From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification


Publisher
Association for Computing Machinery

Version
Author's final manuscript

Accessed
Tue Mar 12 08:29:02 EDT 2019

Citable Link
http://hdl.handle.net/1721.1/91954

Terms of Use
Creative Commons Attribution-Noncommercial-Share Alike

Detailed Terms
http://creativecommons.org/licenses/by-nc-sa/4.0/
From Network Interface to Multithreaded Web Applications
A Case Study in Modular Program Verification

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

Abstract

Many verifications of realistic software systems are monolithic, in the sense that they define single global invariants over complete system state. More modular proof techniques promise to support reuse of component proofs and even reduce the effort required to verify one concrete system, just as modularity simplifies standard software development. This paper reports on a case study applying modular proof techniques in the Coq proof assistant. To our knowledge, it is the first modular verification certifying a system that combines infrastructure with an application of interest to end users. We assume a nonblocking API for managing TCP networking streams, and on top of that we work our way up to certifying multithreaded, database-backed Web applications. Key verified components include a cooperative threading library and an implementation of a domain-specific language for XML processing. We have deployed our case-study system on mobile robots, where it interfaces with off-the-shelf components for sensing, actuation, and control.

Categories and Subject Descriptors E.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs - Mechanical verification; D.2.4 [Software Engineering]: Software/Program Verification - Correctness proofs

Keywords modular program verification; proof assistants; thread libraries; Internet servers; domain-specific languages

1. Introduction

Program verification for realistic systems is very labor-intensive. The proof engineer faces all of the usual challenges of the software engineer and then some. Typically a machine-checked proof of program correctness requires more human effort than implementing the software system in the first place. Modularity is an essential part of the programmer’s arsenal in mastering complexity, and we might ask for the same in verification. We want a program’s proof to mirror its natural decomposition into encapsulated components. Each component should be proved to respect a formal interface, and thereafter we reason about the component only through its interface. Not only does this style allow us to reuse component proofs in verifications of different full systems, but it also frequently lowers the effort to verify just one system, for the same reason the author of a program with a special-purpose data structure may choose to encapsulate that data structure in its own class or module, even with no expectation for future reuse in other programs.

The most recent impressive system verifications are not modular. The L4 verified[22] and Verve[37] projects have both verified operating-system kernels with different semi-automated proof tools. Their proofs are structured around global invariants relating all state of all system components, both in the kernel and in relevant parts of applications. When one part of a kernel changes, the global invariant must often be modified, and the proofs of all other components must be reconsidered. Ideally, proof automation is able to use already-written annotations and proof scripts to reconstruct the new proofs, but in practice the proof engineer is often required to revisit old proof details of system modules beside the one he has changed. Such coupling between proof pieces is undesirable, and we instinctively feel that it should be avoidable for certain kinds of changes, like applying an optimization that does not affect a component’s API-level behavior.

A variety of impressive results have also been published for modular verification systems. Modular program logics have been demonstrated for a variety of tricky features of low-level code including dynamic thread creation[9], function pointers[29], stack-based control operators[12], self-modifying code[3], garbage collection[25], and interrupts[10]. However, these past projects never applied modular verification at scale, certifying a full system of interest to end users. In theory, the frameworks were set up to allow black-box linking of proofs about systems infrastructure and applications, to produce full-system proofs, but the devil is in the details. This paper is about those details.

We present the first realistic case study in modular mechanismized program verification, establishing properties of a deployed application. What exactly we mean by “realistic” we will spell out below. One key element is separate proof of an application and reusable systems infrastructure. In our case, the application is a database-backed dynamic Web application and the infrastructure is a cooperative multithreading library. One subtle source of difficulty is supporting infrastructure that itself needs to play nicely with other modules of infrastructure, some of which may not even have been written or specified yet. Contrast that complication with the L4 verified[22] and Verve[37] proofs, which show correctness of OS kernels assumed to have exclusive control over all privileged elements of machine state; or the proof of the CompCert C compiler[23], which assumes that no other compiler will be used to produce code in the final program. Other elements of realism have produced further challenges, which we needed to solve on the way to deploying an application with real users who are not formal-methods experts.
What properties would we expect to see of modular verifications, in a hypothetical future world where such activities are standard, at least for code of particular importance?

- **Not all verified components will have theorems at the same level of detail.** The reason is the classic formal-methods complaint that “a word processor does not have an obvious specification worth proving.” Also, different parts of a system may be of varying importance, where the proof engineer wants to invest more effort in more important components via more precise theorems.

- **Not all software in the final system will be verified.** It is necessary to have some kind of “foreign function interface” to connect to conventional components.

- **There will be further modular structure within infrastructure components, and this structure must be designed to facilitate effective reasoning in client code.** For instance, in the components of a thread library, we need to think carefully about which interfaces to assign to the data structures used to implement the scheduler, where we feel pressure to make the interfaces as weak as possible to minimize proof effort. However, we are building these modules so that they may eventually be linked with client code, where we feel pressure to make the infrastructure interfaces as strong as possible to give the client flexibility.

- **Some components will be implemented in domain-specific languages (DSLs), but we do not want to have to trust anything about those languages or their implementations.** Our case study uses a DSL for declarative XML processing and access to a relational database.

- **Most proof details should be automated.** It is unacceptable to require detailed manual proof for each line of program code.

- **Most importantly, we must actually link our proofs of infrastructure components with proofs of applications, to form full-program theorems.** A proof of infrastructure is only as good as the sorts of proof composition it enables.

To the best of our knowledge, no past work in modular verification has produced a runnable case study demonstrating any of the above principles, whereas ours involves all of them.

Our verification was done in the Coq proof assistant using the Bedrock library [4, 5], whose basic functionality we review in more detail shortly. Bedrock is a platform for producing verified assembly code by programming, proving, and compiling entirely within Coq.

Our application is a replacement for the so-called Master Server of the popular open-source Robot Operating System (ROS) [32]. ROS is a platform for component-based implementation of robotics software. Systems are structured as networks of nodes (processes) that communicate only via message-passing (mostly with TCP). Messages are largely exchanged via a publish-subscribe architecture. Among other services, the Master Server processes node requests to publish or subscribe to particular named topics. It must maintain an in-memory database mapping node names to IP addresses and ports, mapping topics to their publishers and subscribers, and so on. The Master Server speaks the XML-RPC protocol and runs as a Web server. Our version of it has been deployed on several semi-autonomous vehicles and used to coordinate the interaction of nodes for sensors, actuators, and control algorithms.

The system infrastructure we have verified is a cooperative threading library. As unverified primitives, we assume system-call functions for opening TCP sockets and sending and receiving byte streams over them. Furthermore, we require system calls for efficient polling, to determine which sockets have new data to process. All the assumed primitives beside polling are nonblocking. On top of them, we implement a threading library that supports the standard abstraction of threads blocking during calls to I/O functions, without forcing programmers to refactor their code into continuation-passing style.

We prove a deep functional correctness theorem for the thread library, where bugs can lead to particularly vexing misbehavior where one thread overwrites the private state of another. To demonstrate the possibility to mix proofs at different granularities in one verification, we verify the application at the level of data-structure shape invariants, mostly pertaining to the in-memory relational database. We link the module proofs together into a whole-program proof.

The unifying specification style applied to the final program is: various points in the assembly code (e.g., entry labels of crucial functions) are labeled with invariants in higher-order logic, formalizing properties of machine state that should always hold upon reaching those points. We prove that, starting from an initial program location whose invariant is satisfied, execution will proceed safely forever, never reaching a program point whose invariant is false, regardless of what nondeterministic results system calls return. These invariants can be at many different levels of specificity. Invariants within our thread library encode detailed requirements for functional correctness, while invariants in our application code focus more on shape invariants of data structures like the in-memory relational database. All invariants, throughout the program, must formalize enough of essential isolation properties to guarantee lack of corruption of other modules’ data structures.

