Modular Verification of C Programs in Verifiable C

Lennart Beringer and Andrew W. Appel

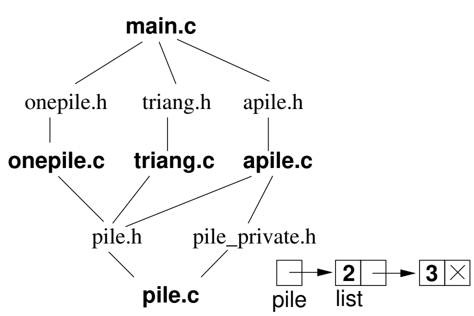
July 2, 2019

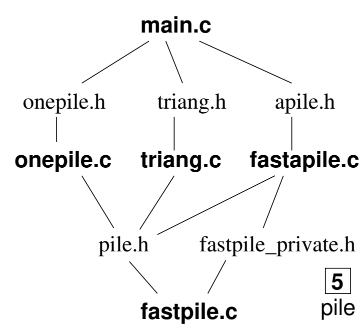
Abstract. C programs are broken into modules (.c files) that import (call upon) functions from each other. VST verifications can follow the modular structure of the program. This tutorial shows how.

We assume the reader is familiar with the use of Verifiable C to prove functional correctness of single-file C programs. Here we illustrate the modular verification of modular C programs.

Our main example is an abstract data type (ADT) for *piles*, simple collections of integers. The complete example (C code and Coq verification) can be found in the VST distribution (or github repo), in directory progs/pile.

Figure 1 (on the next page) shows a modular C program that throws numbers onto a pile, then adds them up.





The diagram at left shows that pile.c is imported by onepile.c (which manages a single pile), apile.c (which manages a single pile in a different way), and triang.c (which computes the nth triangular number). The latter three modules are imported by main.c. Onepile.c and triang.c import the abstract interface pile.h; apile.c imports also the low-level concrete interface pile_private.h that exposes the representation—a typical use case for this organization might be when apile.c implements representation-dependent debugging or performance monitoring.

When—as shown on the right—pile.c is replaced by a faster implementation fastpile.c (code in Figure 3) using a different data structure, apile.c must be replaced with fastapile.c, but the other modules need not be altered, *and neither should their specification or verification.*

Figure 2 presents the specification of the pile module, in the Verifiable C separation logic. Each C-language function identifier (such as _Pile_add) is bound to a funspec, a function specification in separation logic.

```
/* pile.h */
typedef struct pile *Pile;
Pile Pile_new(void);
void Pile_add(Pile p, int n);
int Pile_count(Pile p);
void Pile_free(Pile p);

/* onepile.h */
void Onepile_init(void);
void Onepile_add(int n);
int Onepile_count(void);

/* apile.h */
void Apile_add(int n);
int Apile_count(void);

/* triang.h */
int Triang_nth(int n);

/* triang.c */
#include "pile.h"
int Triang_nth(int n) {
  int i,c;
  Pile p = Pile_new();
  for (i=0; i<n; i++)
    Pile_add(p,i+1);
  c = Pile_count(p);
  Pile_free(p);
  return c;
}

/* onepile.c */
#include "pile.h"
Pile the_pile;
void Onepile_init(void)
 {the_pile = Pile_new();}
void Onepile_add(int n)
 {Pile_add(the_pile, n);}
int Onepile_count(void)
 {return Pile_count(the_pile);}
```

```
/* pile_private.h */
struct list {int n; struct list *next;};
struct pile {struct list *head;};

/* pile.c */
#include <stddef.h>
#include "stdlib.h"
#include "pile.h"
#include "pile_private.h"
Pile Pile_new(void) {
  Pile p = (Pile)surely_malloc(sizeof *p);
  p->head=NULL;
  return p;
}
void Pile_add(Pile p, int n) {
  struct list *head = (struct list *)
      surely_malloc(sizeof *head);
  head->n=n;
  head->next=p->head;
  p->head=head;
}
int Pile_count(Pile p) {
  struct list *q;
  int c=0;
  for(q=p->head; q; q=q->next)
    c += q->n;
  return c;
}
void Pile_free(Pile p) { . . . }

/* apile.c */
#include "pile.h"
#include "pile_private.h"
#include "apile.h"
struct pile a_pile = {NULL};
void Apile_add(int n)
  {Pile_add(&a_pile, n);}
int Apile_count(void)
  {return Pile_count(&a_pile);}
```

