Certified Software

Only if the programmer can prove (through formal machine-checkable proofs) it's free of bugs with respect to a claim of dependability.

Zhong Shao
Yale University
zhong.shao@yale.edu

Abstract

Certified software consists of a machine-executable program plus a formal machine-checkable proof that the software is free of bugs with respect to a claim of dependability. The conventional wisdom is that certified software will never be feasible because the dependability of any real software must also rely on that of its underlying operating system and execution environment which is too low-level to be verifiable. In recent years, however, there have been many advances in the theory and engineering of mechanized proof systems applied to verification of low-level code, including proof-carrying code, certified assembly programming, local reasoning and separation logic, certified linking of heterogeneous components, and certified or certifying compilation. In this article, I give an overview of this exciting new field, focusing on both foundational ideas and key insights that make the work on certified software differ from traditional program verification systems. I will also describe several exciting recent advances and challenging open problems.

1. Introduction

Computer software is one of the most influential technologies ever developed. Software has entered every aspect of our lives and is used to control everything from computing and communication devices (such as computers, networks, cell phones, and Web browsers), to consumer products (such as cameras, TVs, and refrigerators), to cyber-physical systems (such as automobiles, medical devices, and aviation systems), and to critical infrastructure (such as financial, energy, communications, transportation, and national defense).

Unfortunately, software is also sometimes our least dependable engineering artifact. Software companies lack the kind of meaningful warranty most other engineering organizations are expected to provide. Major corporations and government agencies worldwide invest in fixing software bugs, but the prospect of building reliable software is bleak. The pervasive presence of software bugs also makes all existing computing and information systems vulnerable to security and privacy attacks.

An important cause of such difficulty is the sheer complexity of the software itself. If each line of code is viewed as an individual component, software systems are easily the most complicated things we humans have ever built. Unlike hardware components, software execution can easily lead to an unbounded number of states, so testing and model-checking techniques cannot guarantee reliability. As the hardware community moves deep into new multi-core and cyber-physical platforms, and as software is thoroughly integrated into everyday objects and activities, the complexity of future software could get much worse, even as demand for dependable software becomes more urgent.

For most of today's software, especially low-level forms like operating systems, nobody knows precisely when, how, and why they actually work. These systems lack rigorous formal specifications and were developed mostly by large teams of developers using programming languages and libraries with imprecise semantics. Even if the original developers had a good informal understanding of the inner workings, their knowledge and assumptions about system behavior (often implicit) are easily lost or broken in subsequent development or maintenance phases.

The software research community has sought to tackle these problems in recent years but remains hampered by three key difficulties:

Lack of metrics. Metrics are still lacking for measuring software dependability, making it difficult to compare different techniques and build steady progress in the field. Dependability often includes attributes like reliability, safety, availability, and security. A system's availability can be measured retroactively as a percentage of its uptime in a given year; for example, six nines, or 99.9999%, means 31.5 seconds downtime per year, but quantifying other attributes is much more difficult. A program with one bug is not necessarily 10 times more secure than a program with 10 bugs. A system's reliability depends on its formal specification, which is often nonexistent.

Worse, software dependability is often confused with the dependability of the software's execution environment, which consists of not just hardware devices but also human operators and the physical world. Since the dependability of the execution environment is often beyond human control, many people view software as a complex biological system, rather than as a rigorous mathematical entity;

System software. A software application's dependability also relies on the dependability of its underlying system software, including OS kernel, device driver, hypervisor, garbage collector, and compiler. These low-level programs are often profoundly complex and bug-prone, but little has been done to make them truly dependable. For example, if an OS kernel or even a compiler has security holes, the entire system could be compromised, regardless of what software developers do at a higher level [31, 19]; and

Last-mile problem. Despite recent progress in formal-methods research, program verification still involves a vexing "last-mile problem." Most software-verification research concentrates on high-level models rather than on actual programs, valuable for finding bugs but leaving a big gap that must be closed before meaningful dependability claims can be made about actual software. Failure to reason about actual code also has big implications for maintainability; for example, it is difficult for programmers to pinpoint the source and a fix when a new bug is

identified and ensure that subsequent updates (to actual code) will not break the code's high-level model.

Leading research on *certified software* aims to tackle all three. For example, concerning the lack of good metrics, a line is drawn between the actual machine-executable software and the surrounding physical environment (e.g., hardware devices and human operators). We can neither predict the future of the physical world nor formally certify human behavior, but at least under a well-defined, stable hardware platform (such as the x86 instruction set), the behavior of each machine executable is a rigorous mathematical entity. With a formal specification stating its desirable behavior, we can (at least in theory) rigorously "certify" that the machine executable behaves as expected. A good dependability metric is then just the formal claim developers make and certify about each program.

The long-term goal for research on certified software is to turn code—often a system's weakest link—into its most dependable component. The formal specification given may not precisely capture the behavior of the physical environment, so the overall system may still not function properly, but, at least, when a problem occurs, programmers and users alike are assured that the behavior of the software is properly documented and rigorously enforced. The specifications for functional correctness of individual components may occasionally be too large to be comprehensible, but many systemwide safety, liveness, and security properties can be stated succinctly and certified with full confidence.

