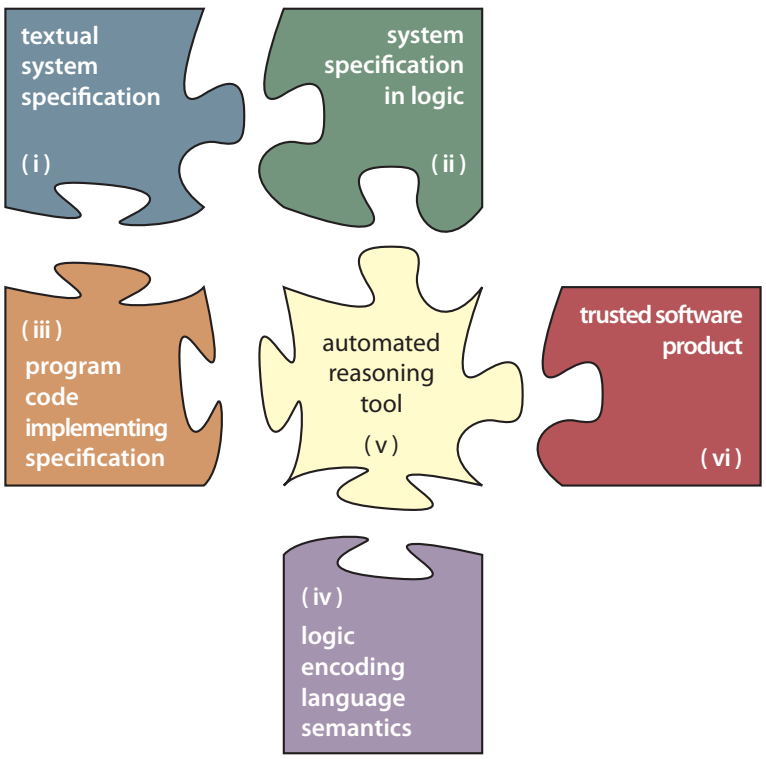

Figure 2. A stylized view of a development process supporting formal verification. Verification alters the nature of the process, reshaping it around a new entity—the automated reasoning tool (v). Applying an automated reasoning tool requires us to specify our system requirements absolutely and unambiguously in whatever encoding of symbolic logic the tool uses (ii). It also requires that we develop a logic primer (iv) on the semantics of our programming language, so that the tool can make sense of our code (iii). If we accomplish both of these tasks, we ask the tool to prove our program code correct relative to its specification. If the tool succeeds, the output of the process is a *trusted* software product that is free from error (vi).

in the traditional process. Piece (ii) however, is new. Here, we use our textual specification as a starting point for expressing our system requirements in formal logic. Just as we converted a series of English-language assertions into logic equation 1, so we work to express our requirements in whatever encoding of symbolic logic our reasoning tool prefers. But just because we have our specification encoded in logic doesn't mean we can ask our tool to verify our code against it—at least, not yet. We also need a logic primer that teaches our tool what the constructs of our programming language mean. Without this primer, our automated reasoning tool would be unable to interpret the meaning of the strings and symbols in our source code. We develop this primer in (iv).

Once we have in hand (ii), (iii), and (iv), we can feed them as input to our automated tool. The tool interprets our source code using our language primer, and tries to prove that the behavior effected by our code is the behavior we've specified as correct. We see this in (v). If the tool succeeds, we hold a positive guarantee that our program functions as specified. With it, we can transform the output of the development process, from an untrusted software product to a trusted one (vi).

5. Automated Reasoning Tools

In an essential sense, the vision of formal verification articulated above still belongs more to the realm of fiction than of fact. All of the pieces exist, but aren't yet fully formed. We are still a long way from a time when automated intelligences can certify complex products at the touch of a button [25]. But the goal of a tool that can automatically sort perfect programs from imperfect ones remains compelling—and a goal that has many researchers in hot pursuit. Speaking at the IFIP's 2005 Verification Grand Challenge Workshop, J. Strother Moore predicted that someday there would emerge machine intelligences that "[autonomously] *understand* why our software is correct." This development, Moore continued, would enable tools capable even of *discovering*—on their own—the reasons for a software product's correctness. On that day, Moore noted, we'll have the kind of verification tools everyone wants. But as Moore soon made clear, we are nowhere near these achievements today [25]. Because formal verification remains an incompletely formed endeavor, we pay special attention to its present limitations. And nowhere are these limitations more visible or profound than in the reasoning tools themselves.

5.1 Limitations of Automated Reasoning Tools

From a purely philosophical perspective, the structure of the universe doesn't seem to encourage the development of computer programs that can prove other computer programs correct. After all, proving *total correctness* in the general case would require us to be able to determine whether or not an arbitrary computer program halts—and this is known to be impossible [11]. Even if we forego the goal of proving total correctness and seek instead to prove only *partial correctness*—that when a program does halt, it halts with the correct output—we still find ourselves in something of a quandary. This is because the general notion of a formal proof fits within the certificate definition of **NP** [24, 11]. Given a string x and a language $L = \{w \mid w \text{ encodes a true theorem}\}$, the act of producing a formal proof is equivalent to deciding whether $x \in L$. In this formulation, a proof y serves as a *certificate* witnessing the membership of string x in language L. If we hope to read our proof in anything less than eons, then it's of a length polynomial in the length of x. This is precisely the certificate definition **NP** [24]. And while the nature of **NP** tells us that we can check the validity of y quickly, everything we know about **NP** also suggests that discovering y in the first place is probably—in the general case—intractable. Thus the general software verification problem is either undecidable or intractable. What ever are we to do?

The answer is to set aside the goal of any general solution and to adopt instead the more modest goal of developing a set of heuristics that can prove theorems about computer programs *in most cases*, *under interactive human direction*. This problem is not only tractable, it's solved by several automated reasoning tools everyday [13, 14].

5.2 Classifying Automated Reasoning Tools

Much like cars or microwave ovens, automated reasoning tools come in differing shapes and sizes, each having its own strengths and weaknesses. For example, a 3500 watt commercial microwave probably wouldn't be at home in a college dorm room, but it might be just the ticket for a busy university dining hall a block away. In the same way, when a project team or organization considers adding automated verification to their development process, the question isn't "what's the best automated reasoning tool?" but is instead "what's the best automated reasoning tool for what we want to do?" Of course, we can only answer this question intelligently once we understand the different kinds of tools available.

The first line of differentiation traced between tools draws a separation between *implicit* tools and *explicit* ones. An implicit reasoning tool doesn't exist as a standalone application, but instead exists only as component in some larger verification environment. A contemporary example of an implicit automated reasoning tool is the *Boogie* theorem-proving component in the Spec# formal programming system developed by Microsoft Research [26]. Among historical examples of implicit tools, foremost is the reasoning engine of the *Stanford Pascal Verifier* (or *SPV*) [27] system of the 70s and 80s.