Section 4 gives a more detailed outline of what we have proved, and Section 5 reviews the relevant aspects of the Bedrock framework. The next few sections present our main contributions:

- Section 4 explains how to extend Bedrock with a treatment of system calls, to model interaction with conventional, unverified libraries.

- Section 5 introduces three verification design patterns that we found essential to meet our standard of proof modularity and automation. We present a new style of definition for recursive predicates that are higher-order and stateful, a new formal interface style for components of thread libraries or others working with first-class code pointers, and a new take on classic separation-logic [33] rules that is compatible with verifying the building blocks of domain-specific languages.

- The next two sections present particular modular verification architectures that we found work well for our components. Section 6 presents the architecture of our cooperative threading library and Section 7 the architecture of our verified compiler for a DSL for XML processing and relational database operations.

- Finally, one contribution that we want to highlight most is that our proofs apply to real code that has been deployed outside a formal-methods context. Section 8 gives some empirical analysis of our proofs and executable code, which we do not believe has been done before for modularly verified systems.

The complete source code of this project, for both rechecking proofs and generating executable Linux programs, is available on the project Web site:

http://plv.csail.mit.edu/bedrock/

2. System Architecture Summary

Before proceeding to technical details, we take a moment to sketch the full layered architecture that we have verified. Different levels of this architecture have separately encapsulated proofs, and we
combine all the proofs, treated as black boxes, into one final theorem about a particular linked assembly program.

To summarize our results: the unifying verification formalism is based on invariant checking. Programs contain assertions written in higher-order logic. That is, they may express much more intricate properties than in executable code-based assertions. Each module contains code with assertions, plus assumptions on other modules, plus proofs that the assumptions imply that no assertions are violated while we are running code from this module. Our final program is a large assembly source file (about 500,000 lines) where each basic block begins with an assertion, but we produce this code by (verified) compilation from higher-level languages with their own notions of where assertions (again, in higher-order logic, not just executable code) are allowed. The final theorem is that the assembly code runs without assertion violations.

The Bedrock platform is fundamentally single-threaded at its lowest level. We verify assembly programs with respect to a single-core operational semantics. We do not take advantage of parallelism as a performance optimization, but rather we use concurrency as a convenient program-structuring idiom, with multiple threads running and participating in different IO interactions. The thread model is cooperative, where threads yield control of the processor explicitly by calling particular library functions.

We prove functional (partial) correctness of our thread library, by giving it very expressive assertions, sufficient to enable easy verification of similar functional correctness of simple applications, e.g., proving that a factorial program really computes factorial directly from its semantics. Every assembly basic block begins with an invariant, and their connection to higher-level interfaces like “finite multiset” to be exported to higher layers.

Our proof is structured into several software layers. We expand on each layer in later sections, but here is a quick summary.

1. Device Abstraction API. We begin from a nonverified layer providing a set of system calls for basic, nonblocking access to a network card and TCP/IP stack. Without impacting our proofs, this layer could just as well be a simple interprocess communication mechanism, as the details of message-passing are orthogonal to the properties we verify.

2. Data Structures & Algorithms (systems-level). Our first verified layer considers core data structures, such as the free list of a malloc() implementation and the FIFO queue at the heart of a round-robin thread scheduler. Challenges here have to do with reasoning about heap-allocated data structures, pointer aliasing, and their connection to higher-level interfaces like “finite multiset” to be exported to higher layers.

3. Thread Context Management. Next, we provide an abstraction of queues of suspended threads. Verification challenges center on the relationship between memory and first-class code pointers (i.e., saved instruction pointers of threads). Each thread must be considered to own some private memory but also to access some shared memory, containing data structures that other threads access as well. Functions at this level must break basic abstractions of C-like code, implementing a coroutine style by storing function return pointers in heap data structures, rather than just returning to them as usual. The interface for this level should hide such details, so that higher layers may pretend that a normal function-call discipline is being followed.

4. Blocking IO Abstraction. The next challenge is to provide a simple interface that enables easy overlapping of IO and computation. While the lowest layer of our system exposes nonblocking IO functions, we combine them with thread queues to implement blocking IO plus dynamic thread creation and destruction. The interface of this level should make blocking IO calls appear very similar to the nonblocking calls, hiding the fact that threads may suspend themselves into queues while waiting for IO events to be enabled.

5. Data Structures & Algorithms (application-level). We implement a simple in-memory relational database, with standard data structures for representing table state.

6. Domain-Specific Language (application-level). Our application must both access the database and do parsing and generation of XML, since it provides an XML-RPC Web service. We implemented a domain-specific language (DSL) called the Bedrock Web Service Language (BWS), which more or less allows us to execute intuitive pseudocode for such operations. Our language has a verified compiler, so programmers do not need to do any manual proof to generate a theorem corresponding to a compiled application, showing that it maintains the invariants of the database, does not interfere with other threads, etc.

7. Application Logic. Finally, we implement the ROS Master service as a BWS program in about 400 lines of code, calling BWS library tactics to automate its proof.

Layers 2 through 4 would be considered as a “user-level thread library” in UNIX parlance. They implement functionality similar to what has been treated in operating-systems-verification projects like L4.verified [22]. Those projects have generally given operational specifications to kernels, via nondeterministic programs expressing how the kernel should respond to hardware and user-level events. In the modular verification style we adopt here, we instead specify all layers of the system in terms of invariants that are respected and with preconditions and postconditions of functions. The two approaches can be related to each other formally, and the invariant style tends to make for easier verification of application-level properties.

Of course, one can always ask for more specific theorems about programs, and it is hard to come up with a formal characterization of “full functional correctness for a full system.” For instance, most theorems assume correctness of hardware at some level, and our proof goes even further in considering a TCP/IP stack to be outside the scope of what we verify. However, we do not assume any other services traditionally provided by an operating system: we need no virtual memory, no process preemption interrupts, and so on. The program style is set up to be compatible with “bare-metal” execution in future work. We start modestly with our application-level specifications, as this case study is targeted at learning pragmatic lessons about modular program proofs from first principles.

3. Review of Bedrock

Our verification is done using Bedrock, a Coq library that provides formal definitions, proof procedures, and syntax macros, which together enable implementing, specifying, verifying, and compiling low-level programs within Coq. At the core of Bedrock is the Bedrock IL, a kind of cross-platform assembly language. In the end, every program we verify is compiled to the Bedrock IL, and our correctness theorems are stated in terms of this language and its semantics. Every assembly basic block begins with an invariant, and we prove that each invariant holds each time it is visited. We compile Bedrock IL to 32-bit or 64-bit x86 assembly code for
2nd-order vars. \( \alpha \)  
Coq propositions \( P \in \text{Prop} \)  
Machine words \( w \in \mathbb{W} \)  
Machine states \( \gamma \in \mathbb{S} \)  
Coq terms \( \nu \)  
PropX \( \phi ::= |P| | \{ \lambda \gamma. \phi \} w | \phi \land \phi | \phi \lor \phi | \phi \Rightarrow \phi | \forall x. \phi | \exists x. \phi | \alpha(v) | \forall \alpha. \phi | \exists \alpha. \phi \)

Figure 1. Syntax of the XCAP assertion logic

execution in Linux. That final translation is currently unverified, but it works in a particularly simple way, essentially macro-expanding each Bedrock IL opcode into a sequence of opcodes in the target language.

Though the core features deal with an assembly language, Bedrock provides infrastructure for combined verification and compilation of an extensible C-like language, the Bedrock Structured Programming System (BSPS) [5], and our explanation in this paper begins at that abstraction layer. We will write code that looks broadly like C. It is important to note that these programs appear within Coq source files that also contain theorem statements and proofs; Bedrock takes advantage of Coq’s extensible parser to embed syntax of other languages. We make further use of the same facility to embed our new Bedrock Web Service Language, whose verified compiler is also built on top of BSPS to avoid redundant proofs about low-level code generation. We refer readers to past Bedrock publications [5] for details of the lower abstraction levels that we elide for the rest of this paper.

