Deep Specifications and Certified Abstraction Layers

Ronghui Gu Jérémie Koenig Tahina Ramananandro Zhong Shao
Xiongnan (Newman) Wu Shu-Chun Weng Haozhong Zhang[†] Yu Guo[†]

Yale University [†]University of Science and Technology of China

Abstract

Modern computer systems consist of a multitude of abstraction layers (e.g., OS kernels, hypervisors, device drivers, network protocols), each of which defines an interface that hides the implementation details of a particular set of functionality. Client programs built on top of each layer can be understood solely based on the interface, independent of the layer implementation. Despite their obvious importance, abstraction layers have mostly been treated as a system concept; they have almost never been formally specified or verified. This makes it difficult to establish strong correctness properties, and to scale program verification across multiple layers.

In this paper, we present a novel language-based account of abstraction layers and show that they correspond to a strong form of abstraction over a particularly rich class of specifications which we call *deep specifications*. Just as *data abstraction* in typed functional languages leads to the important *representation independence* property, abstraction over deep specification is characterized by an important *implementation independence* property: any two implementations of the same deep specification must have *contextually equivalent* behaviors. We present a new layer calculus showing how to formally specify, program, verify, and compose abstraction layers. We show how to instantiate the layer calculus in realistic programming languages such as C and assembly, and how to adapt the CompCert verified compiler to compile certified C layers such that they can be linked with assembly layers. Using these new languages and tools, we have successfully developed multiple certified OS kernels in the Coq proof assistant, the most realistic of which consists of 37 abstraction layers, took less than one person year to develop, and can boot a version of Linux as a guest.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, formal methods; D.3.3 [*Programming Languages*]: Languages Constructs and Features; D.3.4 [*Programming Languages*]: Processors—Compilers; D.4.5 [*Operating Systems*]: Reliability—Verification; D.4.7 [*Operating Systems*]: Organization and Design—Hierarchical design; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Abstraction Layer; Modularity; Deep Specification; Program Verification; Certified OS Kernels; Certified Compilers.

POPL '15, January 15–17, 2015, Mumbai, India.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3330-9/15/01. . . $15.00.
http://dx.doi.org/10.1145/2676726.2676975

1. Introduction

Modern hardware and software systems are constructed using a series of abstraction layers (e.g., circuits, microarchitecture, ISA architecture, device drivers, OS kernels, hypervisors, network protocols, web servers, and application APIs), each defining an interface that hides the implementation details of a particular set of functionality. Client programs built on top of each layer can be understood solely based on the interface, independent of the layer implementation. Two layer implementations of the same interface should behave in the same way in the context of any client code.

The power of abstraction layers lies in their use of a very rich class of specifications, which we will call *deep specifications* in this paper. A deep specification, in theory, is supposed to capture the precise functionality of the underlying implementation as well as the assumptions which the implementation might have about its client contexts. In practice, abstraction layers are almost never formally specified or verified; their interfaces are often only documented in natural languages, and thus cannot be rigorously checked or enforced. Nevertheless, even such informal instances of abstraction over deep specifications have already brought us huge benefits. Baldwin and Clark [1] attributed such use of abstraction, modularity, and layering as the key factor that drove the computer industry toward today's explosive levels of innovation and growth because *complex products can be built from smaller subsystems that can be designed independently yet function together as a whole*.

Abstraction and modularity have also been heavily studied in the programming language community [31, 30]. The focus there is on abstraction over "shallow" specifications. A module interface in existing languages cannot describe the full functionality of its underlying implementation, instead, it only describes type specifications, augmented sometimes with simple invariants. Abstraction over shallow specifications is highly desirable [24], but client programs cannot be understood from the interface alone—this makes modular verification of correctness properties impossible: verification of client programs must look beyond the interface and examine its underlying implementation, thus breaking the modularity.

Given the obvious importance, formalizing and verifying abstraction layers are highly desirable, but they pose many challenges:

- *Lack of a language-based model.* It is unclear how to model abstraction layers in a language-based setting and how they differ from regular software modules or components. Each layer seems to be defining a new "abstract machine;" it may take an existing set of mechanisms (e.g., states and functions) at the layer below and expose a different view of the same mechanisms. For example, a virtual memory management layer—built on top of a physical memory layer— would expose to clients a different view of the memory, now accessed through virtual addresses.
- *Lack of good language support.* Programming an abstraction layer formally, by its very nature, would require two languages: one for writing the layer implementation (which, given the low-

level nature of many layers, often means a language like C or assembly); another for writing the formal layer specification (which, given the need to precisely specify full functionality, often means a rich formal logic). It is unclear how to fit these two different languages into a single setting. Indeed, many existing formal specification languages [34, 18, 16] are capable of building accurate *models* with rich specifications, but they are not concerned with connecting to the actual running code.

- *Lack of compiler and linking support.* Abstraction layers are often deployed in binary or assembly. Even if we can verify a layer implementation written in C, it is unclear how to compile it into assembly and link it with other assembly layers. The CompCert verified compiler [19] can only prove the correctness of compilation for whole programs, not individual modules or layers. Linking C with assembly adds a new challenge since they may have different memory layouts and calling conventions.

In this paper, we present a formal study of abstraction layers that tackles all these challenges. We define a *certified abstraction layer* as a triple (L_1, M, L_2) plus a mechanized proof object showing that the layer implementation M, built on top of the interface L_1 (the *underlay*), indeed faithfully *implements* the desirable interface L_2 above (the *overlay*). Here, the *implements* relation is often defined as some *simulation* relation [22]. A certified layer can be viewed as a "parameterized module" (from interfaces L_1 to L_2), *a la* an SML functor [23]; but it enforces a stronger contextual correctness property: a correct layer is like a "certified compiler," capable of converting any *safe* client program running on top of L_2 into one that has the same behavior but runs on top of L_1 (e.g., by "compiling" abstract primitives in L_2 into their implementation in M).

A regular software module M (built on top of L_1) with interface L_2 may not enjoy such a property because its client may invoke another module M' which shares some states with M but imposes different state invariants from those assumed by L_2. An abstraction layer does not allow such a client, instead, such M' must be either built on top of L_2 (thus respecting the invariants in L_2), or below L_2 (in which case, L_2 itself must be changed).

