VSTlib: Library Components for Verified C Programs

Andrew W. Appel

Princeton University

Abstract

C program components verified for functional correctness in Coq using VST (Verified Software Toolchain) can now rely on a set of standard library components (math functions, malloc/free, atomic load/store, locks, threads) that have formal specifications.

The Verified Software Toolchain [1] is a tool and library in Coq to verify functional correctness of C programs. VST has been used to verify many small C programs: SHA-256 hashing, HMAC authentication, HMAC-DRBG random-number generation, generational garbage collection, graph algorithms (Prim, Dijkstra, union-find, etc.), concurrent messaging, an HTTP key-value server, differential equation solver, Reed-Solomon coding, and more.[1] In scaling up from small to medium-size programs, we can use VST's system for modular verification of modular programs: Verified Software Units (VSUs) [4]. These larger programs will need to use *libraries:* standard APIs for system calls, I/O, memory management, math functions, threads, locks, and so on.

We introduce **VSTlib**, a new set of C libraries each with a formal specification compatible with the VSU system. Some of these libraries are proved correct using VST; others are simply interfaces to standard system-call APIs or standard (unverified) system libraries, in which case we take as an axiom that they satisfy the given spec.

VSTlib is available as coq-vst-lib from opam-coq-archive, released in April 2023. Summary documentation is at `https://github.com/PrincetonUniversity/VST/tree/master/lib` and the formal specifications are in VST/lib/proof/spec_*.v in the GitHub repository.

Currently the following libraries are available:

Name	Header Files	P/A	Description
malloc	⟨stdlib.h⟩	Axiomatized	Posix standard malloc/free
atomics	⟨stdatomic.h⟩, SC_atomics.h	Axiomatized	C11 SC-mode atomic load, store, CAS
threads	⟨threads.h⟩, VSTthreads.h	Axiomatized	shared-memory threads
locks	⟨stdlib.h⟩, VSTthreads.h	Proved	semaphore-style daring locks
math	⟨math.h⟩	Axiomatized	Posix standard math library (sin, cos, etc.)

The concurrency libraries (atomics, threads, locks) were axiomatized and proved by William Mansky [7] and packaged into library VSUs by the author. In the future we will install a verified malloc/free system [3] ("Proved" rather than "Axiomatized") to give the user a choice of memory allocators; the VSU system easily supports having different implementations of the same interface.

These libraries have already been used in higher-level verifications. A verified-accurate matrix-vector library [6] uses **math**. A Jacobi-method linear solver [8] uses **math** and **malloc**. A parallel dot-product program (github.com/VeriNum/pardotprod) uses **malloc, atomics,** and **locks**.

Linkage to library modules is easy and straightforward both in C and in Coq. The library comes with an `include` directory (containing `.h` files) to be used with a `-I` parameter to a C compiler or `clightgen`.[2] The `src` directory contains `.c` programs that can be compiled and linked with the user's verified clients. The `proof` directory has Coq library modules, available within Coq via `Import VSTlib.`[...], assuming that one has installed VSTlib via opam.

[1] `https://github.com/PrincetonUniversity/VST/blob/master/doc/catalog-of-examples.md`

[2] When verifying a C program one runs `clightgen` to use CompCert's parser to generate an AST in Coq upon which to run VST-Floyd's symbolic execution.

Acknowledgments. This work was supported in part by NSF grants CCF-2005545 and CCF-2219757.

The math library contains 58 standard Posix math functions such as sin, cos, fma (fused multiply-add), and so on. Each of these is specified to provide a certain level of floating-point accuracy *on each specific target architecture* as documented in the GNU C library manual [5]. Different architecture-specific implementations of the math library have been measured to have different *observed* worst-case accuracy. In the absence of proofs of all these implementations, we will assume these as worst-case *bounds*.[3] So if you install VST with target architecture AArch64, then you'll get a single-precision arctangent function accurate within 1.5 ulp (unit in last place), but on VST configured for x86-32 it'll be specified as accurate to 0.5 ulp.[4] Accuracy specifications for floating-point functions are written using VCFloat's Coq framework for specifying floating-point accuracy of arbitrary-arity user-defined functions [2].

The threads library supports spawn and exit (without thread-IDs or thread-joining, which can be synthesized using semaphores).

The atomics library contains sequentially consistent (SC) atomic operations: alloc, free, load, store, compare-and-swap, exchange, on atomic integers and atomic pointers.

The locks library supports dynamically allocated daring semaphores (i.e., which can be released by a thread other than the one that locks them), programmed in C using the SC atomics and proved correct.

Conclusion. For many years now, CompCert and VST have supported the concept of "external function" with a specification. But previous organizations for linking C programs and their proofs together (as variously embodied in the verifications described at the link given in Footnote 1) were ad-hoc and clumsy.

More recently, the VSU system [4] gave VST a formal semantics for specifying APIs and data abstraction, proved sound with respect to a simple model of linking programs together. This enables VSTlib: a practical, convenient, and extensible basis for library support. We plan to extend it with other commonly used C libraries.

The appendix shows a worked example of a client-program verification making use of VSTlib.

References

[1] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *ESOP'11: European Symposium on Programming*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.

[2] Andrew W. Appel and Ariel E. Kellison. VCFloat2: Floating-point error analysis in Coq. https://github.com/VeriNum/vcfloat/blob/master/doc/vcfloat2.pdf, 2023.

[3] Andrew W. Appel and David A. Naumann. Verified sequential malloc/free. In *International Symposium on Memory Management (ISMM)*, pages 48–59, 2020.