Bedrock implements the XCAP [9] program logic and inherits the sorts of final program correctness theorems that XCAP supports. XCAP is designed for modular verification of programs that manipulate code pointers as data. In such a setting, it is not obvious what is an appropriate notion of specification for functions. One might try defining a domain of specifications like \( P \subseteq (\mathbb{W} \rightarrow \mathbb{S}) \rightarrow \text{Prop} \). That is, a specification is a predicate over both machine states \( \mathbb{S} \) and mappings from program counters \( \mathbb{W} \) to the specifications of the associated code blocks. Some sort of ability for specifications to refer to specifications or code of other blocks is essential for verifying higher-order programs. Unfortunately, the naïve recursive equation has no solution, as a simple cardinality argument shows.

Several more involved notions of specification have been studied. Many are based on step indexing [2], where otherwise troubling recursive definitions are made well-founded by adding extra parameters counting steps of program execution that remain. XCAP, and thus our work in this paper, is based on an alternative approach. A specification language is defined with a deep embedding, or an explicit definition of syntax trees in Coq. XCAP uses the PropX language shown in Figure [1].

PropX is a second-order assertion logic that includes most of the usual connectives. Unusual features include \( \{ P \} \), for lifting a normal Coq proposition into PropX; and \( \{ \lambda \gamma. \phi \} w \), a kind of Hoare double, for asserting that the precondition of the code block pointed to by \( w \) is the function \( \lambda \gamma. \phi \), from Bedrock IL machine states to PropX formulas. Introducing this connective solves the problem with specifications that reference other specifications. Just as a classic static type system allows, e.g., a map function to constrain its argument using a syntactic function type, XCAP allows, e.g., an assembly-level map function to constrain its argument using a syntactic Hoare double nested within map’s precondition. The soundness of either reasoning pattern is justified by considering that, at each point in program execution, we are running some code fragment that has been proved to meet a syntactic specification, which accurately describes the current state.

Usually one wants a more semantic notion \( \{ \gamma. \phi \} w \) of a Hoare double, saying not that the given specification is literally attached to code block \( w \), but rather that the given specification implies \( w \)’s specification semantically. Both this notation and the Hoare triple of separation logic [3] are defined by macro expansion to expansions of \( \{ \cdot \cdot \} \cdot \cdot \) and other connectives.

Also crucial to the expressiveness of PropX is the inclusion of second-order quantification over predicate variables \( \alpha \). For instance, in our thread library verification, we use this feature to quantify over invariants describing the private state of individual threads. The semantics of PropX formulas is given by a set of natural-deduction rules, where the second-order quantifiers receive the peculiar treatment of being given introduction rules but no elimination rules. As a result, elimination-style reasoning may only be done at the meta level, in Coq’s normal logic. reasoning about the syntactic definition of PropX deduction. Here is where we run into the friction that seems always to accompany sound specification regimes for higher-order, stateful programs, as the soundness proof for the PropX deduction system cannot be adapted naturally to cover elimination rules for these quantifiers.

In the rest of this paper, we often adopt convenient shorthands like omitting lifting brackets \( \{ \cdot \} \) in PropX formulas, pretending that we are instead working in a more conventional higher-order logic.

Earlier we sketched the basic idea of a verified module: a unit of code annotated with invariant assertions, plus assumptions about which invariants (e.g., as precondition-postcondition pairs) are associated with code labels in other modules, plus proof that our own code never encounters assertion violations, if our assumptions are accurate. XCAP instantiates this style to the case of assembly code. Modules encapsulate sets of basic blocks, and a linking theorem justifies combining the basic blocks of two modules, when their assumptions are consistent. Linking module A with B may remove A’s assumptions about B, and vice versa. Eventually, we link enough modules to empty out the assumptions, and we have a closed program. The final theorem about a verified closed program is that it never encounters violations of its invariant assertions. We also get a standard memory safety property, where code never accesses unmapped memory addresses or jumps to nonexistent program labels.

Once again, however, in this paper we take the assembly-level details for granted, hiding below the abstractions that BSPS provides. We think in terms of verified modules, invariants, linking, etc., for modules of C-like code.

4. System Calls

The last section reviewed existing tools that we rely on, and we now turn to new contributions. The next few sections discuss elements of modular verification that only show up when scaling to more
realistic systems. We begin here with the question, how do we support integration with unverified support code?

We extended the Bedrock IL with new system calls, trusted primitives outside the scope of our verification, upon which we depend to provide an efficient interface to the outside world. Since we focus on Internet servers, our system calls all relate to TCP network connections and IO. These system calls expose only nonblocking operations, with no controls on sharing resources (e.g., the network card) across different software components. Instead, we implement such controls in Bedrock and verify our implementations.

Figure 2 shows C prototypes for our trusted system calls. The first group of functions provides a simplified version of the standard BSD sockets API. We write `fd_t` for the type of file descriptors, standing here for open socket connections. Again we emphasize that all of these socket operations are nonblocking. For example, any call to `read()` will return immediately with an error code if no bytes are available to read.

On top of this interface, we built a verified implementation of the standard cooperative multitreading abstraction with blocking IO. To enable efficient polling of the kind needed to implement blocking IO, we use two additional system calls in the style of Linux’s `epoll`[1] A function `declare()` is called to register intent to read or write a particular socket, producing a value called a reservation. Later, when the program makes a `wait()` system call, the result is a reservation whose associated IO operation is now ready to execute. A call to `wait()` itself may either be blocking, if the program has no other useful work to do without further IO opportunities; or nonblocking, if there is further CPU-bound work to do while waiting for IO. Conceptually, in the former case, the system blocks until one of the reserved IO events becomes enabled, at which point that reservation ID is returned.

We have extended the Bedrock IL semantics to include these system calls as primitive operations, explained via one new operational-semantics rule per system call. Figure 3 gives an example rule for `read()`. The semantics is small-step, operating on states of the form \((m, pc, r)\), where \(m\) is memory, \(pc\) the program counter, and \(r\) values of registers. Two registers in particular matter here, to formalize the calling convention that `read()` expects: the stack pointer \(Sp\) and the return pointer \(Rp\), which stores a code address to return to after the call.

The first few premises in Figure 3 formalize the precondition of `read()`. In order, we have that the stack pointer indicates a valid memory segment lasting at least 16 bytes (containing the values of passed parameters), two particular stack slots contain the values of parameters `buf` and `len`, and the memory region signified by those parameters is valid. The remaining premises constrain the result state of this step. We say that the stack pointer \(Sp\) is guaranteed to be restored, and that only memory addresses within the passed buffer may be modified.

Note that the rule is quite nondeterministic. We give a conservative characterization of the unverified code that we link against. In particular, rather than formalizing the effect of the function, we just describe which parts of state it may change.

Though it is convenient to state our final theorems in terms of this relatively simple operational semantics, reasoning about this style of specification directly is unwieldy, so we restate it in the higher-level separation-logic notation that Bedrock provides (and we prove that each restatement implies the original). For a concrete example, consider this Hoare triple that we prove for the `read()` function.

\[
\{ \text{buf} \rightarrow \text{len}\} \text{read(sock, buf, len)} \{ \text{buf} \rightarrow \text{len}\}
\]

The notation \(a \rightarrow n\) asserts that memory address \(a\) is the beginning of an accessible memory region of \(n\) bytes.

The `read()` specification is not fully precise. For instance, the file descriptor `sock` is not mentioned at all. We do not model the detailed semantics of network access, instead conservatively modeling reading from the network as non deterministic. Our concern is more with verifying the correct management of resources internal to a single server, so specifications focus on memory layout and transfers of control.

We implemented the system calls mostly in C, with some assembly code to mediate between the Bedrock calling convention and the native conventions of target platforms.

5. Three Verification Design Patterns

The last section discussed the operational semantics that we use to state our final theorem. Recall that programs at any level of Bedrock are annotated with invariant assertions, and we prove that assertions always hold whenever we reach them. We now describe three design patterns we developed in the course of choosing good invariants and good ways of proving that they always hold.