Our paper makes the following new contributions:

- We present the first language-based account of certified abstraction layers and show how they correspond to a *rigorous* form of abstraction over deep specifications used widely in the system community. A certified layer interface describes not only the precise functionality of any underlying implementation but also clear assumptions about its client contexts. Abstraction over deep specifications leads to the powerful *implementation independence* property (see Sec. 2): any two implementations of the same layer interface have contextually equivalent behaviors.
- We present a new layer calculus showing how to formally specify, program, verify, and compose certified abstraction layers (see Sec. 3). Such a layer language plays a similar role as the module language in SML [23], but its interface checking is not just typechecking or signature matching; instead, it requires formal verification of the *implements* relation in a proof assistant.
- We have instantiated the layer calculus on top of two core languages (see Sec. 4 and 5): **ClightX**, a variant of the CompCert Clight language [5]; and **LAsm**, an x86 assembly language. Both ClightX and LAsm can be used to program certified abstraction layers. We use the Coq logic [35] to develop all the layer interfaces. Each ClightX or LAsm layer is parameterized over its underlay interface, implemented using CompCert's external call mechanisms. We developed new tools and tactic libraries to help automate the verification of the *implements* relation.
- We have also modified CompCert to build a new verified compiler, **CompCertX**, that can compile ClightX abstraction layers into LAsm layers (see Sec. 6). CompCertX is novel because it can prove a stronger correctness theorem for compiling individual functions in each layer—such a theorem requires reasoning about *memory injection* [21] between the memory states of the source and target languages. To support linking between ClightX and LAsm layers, we show how to design the *implements* relation so that it is stable over memory injection.
- Using these new languages and tools, we have successfully constructed several feature-rich certified OS kernels in Coq (see Sec. 7). A certified kernel (L_{x86}, K, L_{ker}) is a verified LAsm implementation K, built on top of L_{x86}, and it *implements* the set of system calls as specified in L_{ker}. The correctness of the kernel guarantees that if a user program P runs *safely* on top of L_{ker}, running the version of P linked with the kernel K on L_{x86} will produce the same behavior. All our certified kernels are built by composing a collection of smaller layers. The most realistic kernel consists of 37 layers, took less than one person year to develop, and can boot a version of Linux as a guest.

The *POPL Artifact Evaluation Committee* reviewed the full artifact of our entire effort, including ClightX and LAsm, the CompCertX compiler, and the implementation of all certified kernels with Coq proofs. The reviewers unanimously stated that our implementation *exceeded their expectations*. Additional details about our work can be found in the companion technical report [13].

2. Why abstraction layers?

In this section, we describe the main ideas behind deep specifications and show why they work more naturally with abstraction layers than with regular software modules.

2.1 Shallow vs. deep specifications

We introduce shallow and deep specifications to describe different classes of requirements on software and hardware components. Type information and program contracts are examples of "shallow" specifications. Type-based module interfaces (e.g., ML signatures) are introduced to support compositional static type checking and separate compilation: a module M can be typechecked based on its import interface L_1 (without looking at L_1's implementation), and shown to have types specified in its export interface L_2.

To support compositional verification of strong functional correctness properties on a large system, we would hope that all of its components are given "deep" specifications. A module M will be verified based on its import interface L_1 (without looking at L_1's implementation), and shown to *implement* its export interface L_2.

To achieve true modularity, we would like to reason about the behaviors of M **solely** based on its import interface L_1; and we would also like its export interface L_2 to describe the full functionality of M while omitting the implementation details.

More formally, a deep specification captures everything we want to know about any of its implementations—it must satisfy the following important "implementation independence" property:

Implementation independence: *Any two implementations (e.g., M_1 and M_2) of the same deep specification (e.g., L) should have* contextually equivalent *behaviors.*

Different languages may define such contextual equivalence relation differently, but regardless, we want that, given any *whole-program* client P built on top of L, running $P \oplus M_1$ (i.e., P linked with M_1) should lead to the same observable result as running $P \oplus M_2$.

Without implementation independence, running $P \oplus M_1$ and $P \oplus M_2$ may yield different observable results, so we can prove a specific whole-program property that holds on $P \oplus M_1$ but not on $P \oplus M_2$—such whole-program property cannot be proved based on the program P and the specification L alone.

```
typedef enum {
  TD_READY, TD_RUN,
  TD_SLEEP, TD_DEAD
} td_state;

struct tcb {
  td_state tds;
  struct tcb *prev, *next;
};

struct tdq {
  struct tcb *head, *tail;
};
// $\nu_{\text{tcbp}}$ and $\nu_{\text{tdqp}}$
struct tcb tcbp[64];
struct tdq tdqp[64];
// $\kappa_{\text{dequeue}}$
struct tcb *
dequeue(struct tdq *q){
  struct tcb *head,*next;
  struct tcb *pid=null;
  if(q == null)
    return pid;
  else {
    head = q -> head;
    if (head == null)
     return pid;
    else {
     pid = head;
     next = head -> next;
     if(next == null) {
      q -> head = null;
      q -> tail = null;
     } else {
      next -> prev = null;
      q -> head = next;
     }
    }
  }
  return pid;
} ...
```

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.

Inductive tcb :=
| TCBUndef
| TCBV (tds: td_state)
       (prev next: Z)

Inductive tdq :=
| TDQUndef
| TDQV (head tail: Z)

Record abs:={tcbp:ZMap.t tcb;
             tdqp:ZMap.t tdq}

Function $\hat{\sigma}_{\text{dequeue}}$ a i :=
match (a.tdqp i) with
|TDQUndef => None
|TDQV h t =>
 if zeq h 0 then
  Some (a, 0)
 else
 match a.tcbp h with
 |TCBUndef => None
 |TCBV _ _ n =>
  if zeq n 0 then
  let q':=(TDQV 0 0) in
   Some (set_tdq a i q', h)
  else
  match a.tcbp n with
  |TCBUndef => None
  |TCBV s' _ n' =>
   let q':=(TDQV n t) in
   let a':=set_tdq a i q' in
   let b:=(TCBV s' 0 n') in
    Some (set_tcb a' n b, h)
  end
 end
end ...
```

Figure 1. Concrete (in C) vs. abstract (in Coq) thread queues

```
Definition tcb := td_state.

Definition tdq := List Z.