To address the second and third difficulties, software developers must also certify the actual system-software code. Most needed is a new "certified" computing platform where programmers have firm control over the behavior of its system software stack, including bootloader, OS kernel, device driver, hypervisor, and other runtime services. Software consisting of mostly certified components would be easier to maintain, because the effects of updating a certified component would be easier to track, and new bugs would quickly be localized down to the noncertified modules.

Constructing large-scale certified software systems is itself a challenge. Still unknown is whether it can be done at all and whether it can be a practical technology for building truly dependable software. In this article, I explore this new field, describing several exciting recent advances and challenging open problems.

2. What Is It?

Certified software consists of a machine-executable program C plus a rigorous formal proof P (checkable by computer) that the software is free of bugs with respect to a particular dependability claim S. Both the proof P and the specification S are written using a general-purpose mathematical logic, the same logic ordinary programmers use in reasoning every day. The logic is also a programming language; everything written in logic, including proofs and specifications, can be developed using software tools (such as proof assistants, automated theorem provers, and certifying compilers). Proofs can be checked automatically for correctness, on a computer, by a small program called a *proof checker*. As long as the logic used by programmers is consistent, and the dependability specification describes what end users want, programmers can be sure that the underlying software is free of bugs with respect to the specification.

The work on certified software fits well into the Verified Software Initiative (VSI) proposed by Hoare and Misra [15], but differs in several distinct ways from traditional program-verification systems.

First, certified software stresses use of an expressive general-purpose metalogic and explicit machine-checkable proofs to support modular reasoning and scale program verification to handle all kinds of low-level code [32, 24, 10, 3]. Using a rich mechanized metalogic allows programmers to define new customized "domain-specific" logics (together with its meta theory), apply them to certify different software components, and link everything to build end-to-end certified software [9]. With machine-checkable proofs, proof-checking is automated and requires no outside assumptions. As long as the metalogic is consistent, the validity of proof P immediately establishes that the behavior of program C satisfies specification S.

Existing verification systems often use a rather restricted assertion language (such as first-order logic) to facilitate automation but do not provide explicit machine-checkable proof objects. Program components verified using different program logics or type systems cannot be linked together to make meaningful end-to-end dependability claims about the whole software system. These problems make it more difficult for independent third parties to validate claims of dependability.

Second, with an expressive metalogic, certified software can be used to establish all kinds of dependability claims, from simple type-safety properties to more advanced safety, liveness, security, and correctness properties. Building these proofs need not follow Hoare-style reasoning [14]; much of the earlier work on proof-carrying code [23] constructed safety proofs automatically using such technologies as type-preserving compilation [30, 29] and typed assembly language [22]. However, most traditional program verifiers concentrate on partial correctness properties only.

Third, certified software emphasizes proving properties for the actual machine executables, rather than their high-level counterparts, though proofs can still be constructed at the high level and then propagated down to the machine-code level using a certifying or certified compiler. On the other hand, most existing program verifiers target high-level source programs.

Fourth, to establish a rigorous dependability metric, certified software aims to minimize the trusted computing base, or TCB, the small part of a verification framework in which any error can subvert a claim of end-to-end dependability. TCB is a well-known concept in the verification and security community, as well as a source of confusion and controversy [6].

The dependability of a computing system rests on the dependable behavior of its underlying hardware devices, human operators, and software. Many program verifiers are comfortable with placing complex software artifacts (such as theorem provers, OS, and compilers) into the TCB because it seems that the TCB of any verification system must include those "hard-to-reason-about" components (such as hardware devices and human operators) so is already quite large.

Of course, all program-verification systems create a formal model about the underlying execution environment. Any theorem proved regarding the software is with respect to the formal model only, so the TCB for any claim made regarding the software alone should not include hardware devices and human operators.

Still, any bug in the TCB would (by definition) compromise the credibility of the underlying verification system. A smaller TCB is generally more desirable, but size is not necessarily the best indicator; for example, a 200-line garbage collector is not necessarily more reliable than a 2,000-line straightforward pretty printer. The TCB of a good certified framework must include only components whose soundness and integrity can also be validated by independent third parties.

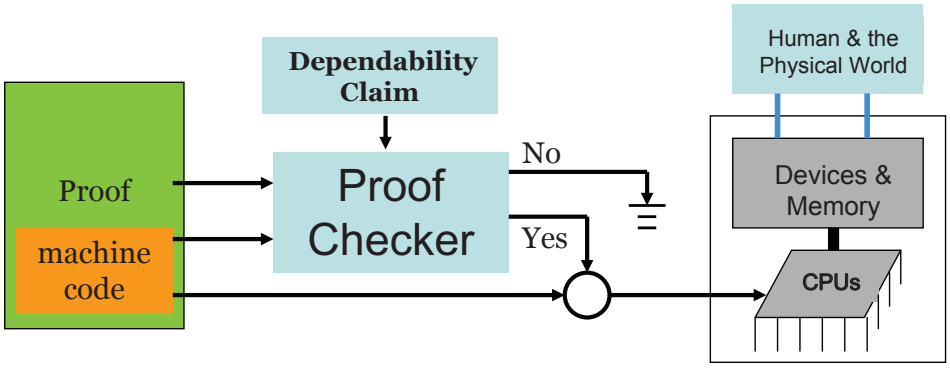


Figure 1. Components of a certified framework

2.1 Components of a certified framework

A typical certified framework (see Figure 1) consists of five components:

- *The certified software itself.* Including both machine code and formal proof;
- *Formal machine model.* Providing the operational semantics for all machine instructions;
- *Formal dependability claim for the software.* Including safety property, security policy, and functional specification for correctness;
- *Underlying mechanized metalogic (not shown).* For coding all proofs, specifications, and machine-level programs; and
- *Proof checker.* For checking the validity of all the proofs following the inference rules of the metalogic.