The key characteristic of all implicit tools is that they are not built for general purpose artificial reasoning. By design, they can prove theorems only for the purpose of software verification, and even then, they can do so only in the context of their specific language and environment. For example, Microsoft's *Boogie* cannot prove that the square root of two is an irrational number. Nor can it prove that a C++ method implementation properly maintains the representational invariants of its class. This is because *Boogie* provides no facility to specify C++ semantics or the properties of the real and rational numbers. *Boogie* is limited to the formal world inside the Spec# environment [26]. But here, *Boogie* is a champion. It can quickly reason over invariants and specifications embedded in Spec#-language[2] code, and can usually verify the correctness of program segments without human interaction [26]. What's more, Microsoft Research has seamlessly and elegantly integrated *Boo-*

gie into their Spec# development infrastructure, which itself plugs neatly into their Visual Studio IDE [26]. If you're already using C# and the Microsoft toolchain, then the transition to Spec# and *Boogie* is all but effortless—at least compared to the cost in effort of adopting other formal tools. More generally, these qualities of ease of use and relatively seamless integration into existing workflows are typical of implicit systems, and are among their most attractive features. But remember, all of this convenience comes at a price. By their very definition, implicit systems are specialists. They function only within the context of some specific environment, and they usually understand only the semantics of some single language for which they were expressly designed. In the case of *Boogie*, this language is Spec# [26]. In the case of the SPV engine, it was Pascal [27].

Unlike implicit tools, explicit tools are generalists. An explicit automated reasoning tool is a standalone software package that can be employed to prove arbitrary mathematical theorems. Provided we are willing to take the time to encode the relevant specification and context precisely, a good explicit automated reasoning tool can prove theorems over almost any unambiguous formal object, from software programs, to hardware designs, to the integers [13, 14, 21].

What's more, this generality is at the heart of the key advantage that explicit systems hold over implicit ones, namely, their flexibility. And this flexibility is no trifle. Indeed, in the kinds of real-world development scenarios where formal methods are likely to be considered, it may well be flexibility that makes the introduction of formal methods possible from a cost-benefit standpoint. The human capital investment associated with introducing formal verification—particularly with explicit systems—is extremely high [13, 14, 21, 28]. For example, the introducing an explicit tool-based formal process may require that some of the most capable staff members—those identified as having the greatest aptitude for formal work, such as senior developers or architects—be re-tasked away from day-to-day development duties to become "theorem prover gurus," who spend most of their time and effort shepherding the operation of the tool [21, 29]. Obviously, this is a steep price to ask any organization to pay. But if a reasoning tool is sufficiently flexible, then it may be possible to amortize the costs of training and maintaining a cohort of reasoning tool experts by using the tool and the staff trained to operate it across several different project arenas. Formal methods are often employed in safety-critical development scenarios in aerospace, defense, and automotive controls [19, 30, 31, 32]. In these industries, computational products are employed in specialized, high-performance applications for which vanilla, off-the-shelf hardware may not be suitable. Indeed, hardware-software co-design is common in high performance applications, where software development often proceeds in tandem with the logic design of custom ASICs or FPGAs [33]. What's more, safety-critical systems often must be developed to meet layered, interconnected, and complex sets of requirements. The development of these systems may be governed not only by the basic functional requirements of the project client, but also by criteria imposed indirectly on the project by military standards boards or government regulatory agencies [32]. In such situations, being able to validate the mutual consistency of such diverse requirements specifications is essential—especially if formal verification is to be employed. After all, it doesn't make much sense to try to verify a software or hardware design against a formal requirements specification that is itself contradictory.

With the flexibility offered by good explicit reasoning tools, we may be able to satisfy all of these formal needs—software verification, hardware verification, and specification consistency checking—with one tool [34, 35, 36, 37]. While we would still be required to either re-task existing staff resources or hire new ones to maintain a corpus of formal methods know-how within our organization, by *multi-targeting* this expertise we could amortize our investment across various project arenas. Figure 3 on the following page illustrates this multi-targeting paradigm.

In figure 3, we imagine that our organization has deployed an Isabelle/HOL reasoning engine [14], an explicit tool, and has cultivated a dedicated support team to operate it. Our organization is developing a safety-critical aerospace computing system that requires the development of custom hardware and software. We're writing our software in a constrained dialect of C++ that is amenable to formal verification [36, 37], and we're expressing the logic designs for our custom ASICs in the VHDL hardware description language [35]. Because the specification is complex and has to meet sophisticated regulatory requirements, we've developed our system specification as a high-level architectural model and have had the powers-that-be sign off on it. Our architectural model is specified in the industry standard Unified Modeling Language (or *UML*) [38]. While UML models themselves have semantics too vague for formal work [34], we've used the OCL (or *Object Constraint Language*) extension to UML [39] to encode much of our specification as representational invariants over classes as well as pre-conditions and post-conditions attached to method entry and exit points. Our formal methods support team then develops *formal glue* or collections of logic definitions, semantic encodings, translators and scripts that are used to map external formal entities into the domain of the Isabelle/HOL engine [14]. Glue is developed to map C++ $\Rightarrow$ HOL, VHDL $\Rightarrow$ HOL, and OCL $\Rightarrow$ HOL. Our formal support team works closely with the system architects, software developers, and logic designers to get each glue kit right, amortizing their expertise across domains. In closing our example, we note that all of the mappings described—C++ $\Rightarrow$ HOL [36, 37], VHDL $\Rightarrow$ HOL [35], and OCL $\Rightarrow$ HOL [34]—exist today (though the various C++mappings encode only a constrained subset of full C++ semantics [36, 37]).

The multi-targeting of explicit tools across project arenas—made possible by the generality and flexibility of these systems—goes a long way toward mitigating the high human capital costs of deploying them. But "explicit automated reasoning tool" isn't a monolithic categorization. In the next section, we'll

(2) Spec# is a strict superset of C# augmented with syntactic facilities for specifying invariants and properties inline in source code. See [26] for details.

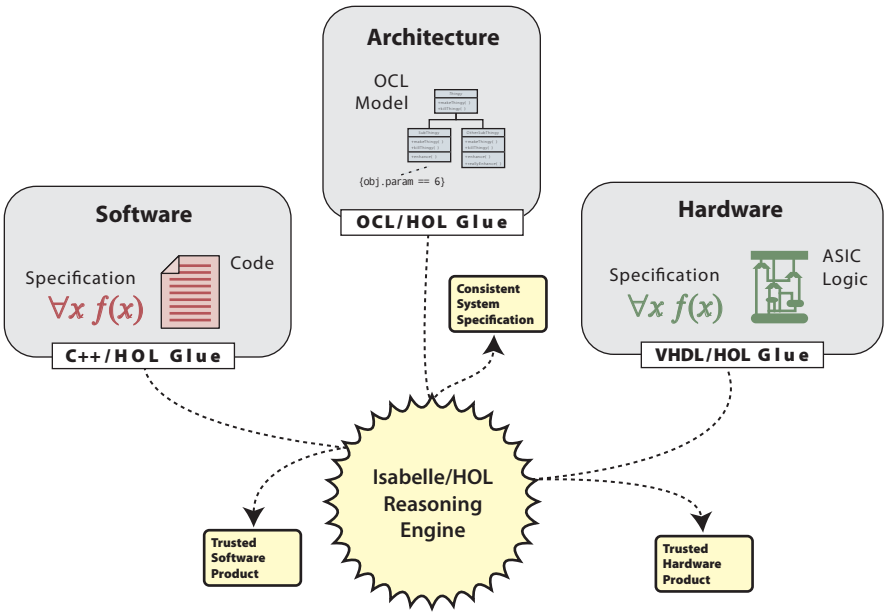

Figure 3. By *multi-targeting* the expertise of a deficated formal methods support team across different project arenas, it may be possible to amortize the sizable human capital investment required of an organization to develop in-house formal expertise. Here, we imagine a situation in which a single, dedicated formal support team develops various *logic glues* in support of activities across the spectrum of a large hardware-software co-development project, including hardware verification [35], software verification [36, 37], and requirements specification consistency checking [34, 39].

explore several points of similarity and difference between the major families of explicit tools in use today.