Record abs':={tcbp:ZMap.t tcb;
              tdqp:ZMap.t tdq}
```

```
Function $\hat{\sigma}'_{\text{dequeue}}$ a i :=
match (a.tdqp i) with
| h :: q' =>
  Some(set_tdq a i q', h)
| nil => None
end ......
```

Figure 2. A more abstract queue (in Coq)

Hoare-style partial correctness specifications are rarely deep specifications since they fail to satisfy implementation independence. Given two implementations of a partial correctness specification for a factorial function, one can return the correct factorial number and another can just go into infinite loop. A program built on top of such specification may not be reasoned about based on the specification alone, instead, we have to peek into the actual implementation in order to prove certain properties (e.g., termination).

In the rest of this paper, following CompCert [20], we will focus on languages whose semantics are *deterministic* relative to external events (formally, these languages are defined as both *receptive* and *determinate* [33] and they support external nondeterminism such as I/O and concurrency by making events explicit in the execution traces). Likewise, we only consider interfaces whose primitives have deterministic specifications. If L is a deterministic interface, and both M_1 and M_2 implement L, then $P \oplus M_1$ and $P \oplus M_2$ should have identical behaviors since they both follow the semantics of running P over L, which is deterministic. Deterministic specifications are thus also deep specifications.

Deep specifications can, of course, also be nondeterministic. They may contain resource bounds [6], numerical uncertainties [7],

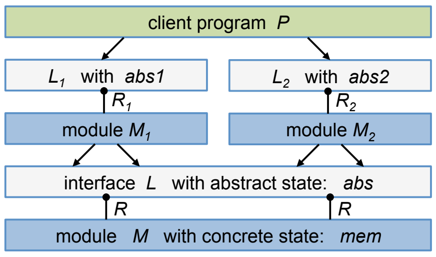


Figure 3. Client code with conflicting abstract states?

etc. Such nondeterminism should be unobservable in the semantics of a *whole* program, allowing implementation independence to still hold. We leave the investigation of nondeterministic deep specifications as future work.

2.2 Layers vs. modules

When a module (or a software component) implements an interface with a shallow specification, we often hide its private memory state completely from its client code. In doing so, we can guarantee that the client cannot possibly break any invariants imposed on the private state in the module implementation.

If a module implements an interface with a deep specification, we would still hide the private memory state from its client, but we also need to introduce an *abstract state* to specify the full functionality of each primitive in the interface.

For example, Fig. 1 shows the implementation of a concrete thread queue module (in C) and its interface with a deep specification (in Coq). The local state of the C implementation consists of 64 thread queues (tdqp) and 64 thread control blocks (tcbp). Each thread control block consists of the thread state, and a pair of pointers (prev and next) indicating which linked-list queue it belongs to. The dequeue function takes a pointer to a queue; it returns the head block if the queue is not empty, or null if the queue is empty.

In the Coq specification (Fig. 1 right; we omitted some invariants to make it more readable), we introduce an abstract state of type abs where we represent each C array as a Coq finite map (ZMap.t), and each pointer as an integer index (Z) to the tdq or tcb array. The dequeue primitive $\hat{\sigma}_{\text{dequeue}}$ is a mathematical function of type $\mathsf{abs} \to \mathsf{Z} \to \mathsf{option}\ (\mathsf{abs} \times \mathsf{Z})$; when the function returns None, it means that the abstract primitive faults. This dequeue specification is intentionally made very similar to the C function, so we can easily show that the C module indeed *implements* the specification.

We define that a module implements a specification if there is a *forward simulation* [22] from the module implementation to its specification. In the context of determinate and receptive languages [33, 20], if the specification is also deterministic, it is sufficient to find a forward simulation from the specification to its implementation (this is often easier to prove in practice).

In the rest of this paper, following CompCert, we often call the forward simulation from the implementation to its specification as *upward (forward) simulation* and the one from the specification to its implementation as *downward (forward) simulation*.

Fig. 2 shows a more abstract specification of the same queue implementation where the new abstract state abs' omits the prev and next links in tcb and treats each queue simply as a Coq list. The dequeue specification $\hat{\sigma}'_{\text{dequeue}}$ is now even simpler, which makes it easier to reason about its client, but it is now harder to prove that the C module implements this more abstract specification. This explains why we often introduce less abstract specifications (e.g., the one in Fig. 1) as intermediate steps, so a complex abstraction can be decomposed into several more tractable abstraction steps.

Deep specification brings out an interesting **new challenge** shown in Fig. 3: *what if a program P attempts to call primitives defined in two different interfaces L_1 and L_2, which may export two* *conflicting views (i.e., abstract states* abs1 *and* abs2*) of the same abstract state* abs *(thus also the same concrete memory state* mem*)?*

Here we assume that modules M, M_1, M_2 implement interfaces L, L_1, L_2 via some simulation relations R, R_1, R_2 (lines marked with a dot on one end) respectively. Clearly, calling primitives in L_2 may violate the invariants imposed in L_1, and vice versa, so L_1 and L_2 are breaking each other's abstraction when we run P. In fact, even without M_2 and L_2, if we allow P to directly call primitives in L, similar violation of L_1 invariants can also occur.

This means that we must prohibit client programs such as P above, and each deep specification must state the clear assumptions about its valid client contexts. Each interface should come with a single abstract state (abs) used by its primitives; and its client can only access the same abs throughout its execution.

This is what *abstraction layers* are designed for and why they are more compositional (with respect to deep specification) than regular modules! Layers are introduced to limit interaction among different modules: only modules with identical state views (i.e., R_1, R_2 and abs1, abs2 must be identical) can be composed horizontally.