If the proof of a given certified software package can be validated by the proof checker, then execution of the software on the formal machine model is guaranteed to satisfy a formal dependability claim.

Things can, however, still go wrong. First, the mechanized metalogic might be inconsistent, a risk that can be minimized if the framework designers choose a simple, well-understood, general-purpose metalogic and prove (perhaps on paper) why it is indeed consistent.

Second, the proof checker is a computer program, so it could go wrong all by itself. But if the framework uses a simple logic with a small number of inference rules, the proof checker can be made quite small, written in assembly, and verified by hand.

Third, the formal machine model might not reflect hardware behavior. Most hardware vendors perform intensive hardware verification, so this risk can be minimized if hardware and software developers share the machine specifications. Even if this is not possible, the framework designer can still validate the model by comparing its operational semantics with the instruction-set reference manuals.

Finally, the formal dependability specification (*SP*) may not accurately capture the behavior of the human or physical world. Nevertheless, *SP* is formally stated and the code is guaranteed to satisfy *SP*. Here, I deliberately decoupled the correctness of verification from the specification process. Existing efforts validating and testing specifications are, of course, very valuable and complementary to the certification process.

Since a dependability claim is made only regarding the formal machine model, the TCB of such a certified framework consists of just the consistency proof of the metalogic and the integrity of the proof checker, both of which should be demonstrable by independent third parties (such as through the peer-review process of a top-quality journal). If the computer science community can agree on a single metalogic (a good thing), this task of standardizing a metalogic would need to be done only once. Certified software would then no longer be the weakest link in a dependable system.

2.2 Mechanized metalogic

A key enabling technology for certified software is to write formal proofs and specifications as typed functional programs, then have a computer automatically check the validity of the proofs, in the same way a static type-checker does type-checking. This idea came from the well-known Curry-Howard correspondence referring to the generalization of a syntactic analogy between systems of formal logic and computational calculi first discovered by the American logicians Haskell Curry and William Howard. Most advances for developing large-scale machine-checkable proofs were made only during the past 10 years; see the excellent survey by Barendregt and Geuvers [2] and a 2008 overview article by Hales [11].

In the context of certified software, there are a few more requirements: the logic must be consistent and expressive so software developers can express everything they want to say. It must also support explicit machine-checkable proof objects and be simple enough that the proof checker can be hand-verified for correctness.

Because software components may be developed using different programming languages and certified using different domain-specific logics and type systems, mechanized metalogic must also support meta-reasoning. It can be used to represent the syntax, inference rules, and meta-proofs (for their soundness) of the specialized object logics.

Much of the current work on certified software is carried out in the Coq proof assistant [16]. Coq itself provides a rich higher-order logic with powerful inductive definitions, both crucial for writing modular proofs and expressive specifications.

2.3 Advantages

With certified software, the dependability of a software system would be measured by the actual formal dependability claim it is able to certify. Because the claim comes with a formal proof, the dependability can be checked independently and automatically in an extremely reliable way.

A formal dependability claim can range from making almost no guarantee, to simple type-safety property, to deep liveness, security, and to correctness properties. It provides a great metric for comparing different techniques and making steady progress toward the system's overall dependability.

If the software community could agree on a metalogic and work out the formal models of a few popular computing platforms, certified software would provide an excellent framework for accumulating dependable software components. Since proofs are incontrovertible mathematical truths, once a software component is certified, its trustworthiness (with respect to its specification) would presumably last for eternity.

Unlike higher-level programming languages, certified software places no restrictions on the efficiency of its underlying code and the way programs are developed. Because the metalogic is as rich as the one programmers use in daily reasoning, and everything running on a computer must eventually be executed as a machine executable, if programmers believe (informally) that their super-efficient and sophisticated code really works as they claim, there should be a way to formally write down their proofs. When dependability is not an issue, the software can be used as is, assuming proper isolation from the rest of the system; when programmers really care about dependability, they must provide the formal machine-checkable proof.

On the other hand, certified software encourages the usual best practices in software engineering and program verification. Certifying large-scale systems clearly benefits from high-level programming abstraction, domain-specific logics, modular decomposition and refinement, model-driven design and development, the correctness-by-construction methodology [12], and automated theorem-proving tools. The only difference is that they now insist on receiving hard evidence (such as machine-checkable proof objects) as a way to deliver quality assurance and measure the effectiveness of the technologies.