[4] Lennart Beringer. Verified software units. In *30th European Symposium on Programming (ESOP'21), LNCS 12648*, pages 118–147. Springer, 2021.

[5] GNU C Library, §19.7: Known maximum errors in math functions. https://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html, 2023.

[6] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. LAProof: a library of formal accuracy and correctness proofs for sparse linear algebra programs. In *30th IEEE International Symposium on Computer Arithmetic*, September 2023. to appear.

[7] William Mansky. Verifying concurrent programs with VST. https://github.com/PrincetonUniversity/VST/blob/master/doc/concurrency.pdf, August 2022.

[8] Mohit Tekriwal, Andrew W. Appel, Ariel E. Kellison, David Bindel, and Jean-Baptiste Jeannin. Verified correctness, accuracy, and convergence of a stationary iterative linear solver: Jacobi method. In *16th Conference on Intelligent Computer Mathematics*, page (to appear), 2023.

[3] In the future we may be able to install a math library implementation with stronger soundness guarantees.

[4] That is, in *Known Maximum Errors in [GNU Library] Math Functions* [5], error for the `atanf` function in column AArch64 is listed as 1 (which means, "1 ulp in addition to the 0.5 ulp that arises just from rounding") but for i686 it is listed as - (which means "0 ulp in addition to the 0.5 from rounding."

Appendix: Worked Example

We demonstrate a simple tutorial example of a VSTlib client: a program that uses malloc and free.

File test.c:

```
#include <stdlib.h>

int myfunc(int x) {
  int *p = (int*)malloc(sizeof(*p));
  if (p) { /* if malloc succeeded, store x at p, fetch it back, add 1, free p */
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1; /* if malloc failed, compute x+1 without using memory */
}
```

File verif_test.v: We start with the standard boilerplate at the beginning of any VST verification:

```
Require Import VST.floyd.proofauto.
Require Import test. (* import the AST of test.c, as parsed by CompCert *)
#[export] Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
```

Now we import the *specifications* of all the library components that we're using. This program just uses the malloc/free component.

```
From VSTlib Require Import spec_malloc.
```

This file defines MallocASI, the *abstract specification interface* of a malloc/free system—this is basically the specification of an API (*application programmer interface*, of course). That interface refers to MallocAPD, an abstract data type description of the encapsulation of a memory manager's free-list data structure (APD stands for *abstract predicate declaration*).

Our example has an ASI and and APD for a malloc/free system, but any component will have an ASI (description the interface) and (sometimes) one or more APDs describing abstract types.

To make our proof of test.c parametric over all possible implementations of MallocASI, we use Coq's Section/Variable feature to parameterize over an instance M of MallocAPD.

```
Section VERIFICATION.
Variable M: MallocAPD.
Existing Instance M.
```

As usual in VST, we write a *function specification* (funspec) of each C function, listing its PREcondition and POSTcondition. The keyword WITH is a quantifier for Coq variables (x,gv) used in the funspec. The variable x holds the mathematical integer represented in the C function-parameter; the corresponding C identifier is referred to as _x in our Coq verification. The variable gv is a technical trick in VST to give separation-logic predicates access to *global variables* (since each .c file can refer to its own globals).

```
Definition myfunc_spec :=
  DECLARE _myfunc
  WITH x: Z, gv: globals
  PRE [ tint ]
     PROP (Int.min_signed <= x < Int.max_signed)
     PARAMS (Vint (Int.repr x)) GLOBALS (gv)
     SEP (mem_mgr gv)
   POST [ tint ]
     PROP()
     RETURN(Vint (Int.repr (x+1)))
     SEP(mem_mgr gv).
```

The PREcondition says, the function parameter's C-language type is int (that is, tint refers to the deep-embedded description of the C type). The PREcondition says, the value of x must be less than the maximum signed integer value (so adding 1 to it will not overflow). It says, the C parameter contains a 32-bit representation of x.

This is SEParation Logic, so the PREcondition must say what memory resources it relies upon; in this case, just the memory-manager's own data structures of free-lists of various-sized object, which is all encapsulated in the mem_mgr predicate. That predicate is built from ordinary separation-logic predicates (see [3]), but here it is fully abstract: mem_mgr gv really means @mem_mgr M gv, which selects one field of the M record; but since M is a Variable (i.e., a parameter), that means it's impossible for the verification of test.c to depend on which implementation of the memory manager it's using.

The POSTcondition returns an integer value (hence, [tint]); the return value is a 32-bit representation of x+1; and the memory manager is still in a consistent state.

When verifying function-bodies in VST, one makes a list of all the function-specs in the program on which our function-bodies might depend. What's new with VSUs is that this list contains not only the locally verified functions, but also the concatenation of all the abstract specification interfaces (ASIs) of the imported components:

```
Definition Gprog := [myfunc_spec] ++ MallocASI.
```

The verification of each function body, when using VSTlib, looks just the way it would if the functions being called (such as malloc) were local instead of external.

```
Lemma body_myfunc: semax_body Vprog Gprog f_myfunc myfunc_spec.
Proof.
 start_function.
 forward_call (malloc_spec_sub(tint)) gv.
 Intros p.
 if_tac. (* is p equal to NULL? *)
 - (* p == NULL *)
   forward_if; subst; try contradiction.
   forward.
 - (* p <> NULL *)
   forward_if; subst; try contradiction.
   Intros.
   forward.
   forward.
   forward.
   forward_call (free_spec_sub (tint)) (p,gv).
   if_tac; try contradiction; cancel.
   forward.
Qed.

End VERIFICATION.
```

The user of VST can now package this verification of client test.c as a VSU (which we will not show here), and link all the VSUs together (test.c and libraries) for a whole-application verification.