5.3 The Big Players

In writing this paper, I surveyed the published literature on automated formal verification and the reasoning tools that make it possible. My survey was extensive, but it didn't take more than a few database queries to see plainly that the automated formal verification communities in academia and industry have largely coalesced around two distinct families of automated reasoning tools: the *LCF family* [40], whose offspring include HOL90, HOL Light, Isabelle/HOL and Nuprl [40], and the *Boyer-Moore family*, whose chief extant descendent is ACL2 [13, 41]. This is not to say that these two families alone comprise every good explicit automated reasoning tool ever developed, or about which a peer-reviewed journal article was ever written. Indeed, Stanford's PVS tool [42, 43, 44] was the subject of considerable academic and industrial interest throughout the 1990s [28, 31, 42], and active development work continues on it today [43, 44]. But even a cursory browse through the ACM and IEEE digital libraries will reveal that PVS has never achieved the level of interest accorded to the contemporary LCF descendents or to ACL2, at least as measured by numbers of published articles, papers, case studies, and conference talks. In this subsection, we'll compare these two tool families. We'll begin by examining their similarities, then catalog their differences.

The most visible similarity linking the LCF family and Boyer-Moore family is their popularity. Both have seen major industrial applications—notably in the area of hardware verification. HOL Light is in regular use at Intel [45], while at AMD, the microcode of all new floating-point designs is checked against the IEEE 754 specification with ACL2 [13, 21, 29, 45]. Both systems are also the subjects of well-written, paced tutorial texts issued by major publishing houses. Springer-Verlag commissioned *Isabelle/HOL: A Proof Assistant for Higher Order Logic* [14] in 1994 and Kluwer Academic issued the ACL2 bible *Computer Aided Reasoning: An Approach* in 2000 [13]. Such ready availability of high quality, professionally edited and issued documentation is unusual in a field as relatively arcane as automated reasoning tools. As a point of comparison, the PVS documentation web page on the Stanford Research Institute site begins with the disclaimer, "These manuals are largely out of date…" [44].

Another point of similarity between the two principal families comes in their histories.

The LCF family of tools began life at Stanford under the direction of Robin Milner in 1972, as *LCF—a Logic for Computable Functions*. A year later, Milner returned to his native Britain to take up a post-doctoral fellowship at the University of Edinburgh. There he met Ph.D. student Michael J.C. Gordon. Gordon joined a working group led by Milner whose aim was to correct several serious deficiencies present in the original formulation of LCF, then called *Stanford LCF* [40]. The result of their effort was an incremental evolution of Stanford LCF that became known as *Edinburgh LCF*. In 1981, Gordon moved to Cambridge, where he set out to make Edinburgh LCF a practical tool for hardware verification. The fruit of Gordon's effort was *HOL* or *a Higher Order Logic*, first released in 1988 as HOL88. The application of HOL88 to protocol verification at AT&T Bell Laboratories yielded several refinements, bundled and released two years later as HOL90. A variant of HOL90 aimed at resource-constrained environments—specifically PCs—was released as HOL Light. At Cambridge with Gordon during the early 80s was a young American post-doctoral fellow, Lawrence Paulson. Paulson evolved LCF independently throughout the 1980s and later merged his changes into the HOL90 source tree to create Isabelle/HOL, a widely used HOL variant with superior support for first order logic and enhanced automation [40].

Curiously, the University of Edinburgh plays a central role in the history of the Boyer-Moore tool family as well: the two progenitors met there in 1971, when Moore was Ph.D. stu-

dent and Boyer was a post-doctoral fellow. The original Boyer-Moore system was demonstrated in 1973 [41]. Soon after, Boyer and Moore—both Americans—left Britain and returned home. They settled in Austin, where Boyer had taken a faculty position at the University of Texas. By 1980, the original Boyer-Moore system had evolved into Thm [41]. In 1983, Boyer, Moore and others founded Computational Logic, Inc. (hereafter CLI) to commercialize Thm [41, 46]. CLI was a closely held company headquartered in Austin whose ownership was divided among its principals and the Institute for Computing Science and Computer Applications of the University of Texas [13, 46]. CLI's biggest clients were various agencies of the U.S. government, for whom CLI performed contract research. CLI also provided consulting services to the government and several Fortune 500 companies, to establish and support installations of the various CLI reasoning tools [13, 46]. CLI consultants, for example, worked hands-on with AMD engineers to develop AMD's in-house ACL2 expertise [13, 21, 29, 46]. By 1986, CLI had evolved Thm into Nqthm, and by 1989, Nqthm had become ACL2—a wordplay on what Boyer and Moore had initially regarded their system: *a Computational Logic for a Common Lisp*. CLI shuttered its doors in 1997, but work on ACL2 continues at the University of Texas, where both Boyer and Moore hold faculty positions [21].

Transatlantic relationships, the University of Edinburgh, and good documentation, however, are where the similarities between the modern LCF derivatives and ACL2 end. The differences between the two tool families, summarized by category below, are substantial.

Basis Logic. The mechanized logic of ACL2 is strictly a first order logic, whereas the current LCF derivatives all use higher order logics. Does this mean that ACL2 is more limited than contemporary members of the LCF family? Not necessarily. On the one hand, any good book on mathematical logic will tell you that higher-order logics are strict extensions of first order logic, and that there are concepts directly expressible in higher-order logics that are not expressible in first order logic [47]. On the other hand, this fact may have little relevance as regards encoding requirements specifications and language semantics for software verification. In a 1997 article published in the IEEE's *Transactions on Software Engineering*, William Young compared the logic of ACL2 to the logic of contemporary higher-order systems, writing "Regarding higher order features...they often are elegant, but seldom necessary" [28]. Young continues, remarking that, in his experience, higher order features are no more than a "notational convenience" [28]. He even offers concrete suggestions on how to translate higher-order specifications into the more restrictive logical framework of ACL2, noting, for example that ACL2's first order approach to quantification can be overcome in many cases by "defining a separate quantified notion rather than using a quantifier in place" [28]. J. Strother Moore and his colleagues echo this notion more generally in their introduction to [13], where they note that in almost all cases where intuition suggests higher order constructs, "some formalizable [in ACL2] concept can be substituted without weakening the meaning of the statement."

Automation. If we're going to deploy an *automated* reasoning tool, then certainly we desire that it operate relatively autonomously—otherwise we might just as well be constructing our proofs by hand. Alas, all major explicit automated reasoning tools in use today are decidedly limited in their autonomy, usually requiring considerable interaction with a human operator—the so called "interactive user"—to prove a theorem successfully [13, 14, 28]. The exact role of interactive users in the proof process varies from tool to tool, but usually comprises such things as declaring subgoals to pursue and suggesting proof strategies [13, 14]. Interestingly, automation is an area in which the first-order internal logic of ACL2 affords advantages over systems employing higher-order logics, such as the LCF derivatives [28]. In his comparison between first-order and higher-order approaches, Young writes "Simply by virtue of its richness, higher order logic can present challenges to automated proof discovery tools that are not encountered for first order logic...use of a more restrictive logic means that more effort sometimes must be invested in the specification, but is often repaid with increased automation in the proof" [28]. In support of his assertion, Young recounts the results of some anecdotal tests he ran, noting that "for the one lemma... ACL2 running on a slower machine dispatched the lemma totally automatically in less than one tenth the time [compared to a higher-order system]" [28].