A layer interface seems to be defining a new "abstract machine" because it only supports client programs with a particular view of the memory state. The correctness of a certified layer implementation allows us to transfer formal reasoning (of client programs) on one abstract machine (the overlay) to another (the underlay).

Programming with certified abstraction layers enables a disciplined way of composing a large number of components in a complex system. Without using layers, we may have to consider arbitrary module interaction or dependencies: an invariant held in one function can be easily broken when it calls a function defined in another module. A layered approach aims to sort and isolate all components based on a carefully designed set of abstraction levels so we can reason about one small abstraction step at a time and eliminate most unwanted interaction and dependencies.

3. A calculus of abstraction layers

Motivation A user of an abstraction layer (L_1, M, L_2) wants to know that its implementation M (on top of the underlay interface L_1) can be used to run any program P written against the overlay interface L_2. If we consider L_1, L_2 as abstract machines and M as a program transformation (which transforms a program P into $M(P)$), then for some notion of refinement $\sqsubseteq$, this property can be stated as $\forall P \,.\, M(P)@L_1 \sqsubseteq P@L_2$, meaning that the behavior of $M(P)$ executing on top of the underlay specification L_1 refines that of the program P executing on top of the overlay specification L_2.

This view of abstraction layers captures a wide variety of situations. Furthermore, two layers (L_1, M, L_2) and (L_2, N, L_3) can be composed as $(L_1, M \circ N, L_3)$, and the correctness of the layer implementation $M \circ N$ follows from that of M and N.

However, the layer interfaces are often not arbitrary abstract machines, but simply instances of a base language, specialized to provide layer-specific primitives and abstract state. The implementation is not an arbitrary transformation, but instead consists of some library code to be linked with the client program. In order to prove this transformation correct, we will verify the implementation of each primitive separately, and then use these proofs in conjunction with a general template for the instrumented language.

Abstract machines and program transformations are too general to capture this redundant structure. The layer calculus presented in this section provides fine-grained notions of layer interfaces and implementations. It allows us to describe what varies from one layer to the next and to assemble such layers in a generic way.

3.1 Prerequisites

To keep the formalism general and simple, we initially take the syntax and behavior of the programs under consideration to be abstract parameters. Specifically, in the remainder of this section we will assume that the following are given:

- a set of identifiers $i \in \mathbb{I}$ which will be used to name variables, functions, and primitives (e.g., dequeue and tcbp in Fig. 1);
- sets of function definitions $\kappa \in \mathrm{K}$, and variable definitions $\nu \in \mathrm{T}$, as specified by the language (e.g., $\kappa_{\mathsf{dequeue}}$ and ν_{tcbp} in Fig. 1);
- a set of behaviors $\sigma \in \Sigma$ for the individual primitives of layers, and the individual functions of programs (e.g., the step relation $\sigma_{\mathsf{dequeue}}$ derived from the Coq function $\hat{\sigma}_{\mathsf{dequeue}}$ in Fig. 1).

More examples can be found in Sec. 4.

We also need to define how the behaviors refine one another. This is particularly important because our layer interfaces bundle primitive specifications, and because a relation between layer interfaces is defined pointwise over these primitives. Ultimately, we wish to use these fine-grained layers and refinements to build complete abstract machines and whole-machine simulations. This can only be done if the refinements of individual primitives are consistent; for example, if they are given in terms of the same simulation relation.

Hence, we index behavior refinement by the elements of a partial monoid $(\mathbb{R}, \circ, \mathbf{id})$. We will refer to the elements $R \in \mathbb{R}$ of this monoid as *simulation relations*. However, note that at this stage, the elements of $\mathbb{R}$ are entirely abstract, and we require only that the composition operator $\circ$ and identity element **id** satisfy the monoid laws $R \circ (S \circ T) = (R \circ S) \circ T$ and $R \circ \mathbf{id} = \mathbf{id} \circ R = R$.

Finally, we need to interpret these abstract simulation relations as refinement relations between behaviors. That is, for each $R \in \mathbb{R}$, we require a relation $\leqslant_R$ on Σ. For instance, if the behaviors $\sigma_1, \sigma_2 \in \Sigma$ are taken to be step relations over some sets of states, $\sigma_1 \leqslant_R \sigma_2$ may be interpreted as the following simulation diagram:

$$\begin{tikzcd}
s_1 \arrow[r, "\sigma_1"] \arrow[d, dash, "R"'] & s_1' \arrow[d, dotted, dash, "R"] \\
s_2 \arrow[r, dotted, "\sigma_2"'] & s_2'
\end{tikzcd}$$

That is, whenever two states s_1, s_2 are related by R in some sense, and σ_1 takes s_1 to s_1' in one step, then there exists s_2' such that σ_2 takes s_2 to s_2' in zero or more steps, and s_2' and s_1' are also related by R. The relations $\leqslant_{-}$ should respect the monoid structure of $\mathbb{R}$, so that for any $\sigma \in \Sigma$ we have $\sigma \leqslant_{\mathbf{id}} \sigma$, and so that whenever $R, S \in \mathbb{R}$ and $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$ such that $\sigma_1 \leqslant_R \sigma_2$ and $\sigma_2 \leqslant_S \sigma_3$, it should be the case that $\sigma_1 \leqslant_{S \circ R} \sigma_3$.

3.2 Layer interfaces and modules

The syntax of the calculus is defined as follows:

$$L ::= \varnothing \mid i \mapsto \sigma \mid i \mapsto \nu \mid L_1 \oplus L_2$$
$$M ::= \varnothing \mid i \mapsto \kappa \mid i \mapsto \nu \mid M_1 \oplus M_2$$

The layer interfaces L and modules M are essentially finite maps; constructions of the form $i \mapsto \_$ are elementary single-binding objects, and $\oplus$ computes the union of two layers or modules. This is illustrated by the proof-of-concept interpretation given in the companion technical report [13]. For example, the thread queue module, shown in Fig. 1, can be defined as $M_{\mathsf{thread\_queue}} := \mathsf{tcbp} \mapsto \nu_{\mathsf{tcbp}} \oplus \mathsf{tdqp} \mapsto \nu_{\mathsf{tdqp}} \oplus \mathsf{dequeue} \mapsto \kappa_{\mathsf{dequeue}}$, while the overlay interface can be defined as $L_{\mathsf{thread\_queue}} := \mathsf{dequeue} \mapsto \sigma_{\mathsf{dequeue}}$.