5.1 Defining Higher-Order Recursive Predicates

A challenging question arising in many verification tasks is how should we define recursive, stateful, higher-order predicates? That is, we need self-referential logical predicates that constrain both mutable state and first-class code pointers. Small changes in the formalism may bring large consequences for the ease of proof automation. Past work on verifying a thread library in XCAP[30] faced this challenge by extending XCAP with a new recursive \(\mu\) connective. That specification style turns out to make automation especially challenging, for reasons that we explain shortly.

The natural use of recursive predicates in our setting is for queues of suspended threads. The associated data representation predicate must be higher-order because it deals with code pointers (for suspended threads) and recursive because the preconditions of those pointers must mention the same predicate we are defining. The last condition follows because we expect threads to be able to call into the thread library, whose methods will naturally be specified in terms of the same thread queue predicate.

In this section, we discuss sound encodings of such predicates, using a simpler motivating example, that of a function memo table structure that stores both a function and an optional cached return value for it. We would like to expose our memo table as an abstract data type with an abstraction relation that hides implementation details. The code module implementing memo tables should expose methods for allocating new tables, setting the function stored in a table, and querying the return value of a current function. For simplicity, we consider only stored functions with no arguments and no private state, satisfying deterministic specifications.

Since we model simple pure functions, a number \(n\) serves as a “specification” for such a function, giving the result that it must always return, so it makes sense to define the invariant of memo tables as a predicate `MemoTable\((p, n)\)`, over the pointer \(p\) to the table and the function specification \(n\). Here is a first cut at the
An interesting implementation detail is how we can derive the new connective from the original XCAP connectives. First, a recursive definition of predicate name as P in a module is encoded as a code block labeled \texttt{name} whose specification is \( \lambda x. P(x) \), embedding P into an XCAP assertion in an arbitrary way. Next, we define \texttt{name} \equiv P as \( \{ \lambda x. P(x) \} \texttt{name} \), using the syntactic Hoare double to look up the specification associated with the right code block and assert it equal to a template built from P. One subtle point here is that we want to allow any module to include predicates over any Coq types. The \( \forall \) quantifiers in the templates above are implicitly annotated with the predicate type, and at some point in a related proof we need to reason from the equality of two such formulas to the equality of their embedded predicates. That reasoning step turns out to require the opaquely named Axiom K [33], which is the only Coq axiom used anywhere in our development.

An important benefit of this new encoding compared to past work on a \( \mu \) connective for XCAP shows up in the definition of \texttt{MemoTable} above: the higher-order quantification over P appears
only in one conjunct of the definition of MemoTable, so only when reasoning about that conjunct do we need to deal with the “incompleteness” of XCAP’s natural-deduction rules for higher-order quantification. Only lines of code in the memo table implementation that actually call the function \( f \) will need to venture into that conjunct, so straightforward proof automation can help us verify the rest of the code automatically, applying the usual manipulations to the connectives. This pattern persists into our verification of thread queues, where manual human proving effort is only necessary for lines of code that resume suspended threads.

5.2 Specifying Layers of Threading Abstractions

In verifying a thread library and its client code, we benefit from several dimensions of modularity. Most obviously, it should be possible to verify client code without knowing library implementation details. However, we also want to take advantage of proof modularity within the library, raising the question, which specification idioms apply effectively to modules within a thread library, where each layer of the library manages suspended threads in some form? Past work with XCAP on cooperative [30] and pre-emptive [11] thread libraries dodges this question by not attempting to verify any actual application code against the libraries, let alone focus on minimizing human effort to verify a new client program. Creating a pleasant client experience requires widening the library specifications, and in turn that widening reveals the benefits of decomposing the somewhat intricate proofs across modules with encapsulated state.

Take as an example our lowest-level component that encapsulates suspended threads: ThreadQueue, providing a FIFO queue of thread entries. Our definition is very close to that from a prior XCAP cooperative threading library [30], but the key difference is in modeling parts of memory shared across threads. That past work forced threads to operate only on their own private memory partitions, modulo the ability to allocate and free memory blocks. Realistic applications need to share some state. For instance, in our final application, the in-memory database is shared across worker threads.

Consider the schematic presentation of (most of) ThreadQueue’s data-structure invariant in Figure 5. We maintain a queue as a linked list of saved instruction pointer/stack pointer pairs. Each saved stack pointer points to some memory region that is private to the associated thread, and there is also a shared memory region described by the global invariant. The trickiness appears in deciding which requirements to assert for the saved code pointers. We face the challenges in recursive predicate definition from the prior subsection, plus additional complexities associated with state, both thread-local and shared.

Figure 6 gives the formal definition that we settled on, which we will explain in pieces.

Though many single-threaded data structures have obvious notions of correctness, a concurrent structure like the ThreadQueue raises questions of how we specify the rules for sharing of state across threads. Many options are possible, and we adopted one simple choice here. Our specifications would be quite inflexible if we took literally the idea of a global invariant, or a fixed property that must always hold of shared memory. Invariants may evolve over time. For instance, in the overall thread scheduler, we want the invariant to change as the set of open files expands, since the invariant ought to enforce that certain pointers only go to valid open files.

We formalize a more general shared-state regime using the idea of monotonically evolving state. A client programmer of ThreadQueue is allowed to choose a set of worlds \( W \), such as “sets of open file pointers.” Furthermore, an evolution relation \( \sqsubseteq \) must be provided, to characterize how the world is allowed to change over time. This relation must be reflexive and transitive, but no further restrictions are enforced. Monotonic evolution of state is not sufficient for stating all of the invariants one might care about, like that there are no dangling references to closed file records, but it provides a useful starting point for investigating logical interfaces to expose to applications. (In higher layers of our stack, we dodge the need to rule out dangling file references by recycling freed file records in a free list and keeping dummy data in any free records.)

The global invariant may now be given as a predicate over \( W \) and the root pointer of the ThreadQueue. Concretely, the ThreadQueue module in Coq is a functor in the sense of ML module systems [24]. Its parameters are \( W, \sqsubseteq, \) and the global invariant predicate \( \text{ginv} \), plus some logical axioms that must be proved about them (e.g., transitivity).

This use of worlds and an evolution relation is inspired by the various semantic techniques based on Kripke structures, such as Kripke logical relations for compiler verification [19]. Binary state-evolution relations for concurrent program verification have a long history, including in rely-guarantee proof methods [20], which have been used both for Hoare-logic-style proofs and in concert with refinement types [13]; and in the two-state object invariants of VCC [17].

As an example of one of the proved specifications for ThreadQueue’s methods, here is the specification for \( \text{yield}() \).

\[
\forall w \in W. \{ \text{tq}(w, q) \ast \text{ginv}(w, q) \ast \text{malloc} \}
\]

\[
\text{yield}(q) \}
\]

\[
\exists w' \in W. w \sqsubseteq w' \land \text{tq}(w', q) \ast \text{ginv}(w', q) \ast \text{malloc} \]

The specification is in the style of separation logic [33], which uses the \(*\) operator to indicate by \( P \ast Q \) that memory can be split into two disjoint regions, one satisfying \( P \) and the other satisfying \( Q \). We write \( \text{tq} \) for the invariant sketched in Figure 5, \( \text{ginv} \) for the global invariant predicate, and \( \text{malloc} \) for the free-list data-structure invariant of our \( \text{malloc} \) library. Universal quantification over \( w \) captures the abstract global state at the point where \( \text{yield}() \) is called, while in the postcondition, existential quantification over \( w' \) introduces the new, possibly evolved, world that characterizes state when the yielding thread is finally resumed.

Note that this specification hides an important complexity: \( \text{yield}() \) does not behave as a normal function, but rather grabs the function return pointer and stores it in a heap data structure. Nonetheless, the specification looks like one for a normal function, and we successfully encapsulate the details of code-pointer manipulation within the correctness proof for ThreadQueue.