Encoding and Control Language. All explicit reasoning tools require some encoding language in which users can express the logical statements, computational procedures, and other abstractions over which reasoning is performed and theorems proved. This encoding language may also be used to support interactive control, defining constructs to start and stop proof attempts, to view intermediate results, and to load collections of definitions or previous results into tool memory from disk. ACL2 uses a dialect of Lisp for this purpose [13], while all members of the LCF family use Standard ML [14, 40, 48, 49]. The ML programming language is tied into all the LCF-derived tools in an essential way. ML is an impure functional programming language (it allows side effects) that enjoys—at least as far as functional languages go—a surprisingly high level of industrial adoption [49]. Like the original Stanford LCF system, the ML programming language was the brainchild of Robin Milner [40], formulated expressly for the purpose of logic encoding and control in the first LCF systems. Indeed, a residue of its intended purpose is still seen in the name *ML* today: *ML* is an acronym for *metalanguage* [48]. Knowing this link between the ML language and the LCF family of higher order logic reasoning tools, it is not surprising that ML is renowned for its unique, polymorphic type system making extensive use of type inference [40, 49]. After all, "higher order" logic can be viewed as really nothing more than first-order logic augmented with type theory [47]. What's more, the strong typing mechanism present in ML and LCF-derived reasoning tools stands in sharp contrast to the typing support—or lack thereof—provided by ACL2 [13]. Like most variants of Lisp, the Lisp dialect used to drive ACL2 is quite weakly typed [13, 28].

Expressiveness. Even ACL2's creators acknowledge that the higher-order logics and rich type systems supported by con-

temporary LCF descendents make these tools more "expressive" than ACL2 [13, 28]. Here, the *expressiveness* of a tool measures the ease with which external notions of interest—like language semantics and correctness specifications—may be captured formally in the tool's encoding language [28]. But there is considerable debate about whether the added expressiveness enjoyed by LCF-derived systems is a blessing or a curse [13, 14, 28]. Indeed, we saw some of this debate in our discussion of automation above. But while the more limited logic of ACL2 enables greater automation, there does seem to be one arena in which the greater expressiveness of the LCF descendants affords them a distinct advantage—glue development. Recall that glue comprises translators, scripts, and pre-defined logic encodings that are used to translate unambiguous external objects—like computer programs and hardware designs—into the formal world inside the reasoning tool [13, 14, 34, 35, 36, 37]. As the ML encoding language used by the LCF family is more expressive than is the dialect of Lisp used by ACL2, ML enables more external concepts to be encoded directly. This in turn makes the development of glue tools to automatically encode external objects easier. Indeed, I found in my survey that most of the seminal achievements of glue development—particularly in recent years—have come as glue supporting LCF-derived tools [34, 36, 37]. Consider, for example [36], the 2006 formalization of most of the C++ object model—including its dubious multiple inheritance mechanism—by Wasserab, Nipkow, Snelting, and Tip, published in the OOPSLA 2006 proceedings. Their key glue was an Eclipse plug-in that translated source code from a restricted subset of C++ (which they called "Core-C++") into Standard ML for verification in Isabelle/HOL [36] (this in effect furnished axiomatic semantics for most of the C++ object model). Likewise, this year's formalization of the UML Object Constraint Language (OCL) relied heavily on glue translating OCL constraints into ML expressions [34]. Does this mean that it's impossible to encode C++ or OCL semantics as glue that translates C++ code or OCL constraints into the first-order logic of ACL2? Of course not. This is only to say that it is probably easier to accomplish these mappings when the target logic is higher order [31].

6. Language Semantics

We closed the last section with a discussion about the relative ease—or difficulty—of developing glue, which for our purposes comprises tools that translate source code written in a real programming language into symbolic logic, expressed in some encoding language, that captures what the code does or means. In the literature, glue takes various forms and goes by various names, from the dowdy *translator* of [36] to the even rather romantic *logic compiler* of [37]. In both of these examples, an Isabelle/HOL reasoning tool is used, and glue takes the form of a utility program that reads in C++ source code, interprets its statements, and generates as output an ML language logic program encoding the meaning of those C++ statements in their ensemble [36, 37]. Before we can develop glue to automate the process of assigning meaning however, we ourselves must be very clear about what exactly the statements in our program code mean.

There are three traditional approaches to assigning meaning to program code, and we describe them briefly here. These descriptions are inspired by [50]:

Operational Semantics, where the meaning of a piece of code is how it transforms the state of an abstract machine as it runs to completion.

Denotational Semantics, where the meaning of a piece of code is whatever that code can be mapped to in some domain of meanings we've chosen. This approach requires that we define valid interpretation functions to map language constructs into our domain of meanings.

Axiomatic Semantics, where the meaning of a piece of code is the ensemble of properties we can prove about it.

For automated verification, we usually employ axiomatic semantics—though having acess to journal articles or language manuals that describe language constructs rigorously through either of the other two methods can be very useful in formulating the a set of axiomatic semantics to use in preparing for verification.

Axiomatic semantics had its beginnings in two seminal works of the 1960s: Robert Floyd's paper "Assigning Meaning to Programs" [51], presented at the American Mathematical Society's Applied Mathematics Symposium 1967 and published in its proceedings, and C. A. R. Hoare's "An Axiomatic Basis for Computer Programming" [52], first printed in the October, 1969 issue of *Communications of the ACM*. The aim of both papers was identical: to introduce mechanisms by which programming language constructs could be absolutely formalized, such that it would become possible to discern the meaning of entire blocks of code by deductive reasoning over its constituent statements alone. Though Floyd's work was presented first, he confined his presentation to flowcharts of algorithms, and kept his conclusions abstract, while Hoare got into the concrete nitty gritty of applying his analyses to pseudocode fragments whose syntax strikingly resembled that of the dominant imperative languages of the day—FORTRAN and ALGOL60 [51, 52]. Perhaps for these reasons, we now know the fruits of Floyd and Hoare's work collectively by the term *Hoare Logic* [50].

Hoare Logic is the dominant paradigm for formalizing language semantics for automated verification [32, 35, 36, 37, 50]. What's more, the basics of Hoare Logic are relatively easy to communicate to anyone who has done some programming and understands deductive logic—which is more-or-less everyone with an undergraduate degree in computer science or mathematics. The principal construct of Hoare Logic is the *Hoare Triple*, a statement of the form $\{P\}$ `St` $\{Q\}$, where P is a logical assertion called a *precondition*, Q is a logical assertion called a *postcondition*, and `St` is an executable language statement [51, 52, 53]. A Hoare Triple is *true* or *valid* if P implies Q given the successful execution of `St`. This is to say, if P is true before we execute `St`, then on executing `St`, Q becomes true. The semantics of language constructs can then be encoded in a set of Hoare Triples defined to be true, given our understanding of how the language works. These then become the *axioms* of the language, and the language itself is said to have been *axiomatized*. To maximize flexibility, we usually seek to encode Hoare

Triples that are true given the weakest possible preconditions and that guarantee the strongest possible postconditons [52]. We may also encode more sophisticated axioms not as simple triples but as rules of inference, such as below:

$$\frac{P\,\{\,\mathtt{St_1}\,\}\,Q \qquad Q\,\{\,\mathtt{St_2}\,\}\,R}{\therefore\quad P\,\{\,\mathtt{St_1\ ;\ St_2}\,\}\,R} \qquad \text{(eqn. 2)}$$

This rule, of course, simply says that if we know that when we execute statement $\mathtt{St_1}$ with P true, then Q becomes true, and if we also know that when we execute statement $\mathtt{St_2}$ when Q is true, then R becomes true, then it must also be true that when we execute statements $\mathtt{St_1}$ and $\mathtt{St_2}$ sequentially when P is true, then R becomes true when execution completes. This is nothing more than the imperative programming languages version of the *transitivity of logical implication*, namely that $(P \rightarrow Q) \wedge (Q \rightarrow R) \rightarrow (P \rightarrow R)$.