Fig. 1. The pile.h abstract data type has operations *new, add, count, free.* The triang.c client adds the integers 1–n to the pile, then counts the pile. The pile.c implementation represents a pile as header node (struct pile) pointing to a linked list of integers. At bottom, there are two modules that each implement a single "implicit" pile in a module-local global variable: onepile.c maintains a pointer to a pile, while apile.c maintains a struct pile for which it needs knowledge of the representation through pile_private.h.

```
(* spec_pile.v *)
(* representation of linked lists in separation logic *)
Fixpoint listrep (σ: list Z) (x: val) : mpred :=
 match σ with
 | h::hs ⇒ EX y:val, !! (0 ≤ h ≤ Int.max_signed) &&
      data_at Ews tlist (Vint (Int.repr h), y) x
      ∗ malloc_token Ews tlist x ∗ listrep hs y
 | nil ⇒ !! (x = nullval) && emp
 end.

(* representation predicate for piles *)
Definition pilerep (σ: list Z) (p: val) : mpred :=
 EX x:val, data_at Ews tpile x p ∗ listrep σ x.

Definition pile_freeable (p: val) :=
  malloc_token Ews tpile p.

Definition Pile_new_spec :=
 DECLARE _Pile_new
 WITH gv: globals
 PRE [ ] PROP() LOCAL(gvars gv) SEP(mem_mgr gv)
 POST[ tptr tpile ]
   EX p: val,
     PROP() LOCAL(temp ret_temp p)
     SEP(pilerep nil p; pile_freeable p; mem_mgr gv).

Definition Pile_add_spec :=
 DECLARE _Pile_add
 WITH p: val, n: Z, σ: list Z, gv: globals
 PRE [ _p OF tptr tpile, _n OF tint ]
    PROP(0 ≤ n ≤ Int.max_signed)
    LOCAL(temp _p p; temp _n (Vint (Int.repr n));
            gvars gv)
    SEP(pilerep σ p; mem_mgr gv)
 POST[ tvoid ]
    PROP() LOCAL()
    SEP(pilerep (n::σ) p; mem_mgr gv).

Definition sumlist : list Z → Z := List.fold_right Z.add 0.

Definition Pile_count_spec :=
 DECLARE _Pile_count
 WITH p: val, σ: list Z
 PRE [ _p OF tptr tpile ]
    PROP(0 ≤ sumlist σ ≤ Int.max_signed) LOCAL(temp _p p)
    SEP(pilerep σ p)
 POST[ tint ]
    PROP() LOCAL(temp ret_temp (Vint (Int.repr (sumlist σ))))
    SEP(pilerep σ p).
```

Notation key

mpred   predicate on memory

EX   existential quantifier
!!   injects Prop into mpred
&&   nonseparating conjunction
data_at π τ v p is $p \mapsto v$, separation-logic mapsto at type τ, permission π

malloc_token π τ x represents "capability to deallocate x"

Ews   the "extern write share" gives write permission

_Pile_new is a C identifier

WITH   quantifies variables over PRE/POST of funspec

The C function's return type, tptr tpile, is "pointer to **struct** pile"

PROP(...) are pure propositions on the WITH-variables

LOCAL(... temp _p p ...) associates C local var _p with Coq value p

gvars gv establishes gv as mapping from C global vars to their addresses

SEP(R_1; R_2)   are separating conjuncts $R_1 * R_2$

mem_mgr gv represents *different* states of the malloc/free system in PRE and POST of any function that allocates or frees

Fig. 2. Specification of the pile module (Pile_free_spec not shown).

Verifying that pile.c's functions satisfy the specifications in Fig. 2 using VST-Floyd is done by proving Lemmas like this one (in file verif_pile.v):

Lemma body_Pile_new: semax_body Vprog Gprog f_Pile_new Pile_new_spec.
Proof. ... *(∗7 lines of Coq proof script∗)*.... **Qed**.

This says, in the context Vprog of global-variable types, in the context Gprog of function-specs (for functions that Pile_new might call), the function-body f_Pile_new satisfies the function-specification Pile_new_spec.

1 Specification files

In the VST distribution directory progs/pile, examine the files spec_∗.v and verif_∗.v. Let us take spec_onepile.v as an example:

```
(* spec_onepile.v *)
Require Import VST.floyd.proofauto.
Require Import onepile.
Require Import spec_stdlib.
Require Import spec_pile.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
```

The CompSpecs describes the fields struct and union declarations in the C program. Each module may use different local structs, or some structs may be declared in header files so they appear in several modules. Here, prog refers to onepile.prog, so the CompSpecs is built based on the structs in onepile.c. It's important that this **Instance** CompSpecs is built *after* importing spec_stdlib and spec_pile, otherwise their CompSpecs would shadow the one we want here.

```
(* spec_onepile.v, continued *)
Definition onepile (gv: globals) (sigma: option (list Z)) : mpred :=
 match sigma with
 | None ⇒ data_at_ Ews (tptr tpile) (gv _the_pile)
 | Some il ⇒ EX p:val, data_at Ews (tptr tpile) p (gv _the_pile) ∗
                                pilerep il p ∗ pile_freeable p
 end.
```

The separation-logic predicate for onepile refers to the abstract predicates pilerep and pile_freeable imported from spec_pile.