Verification of methods like \( \text{yield}() \) depends on giving a proper definition of \( \text{tq} \), drawing on the recursive definition approach from Section 5.1. First, we define what is a valid suspension of a paused thread in Figure 6 via a predicate \( \text{ssuspend}(w, q, p, s) \) over world (at time of suspension) \( w \), thread queue pointer \( q \), saved stack counter \( p \), and saved stack pointer \( s \). The definition starts by using Section 5.1 to trick to obtain in local predicate variable \( P \) a reference to \( \text{tq} \). Next, we quantify existentially (and impredica-
tq(w, q) \triangleq \exists I, \text{tq}(q) \land \text{backupStack}(s') \times \text{queue}(I, q') \times \bigotimes_{(p, q) \in \gamma} \\text{susp}(w, q, p, s)

Figure 6. The thread queue predicate and an auxiliary definition

tively) over another predicate variable \( I \) capturing the suspended thread’s local memory invariant. We assert that this invariant does indeed describe \( m \), the partial memory that is the implicit parameter to a separation-logic predicate like \( \text{susp} \).

The formal definition is a semantic Hoare double ascribing a precondition to the saved program counter. Here \( \gamma \) is the Bedrock IL state at the point where the thread is resumed, we require that the stack pointer of \( \gamma \) has been restored to the saved value \( s \). Next, we require that the world \( w \) has evolved legally to some \( w' \). A separation-logic assertion applied to \( \gamma \) breaks memory into four pieces, satisfying the local invariant \( I \), the thread queue invariant \( \text{tq} \) (referred via its local name \( P \)), the global invariant in the new world, and the \( \text{malloc} \) library invariant.

Figure 6 also gives the final definition of the \( \text{tq} \) predicate, for parameters current world \( w \) and queue pointer \( q \). We quantify over a bag (or multiset) \( I \) of suspended threads, represented as pairs of saved program counters and stack pointers, as well as over two values \( q' \) and \( s' \) that \( q \) points to. The pointer \( q' \) is to a more primitive queue structure, while \( s' \) points to the backup stack used while deallocating the current thread in exit ( ). Multisets form the interface with queues via the queue predicate. The most interesting part of the specification is its last conjunct, which uses \( \bigotimes \) as a notation for iterated separating conjunction over some index bag. Here we generate one \( \text{susp} \) conjunct per suspended thread in \( I \).

All the quantifiers appearing directly in \( \text{tq} \)’s definition are first-order, since their domains do not involve specifications. As a result, this definition stays within the fragment of XCAP assertions that support simple proof automation, using connectives that have both introduction and elimination rules. In contrast, \( \text{susp} \) uses second-order quantification, so less clean proofs are required to reason about indirect jumps to suspended threads. Fortunately, only four such jumps appear in the ThreadQueue implementation, and we are able to apply predictable proof automation for the other 60-some lines of executable code. Past work on verifying a similar library with XCAP [30] used a new \( \mu \) recursive predicate connective that also lacks an elimination rule, and perhaps partly as a result, their proofs were highly manual, using about 10 times more lines of proof to establish a weaker interface for a library with fewer features.

5.2.1 Layering More Instances of the Same Pattern

Not only does the world-based specification style facilitate flexible use of shared state in applications, it also composes nicely in layers within a thread library. To demonstrate, we discuss verifying two further components on top of ThreadQueue: ThreadQueues, which exposes multiple queues corresponding to multiple possible IO events; and Scheduler, which exposes standard blocking IO operations on TCP sockets. To our knowledge, ours is the first work to consider a certified implementation of blocking IO in a modular verification framework.

First consider ThreadQueues, which abstracts over a dynamically growing set of thread queues. It provides our first chance to take advantage of the generality of ThreadQueue’s handling of global invariants. Specifically, ThreadQueues asks the programmer to choose \( W \) and \( \subseteq \) like before, though now \( \text{ginv} \) is a predicate over \( W \) and bags of queues, where the latter component was just a single queue for ThreadQueue. For any such choices by the programmer for the parameters of ThreadQueues, we must come up with appropriate parameters to use internally for ThreadQueue. Concretely, both of these modules are ML-style functors, and ThreadQueues instantiates the ThreadQueue functor with derived parameters based on its own parameters.

The main idea is to define a ThreadQueue-level world to contain not just a ThreadQueues-level world, but also the set of all queues that have been allocated so far. To formalize this notion for given \( W, \subseteq \), and \( \text{ginv} \), here is how we derive the ThreadQueue parameters, where \( P(Q) \) stands for the set of sets of queues pointers:

\[
W' \triangleq P(Q) \times W \\
(\tau_1, s_1) \subseteq (\tau_2, s_2) \triangleq \tau_1 \subseteq \tau_2 \land s_1 \subseteq s_2 \\
\text{g} \text{inv}'((\tau, s), q) \triangleq \text{g} \text{inv}(\tau, s) \times \bigotimes \text{tq}(\tau, s), q')
\]

We focus on explaining the \( \text{g} \text{inv}' \) definition, which formalizes what a particular thread queue expects to see in the remainder of memory. The definition above says “a thread queue sees the global invariant plus all of the other thread queues.” The “other” part is encoded via the \(-\) operator in the index bag of \( \bigotimes \). This sort of explicit capture of all library state may seem verbose compared to state-hiding approaches like anti-frame rules [31] and dynamic locks with invariants [15] for separation logic. An advantage of our approach here is that it may be developed as a “specification design pattern” on top of XCAP’s simple Hoare logic, with no requirement to build supporting features into the trusted base, while state-hiding approaches seem to require such support.

Thankfully for the application programmer, these details are encapsulated inside the ThreadQueue correctness proof. In the end, we export method specifications that refer to an abstract predicate \( \text{tq}s \), whose arguments are a world value and a set of available queues. Here is the revised specification for \( \text{yield}() \) at the ThreadQueues level, for enqueuing a thread into \( \text{inq} \) while dequeuing a thread to resume from \( \text{outq} \).

\[
\forall w \in W, \tau \in P(Q). \{ \text{inq} \in \tau \land \text{outq} \in \tau \land \text{tq}s(w, \tau) \times \text{g} \text{inv}(w, \tau) \times \text{malloc} \} \\
yield(\text{inq}, \text{outq}) \\
\{ \exists w' \in W, \tau' \in P(Q). w' \subseteq w \land \tau' \subseteq \tau' \land \text{tq}s(w', \tau') \times \text{g} \text{inv}(w', \tau') \times \text{malloc} \}
\]

We come finally to Scheduler, which encapsulates ThreadQueues along with other state to provide both thread management and blocking TCP IO. Scheduler is parameterized over a choice of a simple global invariant whose only argument is \( \tau \), a set of available file record pointers. We thus instantiate ThreadQueues with \( W \) as the set of sets of file pointers, \( \subseteq \) as the subset relation, and \( \text{g} \text{inv} \) as the invariant that we sketch later in Figure 6. Our choice of this particular interface prevents the final clients of Scheduler from choosing global invariants that depend on custom world concepts. The proof architecture would support a stronger interface preserving such freedom, but we decided to compromise flexibility in favor of more straightforward proof automation.

Now we arrive at a specification for the final repackaging of \( \text{yield}() \), where \( P(F) \) is the set of file record pointers:

\[
\forall \tau \in P(F). \{ \text{sched}(\tau) \times \text{g} \text{inv}(\tau) \times \text{malloc} \} \\
yield() \\
\{ \exists \tau \in P(F). \tau \subseteq \tau \land \text{sched}(\tau) \times \text{g} \text{inv}(\tau) \times \text{malloc} \}
\]
The final interface hides all details of code-pointer manipulation, using just an abstract predicate to capture the state of the scheduler library. The four layers ThreadQueue, ThreadQueues, Scheduler, and application are truly modular, where proofs in each layer treat only the next lower layer, only through Hoare triples for methods specified with abstract predicates. We may tweak each layer without needing to reexamine proofs for others, so long as we maintain its formal interface. For instance, the specifications of the lower-level components are general enough to allow us to hone the data structures of the scheduler without modifying any other module or its proofs.