You may have noticed the similarity between eqn. 2 above and eqn. 1 on page 2. This is natural, and is a consequence of the fact that Hoare Logic's *raison d'être* is to map the constructs of a programming language into a consistent logical calculus, to which we can apply all of the timeless rules of deductive reasoning—even those that we learned as far back as grade school—and hopefully even apply them automatically. But this is where the similarities between Hoare Logic and grade-school logic puzzles end. Developing a consistent Hoare Logic for even a restricted subset of a real programming language requires colossal intellectual investment. Consider, for example, the Hoare-style rule of inference shown in figure 4. This rule axiomatizes the method call mechanism for a restricted subset of Java and is excerpted from [53].

What's more, there are language constructs that are still gravely difficult to axiomatize. Are these only arcane constructs that no one uses? Not at all. Side effects, static initialization, and pointers (and the descendents of pointers—references, object handles, or any construct that enables objects to be aliased) still present serious problems in the general case, and even axiomatizing some significant subset of cases for a real programming language usually merits publication in a peer-reviewed journal [36, 37, 53].

It is for precisely this reason that figure 2 on page 4 appears as it does. When automated verification is introduced into the development process, the whole of the process comes to revolve around it. And nowhere is this more true than as regards language semantics. In a traditional development process, as shown in figure 1 on page 4, source code itself is the limiting object—a software product's behavior and the requirements governing it are ultimately constrained by what can be *implemented*. But formal verification changes this. We become constrained not only by what can be implemented, but also by what can be *verified*.

Imagine a project team tasked with the development of some software product that must be formally verified. Before that team can ever write a single line of code, they have to decide *what language constructs can we realistically develop good axiomatizations for?* And if pointers and statically-initialized objects aren't among them, then these language features simply cannot be used. Of course, restricting ourselves to some limited, verifiable subset of our implementation language clearly makes the language less expressive. This may well mean that certain problems become harder to solve.

Indeed, the idea that automated verification imposes constraints on the development process that may be unbearable—even if they enable us to obtain trusted output products—is a recurring one in the debate over formal verification [50]. We'll address it more fully later, in our concluding remarks.

7. Formalizing Requirements Specifications

We have left this section to the end, and for good reason. The capture and encoding of complex system requirements in the absolute and unforgiving vocabulary of symbolic logic is no easy trick [13, 14, 34, 54, 55]. What's more, this difficult endeavor must be completed perfectly. After all, if we verify our software product against a specification that does not correctly express the desires of our project stakeholders, then all of our efforts will be in vain. What we invest in axiomatizing our language semantics or shepherding the operation of our reasoning tool can return for us only a proof of something fallacious or irrelevant. If that's not bad enough, this is also an area in which the published literature seems able to give little guidance. There don't seem to be many good, published case studies describing processes or methodologies for capturing general software requirements in a form exact enough to enable them to be encoded as logical specifications. The literature contains good articles applicable in specific cases. Indeed, one is Crow and Di Vito's "Formalizing Space Shuttle Software Requirements: Four Case Studies," which appeared in the July 1998 issue of *ACM Transactions on Software Engineering and Methodology* [54]. But unless you have previous experience with aerospace controls development and know about things like *engine-out contingency abort maneuvers* and *reaction control system jet selections*, the Crow and Di Vito paper may raise more questions than answers. Likewise, significant publication activity surrounds the Naval Research Laboratory's *SCR Methodology* [55]—a paradigm for requirements analysis and specification that holds encoding for formal analysis among its explicit aims. But SCR is a big system, built on complicated tools, expressly targeted at the aerospace and defense industries. Un-

$$\textit{Call}\ \frac{\begin{array}{c}\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e\text{–}\!\succ\{Q\}\quad \forall a.\ \Gamma,A\vdash\{Q\leftarrow\mathsf{Val}\ a\}\ args\doteq\!\succ\{R\ a\}\\ \forall a\ vs\ D\ l.\ \Gamma,A\vdash\{(R\ a\leftarrow\mathsf{Vals}\ vs\ \wedge.\ (\lambda(x,s).\ D=\mathsf{target}\ mode\ s\ a\ md\ \wedge\ l=\mathsf{locals}\ s)\ ;.\\ \mathsf{init\_lvars}\ \Gamma\ D\ (mn,pTs)\ mode\ a\ vs)\wedge.\ (\lambda\sigma.\ \mathsf{normal}\ \sigma\longrightarrow\Gamma\vdash mode\rightarrow D\preceq t)\}\\ \mathsf{Methd}\ D\ (mn,pTs)\text{–}\!\succ\{\mathsf{set\_lvars}\ l\ .;\quad S\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \{t,md,mode\}e..\quad mn(\{pTs\}args)\text{–}\!\succ\{S\}}$$

Figure 4. A Hoare-style axiomatization of the method call construct in a constrained dialect of Java, excerpted from [53].

less your development organization works in these industries, adopting SCR may be prove more trouble than it's worth. In the end, J. Strother Moore and his colleagues summarized the requirements challenge plainly in [13], writing

> *How hard is it to use ACL2 to prove theorems of commercial interest? Modeling "the product" and capturing…important properties characterizing its "correctness" are the first and often hardest steps. These steps usually involve talking to the implementors, reading design documents, and experimenting with "toy models." You will often find that the implementors approach the project very differently than you do. They may not have in mind a correctness criterion that can be expressed as a property you can formalize. Their notion of correctness is often simply the absence of "bugs." Bugs, like good art, can evidently be recognized when seen.*

At its heart, the requirements challenge does appear more an art than a science. Nevertheless, we are not wholly lost. There are still useful things I think we can say about the process of unambiguously capturing and logically encoding system requirements, however challenging this process may be. But first, we need to get an idea of the shape that this endeavor usually takes.

7.1 Requirements—the Lay of the Land

The best way to come to understand the shape of requirements capture and encoding for verification is to compare the process to an analogous one most programmers are familiar with—*modular decomposition*, or the process of factoring a complex software system into a collection of independently evolvable "modules" [6, 56, 57]. Now, I want to be very clear that I don't strictly mean "modules" in the modular programming sense of separately compilable translation units [56, 57]. Those are indeed one kind of "module," but I use "module" here in the abstract. By *module*, I mean *any discrete software artifact that has an identity and existence independent of other, similar artifacts that in their ensemble constitute some complete system* [6]. What's more, I admit to the existence of many different kinds of modules, depending on the decomposition used. In a classical functional or procedural decomposition, the "modules" are procedures and functions [6, 56, 57]; in an object-oriented decomposition, the "modules" are classes [6].

We illustrate one interpretation of modular decomposition in figure 5. Here, we stylize a traditional procedural decomposition, in which the modules over which we factor the functionality of our system are subroutines [6, 56, 57]. Indeed, this should be a process with which most readers are familiar. Given a textual requirements specification, we enumerate a set of responsibilities. We then use these responsibilities to shape the *module architecture* of our system. For example, if our system's overall responsibility is to process the company payroll, we may decide that one part of the overall enterprise is to spool all employee paystubs to a network printer. We may also decide that we want the paystubs to emerge from the printer in name-sorted order so that employees can easily pick their stub from the output tray. So, in order to print our paystubs correctly, we need to be able to sort a list of records on a key that's a text string. If we're lucky of course, we already have a good sort function ready to be linked in to our project. And if we don't, then we'd want to write our sort function in a fashion general enough to enable its reuse in a later project [6].