Certified software also decouples the tools for proof construction and program development from the proof-checking infrastructure. The rich metalogic provides the ultimate framework for building up layers of abstraction for complex software. Once they are formed, programmers can build different software components and their proofs using completely different methods. Because specifications and proofs are both represented as programs (within a computer), they can be debugged, updated, transformed, analyzed, and reused by novel proof-engineering tools.

Certified software also significantly improves the maintainability of the underlying system. A local change to an individual component can be checked quickly against its specification, with its effect on the overall system known immediately. A major reorganization of the system can be done in a principled way by comparing the changes against high-level specifications programmers have for each certified component.

2.4 Challenges

The main challenge of certified software is the potentially huge cost in constructing its specifications and proofs, though such cost can be cut dramatically in the following ways.

First, how software is developed makes a huge difference in the system's future dependability. If the software is full of bugs or developed without consideration of the desirable dependability claim, post-hoc verification would be extremely expensive in terms of time and money, or simply impossible. A proactive approach (such as correctness-by-construction [12]) should lower the cost significantly.

Second, building certified software does not mean that programmers must verify the correctness of every component or algorithm used in its code; for example, in micro-kernels or virtual-machine monitors, it is often possible for programmers to verify a small set of components that in turn perform run-time enforcement of security properties on other components [33].

Dynamic validation (such as translation validation for compiler correctness [26]) also simplifies proofs significantly; for example, it may be extremely difficult to verify that a sophisticated algorithm A always takes an input X and generates an output Y such that $R(X, Y)$ holds; instead, a programmer could extend A by adding an additional validation phase, or a validator, that checks whether the input X and the output Y indeed satisfy the predicate R, assuming R is decidable. If this check fails, the programmer can invoke an easier-to-verify (though probably less-efficient) version of the algorithm A. To build certified software, all the programmer needs is to certify the correctness of the validator and the easier version of the algorithm, with no need to verify algorithm A anymore.

Third, the very idea that proofs and specifications can be represented as programs (within a computer) means that developers should be able to exploit the synergy between engineering proofs and writing large programs, building a large number of tools and proof infrastructures to make proof construction much easier.

Finally, formal proofs for certified software ought to be much simpler and less sophisticated than those used in formal mathematics [11]. Software developers often use rather elementary proof methods to carry out informal reasoning of their code. Proofs for software are more tedious but also more amenable for automatic generation [28, 5].

Certified software also involves other challenges. For example, the time to market is likely terrible, assuming dependability is not a concern, so the cost of certification would be justified only if end users truly value a dependability guarantee. Deployment would be difficult since most real-world engineers do not know how to write formal specifications, let alone proofs. Pervasive certification requires fundamental changes to every phase in most existing software-development practices, something few organizations are able to undertake. The success of certified software critically relies on efforts initially developed in the research community.

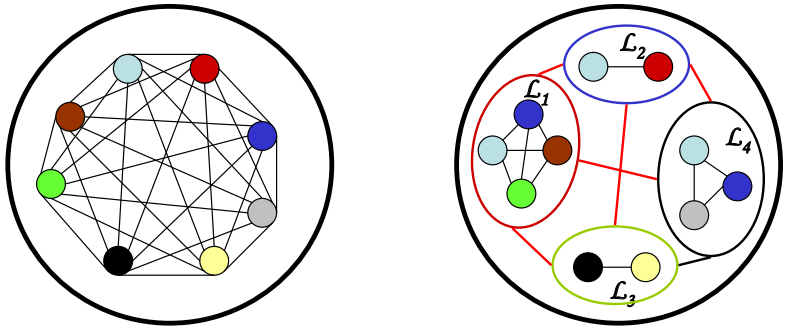
3. Recent Advances

Advances over the past few years in certified software have been powered by advances in programming languages, compilers, formal semantics, proof assistants, and program verification. Here, I sample a few of these efforts and describe the remaining challenges for delivering certified software.

3.1 Proof-carrying code

Necula's and Lee's 1996 work [23] on proof-carrying code (PCC) is the immediate precursor to the large body of more recent work on certified software. PCC made a compelling case for the importance of having explicit witness, or formal machine-checkable evidence, in such applications as secure mobile code and safe OS kernel extensions. PCC allows a code producer to provide a (compiled) program to a host, along with a formal proof of safety. The host specifies a safety policy and a set of axioms for reasoning about safety; the producer's proof must be in terms of these axioms.

PCC relies on the same formal methods as program verification but has the advantage that proving safety properties is much easier than program correctness. The producer's formal proof does not, in general, prove that the code produces a correct or meaningful result



(a) The One-for-All Logic (b) Domain-Specific Logics

Figure 2. Using domain-specific logics to verify modules

but does guarantee that execution of the code satisfies the desirable safety policy.

Checking proofs is an automated process about as simple as programming-language type-checking; on the other hand, finding proofs of theorems is, in general, intractable. Subsequent work on PCC focused on building a realistic certifying compiler [4] that automatically constructs proofs (for simple type-safety properties) for a large subset of Java and on reducing the size of proof witness, an important concern in the context of mobile code.

An important PCC advantage inherited by certified software is that the software does not require use of a particular compiler or of any compiler. As long as the code producer provides the proof, the code consumer is assured of safety. This significantly increases the flexibility available to system designers.