The reader may have noticed an apparent weakness in our functor-based approach to specifying global invariants: when multiple applications are sharing a Scheduler instance, they must somehow coordinate to choose a mutually satisfactory global invariant. However, it is fairly straightforward to build a little plumbing that still allows each application to be proved modularly, when global state of each application can be kept in a distinct region that other applications may not touch. Each application (with proof) is implemented as a functor over the part of the global invariant contributed by other applications, and the overall global invariant is denoted as the conjunction of the part known locally and the part passed as functor parameter. A single global invariant may be seen through a different lens of this kind for each application. Variants of this approach may also support different modes of state-sharing between applications. The point is that the Scheduler specification allows these modularity disciplines to be constructed on top, with Scheduler’s implementation and proof remaining a black box.

One last note may be worthwhile, about the malloc predicate used throughout our example specifications. Why does this predicate not take some kind of state-tracking argument, too, which we are forced to propagate through the spec of Scheduler? For instance, one might expect to see malloc parameterized over a description of which memory blocks have been allocated so far, so that we can require that any block being freed really did come from a past allocation. Our malloc library dodges this complication by allowing any block of memory (disjoint from the current malloc state) to be donated via “deallocation,” even if it previously had nothing to do with malloc.

5.3 Patterns for DSL Implementation

Many verification tasks are practical to carry out over code in low-level languages like C, but, in scaling to larger systems, we may realize large productivity improvements by coding in and reasoning about higher-level languages. Ideally we raise the abstraction level while keeping our final theorems at the level of assembly code. The relevant question is, what is the right way to incorporate a verified DSL implementation within a modular proof?

In particular, we focus on how the DSL implementation and proof may themselves be factored into modules. We developed our DSL, the Bedrock Web Services Language (BWS), modularly at the level of language constructs. BWS supports the mostly orthogonal features of relational database access and XML processing. Each of these factors is further decomposed into constructs, like database querying vs. modification, and XML pattern-matching vs. generation. Each construct is implemented with a separately encapsulated proof. We hope these building blocks are suitable to assemble in different ways to build different DSLs, but we also found that modularity paid off, even just for the construction of this one DSL.

The Bedrock Structured Programming System [5] already provides a notion of certified low-level macro, where we package a code generator with a Hoare-logic proof rule and a proof that the two are compatible. Thus, it was natural for us to implement our DSL building blocks as BSPS macros.

The Bedrock module formalism does not currently support statically allocated data, such as for string literals extracted from source code. No doubt a more practical evolution of the framework would add such a feature, but we can take advantage of the omission to motivate an example macro, for checking whether a byte array matches a string literal. In other words, we effectively represent static character arrays by inlining appropriate unrolled loops into assembly code.

BSPS includes basic combinators for, e.g., sequencing, conditionals, and loops. Merely composing these combinators and their proof rules leads to a derived proof rule that may expose too much code detail. We often want to wrap a macro, applying an equivalent of the rule of consequence to assign a more abstract proof rule. The BSPS design includes a basic wrapping combinator, which exposes a view of programs at the level of Bedrock IL control-flow graphs, with no concept of local variables. We found that interface to be too cumbersome for the more abstract macros that interest us in this project, so we built a higher-level wrapping combinator that allows the preconditions and postconditions of chunks to mention the set \( \mathcal{V} \) of declared local variables.

To illustrate, consider our example of a macro to check whether some substring of a string equals a constant. Let us fix four program variables: \( \text{str} \), for a pointer to the string we should check; \( \text{len} \), a variable holding its length; \( \text{pos} \), indicating the first index of the substring to check; and \( \text{output} \), a variable where we write either 0 or 1, to indicate the Boolean result of the test. In Figure 7, a simple recursive definition of a function \( \text{SE} \) suffices to generate the code, on top of more primitive BSPS macros.

We implement definitions like this one in Coq’s functional programming language Gallina. The \([…]\) syntax indicates antiquoting some computed code pieces within BSPS syntax. We define the overall macro via \( \text{StringEq(s) ≜ SE(s, 0)} \).

BSPS automatically builds a sound Hoare-style proof rule for any concrete application of StringEq. However, the rule reveals the exact low-level code structure, when we would prefer a more abstract presentation, encapsulating details like checking array bounds. Here is the rule we derive with our expanded wrapping combinator; we will step through explaining it in stages. We make the four variable names (e.g., \( \text{str} \)) into explicit parameters of the macro (e.g., \( \text{str} \)), so that client code may choose appropriate concrete names for its context.

\[
\forall I, P. \{ P \land \text{array8}(I, \text{str}) \land |I| = \text{len} \land \text{pos} \leq \text{len} \\
\land \{ \text{str}, \text{len}, \text{pos}, \text{output} \} \subseteq \mathcal{V} \\
\land \text{NoDup}([\text{str}, \text{len}, \text{pos}, \text{output}]) \land |s| < 2^{32} \\
\land \text{FV}(P) \cap \{ \text{output}, \text{pos} \} = \emptyset \}
\]

\[
\text{StringEq(str, len, pos, output, s)} \\
\{ P \land \text{array8}(I, \text{str}) \land |I| = \text{len} \land \text{pos} \leq \text{len} \}
\]

Most of the precondition and postcondition are heap-independent facts combined with \( \land \). The only heap-dependent parts of the predicates are \( P \), some frame predicate; and \( \text{array8}(I, \text{str}) \), indicating that the requested variable \( \text{str} \) points to a byte array whose contents are \( I \). These parts repeat verbatim before and after, indicating that

\[
\text{SE("", offset) ≜ output} \leftarrow 1 \\
\text{SE(c :: s, offset) ≜ if (pos + [offset] < \text{len}) \land \&\& \text{str}[pos + [offset]] = [c]} \{ \\
\{ \text{SE}(s, offset + 1) \} \} \text{ else } \{ \text{output} \leftarrow 0 \}
\]
the macro may only produce code that has no heap effects (or fails to terminate).

Most of the heap-independent parts of the precondition are variable-related side conditions, of the sort familiar to users of separation logic. We require that the variable parameters are all members of the set $V$ of declared local variables, and we also require that no variable name is duplicated as the value of multiple parameters.

The last precondition conjunct imposes a condition on the free variables of the frame predicate $P$, which is inspired by conditions in the classic separation-logic frame rule. That rule assumes \{P \land \exists c (Q)\} and concludes \{P \land F \land \exists c (Q \land \neg F)\} for an arbitrary $F$ that does not mention variables that command $c$ could modify. The classic rule is able to inspect $c$ syntactically to determine which variables it may write, but in the macro setting we want to hide code details through encapsulation. Thus, we expose a formal interface for StringEq that in effect promises not to change any outside the set \{output, pos\}, by only allowing useful postconditions with $P$ that does not depend on those variables.

This technique generalizes to macros that take blocks of code as arguments. For instance, consider some macro Iterate(buf, len, $k$) implementing a foreach loop over all cells of a buffer (indicated by base-pointer variable buf and length variable len), running some body code $k$ for each. Where $\mathcal{V}$ is some set of program variables that the macro needs to use in generated bookkeeping code, we might assign this proof rule:

$$\forall I. \{P \land \text{array8}(I, \text{buf}) \land |I| = \text{len}\}$$

$$k \{P \land \text{array8}(I, \text{buf}) \land |I| = \text{len}\}$$

$$\forall I. \{P \land \text{array8}(I, \text{buf}) \land |I| = \text{len}\}$$

$$\land \{\text{buf}, \text{len}\} \cup \mathcal{V} \subseteq \mathcal{V} \land \forall (P) \land \forall (V) = 0\}$$

Iterate(buf, len, $k$) \{P \land \text{array8}(I, \text{buf})\}

Let us read the conclusion first. We assert a particular Hoare triple for the macro, for arbitrary choices of buffer contents $I$ (which should remain unchanged by execution). As before, two pieces of state appear identically in precondition and postcondition: a frame predicate $P$ and the buffer itself via the array8 predicate. We require that variable len hold an accurate array length, that all variables we need are included in the available variables $\mathcal{V}$, and that the frame predicate $P$ does not depend on any of the bookkeeping variables $\mathcal{V}$ that may be modified.