All of these practices—of course—are nothing more than the bread-and-butter of good software engineering [3]. But by generalizing these notions, we can get a feel for how specifications and implementations are linked together to be automatically verified. To prepare for automated formal verification, we break down our requirements in a *specificational decomposition*. This—like any decomposition—outputs a collection of modules. Only here, our modules are *not* implementation artifacts like procedures or classes, but are instead *correctness conditions* for implementation artifacts that haven't yet been developed. We illustrate this in Figure 6 on the following page. We don't a get C++ implementation of a sort routine out of a specificational decomposition; we get a set of logical conditions that describe what it means for a sort routine to be correct. Obviously, we're likely to have requirements far more nebulous

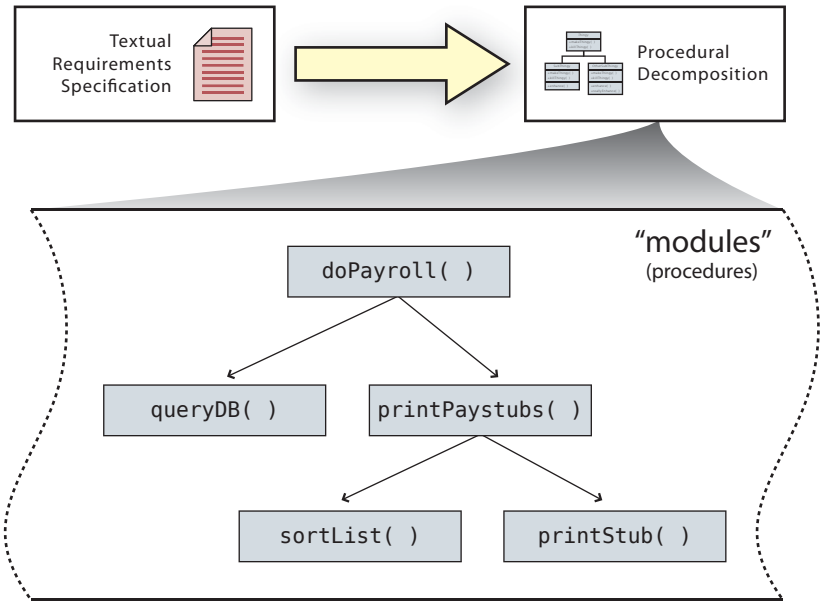

Figure 5. A stylized view of a traditional procedural decomposition, in which the "modules" over which we factor the functionality of our system are subroutines [6, 56, 57].

than a sorted list. But to prepare for automated verification, we must generate correctness conditions for such requirements all the same. For example, what does it mean for a pay stub record to be *correctly* spooled to a printer? Perhaps we can develop some understanding of the printer's command language, and seek to prove that the bytestream we send to the printer over the network will always encode a valid command program that prints a paystub with accurate information.

Once we have completed a specificational decomposition, we then code implementations for each of the *correctness modules* we've specified. Hopefully, each of the implementations we've coded meet the conditions imposed on them by their corresponding correctness module. In any case, we'll know soon enough. Now, we feed each of our correctness modules, along with their corresponding implementations, to an automated reasoning tool. Given appropriate semantic glue, the reasoning tool interprets the meaning of our code, and determines whether or not that code meets the correctness specifications imposed on it. All of this occurs on a *per-module basis*. Indeed, J. Strother Moore says as much when he writes in [23], "The verification of a system of methods is no harder than the combined verification of each method in the system. This remark, while trivial, has profound consequences if one clearly views the software verification problem as the specification and verification of the component pieces."

We hope that Moore is correct—because we're a long way from reasoning tools smart enough to test monolithic specifications against monolithic implementations [25]. The best we can hope for with contemporary tools is per-module correctness.

7.2 Automated Verification and Methodology

We can try to ensure the validity of Moore's assertion in our own projects by decomposing our specification using the best methods available [3, 6]. If we are to be able to verify a software system in the ensemble by checking the correctness of its individual modules, then those modules—be they classes or procedures—must be factored with certain guidelines in mind. We need to ensure, for example, that our modules interact only through well-defined interfaces that can we can formally specify and verify against. In general, we can't have interactions through side-effects mutating some sort of shared, global state.

The methodology community—particularly the object-oriented methodology community—has a lot to say about ways to divide responsibilities among modules and to specify interfaces through which modules collaborate to effect overall system behavior [3, 6, 16, 39, 59, 60]. As early as the 1970s, the Smalltalk community had developed *class-responsibility-collaboration* (or CRC) cards to encourage intuitive reasoning about the most natural way to factor responsibilities among classes [59]. In the 1980s, Bertrand Meyer, creator of the Eiffel programming language, pioneered a methodology called *design by contract*, in which the responsibilities of a class were

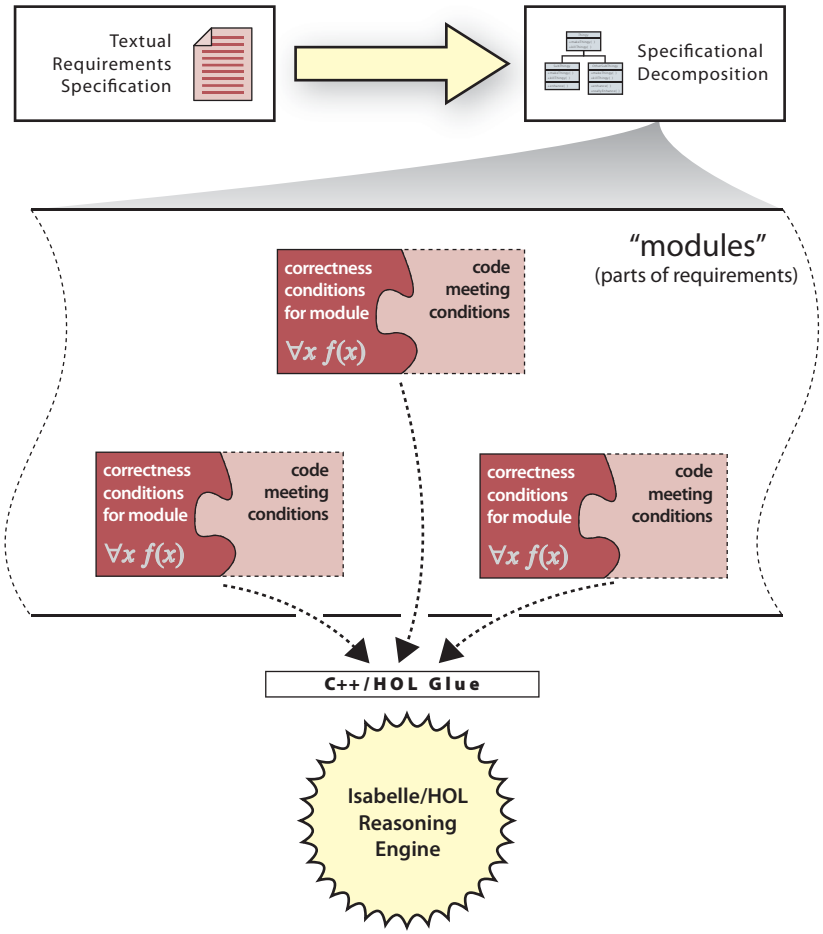

Figure 6. A stylized view of a *specificational decomposition* performed in preparation for automated verification. A specificational decomposition is analagous to the traditional procedural decomposition shown in Figure 5, but here our modules are *not* implementation artifacts like procedures or classes, but are instead *correctness conditions* for implementation artifacts that have yet to be developed

expressed as logical preconditions and postconditions over its methods. In this scenario, each method of a class guarantees that if its callers correctly satisfy its preconditions, then they can be assured that the method's postconditions will hold when control returns to them. Of course, these guarantees are informal only, since no static verification is performed [60].