The rules are presented in Fig. 4. The inclusion preorder defined on modules corresponds to the intuition that when $M \subseteq N$, any definition present in M must be present in N as well. The composition operator $\oplus$ behaves like a join operator. However, while $M \oplus N$ is an upper bound of M and N, we do not require it to be the *least* upper bound. The order on layer interfaces extends the

$$\boxed{M_1 \subseteq M_2}$$

$$\begin{array}{rr}
M \subseteq M & \text{MLE-REFL} \\
\varnothing \subseteq M & \text{MLE-EMPTY} \\
M \oplus \varnothing \subseteq M & \text{MLE-ID-RIGHT} \\
(M_1 \oplus M_2) \oplus M_3 \subseteq M_1 \oplus (M_2 \oplus M_3) & \text{MLE-ASSOC} \\
M_2 \oplus M_1 \subseteq M_1 \oplus M_2 & \text{MLE-COMM} \\
M_1 \subseteq M_1 \oplus M_2 & \text{MLE-UB-LEFT} \\
M_1 \subseteq M_2 \wedge M_2 \subseteq M_3 \Rightarrow M_1 \subseteq M_3 & \text{MLE-TRANS} \\
M_1 \subseteq M_1' \wedge M_2 \subseteq M_2' \Rightarrow M_1 \oplus M_2 \subseteq M_1' \oplus M_2' & \text{MLE-MON}
\end{array}$$

$$\boxed{L_1 \leqslant_R L_2}$$

$$\begin{array}{rr}
L \leqslant_{\mathbf{id}} L & \text{LLE-REFL} \\
\varnothing \leqslant_R L & \text{LLE-EMPTY} \\
L \oplus \varnothing \leqslant_{\mathbf{id}} L & \text{LLE-ID-RIGHT} \\
(L_1 \oplus L_2) \oplus L_3 \leqslant_{\mathbf{id}} L_1 \oplus (L_2 \oplus L_3) & \text{LLE-ASSOC} \\
L_2 \oplus L_1 \leqslant_{\mathbf{id}} L_1 \oplus L_2 & \text{LLE-COMM} \\
L_1 \leqslant_{\mathbf{id}} L_1 \oplus L_2 & \text{LLE-UB-LEFT} \\
L \oplus L \leqslant_{\mathbf{id}} L & \text{LLE-IDEMPOTENT} \\
L_1 \leqslant_R L_2 \wedge L_2 \leqslant_S L_3 \Rightarrow L_1 \leqslant_{S \circ R} L_3 & \text{LLE-TRANS} \\
L_1 \leqslant_R L_1' \wedge L_2 \leqslant_R L_2' \Rightarrow L_1 \oplus L_2 \leqslant_R L_1' \oplus L_2' & \text{LLE-MON} \\
\sigma_1 \leqslant_R \sigma_2 \Rightarrow i \mapsto \sigma_1 \leqslant_R i \mapsto \sigma_2 & \text{LLE-INTRO-PRIM}
\end{array}$$

$$\boxed{L_1 \vdash_R M : L_2}$$

$$\frac{}{L \vdash_{\mathbf{id}} \varnothing : L}\ \text{EMPTY}$$

$$\frac{}{L \vdash_{\mathbf{id}} i \mapsto \nu : i \mapsto \nu}\ \text{VAR}$$

$$\frac{L_1 \vdash_R M : L_2 \qquad L_2 \vdash_S N : L_3}{L_1 \vdash_{R \circ S} M \oplus N : L_3}\ \text{VCOMP}$$

$$\frac{L \vdash_R M : L_1 \qquad L \vdash_R N : L_2}{L \vdash_R M \oplus N : L_1 \oplus L_2}\ \text{HCOMP}$$

$$\frac{L_1 \leqslant_R L_1' \qquad L_1 \vdash_S M : L_2 \qquad L_2' \leqslant_T L_2}{L_1' \vdash_{R \circ S \circ T} M : L_2'}\ \text{CONSEQ}$$

Figure 4. The fine-grained layer calculus

underlying simulation preorder $\leqslant_R$ on behaviors. Compared to $\subseteq$, it should satisfy the additional property LLE-IDEMPOTENT.

The judgment $L_1 \vdash_R M : L_2$ is akin to a typing judgment for modules. It asserts that, using the simulation relation R, the module M—running on top of L_1—faithfully implements L_2. Because modules consist of code ultimately intended to be linked with a client program, the empty module $\varnothing$ acts as a unit, and can implement any layer interface L (EMPTY). Moreover, appending first N, then M to a client program is akin to appending $M \oplus N$ in one step (VCOMP). These rules correspond to the identity and composition properties already present in the framework of abstract machines and program transformations. However, the fine-grained calculus also provides a way to split refinements (HCOMP): when two different layer interfaces are implemented *in a compatible way* by two different modules on top of a common underlay interface, then the union of the two modules implements the union of the two interfaces.

This allows us to break down the problem of verifying a layer implementation in smaller pieces, but ultimately, we need to handle individual functions and primitives. The consequence rule (CONSEQ) can be used to tie our notion of behavior refinement into the calculus. However, to make the introduction of certified code possible, we need a semantics of the underlying language.

3.3 Language semantics

Assume that layers and modules are interpreted in the respective sets $\mathbb{L}$ and $\mathbb{M}$. The semantics of a module can be understood as the effect of its code has on the underlay interface, as specified by a function

$$\begin{array}{rr}
[\![-]\!] \ : \ \mathbb{M} \to (\mathbb{L} \to \mathbb{L}) & \\
i \mapsto \nu \leqslant_{\mathbf{id}} [\![i \mapsto \nu]\!]L & \text{SEM-VAR} \\
[\![M]\!](L \oplus [\![N]\!]L) \leqslant_{\mathbf{id}} [\![M \oplus N]\!]L & \text{SEM-COMP} \\
M_1 \subseteq M_2 \wedge L_1 \leqslant_R L_2 \Rightarrow [\![M_1]\!]L_1 \leqslant_R [\![M_2]\!]L_2 & \text{SEM-MON}
\end{array}$$

Figure 5. Semantics of modules

$[\![-]\!] : \mathbb{M} \to \mathbb{L} \to \mathbb{L}$. Given such a function, we can interpret the typing judgment as:

$$L_1 \vdash_R M : L_2 \quad \Leftrightarrow \quad L_2 \leqslant_R L_1 \oplus [\![M]\!]L_1.$$

Then the properties in Fig. 5 are sufficient to ensure the soundness of the typing rules with respect to this interpretation.

Here, surprisingly, we require that the specification refine the implementation! This is because our proof technique involves turning such a *downward* simulation into the converse *upward* simulation, as detailed in Sec. 5 (Theorem 1) and Sec. 4.3. Also, we included L_1 on the right-hand side of $\leqslant_R$ to support pass-through of primitives in the underlay L_1 into the overlay L_2.

The property SEM-COMP can be understood intuitively as follows. In $[\![M]\!](L \oplus [\![N]\!]L)$, the code of M is able to use the functions defined in N in addition to the primitives of the underlay interface L, but conversely the code of N cannot access the functions of M. However, in $[\![M \oplus N]\!]L$, the functions of M and N can call each other freely, and therefore the result should be more defined. The property SEM-MON states that making the module and underlay larger should also result in a more defined semantics.

Once a language semantics is given, we introduce a language-specific rule to prove the correctness of individual functions:

$$\frac{\mathrm{VC}(L, \kappa, \sigma)}{L \vdash_{\mathbf{id}} i \mapsto \kappa : i \mapsto \sigma}\ \text{FUN}$$

where the language-specific predicate $\mathrm{VC}(L, \kappa, \sigma)$ asserts that the function body κ faithfully implements the primitive behavior σ on top of L. This rule can be combined with the rules of the calculus to build up complete certified layer implementations.