The PCC framework is itself quite general, but the original PCC systems suffered from several major limitations: Most notable was that the proof checker had to rely on a rather specific set of typing rules so did not support more expressive program properties; the typing rules were also error-prone, with their soundness often not proved, so a single bug could undermine the integrity of the entire PCC system.

Foundational PCC, or FPCC [1, 13], tackled these problems by constructing and verifying its proofs using a metalogic, with no type-specific axioms. However, FPCC concentrated on building semantic models for high-level type-safe languages, rather than performing general program verification.

3.2 Certified assembly programming

Certified Assembly Programming (CAP) [32] is a logic-based approach for carrying out general program verification inside a rich mechanized metalogic (such as the one provided by Coq). Like Hoare logic, a CAP program consists of assembly code annotated with pre- and post-conditions and program invariants. Unlike traditional Hoare-style verification, all CAP language constructs (such as assembly instruction sets), program assertions, inference rules, operational semantics, and soundness proofs are implemented inside the mechanized metalogic. This design makes it possible to build a complete certified software package with formal dependability-claim and machine-checkable proofs. With help from a proof assistant, programmers are able to combine manually developed proof scripts with automated proof tactics and theorem provers, allowing CAP to support verification of even undecidable program properties.

CAP marries type-based FPCC with Hoare-style program verification, leading to a great synergy in terms of modularity and expressiveness. Hoare logic is well-known for its limited support for higher-order features; most Hoare systems do not even support verification of simple type-safety properties. However, both shortcomings are easily overcome in type-based approaches. Subsequent work on CAP over the past five years developed new specialized program logics for reasoning about such low-level constructs as embedded code pointers [24], stack-based control abstractions [10], self-modifying code [3], and garbage collectors [21].

Under type-based FPCC, function returns and exception handlers are often treated as first-class functions, as in continuation-passing style (CPS), even though they have more limited scope than general first-class continuations. For functional programmers, CPS-based code is conceptually simple but requires complex higher-order reasoning of explicit code pointers (and closures). For example, if a function needs to jump to a return address (treated as continuation), the function must assert that the return address is indeed a valid code pointer to jump to . But the function does not know exactly what the return address will be, so it must abstract over properties of all possible return addresses, something difficult to do in first-order logic.

In the work on stack-based control abstraction [10], my colleagues and I showed that return addresses (or exception handlers) are much more disciplined than general first-class code pointers; a return address is always associated with some logical control stack, the validity of which can be established statically; a function can cut to any return address if it establishes the validity of its associated logical control stack. Such safe cutting to any return address allows programmers to certify the implementation of sophisticated stack operations (such as setjmp/longjmp, weak continuations, general stack cutting, and context switches) without resorting to CPS-based reasoning. For example, when programmers certify the body of a function, they do not need to treat its return address as a code pointer; all they need is to make sure that at the return, the control is transferred to the original return address. It is the caller's responsibility to set up a safe return address or valid code pointer; this is much easier because a caller often knows the return address that must be used.

3.3 Local reasoning and separation logic

Modular reasoning is the key technique for making program verification scale. Development of a certified software system would benefit from a top-down approach where programmers first work out the high-level design and specification, then decompose the en-

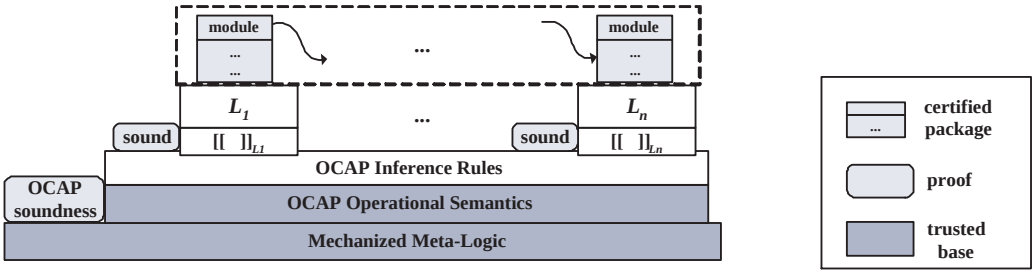


Figure 3. An open framework for building certified software

tire system into smaller modules, refine high-level specifications into actual implementation, and finally certify each component and link everything together into a complete system.

However, there is yet another critical dimension to making program verification modular. Traditional Hoare logics often use program specifications with arbitrarily large "footprints." Separation logic [27, 17] advocates "local reasoning" using small-footprint specifications; that is, the specification of each module (or procedure) should refer only to data structures actually touched by a module's underlying code. By concisely specifying the separation of heap and other resources, separation logic provides succinct yet powerful inference rules for reasoning about shared mutable data structures and pointer anti-aliasing.

Concurrent separation logic (CSL) [25] applies the same idea to reason about shared-memory concurrent programs, assuming the invariant that there always exists a partition of memory among different concurrent entities and that each entity can access only its own part of memory. This assumption might seem simple but is surprisingly powerful. There are two important points about this invariant. First, the partition is *logical*; programmers do not need to change their model of the physical machine, which has only one global shared data heap, and the logical partition can be enforced through separation logic primitives. Second, the partition is not static and can be adjusted dynamically during program execution by transferring the ownership of memory from one entity to the other.