In spite of the informality of these methods, I believe that they can be good first steps in the process of decomposing requirements into verifiable modules. In all cases, their goal is to encourage deeper understanding of whatever underlying problems the system requirements dictate be solved. And they do this by enabling developers to play around with different problem decompositions. Indeed, such deep understanding is crucial if automated verification is to be employed: by better understanding a problem, we can better specify what correct solutions to it look like. Young says as much in [28], where he writes "We make the argument that the advantages of [the new proof] over the earlier…proof are due much more to cleverness in the formalization…than to any strengths of the logic or automated proof system…We believe strongly that the obvious improvement of Rushby's formalization…is due to his greater insight into the problem."

8. Conclusion—the State of the Art

What is the state of the art in automated formal verification? And let's not measure it by asking, *how much the field has advanced since its inception?* Instead, let's answer *what we can do with it today?* Some might answer, "well—for most projects—not much." This was the certainly the view of Tom Melham, saying:

> *Formal methods will never have a significant impact until they can be used by people that don't understand them.*

in a quotation attributed to him in [50].

And it is hard to disagree. After all, only subsets of industrial-strength programming languages have been axiomatized, much axiomatization in real projects is still done by hand, and formalizing correctness specifications is a nightmare [13].

All of that said, I do believe that automated formal verification has a role in the development of real-world, industrial computing systems—even today.

At the lowest-levels of system development—where *foundation components*, like transaction kernels in financial trading systems and device drivers in operating systems execute—much code is still written in C [61] or in a very restricted subset of C++ [62]. What's more, in these performance critical arenas—where code often operates close to bare hardware—the added indirection and dereferencing stages that pointers and other aliasing constructs introduce may cause one to think twice before using them. Indeed, we should remember that the C++ Standard Template Library—perhaps the most performance-critical basic collections framework ever introduced into a mainstream language's standard library—uses value semantics wherever possible [57, 58]. And the STL doesn't eschew pointers for safety—indeed, the STL is extremely *unsafe*—it does so for performance [57]. This encourages the idea that, at least in the arena of foundation components, it may well be possible for us to express most designs of interest without using language features that are difficult to axiomatize. Indeed, it may even be advantageous— from a performance standpoint—never to consider such features at all. What's more, requirements must be more clear-cut in the foundation components universe: after all in many cases, the *raison d'être* for these systems is to stream bits out of raw hardware [61, 62].

To be sure, a highly-visible example of formal methods being employed at the foundation components level already exists. Microsoft encourages the use of the *Windows Driver Verifier*—a lightweight, implicit, automated formal verification tool that developers can use to statically verify their code against the Windows driver model [61]. In effect, this means that some of the most important foundation components in the world's most popular operating system are already formally verified. And other examples will surely follow. In the end, any profit-driven organization decides to employ formal verification when the cost of failure outweighs the labor cost of developing and employing formal expertise. At the foundation components level, failure costs are high—an entire system crashes when a device driver executing in kernel mode fails. And we've already addressed how performance concerns naturally restrict the language features usable in low-level code, as well as how systems targeted so close to raw hardware are necessarily subject to simpler requirements. It is this magic combination of high failure costs and tractable verification conditions that I think ultimately makes widespread automated verification of foundation components inevitable. Indeed, to anyone working in driver or kernel development, I'd say—get ready. The time for automated formal verification has come.

Works Cited

[1] Blair, M., Obenski, S., and Bridickas, P. *Report to the Honorable Howard Wolpe on the Patriot Missile Software Problem* (as GAO/IMTEC-92-96). Washington, D.C.: United States General Accounting Office, 1992.

[2] Marshall, Eliot. "Fatal Error: How a Patriot Overlooked a Scud." *Science*. 255(5050): March 13, 1992.

[3] Schach, Stephen. *Object-Oriented and Classical Software Engineering 7/e*. New York: McGraw-Hill, 2007.

[4] Leveson, N. and Turner, C. "An Investigation into the Therac-25 Accidents." *IEEE Computer*. 26(7): July, 1993.

[5] Gibbs, W. Wayt. "Software's Chronic Crisis." *Scientific American*. 271(3): September, 1994.

[6] Booch, Grady. *Object-Oriented Analysis and Design with Applications 2/e*. Reading, Massachusetts: Addison-Wesley, 1994.

[7] Bryant, Antony. "It's Engineering, Jim…but not as we know it: Software Engineering—a Solution to the Software Crisis, or Part of the Problem?" *Proceedings ot the 22nd Annual ACM SIGSOFT Conference on Software Engineering*. New York: ACM Press, 2000.

[8] Jorgensen, M. and Molokken, K. "How large are software cost overruns? A Review of the 1994 Chaos Report." *Information and Software Technology.* 48(4): April, 2006.

[9] Knuth, Donald. *The Art of Computer Programming 3/e: Volume 1*. Reading, Massachusetts: Addison-Wesley, 1997.

[10] Patterson, D. and Hennessy, J. *Computer Organization and Design: the Hardware/Software Interface 3/e*. San Francisco: Morgan-Kaufmann, 2005.

[11] Sipser, Michael. *Introduction to the Theory of Computation 2/e*. Boston: Thomson Learning, 2006.

[12] Stroll, Robert R. *Set Theory and Logic*. New York: Dover, 1979.

[13] Kaufmann, M., Manolios, P. and Moore, J.S. *Computer Aided Reasoning*: *An Approach*. Boston: Kluwer Academic Publishers, 2000.

[14] Nipkow, T., Paulson, L., and Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher Order Logic*. Berlin: Springer-Verlag 2002.

[15] Royce, Winston. "Managing the Development of Large Software Systems." *Proceedings of IEEE WESCON '70*. New York: IEEE, 1970.

[16] Kruchten, Phillipe. *The Rational Unified Process: An Introduction 2/e*. Reading, Massachusetts: Addison-Wesley, 2000.

[17] Gödel, Kurt (translated by B. Meltzer). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. New York, Dover, 1992.

[18] Strecker, Martin. "Formal Verification of a Java Compiler in Isabelle." *CADE-18: Proceedings of the 18th International Conference on Automated Deduction.* Berlin: Springer-Verlag, 2002.

[19] Botaschanjan, J., Kof, L., Kühnel, C., and Spichkova, M. "Towards Verified Automotive Software." *Proceedings of the Second International Workshop on Software Engineering for Automotive Systems.* New York: ACM Press, 2005.

[20] Blech, J., Gesellensetter, L., and Glesner, S. "Formal Verification of Dead Code Elimination in Isabelle/HOL." *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods.* New York: IEEE, 2005.

[21] Kauffman, M., Manolios, P., and Moore, J.S. *Computer-Aided Reasoning: ACL2 Case Studies.* Boston: Kluwer Academic Publishers, 2000.

[22] Hanbing, L. and Moore, J. S., "Java Program Verification via a JVM Deep Embedding in ACL2." *Theorem Proving in Higher Order Logics: Proceedings of the 17th International Conference.* Berlin: Springer-Verlag, 2004.

[23] Moore, J. Strother. "Proving Theorems about Java-like Bytecode." *Correct System Design: Recent Insights and Advances*, (Olderog, E. and Steffen, B., eds.). Berlin: Springer-Verlag, 1999.