Similarly, given a concrete language semantics, we will want to tie the calculus back into the framework of abstract machines and program transformations. For a layer interface L, we will define a corresponding abstract machine meant to execute programs written in a version of the language augmented with the primitives specified in L. The program transformation associated with a module M will simply concatenate the code of M to the client program. Then, for a particular notion of refinement $\sqsubseteq$, we will want to prove that the typing judgments entail the contextual refinement property:

$$\frac{L_1 \vdash_R M : L_2}{\forall P \,.\, (P \oplus M)@L_1 \sqsubseteq P@L_2}$$

Informally, if M faithfully implements L_2 on top of L_1, then invocations in P of a primitive i with behavior σ in L_2, can be satisfied by calling the corresponding function κ in M.

Indeed in Sec. 4 and Sec. 5, the primitive specifications in $[\![M]\!]L$, based on step relations, are defined to reflect the possible executions of the function definitions in M. Therefore, $L_2 \leqslant_R L_1 \oplus [\![M]\!]L_1$ implies that, for any primitive implementation in M, the corresponding deep specification in L_2 refines the execution of that function definition. Hence the execution of program P with underlay L_2 refines that of $P \oplus M$ with underlay L_1 (the properties enumerated in Fig. 5 hold for a similar reason). Properties of the language (i.e., being determinate and receptive) can then be used to reverse this refinement into the desired $(P \oplus M)@L_1 \sqsubseteq P@L_2$.

4. Layered programming in ClightX

In this section, we provide an instantiation of our framework for a C-like language. This instantiation serves two purposes: it illustrates a common use case for our framework, showing its usability and practicality; and it shows that our framework can add modularization and proof infrastructure to existing language subsets at minimal cost.

Our starting point: CompCert Clight Clight [5] is a subset of C and is formalized in Coq as part of the CompCert project. Its formal semantics relies on a memory model [21] that is not only realistic enough to specify C pointer operations, but also designed to simplify reasoning about non-aliasing of different variables. From the programmer's point of view, Clight avoids most pitfalls and peculiarities of C such as nondeterminism in expressions with side effects. On the other hand, Clight allows for pointer arithmetic and is a true subset of C. Such simplicity and practicality turn Clight into a solid choice for certified programming. However, Clight provides little support for abstraction, and proving properties about a Clight program requires intricate reasoning about data structures. This issue is addressed by our layer infrastructure.

4.1 Abstract state, primitives, and layer interfaces

We enable abstraction in Clight and other CompCert languages by instrumenting the memory states used by their semantics with an *abstract state* component. This abstract state can be manipulated using *primitives*, which are made available through CompCert's external function mechanism. We call the resulting language ClightX.

Abstract state and external functions The abstract state is not just a ghost state for reasoning: it does influence the outcome of executions! However, we seek to minimize its impact on the existing proof infrastructure for program and compiler verification. We do not modify the semantics of the basic operations of Clight, or the type of values it uses. Instead, the abstract state is accessed exclusively through Clight's external function mechanism.

Primitives and layer interfaces CompCert offers a notion of *external functions*, which are useful in modeling interaction with the environment, such as input/output. Indeed, CompCert models compiler correctness through traces of events which can be generated only by external functions. CompCert axiomatizes the behaviors of external functions without specifying them, and only assumes they do not behave in a manner that violates compiler correctness. We use the external function mechanism to extend Clight with our primitive operations, and supply their specifications to make the semantics of external functions more precise.

Definition 1 (Primitive specification)**.** *Let* mem *denote the type of memory state, and let* val *denote the type of concrete values. A* primitive specification σ *over the abstract state type* A *is a predicate on* $(\mathsf{val}^* \times \mathsf{mem} \times A) \times (\mathsf{val} \times \mathsf{mem} \times A)$*: when* $\sigma(args, m, a, res, m', a')$ *holds, we say that the primitive takes arguments args, memory state m and abstract state a, and returns a result res, a memory state* m' *and an abstract state* a'.

The type of abstract state and the set of available primitives will constitute our notion of layer interface.

Definition 2 (Layer interface)**.** *A layer interface* L *is a tuple* $L = (A, P)$ *where* A *is the type of abstract state, and* P *is the set of primitives as a finite map from identifiers to primitive specifications over the abstract state* A.

4.2 The ClightX parametric language

Syntax The syntax of ClightX (parameterized over a layer interface L) is identical to that of Clight. It features global variables (including function pointers), stack-allocated local variables, and temporary variables t . Expressions have no side effects; in particular, they cannot contain any function call. They include full-fledged pointer arithmetics (comparison, offset, C-style "arrays").

$$\begin{array}{lcll} e & ::= & n \,|\, x \,|\, t & \text{Constant, variable, temporary} \\ & | & \texttt{\&}e \,|\, \texttt{*}e \,|\, e_1 \; op \; e_2 \,|\, \ldots & \end{array}$$

Statements include assignment to a memory location or a temporary, function call and return, and structured control (loops, etc.).

$$\begin{array}{lcll} S & ::= & e_1 = e_2 & \text{Assignment to a memory location} \\ & | & t := e & \text{Assignment to a temporary variable} \\ & | & t \leftarrow e(e_1, \ldots) & \text{Function call} \\ & | & \texttt{return}(e) & \text{Function return} \\ & | & S_1; S_2 \mid \texttt{if}(e)\; S_1 \;\texttt{else}\; S_2 \mid \texttt{while}(e)\; S & \end{array}$$

Function calls may refer to internal functions defined as part of a module, or to primitives defined in the underlay L. However these two cases are not distinguished syntactically. In fact, the layer calculus allows for replacing primitive specifications with actual code implementation, with no changes to the caller's code.

Definition 3 (Functions, modules)**.** *A ClightX function is a tuple* $\kappa = (targs, lvars, S)$*, where targs is the list of temporaries to receive the arguments, lvars is the list of local stack-allocated variables with their sizes, and* S *is a statement, the function body. A* module M *is a finite map from identifiers to ClightX functions.*