Under CSL, a shared-memory program can be certified as if it were a sequential program since it is always manipulating its private heap; to access shared memory, it must invoke an atomic operation that transfers resources between the shared heap and the local heap. Several recent efforts have extended CSL with rely-guarantee reasoning, so even lock-free concurrent code can be certified using modular small-footprint specifications.

3.4 Domain-specific logics and certified linking

A key first step toward making certified software practical is to show it is possible to carry out end-to-end certification of a complete software system. Large software systems, especially low-level system software, use many different language features and span many different abstraction levels. For example, the Yale FLINT group's (http://flint.cs.yale.edu) ongoing project [8] to verify a simplified OS kernel exposes such challenges. In it, the kernel includes a simple bootloader, kernel-level threads and a thread scheduler, synchronization primitives, hardware interrupt handlers, and a simplified keyboard driver. Although it has only 1,300 lines of x86 assembly code, it uses such features as dynamic code loading, thread scheduling, context switching, concurrency, hardware interrupts, device drivers, and I/O. How would a programmer use machine-checkable proofs to verify the safety or correctness properties of such a system?

Verifying the whole system in a single program-logic or type system is impractical because, as in Figure 2(a), such a verification system would have to consider all possible interactions among these features, including dynamic code loading, concurrency, hardware interrupts, thread scheduling, context switching, and embedded code pointers, many at different abstraction levels. The resulting logic, if it exists, would be highly complex and difficult to use. Fortunately, software developers seem to never use all features simultaneously. Instead, they use only a limited combination of features—at a certain abstraction level—in individual program modules. It would be much simpler to design and use specialized "domain-specific" logics (DSL) to verify individual program modules, as in Figure 2(b). For example, for the simplified OS kernel, dynamic code loading is used only in the OS boot loader, and interrupts are always turned off during context switching; embedded code pointers are not needed if context switching can be implemented as a stack-based control abstraction.

To allow interactions of modules and build a complete certified software system, programmers must also support interoperability of different logics. In 2007 my colleagues and I developed a new open framework for CAP, or OCAP [7], to support verification using specialized program logics and for certified linking of low-level heterogeneous components. OCAP lays a set of Hoare-style inference rules over the raw operational semantics of a machine language (see Figure 3), and the soundness of these rules is proved in a mechanized metalogic so it is not in the TCB.OCAP uses an extensible and heterogeneous program-specification language based on the higher-order logic provided by Coq. OCAP rules are expressive enough to embed most existing verification systems for low-level code. OCAP assertions can be used to specify invariants enforced in most type systems and program logics (such as memory safety, well-formedness of stacks, and noninterference between concurrent threads). The soundness of OCAP ensures that these invariants are maintained when foreign systems are embedded in the framework.

To embed a specialized verification system $\mathcal{L}$, OCAP developers first define an interpretation $[\![\ \_\ ]\!]_{\mathcal{L}}$ that maps specifications in $\mathcal{L}$ into OCAP assertions; then they prove system-specific rules/axioms as lemmas based on the interpretation and OCAP rules. Proofs constructed in each system can be incorporated as OCAP proofs and linked to compose the complete proof.

There are still many open issues on the design of OCAP: For example, to reason about information-flow properties, OCAP must provide a semantic-preserving interpretation of high-order types (in an operational setting). And to support liveness properties, OCAP must support temporal reasoning of program traces.

3.5 Certified garbage collectors and thread libraries

In 2007, my colleagues and I used OCAP to certify several applications involving both user-program code and low-level runtime

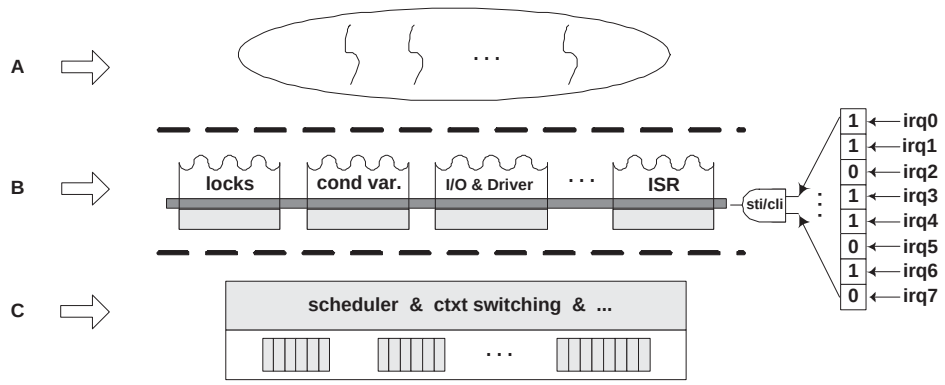


Figure 4. Decomposition of a preemptive thread implementation

code. In one application [7], we successfully linked programs in typed assembly language (TAL) [22] with a certified memory-management library. TAL supports only type-preserving memory updates, and the free memory is invisible to TAL code. We certified the memory-management library in stack-based CAP, or SCAP [10], which supports reasoning about operations over free memory while ensuring that the invariants of TAL code are maintained.