Normally one would add here lemmas onepile_local_facts and onepile_valid_pointer, but we omit those here.

```
(* spec_onepile.v, continued *)
Local Open Scope assert.

Definition Onepile_init_spec :=
 DECLARE _Onepile_init
 WITH gv: globals
```

```
 PRE [ ]
   PROP() LOCAL(gvars gv) SEP(onepile gv None; mem_mgr gv)
 POST[ tvoid ]
   PROP() LOCAL() SEP(onepile gv (Some nil); mem_mgr gv).

Definition Onepile_add_spec :=
 DECLARE _Onepile_add
 WITH n: Z, sigma: list Z, gv: globals
 PRE [ _n OF tint ]
   PROP(0 ≤ n ≤ Int.max_signed)
   LOCAL(temp _n (Vint (Int.repr n)); gvars gv)
   SEP(onepile gv (Some sigma); mem_mgr gv)
 POST[ tvoid ]
   PROP() LOCAL() SEP(onepile gv (Some (n::sigma)); mem_mgr gv).

Definition sumlist : list Z → Z := List.fold_right Z.add 0.

Definition Onepile_count_spec :=
 DECLARE _Onepile_count
 WITH sigma: list Z, gv: globals
 PRE [ ]
   PROP(0 ≤ sumlist sigma ≤ Int.max_signed)
   LOCAL(gvars gv) SEP(onepile gv (Some sigma))
 POST[ tint ]
     PROP() LOCAL(temp ret_temp (Vint (Int.repr (sumlist sigma))))
     SEP(onepile gv (Some sigma)).
```

We have here a funspec corresponding to each function definition in the .c file.

```
(* spec_onepile.v, continued *)
Definition specs := [Onepile_init_spec; Onepile_add_spec; Onepile_count_spec].
Definition ispecs : funspecs := [].
```

This is the key point for modular verification: In each spec_X.v, define two lists of funspecs:

specs: Function specifications of exported functions

ispecs: Function specifications of internal functions, that are not called from other .c files. (In principle, these could be declared static in the .c program, but VST support for static functions is not very good right now.)

```
(* spec_onepile.v, continued *)
Lemma make_onepile: ∀ gv,
  data_at_ Ews (tptr (Tstruct onepile._pile noattr)) (gv onepile._the_pile)
   ⊢ onepile gv None.
Proof. intros. unfold onepile. cancel. Qed.
```

The module onepile.c has an extern global variable the_pile. When the program is linked together, this variable will appear in the SEPpart of the precondition of main, along with global variables from all other modules. It will appear in its concrete form, that is, as a data_at. But the verification of main (and other client modules) would rather see it in abstract form, that is, as onepile gv None. This lemma, provided by spec_onepile.v and used by verif_main.v, converts the initialized global variable from its concrete to abstract specification form.

2 Verification files

Now examine the verification of onepile.c:

```
(* verif_onepile.v *)
Require Import VST.floyd.proofauto.
Require Import linking.
Require Import onepile.
Require Import spec_stdlib spec_pile spec_onepile.
```

After importing VST.floyd.proofauto as usual, we import linking. The file VST/progs/pile/linking.v is an experimental linking system that will someday be added as a standard feature to VST Floyd. Then we import onepile, that is, the abstract syntax trees of onepile.c that we are verifying; and the spec_ modules of all the C functions called upon by onepile.c.

```
(* verif_onepile.v, continued *)
Definition Gprog : funspecs := spec_pile.specs ++ spec_onepile.specs.
```

We build the Gprog for verifying this module by concatenating together the specs lists of all the modules we rely upon.

```
(* verif_onepile.v, continued *)
Lemma body_Onepile_init: semax_body Vprog Gprog f_Onepile_init Onepile_init_spec.
Proof. ... Qed.

Lemma body_Onepile_add: semax_body Vprog Gprog f_Onepile_add Onepile_add_spec.
Proof. ... Qed.

Lemma body_Onepile_count: semax_body Vprog Gprog f_Onepile_count Onepile_count_spec.
Proof. ... Qed.

Definition module :=
  [mk_body body_Onepile_init; mk_body body_Onepile_add;
   mk_body body_Onepile_count].
```

Verification of individual function bodies proceeds just as usual in VST. Then we collect this module's semax_body lemmas into a module.