Semantics Compared with Clight, the semantics of ClightX(L) adds a notion of abstract state, and permits calls to the primitives of L. We will write $L(i)(args, m, a, res, m', a')$ to denote the semantics of the primitive associated with identifier i in L.

We present the semantics of ClightX under the form of a big-step semantics. We fix an injective mapping Γ from global variables to memory block identifiers. We write $[\![e]\!](l, \tau, m)$ for the evaluation of expression e under local variables l, temporaries τ and memory state m. We write $\Gamma, L, M, l \vdash S : (\tau, m, a) \downarrow (res; \tau', m', a')$ for the semantics of statements: from the local environment l, the temporary environment τ, the memory state m, and the abstract state a, execution of S terminates and yields result res (or $\cdot$ if no result), temporary environment τ', memory state m', and abstract state a'. For instance, the rule for `return` statements is:

$$\frac{[\![e]\!](l, \tau, m) = res}{\Gamma, L, M, l \vdash \texttt{return}(e) : (\tau, m, a) \downarrow (res; \tau, m, a)}$$

We write $\Gamma, L, M \vdash f : (args; m, a) \Downarrow (res; m', a')$ to say that a function f defined either as an internal function in the module M, or as a primitive in the layer interface L, called with list of arguments $args$, from memory state m and abstract state a, returns result res, memory m' and abstract state a'.

For internal function calls, we first initialize the temporary environment with the arguments, and allocate the local variables of the callee ($\mathsf{next}(m)$ denotes the next available block identifier in memory m, not yet allocated). Then, we execute the body. Finally, we deallocate the stack-allocated variables of the callee.

$$\frac{\begin{array}{c} M(f) = ((t_1, \ldots, t_n), ((x_1, sz_1), \ldots, (x_k, sz_k)), S) \\ m_1 = \mathsf{alloc}(sz_k) \circ \cdots \circ \mathsf{alloc}(sz_1)(m) \\ l = \varnothing[x_1 \leftarrow \mathsf{next}(m)] \ldots [x_k \leftarrow \mathsf{next}(m) + k - 1] \\ \tau = \varnothing[t_1 \leftarrow v_1] \ldots [t_n \leftarrow v_n] \\ \Gamma, L, M, l \vdash S : (\tau, m_1, a) \downarrow (res; \tau', m_2, a') \\ m' = \mathsf{free}(\mathsf{next}(m), sz_1) \circ \cdots \circ \mathsf{free}(\mathsf{next}(m) + k - 1, sz_k)(m_2) \end{array}}{\Gamma, L, M \vdash f : (v_1, \ldots, v_n; m, a) \Downarrow (res; m', a')}$$

For primitive calls, we simply query the layer interface L:

$$\frac{L(f)(args, m, a, res, m', a')}{\Gamma, L, M \vdash f : (args; m, a) \Downarrow (res; m', a')}$$

Using the function judgment, we can state the rule for function call statements as:

$$\frac{\begin{array}{c} \forall i, [\![e_i]\!](l, \tau, m) = v_i \qquad [\![e]\!](l, \tau, m) = (b, 0) \\ \Gamma(f) = b \qquad \Gamma, L, M \vdash f : (v_1, \ldots, v_n; m, a) \Downarrow (res; m', a') \\ \tau' = \tau[t \leftarrow res] \end{array}}{\Gamma, L, M, l \vdash t \leftarrow e(e_1, \ldots, e_n) : (\tau, m, a) \downarrow (\cdot; \tau', m', a')}$$

$$\dfrac{\dfrac{\forall i \,.\, L_1 \vdash_{\textbf{id}} i \mapsto \kappa_i : i \mapsto \sigma'_i}{L_1 \vdash_{\textbf{id}} M : L'_1} \qquad \dfrac{\dfrac{\forall i \,.\, \sigma_i \leqslant_R \sigma'_i}{\forall i \,.\, i \mapsto \sigma_i \leqslant_R i \mapsto \sigma'_i}}{L_2 \leqslant_R L'_1}}{L_1 \vdash_R M : L_2}$$

where L_1 is the underlay, the module $M = \bigoplus_i i \mapsto \kappa_i$, the intermediate layer $L'_1 = \bigoplus_i i \mapsto \sigma'_i$, and the overlay $L_2 = \bigoplus_i i \mapsto \sigma_i$.

Figure 6. Building a certified ClightX layer

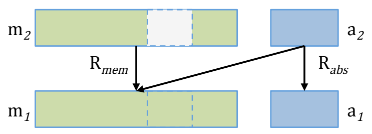


Figure 7. Layer simulation relation

The full semantics of ClightX is given in the companion TR [13].

Definition 4 (Semantics of a module)**.** *Let M be a ClightX module, and L be a layer interface. Let Γ be a mapping from global variables to memory blocks. The semantics of a module M in ClightX(L), written $[\![M]\!]L$, is the layer interface defined as follows:*

- *the type of abstract state is the same as in L;*
- *the semantics of primitives are defined by the following rule:*

$$\frac{f \in \mathrm{dom}(M) \qquad \Gamma, L, M \vdash f : (args; m, a) \Downarrow (res; m', a')}{([\![M]\!]L)(f)(args, m, a, res, m', a')}$$

4.3 Layered programming and verification

To construct a certified abstraction layer (L_1, M, L_2), we need to find a simulation R such that $L_1 \vdash_R M : L_2$ holds. Fig. 6 gives an overview of this process. We write $M = \bigoplus_i i \mapsto \kappa_i$, where i ranges over the function identifiers defined in module M, and κ_i is the corresponding implementation. Global variables in M should not be accessible from the layers above: their permissions are removed in the overlay interface L_2. The interface L_2 also includes a specification σ_i for each function i defined in M.

We decouple the task of code verification from that of data structure abstraction. We introduce an intermediate layer interface, $L'_1 = \bigoplus_i i \mapsto \sigma'_i$, with its specifications σ'_i expressed in terms of the underlay states. We first prove that $L_1 \vdash_{\textbf{id}} M : L'_1$ holds. For each function i in M, we show that its implementation κ_i is a downward simulation of its "underlay" specification σ'_i, that is, $L_1 \vdash_{\textbf{id}} i \mapsto \kappa_i : i \mapsto \sigma'_i$. We apply the HCOMP rule to compose all the per-function simulation statements. Note the simulation relations here are all **id**, meaning there is no abstraction of data structures in these steps. We then prove $L_2 \leqslant_R L'_1$, which means that each specification σ_i in L_2 is an abstraction of the intermediate specification σ'_i via a simulation relation R. From $i \mapsto \sigma_i \leqslant_R i \mapsto \sigma'_i$, we apply the monotonicity rule LLE-MON to get $L_2 \leqslant_R L'_1$. Finally, we apply the CONSEQ rule to deduce $L_1 \vdash_R M : L_2$.

Verifying ClightX functions L_1 and L'_1 share the same views of both concrete and abstract states, so no simulation relation is involved during this step of verification (the FUN rule in Sec. 3.3). Using Coq's tactical language, we have developed a proof automation engine that can handle most of the functional correctness proofs of ClightX programs. It contains two main parts: a ClightX statement/expression interpreter that generates the verification conditions by utilizing rules of ClightX big-step semantics, and an automated theorem prover that discharges the generated verification conditions