Also in 2007, in another application [21], we developed a general framework for certifying a range of garbage collectors and their mutators. If we had tried to develop a single type system to type-check both an ML-like type-safe language and the underlying garbage collector (requiring fancy runtime type analysis), the result would have involved analyzing polymorphic types, which is extremely complex. However, the ML type system never needs to know about runtime tagging and the internals of the garbage collector. Moreover, implementation of the collector need not understand the polymorphic type system used in type-checking ML code; it needs to only distinguish pointers from non-pointers. A better approach, which we followed in 2007, is to certify these modules using different domain-specific logics, thus avoiding the difficult task of designing a universal program logic. Certified garbage collectors can then be linked with certified mutator code to form a complete system.

A year later, in a third application [8], we successfully certified the partial correctness of a preemptive thread library extracted from our simplified OS kernel. The kernel was implemented in 16-bit x86 assembly and worked in real mode for uniprocessor only. It consisted of thread context switching, scheduling, synchronizations, and hardware interrupt handlers. We stratified the thread implementation by introducing different abstraction layers with well-defined interfaces. In Figure 4, at the highest level (Level A), preemptive threads follow the standard concurrent programming model. The execution of a thread can interleave with or be preempted by other threads. Synchronization operations are treated as primitives. Hardware interrupts are abstracted away and handled at Level B where code involves both hardware interrupts and threads; synchronization primitives, input/output operations, device drivers, and interrupt handlers are all implemented at this level; interrupt handling is enabled/disabled explicitly using `sti/cli`. At the lowest level (Level C), the thread scheduler and the context switching routine manipulate the threads' execution contexts stored in thread queues (on the heap). Interrupts are invisible at this level because they are always disabled. Libraries implemented at a lower level are exposed as abstract primitives for the level above it, and their operational semantics in the high-level abstract machine serve as formal specifications for the low-level implementation.

The stratified system model gives programmers a systematic and principled approach for controlling complexity. Programmers can thus focus on a subset of language features at each level and certify different software components using specialized program logics.

3.6 Certified and certifying compilation

Much work in the program-verification community concentrates on source-level programs written in high-level languages (such as C, Java, and C#). In order to turn these programs into certified assembly components suitable for linking in the OCAP framework, OCAP developers must show that their corresponding compiler is also trustworthy.

CompCert is a certified compiler for a subset of C (called C minor, or Cm) developed in 2006 by Leroy [20]. By "certified" compiler, I mean the compiler itself is proved correct. Indeed, Leroy specified formal operational semantics for Cm, as well as for the machine language, building a machine-checkable proof in Coq whereby the compiler preserves behavior from one operational semantics to another. However, the current CompCert compiler supports only sequential Cm programs. It also must be bootstrapped by the OCaml compiler, even though the OCaml compiler is not verified.

On the other hand, a certifying compiler is not necessarily correct but will take a (certified) source program and generate certified assembly code. Much work on certifying compilation focuses on type-safe source languages and can preserve only type-safety properties. A challenging open problem is to extend certifying compilation to preserve deep correctness and security properties.

3.7 Lightweight formal methods

Building large-scale certified software systems does not always require heavyweight program verification. Most software systems are built from modular components at several levels of abstraction. At the lowest levels are the kernel and runtime-system components discussed earlier. At the highest levels are components with a restricted structure operating on well-defined interfaces. The restricted structure can use a type-safe, high-level programming language with high-level concurrency primitives or C programs (even concurrent C programs) in a style understandable to static-analysis tools. Both restricted styles are in widespread commercial use today.

Lightweight formal methods (such as high-level type system, specialized program logic, with decidable decision procedure, and static analysis) can help guarantee important safety properties with

moderate programmer effort; error messages from the typechecker, decision procedure, and static-analyzer usually give appropriate feedback in the programming process. These safety properties are sometimes also even security properties, as in this example: "Module A cannot read the private variables of module B, except through the public methods provided by B." Using information-flow type systems or static analysis a programmer can obtain a stronger version of the same guarantee while also adding "... and not only that, but the public methods of module B do not leak the value of private variable x."

Lightweight formal methods can be used to dramatically cut the cost of building certified software. For a programmer, the challenge is to make them generate explicit proof witness (automatically) and link them with certified low-level kernel and runtime components. With proper embedding, lightweight formal methods would fit nicely into the DSL-centric OCAP framework for constructing end-to-end certified software.

3.8 Automation and proof engineering

The end goal of certified software is a machine-checkable dependability metric for high-assurance software systems. Certified software advocates the use of an expressive metalogic to capture deep invariants and support modular verification of arbitrary machine-code components. Machine-checkable proofs are necessary for allowing third parties to quickly establish that a software system indeed satisfies a desirable dependability claim. Automated proof construction is extremely important and desirable but should be done only without violating the overall integrity and expressiveness of the underlying verification system.

Much previous research on verification reflected full automation as a dominating concern. This is reasonable if the primary goal is finding bugs and having an immediate effect on the real world's vast quantity of running software. Unfortunately, insisting on full automation also severely hinders the power and applicability of formal verification; many interesting program properties (that end users care about) are often undecidable (full automation is impossible), so human intervention is unavoidable. Low-level program modules often have subtle requirements and invariants that can be specified only through high-order logic; programming libraries verified through first-order specifications often have to be adapted and verified again at different call sites.