[24] Arora, S. and Barak, B. *Complexity Theory: a Modern Approach.* Publication expected January, 2008. Draft available at `http://www.cs.princeton.edu/theory/complexity/`. Electronic publication of Princeton University: retrieved 4 November 2007.

[25] Moore, J. Strother. "A Mechanized Program Verifier." Slides from a lecture presented at the *IFIP Verification Grand Challenge Workshop 2005*. Available at `http://www.cs.utexas.edu/users/moore/publications/zurich-05/talk.html`. Electronic publication of the University of Texas: retrieved 8 November 2007.

[26] Barnett, M., Leino, K. R. M., Schulte, W. "The Spec# Programming System: An Overview." *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of CASSIS 2004*. Berlin: Springer-Verlag, 2005.

[27] Polak, Wolfgang. "An Exercise in Automatic Program Verification." *IEEE Transactions on Software Engineering*. 5(5): September, 1979.

[28] Young, William D. "Comparing Verification Systems: Interactive Consistency in ACL2." *IEEE Transactions on Software Engineering*. 23(4): April, 1997.

[29] Brock, B., Kaufmann, M., and Moore, J. S. "ACL2 Theorems about Commercial Microprocessors." *Formal Methods in Computer-Aided Design: Proceedings of FMCAD '96 Palo Alto.* Berlin: Springer-Verlag, 1996.

[30] Randimbivololona, F. "Orientations in Verification Engineering of Avionics Software." *Informatics: 10 Years Back, 10 Years Ahead*, (ed. Wilhelm, R.). Berlin: Springer-Verlag, 2001.

[31] Rushby, J. and von Henke, F. "Formal Verification of Algorithms for Critical Systems." *IEEE Transactions on Software Engineering.* 19(1): January, 1993.

[32] Mackenzie, D. and Pottinger, G. "Mathematics, Technology, and Trust: Formal Verification, Computer Security, and the U.S. Military." *IEEE Annals of the History of Computing.* 19(3): July-September 1997.

[33] Wolf, W. H. "Hardware-Software co-Design of Embedded Systems." *Proceedings of the IEEE.* 82(7): July 1994.

[34] Brucker, Achim. *An Interactive Proof Environment for Object-Oriented Specifications*. Ph.D. Thesis: ETH Zurich, 2007.

[35] Reetz, R. and Kropf, T. "A Flowgraph Semantics of VHDL: Toward a VHDL Verification Workbench in HOL." *Formal Methods in System Design*. 7(1-2): August 1995.

[36] Wasserrab, D., Nipkow, T., Snelting, G., and Tip, F. "An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++." *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*. New York: ACM Press, 2006.

[37] Hohmuth, M., Stephens, S., and Tews, H. *Applying Source-Code Verification to a Microkernel—the VFiasco Project* (as ISSN 1430-211X, a Technical Report of the University

of Dresden). Dresden, Germany: 2002.

[38] Rumbaugh, J. Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1999.

[39] Clark, T. and Warmer, J. *Object Modeling with the OCL: The Rationale Behind the Object Constraint Language (Lecture Notes in Computer Science Edition)*. Berlin: Springer-Verlag, 2002.

[40] Gordon, Michael J. C. "From LCF to HOL: a Short History." *Proof, Language, and Interaction* (eds. Plotkin, G., Stirling, C. and Tofte, M.). Cambridge, Massachusetts: MIT Press, 2000.

[41] Boyer, R. S., Kaufmann, M., and Moore, J. S. "The Boyer Moore Theorem Prover and its Interactive Enhancement." *Computers & Mathematics with Applications*. 29(2): January 1995.

[42] Owre, S., Rushby, J. M. and Shankar, N. "PVS: a Prototype Verification System." *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*. Berlin: Springer-Verlag: 1992.

[43] Owre, Sam. *PVS Specification and Verification System—Home*. Available at: `http://pvs.csl.sri.com/`. Electronic publication of Stanford University Research Institute, Inc.: retrieved 13 November 2007.

[44] Owre, Sam. *PVS Specification and Verification System—PVS Documentation*. Available at: `http://pvs.csl.sri.com/documentation.shtml`. Electronic publication of Stanford University Research Institute, Inc.: retrieved 13 November 2007.

[45] Harrison, John. "Formal Verification at Intel." Slides from a lecture presented at *LICS 2003: the 18th Annual IEEE Symposium on Logic in Computer Science*. Available at: `http://www.cl.cam.ac.uk/~jrh13/slides/lics-22jun03.pdf`. Electronic publication of Intel Corp. and Cambridge University: retrieved 13 November 2007.

[46] Staff of Computational Logic, Inc. *Company Background*. Available at: `http://www.computationallogic.com/corp/cli-background.html`. Electronic publication of Computational Logic, Inc.: retrieved 16 November, 2007.

[47] Andrews, Peter B. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof* 2/e. Berlin: Springer-Verlag, 2002.

[48] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML—Revised Edition*. Cambridge, Massachusetts: MIT Press, 1997.

[49] Paulson, Lawrence C. *ML for the Working Programmer*. Cambridge, England: Cambridge University Press, 1996.

[50] Pierce, Benjamin C. *Languages and Types*. Cambridge, Massachusetts: MIT Press, 2002.

[51] Floyd, Robert W. "Assigning Meaning to Programs." *Proceedings of the 19th Annual AMS Symposium on Applied Mathematics* (ed. Shwartz, J. T.). Providence, Rhode Island: American Mathematical Society, 1967.

[52] Hoare, C. Anthony R. "An Axiomatic Basis for Computer Programming." *Communications of the ACM*. 12(10): October, 1969.

[53] von Oheimb, David. "Hoare Logic for Java in Isabelle/HOL." *Concurrency and Computation: Practice and Experience*. 13(13): November, 2001.

[54] Crow, J. and Di Vito, B. "Formalizing Space Shuttle Requirements: Four Case Studies." *ACM Transactions on Software Engineering and Methodology*. 7(3): July, 1998.

[55] Heitmeyer, C., Bull, A., Gasarch, C. and Labaw, B. "SCR: a Toolset for Specifying and Analyzing Requirements." *COMPASS '95: Proceedings of the Tenth Annual IEEE Conference on Systems Integrity, Software Safety, and Process Security*. New York: IEEE, 1995.

[56] Kernighan, B. W. and Ritchie, D. M. *The C Programming Language 2/e*. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[57] Stroustrup, Bjarne. *The C++ Programming Language 3/e*. Reading, Massachusetts: Addison-Wesley, 1997.

[58] Josuttis, Nicolai M. *The C++ Standard Library: a Tutorial and Reference*. Reading, Massachusetts: Addison-Wesley, 1999.

[59] Wirfs-Brock, R., Wilkerson, B., Wiener, L. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice-Hall, 1990.

[60] Meyer, Bertrand. *Object-Oriented Software Construction 2/e*. Englewood Cliffs, New Jersey: Prentice-Hall, 2000.

[61] Staff of Microsoft, Corp. *WDF Driver Verification Tools Home*. Available at: `http://www.microsoft.com/whdc/driver/wdf/Drv-VerTools.mspx`. Electronic publication of Microsoft Corporation: retrieved 1 December 2007.

[62] Staff of Apple, Inc. *I/O Kit Fundamentals for Mac OS X*. Available at: `http://developer.apple.com/DOCUMENTATION/DeviceDrivers/Conceptual/IOKitFundamentals/`. Electronic publication of Apple, Inc: retrieved 1 December 2007.