The premise formalizes expectations for the body code block $k$. We require that a Hoare triple is valid for $k$, again for any buffer contents $I$. Specifically, $k$ must preserve a familiar-looking statement about $P$, buf, and len. Technically, $k$ might change the values of these variables, but at least the new versions must still satisfy the invariant that buf points to an array whose length is len. Note that $P$ may be chosen to include any state known only to $k$ and not to the Iterate macro.

We want to emphasize one key element of rules like this one: the body $k$ may be instantiated with arbitrary assembly code that can be proved semantically to satisfy the Hoare triple. That is, in coding and verifying our DSL building blocks, we need not commit to the DSL design in advance. Subsets of our building blocks ought to be usable to construct different verified DSL implementations, though so far we have only assembled them into the DSL of Section 5.

6. A Thread-Library Component Stack

This section overviews our concrete architecture for a modularly verified (cooperative) thread library that exposes a rich interface to client code. Sections 5.1 and 5.2 presented the two key novel design patterns that enabled this architecture, and we show the details in Figure 8.

We follow visual conventions introduced in Figure 4. Mixed dotted and dashed borders group elements of state within a single encapsulation boundary, to aid understanding. Arrows indicate pointers, which terminate at precise memory elements, when they do not span encapsulation boundaries; or terminate at the edges of dashed boxes, when pointing to abstract objects provided by other modules. Basically, this picture shows how to take a snapshot of memory in a running system and divide it into pieces encapsulated within separately verified modules.

The bulk of the diagram portrays state of the Scheduler, which provides functions for thread creation and destruction, TCP connection management, and blocking IO. These functions are built on top of the nonblocking IO system calls from Figure 2. The Scheduler is implemented via a number of data structures, including three more encapsulated components (sketched in Section 5), but an abstract interface is exported for use by client code.

The Scheduler manages both threads and open files. A Scheduler instance is rooted in a four-element struct, containing ready, a pointer to a queue of suspended threads ready to be resumed; free, a pointer to a pool of available structs to use in representing open files; wait, a pointer to a table to be consulted in interpreting the results of polling via the wait() system call; and waitLen, the length of the wait array.

The Scheduler maintains a pool of file records, which are recycled after previous open files are closed. Each file record includes an underlying file descriptor and pointers to two queues of suspended threads, one for threads waiting to read from the file and the other for threads waiting to write. In any blocking IO call associated with one of these file records, a pointer to the appropriate one of the two queues is saved in the wait array, at an offset equal to the reservation assigned to this request by the declare() system call. Thus, when a wait() system call returns, its return value may be treated as an index into wait, determining which thread is woken up to handle the new IO event. Since the trusted specs of declare() and wait() are simple and nondeterministic, we do dynamic checking of reservations returned by wait(), to make sure they are in-bounds for the wait array. We also check that the
Figure 9. An example client program of the thread library

```
bfunctionNoRet "handler"("buf", "listener", "accepted", "n", "Sn") [handlerS]
  "listener" ← Call "scheduler"("listen"(port))
  [V fs, PREEvery[V, R | R ∈ fs] & sched fs = mallocHeap0];
  "buf" ← Call "buffers"("mballoc"(inbuf_size))
  [V fs, PREEvery[V, R | R ∈ fs] & sched fs = mallocHeap0];
  "accepted" ← Call "scheduler"("accepted"("listener"))
  [V fs, PREEvery[V, R | R ∈ fs] & V "buf" → bsize × (freeable (V "buf") inbuf_size) & V "listener" ∈ fs & sched fs = mallocHeap0];
  "n" ← Call "scheduler"("read"("accepted", "buf", bsize))
  [V fs, PREEvery[V | V "accepted" ∈ fs] & V "buf" → bsize × (freeable (V "buf") inbuf_size) & sched fs = mallocHeap0];
  "Sn" ← "n" + 1;
  Call "scheduler"("close"("accepted"))
  [V fs, PREEvery[V | V "buf" → bsize × (freeable (V "buf") inbuf_size) & V "listener" ∈ fs] & sched fs = mallocHeap0 0 = [V "Sn" = V "n" + $1] ];
  Call "scheduler"("close"("listener"))
  [V fs, PREEvery[V | V "buf" → bsize × (freeable (V "buf") inbuf_size) & sched fs = mallocHeap0]$1 = [V "Sn" = V "n" + $1] ];
  Call "buffers"("mbfreecake"("buf", inbuf_size))
  [V fs, PREEvery[V] & sched fs = mallocHeap0];
  Call "sys"("printInt"("Sn"))
  [V fs, PREEvery[V] & sched fs = mallocHeap0];
  Exit 100
end

Ltac t := try solve [ sep unf hints; auto ];
  unf; unfold localsInvariantMain; post; evaluate hints; descend;
  try match locals; sep unf hints; auto.

Theorem ok: moduleOk.
Proof.
  vcgen. abstract t.
Qed.
```

corresponding array cell has been initialized with a non-null queue pointer. Failure of either check signals a top-level runtime program error.

The concept of threads in Scheduler is lighter-weight than in most thread libraries. There is no persistent data structure in the library standing for a thread. Instead, we think of each queue as storing continuations allocated opportunistically as "threads" decide to block. The notations in our specification style so far have been hiding details of call-stack representation. Effectively, we require each thread to have a valid stack at the point where it calls any library function, but it is legal for thread code to grow and shrink its stack through manual reallocation between blocking operations. When such an operation is triggered, the associated stack is folded into the idea of local state for the continuation that we save.

Figure 9 shows an example of a client program verified against the thread library. We refer the reader to the BSPS paper[5] for more detail on the C-like programming language used here. At a high level, the code alternates between executable statements and invariants enclosed in square brackets. The former are mostly assignments and function calls. We prove that every invariant holds every time it is reached.

This program accepts a TCP connection on a port, reads some bytes from that connection, and then stashes in a local variable a value one greater than the number of bytes read. After a few more calls into the thread library, that saved value is printed. Invariants ensure that the proper value is printed in the end, modulo the non-determinism of our specs for the underlying IO operations. For instance, the later invariants include clauses V "Sn" = V "n" + $1, asserting that the value of local variable Sn equals the value of variable n plus one, in the domain of 32-bit machine words. Other interesting conjuncts appearing throughout the function include sched fs, representing the state of the scheduler with open-file set fs; and V "accepted" ∈ fs, indicating that local variable accepted holds a file pointer belonging to that set.

Figure 10 shows an excerpt from a real Coq source file, ending in the full code to prove that invariants are never violated. The proof is almost entirely automated, appealing to the separation-logic proof procedures that Bedrock provides[25]. This level of automation persists in verifications of more featureful client code, and it represents a qualitative improvement to the ease of building on verified systems libraries, compared to past work.

7. A Library of Verified Compiler Features

We summarize our architecture for modular verification of a DSL implementation for database access and XML processing. Section 5.3 introduced the basic design patterns we use for splitting a language implementation into a set of certified macros, whose definitions and proofs are independent of the full language that they will be used to implement. Recall that we decided to verify our thread library at the level of functional correctness, and verify client applications mostly at the level of data-structure shape invariants. Figure 10 shows how we decomposed the verification effort at that level for the implementation of our DSL, the Bedrock Web Service Language (BWS). Arrows indicate dependencies between modules.

The two core sets of features are relational database access and XML processing. We decompose the former into macros for filtering (DbCondition), querying (DbSelect), adding (DbInsert), and removing (DbDelete) database rows; and the latter into macros for XML pattern-matching (XmlSelect) and generation (XmlOutput). All of the above are combined into a macro XmlLang, whose parameter is a syntactic BWS program in an AST type. We are able to write the equivalent of a type-checker as a functional program over these ASTs. The final theorem for our main BWS macro establishes that any type-correct program is compiled to assembly code that maintains the invariants of the database and does not interfere with the state of other components.