Nevertheless, there is still a great synergy in combining these two lines of software-verification work. The OCAP framework described here emphasizes domain-specific (including decidable first-order) logics to certify various components in a software system. Successful integration would allow programmers to get the best of both lines.

Developing large-scale mechanized proofs and human-readable formal specifications will become an exciting research field on its own, with many open issues. Existing automated theorem provers and Satisfiability Modulo Theories solvers [5] work only on first-order logic, but this limited functionality is in conflict with the rich metalogic (often making heavy use of quantifiers) required for modular verification of low-level software. Proof tactics in existing proof assistants (such as Coq) must be written in a different "untyped" language, making it painful to develop large-scale proofs.

4. Conclusions

Certified software aligns well with a 2007 study on software for dependable systems [18] by the National Research Council (http://sites.nationalacademies.org/NRC/index.htm) that argued for a direct approach to establishing dependability whereby software developers make explicit the dependability claim and provide direct evidence that the software indeed satisfies the claim. However, the study did not explain what would make a clear and explicit dependability claim, what serves as valid evidence, and how to check the underlying software to ensure that it really satisfies the claim without suffering credibility problems [6].

The study also said that the dependability of a computer system relies not only on the dependability of its software but also on the behavior of all other components in the system, including human operators and the surrounding physical environment. Certified software alone cannot guarantee the dependability of the computer system. However, as explained earlier, many advantages follow from separating the dependability argument for the software from the argument for the software's execution environment.

Computer software is a rigorous mathematical entity for which programmers can formally certify its dependability claim. However, the behavior of human operators depends on too many factors outside mathematics; even if they try hard, they would probably never achieve the kind of rigor they can for software. By focusing on software alone and insisting that all certified software come with explicit machine-checkable proofs, we can use the formal dependability claim as a metric for measuring software dependability. Formal specifications are also more complete and less ambiguous than informal specifications written in natural languages; this should help human operators better understand the behavior of the underlying software.

A key challenge in building dependable systems is to identify the right requirements and properties for verification and decide how they would contribute to the system's overall dependability. Certified software does not make this task easier. Research on certifying low-level system software would give software developers more insight into how different programming-abstraction layers would work together. Insisting on machine-checkable proof objects would lead to new high-level certified programming tools, modular verification methodologies, and tools for debugging specifications, all of which would make developing dependable software more economical and painless.

Acknowledgments

I would like to thank Xinyu Feng, Daniel Jackson, George Necula, Ashish Agarwal, Ersoy Bayramoglu, Ryan Wisnesky, and the anonymous reviewers for their valuable feedback.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] H. P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier Scientific Publishing B.V., 2001.

[3] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 66–77, 2007.

[4] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conference on Programming Language Design and Implementation*, pages 95–107, 2000.

[5] L. M. de Moura and N. Bjørner. Z3: An Efficent SMT Solver. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340, April 2008.

[6] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 206–214, Jan. 1977.

[7] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 67–78, Jan. 2007.

[8] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. 2008 ACM Conference on Programming Language Design and Implementation*, pages 170–182, 2008.

[9] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *Proc. 2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 54–69. Springer-Verlag, October 2008.

[10] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conference on Programming Language Design and Implementation*, pages 401–414, June 2006.

[11] T. C. Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, December 2008.

[12] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, January/February 2002.

[13] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS'02)*, July 2002.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), Oct. 1969.

[15] T. Hoare and J. Misra. Verified software: theories, tools, experiments. In *Proc. 1st IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'05)*, volume 4171 of *LNCS*, pages 1–18. Springer-Verlag, October 2005.

[16] G. Huet, C. Paulin-Mohring, et al. The Coq proof assistant reference manual. The Coq release v6.3.1, May 2000.

[17] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symposium on Principles of Programming Languages*, pages 14–26, Jan. 2001.

[18] D. Jackson, M. Thomas, and L. Millett. *Software for Dependable Systems: Sufficient Evidence?* The National Academic Press, 2007.

[19] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (Oakland)*, pages 314–327, May 2006.

[20] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, Jan. 2006.

[21] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 468–479, 2007.

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 85–97, Jan. 1998.

[23] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symposium on Operating System Design and Impl.*, pages 229–243, 1996.

[24] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd Symp. on Principles of Prog. Lang.*, pages 320–333, Jan. 2006.

[25] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.

[26] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 151–166. Springer-Verlag, March 1998.

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

[28] W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. In *Proceedings of the C/C++ Verification Workshop*, July 2007.

[29] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[30] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. 1996 ACM Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[31] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[32] D. Yu, N. A. Hamid, and Z. Shao. Building certified libaries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 363–379. Springer-Verlag, Apr. 2003.

[33] N. Zeldovich. *Securing Untrustworthy Software Using Information Flow Control*. PhD thesis, Department of Computer Science, Stanford University, October 2007.

Zhong Shao (zhong.shao@yale.edu) is a professor in the Department of Computer Science at Yale University, New Haven, CT.