3 Main

The specification and verification of main is special, because we need to account for *all* the modules' global variables.

```
(* spec_main.v *)
Require Import VST.floyd.proofauto.
Require Import main.
Require Import spec_stdlib spec_onepile spec_apile spec_triang.

Definition linked_prog : Clight.program :=
 ltac: (linking.link_progs_list [
   stdlib.prog; pile.prog; onepile.prog; apile.prog;
   triang.prog; main.prog]).
```

We start by importing all the spec_ files, then define the linked_prog as the combination of all the .c programs. This simulates what the Unix linker (ld) will do. In particular, the linked_prog has all the extern global variables of all the modules.

```
(* spec_main.v *)
Instance CompSpecs : compspecs. make_compspecs linked_prog. Defined.
Definition Vprog : varspecs. mk_varspecs linked_prog. Defined.
Local Open Scope assert.

Definition main_spec :=
 DECLARE _main
 WITH gv: globals
 PRE [ ] main_pre linked_prog nil gv
 POST[ tint ]
    PROP() LOCAL(temp ret_temp (Vint (Int.repr 0))) SEP(TT).

Definition specs := [main_spec].
```

Now, when we calculate the precondition of main, that is, main_pre linked_prog nil gv, all those global variables will be present in the SEPpart of the precondition.

Finally, we export a specs list as usual from this module, containing just main_spec.

Verification of main

```
(* verif_main.v *)
Require Import VST.floyd.proofauto.
Require Import linking.
Require Import main.
Require Import spec_stdlib spec_onepile spec_apile spec_triang spec_main.
Require verif_triang.
```

```
Definition Gprog : funspecs :=
  spec_apile.specs ++ spec_onepile.specs ++ spec_triang.specs ++ spec_main.specs.
```

The beginning of verif_main is just like any other verif_ file: Import the specs of the modules with functions that you call.

Because main.c does not call pile.c directly, there's no need to include spec_pile.specs in the Gprog.

```
(* verif_main.v, continued *)
Lemma body_main: semax_body Vprog Gprog f_main main_spec.
Proof.
start_function.
sep_apply (make_mem_mgr gv).
sep_apply (make_apile gv).
```

After the start_function of body_main, the precondition has (in its SEPclause) many data_ats describing the initialized global variables. Here we use (via sep_apply) lemmas provided by spec_stdlib and spec_apile to abstract these predicates.

```
(* verif_main.v, continued *)
generalize (make_onepile gv).
assert (change_composite_env spec_onepile.CompSpecs CompSpecs).
make_cs_preserve spec_onepile.CompSpecs CompSpecs.
change_compspecs CompSpecs.
intro Hx; sep_apply Hx; clear Hx.
```

In principle, we should do exactly the same with the make_onepile lemma, but it doesn't work; there's a problem with the CompSpecs that we fix with this work-around. This needs to be improved.

```
(* verif_main.v, continued *)
forward_call gv.
. . .
Qed.

Definition module := [mk_body body_main].
```

Finally, after sep_applying all the initialized-global-variable abstraction lemmas, we verify the main function in the ordinary way.

4 Linking

A modular proof of a modular program is organized as follows: CompCert parses each module M.c into the AST file M.v. Then we write the specification file spec_M.v containing funspecs as in Figure 2. We write verif_M.v which imports spec files of all the modules from which M.c calls functions, and contains semax_body proofs of correctness, for each of the functions in M.c.

Now we prove that everything links together:

```
(* link_pile.v *)
Require Import VST.floyd.proofauto.
```

```
Require Import linking.
Require main.
Require verif_stdlib verif_pile verif_onepile verif_apile.
Require verif_triang verif_main.

Definition allmodules :=
   verif_stdlib.module ++ verif_pile.module ++
   verif_onepile.module ++ verif_triang.module ++
   verif_apile.module ++ verif_main.module ++ nil.

Definition Gprog := ltac:
  (let x := constr:(merge_Gprogs_of allmodules) in
   let x := eval hnf in x in
   let x := eval simpl in x in
   exact x).

Lemma prog_correct:
  semax_prog spec_main.linked_prog spec_main.Vprog Gprog.
Proof.
  prove_semax_prog.
  do_semax_body_proofs (SortBodyProof.sort allmodules).
Qed.
```

5 Replacement of implementations

We now turn to the replacement of pile.c by a more performant implementation, fastpile.c, and its specification—see Figure 3. As fastpile.c employs a different data representation than pile.c, its specification employs a different representation predicate pilerep. As pilerep's type remains unchanged, the function specifications look virtually identical[1]; however, the VST-Floyd proof scripts (in file verif_fastpile.v) necessarily differ. Clients importing only the pile.h interface, like onepile.c or triang.c, cannot tell the difference (except that things run faster and take less memory), and are specified and verified only once (files spec_onepile.v / verif_onepile.v and spec_triang.v / verif_triang.v).