Figure 11 shows an example BWS function definition from our ROS Master Server. Any such function is callable remotely over HTTP, following the XML-RPC protocol. This particular function installs a key-value pair in the configuration store that the Master maintains. One complication is that ROS nodes are allowed to subscribe to updates on particular keys, and we must initiate further
HTTP callbacks to all interested parties. In order, the body commands of this function delete any old mapping for the key, insert the new mapping, loop over callbacks to all nodes subscribed to updates, and return a response document to the calling node. The code here uses special ROS-specific Coq notations to hide the details of XML parsing and generation, but we use BWS macros for those functions under the hood. We show the complete program-specific correctness proof at the end of the figure. This proof actually applies to the full set of 20 methods in the Master. It works by calling a BWS library tactic \( \text{wf} \) for discharging program well-formedness obligations automatically. We only need that sort of shallow fact to instantiate the generic correctness proof of the BWS implementation, yielding an assembly-level result that can be composed with the main theorem about our thread library. The generated main() function spawns a set of worker threads servicing HTTP requests.

8. Empirical Results

The premise of this paper is that some lessons about modular verification can only be learned by building and deploying systems at realistic scales. Our ROS Master Server application has both an assembly-level Coq proof and real users without formal-methods expertise. In this section, we summarize the human cost of carrying out our case study, the run-time performance of our application, and its usability.

8.1 Developing the Proofs

Overall, we spent about 2 person-months implementing and verifying the thread library, and then another 2 person-months doing the same for the BWS implementation. The ROS Master Server itself is trivial to code and verify with that infrastructure in place. The thread library includes about 600 lines of implementation code and about 3000 additional lines to state and prove theorems, making for a ratio of verification code to implementation of about 5:1. The BWS implementation includes about 1000 lines for actual executable code generation plus about 20,000 lines for verification, for a ratio of about 20:1. We ascribe the higher overhead in the latter case to the greater “meta-ness” of a compiler, where the code has a higher density of interesting operations. These ratios in general compare reasonably to related projects, like L4 verified [22] with a ratio of 20:1 in a project over 2.2 person-years (with non-modular proofs); and past work in modular verification of thread libraries [11, 30] achieving ratios on the order of 100:1, spend-

Figure 11. Example BWS function with proof

Figure 12. Throughput of our Web server (static content)

Our general workflow is to verify each component fully before testing it. We have been pleasantly surprised that not a single bug has been found in the systems components through testing; verification caught every mistake we have found so far. Testing has revealed about 10 bugs in the application-level components, for which we prove only the sorts of invariant annotations from Figure 9. In each case, we corrected the implementation and found that our automation adjusted the proofs without any need to edit proof code manually.

The current biggest practical obstacle to our verification methodology is the performance of the proof automation (taking several hours to check our whole stack), which we hope to ameliorate in the future by running proof search in parallel on compute clusters.

8.2 Running the Code

We have not attempted to build a competitive high-performance Web server, but we wanted to make sure that we had not simplified our program code too much during verification, in a way that led to performance too poor to support any real use. Therefore, we ran a basic load test experiment between two servers in the same cloud data center. Our benchmark input is the full set of static files from a recent snapshot of a conference Web site. There are about 60 different files, with one outlier at size about 3 MB and the rest each smaller than 1 MB, with several HTML files of only a few kB each. Our experiment compares our verified server against the Apache server from Ubuntu 13 with default configuration. We load-test each server by launching on one of the two machines a set of concurrent processes to send repeated requests to a server running on the other machine.

Figure 12 shows the results of our experiment, comparing the throughput of Apache and our verified server running either 1 worker thread or 10. As expected, the single-threaded server fails to scale throughput well with increased request concurrency. In contrast, our 10-thread verified server keeps up with Apache. Here the network link and the underlying Linux implementation are the real limiting factors. Our coarse experimental results show just that we (and Apache) attain sufficient performance to saturate that substrate. If anything, the results are skewed in favor of Apache, which runs multiple hardware threads, while the Bedrock model is inherently single-core. We can serve real Web sites at rates that will not aggravate users, where to our knowledge previous mechanized verifications of applications had not yielded any such practical results.

The real test of our case-study program is the reactions of its real users. The robotics experts who helped us with our test deployment pay little attention to verification details and are much more interested in how our Master server performs in practice. In the demo set-up, ROS nodes split across the two computers (robot and control terminal) include those associated with a joystick, wheel controls, and sensors including GPS and radar. Where the traditional Python Master server leads to a 2-second start-up process for the robot to enable driving it with the joystick, our BWS server requires 8 seconds, which our users deem tolerable in a high-assurance context. Bedrock-based programs should have an inherent performance advantage over interpreted Python, thanks to specialized generation
of assembly code doing manual memory management. However, our very elementary database implementation, which does not use any kind of index structures, should bring an offsetting asymptotic performance disadvantage, compared to the standard dictionaries used to store corresponding data in the Python server. We also, to make proofs more straightforward, simplified some aspects of our code in ways that hurt performance, like always allocating certain buffers at a fixed size and zeroing them out in full per request, even when network inputs do not take up the full space; and making copies of buffers before sending them, to deal with a library specification weakness that we explain in the next paragraph. We believe that the performance differential stems from these coding patterns.

In the process of building and deploying the application, we noticed a few weaknesses in the thread-library formal interface. For instance, the blocking `write()` function is assigned a specification like:

\[
\begin{align*}
\{ \text{sched } \{ \text{giv} \text{ } \text{buf} \mapsto \text{len} \} \text{write} & (\text{sock, buf, len}) \{ \ldots \}
\end{align*}
\]

We realized late that this specification precludes sending from string buffers that logically belong within the global invariant, such as strings for column values within the database. To compensate, we copy column values into temporary buffers before sending them, incurring additional overhead.

Our server does defend successfully against a security attack that works on the Python server. A carefully crafted XML input\(^1\) leads the Python server to loop allocating an effectively unbounded amount of memory. In contrast, our final theorem establishes a modest static bound on how much memory our server can allocate, by defining the set `ValidMem` (demonstrated in Figure 3) to be small enough.

9. Related Work

Several recent projects have verified systems infrastructure mechanically. L4.verified \(^2\) is the best-known project in this space, based on an Isabelle/HOL proof that a C-language microkernel refines a specification of system calls and so forth, given by a functional program. There has been preliminary work \(^2\) on using the L4.verified proof to establish theorems about applications running on the kernel, but we are not aware of results with applications on the scale of those reported here with our Web service. The Verve project \(^2\) establishes type safety but not functional correctness of .NET programs running on a platform that includes a verified garbage collector. The Verisoft project \(^3\) has also produced kernel correctness proofs, using a decomposition of the proof effort via layering several abstract machines, from hardware to user-level. Theorems have been proved for kernel composed with embedded applications \(^3\) following a simple communication model without dynamic thread creation or freeform calls to IO operations.

Past work has designed verification frameworks to support modular reasoning. For instance, the CAP project has demonstrated several verification frameworks for systems-level programs, including the XCAP logic \(^4\) that we use here.

One project used XCAP to verify a user-level thread library \(^5\) that inspired our new work. Our library follows theirs roughly to the level of ThreadQueue, where their work ends. They also assign their version of this module a weaker specification that does not support sharing of memory across threads in application-specific ways, and they did not report on experience verifying applications against the library.

A follow-on project applied a compositional program logic for hardware interrupts to certify kernel-level code for process scheduling (without dynamic process creation) plus user-level code imple-

---

\(^1\) [http://en.wikipedia.org/wiki/Billion_laugh](http://en.wikipedia.org/wiki/Billion_laugh)

\(^2\) Theorem types

\(^3\) Regular expressions

\(^4\) Verisoft

\(^5\) Ynot project
mous reviewers, for their helpful comments on an earlier version of this paper. This work has been supported in part by NSF grant CCF-1253229 and by DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References