6 Subsumption of function specifications

But we may also equip fastpile.c with a more low-level specification (see Figure 4) in which the function specifications refer to a different representation predicate, countrep—clients of this interface do not need a notion of "sequence." The new specification is less abstract than the one in Fig. 3, and closer to the

[1] Existentially abstracting over the internal representation predicates would further emphasize the uniformity between fastpile.c and pile.c—a detailed treatment of this is beyond the scope of the present article.

```
/* fastpile_private.h */
struct pile { int sum; };

/* fastpile.c */
#include . . .
#include "pile.h"
#include "fastpile_private.h"
Pile Pile_new(void)
  {Pile p = (Pile)surely_malloc(sizeof *p); p→sum=0; return p; }
void Pile_add(Pile p, int n)
  {int s = p→sum; if (0≤ n && n≤ INT_MAX-s) p→sum = s+n; }
int Pile_count(Pile p) {return p→sum;}
void Pile_free(Pile p) {free(p);}

(* spec_fastpile.v *)
Definition pilerep (σ: list Z) (p: val) : mpred :=
 EX s:Z, !! (0 ≤ s ≤ Int.max_signed ∧ Forall (Z.le 0) σ ∧
            (0 ≤ sumlist σ ≤ Int.max_signed → s=sumlist σ))
   && data_at Ews tpile (Vint (Int.repr s)) p.

Definition pile_freeable := (* looks identical to the one in fig.2 *)
Definition Pile_new_spec := (* looks identical to the one in fig.2 *)
Definition Pile_add_spec := (* looks identical to the one in fig.2 *)
Definition Pile_count_spec := (* looks identical to the one in fig.2 *)
```

Fig. 3. fastpile.c, a more efficient implementation of the pile ADT. Since the only query function is count, there's no need to represent the entire list, just the sum will suffice. In the verification of a client program, the pilerep separation-logic predicate has the same signature: list Z → val → mpred, even though the representation is a single number rather than a linked list.

implementation. The subsumption rule allows us to exploit this relationship: we only need to explicitly verify the code against the low-level specification and can establish satisfaction of the high-level specification by recourse to subsumption. This separation of concerns extends from VST specifications to model-level reasoning: for example, in our verification of cryptographic primitives we found it convenient to verify that the C program implements a *low-level functional model* and then separately prove that the low-level functional model implements a high-level specification (e.g. cryptographic security). In our running example, fastpile.c's low-level functional model is *integer* (the Coq Z type), and its high level specification is list Z.

To learn about funspec_sub, its principles and how to use it, see the paper, "Abstraction and Subsumption in Modular Verification of C Programs," by Lennart Beringer and Andrew W. Appel, in *FM'19: 3rd World Congress on Formal Methods*, October 2019.

```
(* spec_fastpile_concrete.v *)
Definition countrep (s: Z) (p: val) : mpred := EX s':Z,
  !! (0 ≤ s ∧ 0 ≤ s' ≤ Int.max_signed ∧ (s ≤ Int.max_signed → s'=s)) &&
  data_at Ews tpile (Vint (Int.repr s')) p.

Definition count_freeable (p: val) := malloc_token Ews tpile p.

Definition Pile_new_spec := ...

Definition Pile_add_spec :=
 DECLARE _Pile_add
 WITH p: val, n: Z, s: Z, gv: globals
 PRE [ _p OF tptr tpile, _n OF tint ]
    PROP(0 ≤ n ≤ Int.max_signed)
    LOCAL(temp _p p; temp _n (Vint (Int.repr n)); gvars gv)
    SEP(countrep s p; mem_mgr gv)
 POST[ tvoid ]
    PROP() LOCAL() SEP(countrep (n + s) p; mem_mgr gv).

Definition Pile_count_spec := ...
```

Fig. 4. The fastpile.c implementation could be used in applications that simply need to keep a running total. That is, a *concrete* specification can use a predicate countrep: $Z \to val \to mpred$ that makes no assumption about a sequence (list Z). In countrep, the variable s' and the inequalities are needed to account for the possibility of integer overflow.