```
typedef enum {
  PG_RESERVED, PG_KERNEL,
  PG_NORMAL
} pg_type;

struct page_info {
  pg_type t;
  uint u;
};
struct page_info AT[1<<20];
```

```
Notation RESV := 0.
Notation KERN := (RESV + 1).
Notation NORM := (KERN + 1).

Inductive page_info :=
| ATV (t: Z) (u: Z)
| ATUndef.

Record abs'' :=
  {AT: ZMap.t page_info}.
```

Figure 8. Concrete (C) vs. abstract (Coq) memory allocation table

```
// κat_get
uint at_get (uint i){
  uint allocated;
  allocated = AT[i].u;
  if (allocated != 0)
      allocated = 1;
  return allocated;
}

// κat_set
void at_set (uint i, uint b){
    AT[i].u = b;
}
```

```
Function σ̂at_get a i :=
match (a.AT i) with
| ATV _ 0 => Some 0
| ATV _ _ => Some 1
| _ => None
end.

Function σ̂at_set a i b :=
match (a.AT i) with
| ATV t _ =>
Some (set_AT a i (ATV t b))
| _ => None
end.
```

Figure 9. Concrete vs. abstract getter-setter functions for AT

```
Inductive σ'at_set :=
| ∀ m m' a ofs v n,
   m.store AT ofs v = m'
  -> ofs = n * 8 + 4
  -> 0 <= n < 1048576
  -> σ'at_set (n::v::nil)
        m a Vundef m' a.
```

```
Inductive σat_set :=
| ∀ m a a' n v,
   σ̂at_set a n v = Some a'
  -> 0 <= n < 1048576
  -> σat_set (n::v::nil)
       m a Vundef m a'.
```

Figure 10. High level and low level specification for at_set

on the fly. The automated theorem prover is a first order prover, extended with different theory solvers, such as the theory of integer arithmetic and the theory of CompCert style partial maps. The entire automation engine is developed in Coq's Ltac language.

Data abstraction Since primitives in L'_1 and L_2 are atomic, we prove the single-step downward simulation between L'_1 and L_2 only at the specification level. The simulation proof for the abstraction can be made language independent. The simulation relation R captures the relation between the underlay state (concrete memory and abstract state) and the overlay state, and can be decomposed as R_{mem} and R_{abs} (see Fig. 7). The relation R_{mem} ensures that the concrete memory states m_1 and m_2 contain the same values, while making sure the memory permissions for the part to be abstracted are erased in the overlay memory m_2. The component R_{abs} relates the overlay abstract state a_2 with the full underlay state (m_1, a_1).

Through this decomposition, we achieve the following two objectives: the client program can directly manipulate the abstract state without worrying about its underlying concrete implementation (which is hidden via R_{mem}), and the abstract state in the overlay is actually implementable by the concrete memory and abstract state in the underlay (via R_{abs}).

Common patterns We have developed two common design patterns to further ease the task of verification. The *getter-setter* pattern establishes memory abstraction by introducing new abstract states and erasing the corresponding memory permissions for the overlay. The overlay only adds the get and set primitives which are implemented using simple memory load/store operations at the underlay. The *abs-fun* pattern implements key functionalities, but does not introduce new abstract state. Its implementation (on underlay) does not touch concrete memory state. Instead, it only accesses the states

```
// κ_palloc
uint palloc(uint nps){
  uint i = 0, u;
  uint freei = nps;
  while(freei == nps
          && i < nps) {
     u = at_get(i);
     if (u == 0)
       freei = i;
     i ++;
  }
  if (freei != nps)
    at_set(freei, 1);
  return freei;
}
```

```
Definition first_free a n:
  {v| 0<= fst v < n
   /\ a.AT (fst v) = ATV (snd v) 0
   /\ ∀ x, 0 <= x < fst v
        ->  ∼ a.AT x = ATV _ 0}
+ {∀ x, 0 <= x < n
        ->  ∼ a.AT x = ATV _ 0}.

Function σ̂_palloc a nps :=
match first_free a nps with
 | inleft (exist (i, t) _) =>
   (set_AT a i (ATV t 1), i)
 | _ => (a, nps)
end.
```

Figure 11. Concrete (in C) vs. abstract (in Coq) palloc function

```
Inductive σ'_palloc : spec :=
| ∀ m a a' nps n,
    σ̂_palloc a nps = (a', n)
   -> 0 <= nps < 1048576
   -> σ'_palloc (nps::nil) m a n m a'.

Definition σ_palloc := σ'_palloc.
```

Figure 12. High level and low level specification for palloc function

that have already been abstracted, and it only does so using the primitives provided by the underlay interface.

Figs. 8-12 show how we use the two patterns to implement and verify a simplified physical memory allocator palloc, which allocates and returns the first free entry in the physical memory allocation table. Fig. 8-10 shows how we follow the *getter-setter* pattern to abstract the allocation table into a new abstract state. As shown in Fig. 8, we first turn the concrete C memory allocation table implementation into an abstract Coq data type. Then we implement the getter and setter functions for the memory allocation table, both in C and Coq (see Fig. 9). The Coq functions $\hat{\sigma}_{\mathsf{at\_get}}$ and $\hat{\sigma}_{\mathsf{at\_set}}$ are just intermediate specifications that are used later in the overlay specifications. The actual underlay and overlay specifications of the setter function at_set are shown in Fig. 10.

We then prove $L_1 \vdash_{\mathbf{id}} \mathsf{at\_set} \mapsto \kappa_{\mathsf{at\_set}} : \mathsf{at\_set} \mapsto \sigma'_{\mathsf{at\_set}}$, and also $\mathsf{at\_set} \mapsto \sigma_{\mathsf{at\_set}} \leqslant_R \mathsf{at\_set} \mapsto \sigma'_{\mathsf{at\_set}}$.

The code verification (first part) is easy for this pattern because the memory load and store operations in the underlay match the source code closely. The proof can be discharged by our automation tactic. The main task of this pattern is to prove refinement (second part): we design a simulation relation R relating the memory storing the global variable at underlay with its corresponding abstract data at overlay. The component R_{mem} ensures that there is no permission for allocation table `AT` in overlay memory state m_2, while the component R_{abs} is defined as follows:

- $\forall i \in [0, 2^{20})$, R_{abs} enforces the *writable* permission on `AT[i]` at underlay memory state m_1, and requires (a_2.`AT i`) at overlay to be `(ATV AT[i].t AT[i].u)`.
- Except for `AT`, R_{abs} requires all other abstract data in underlay and overlay to be the same.

The refinement proof for $L_2 \leqslant_R L'_1$ involves the efforts to prove that this relation R between underlay memory and overlay abstract state is preserved by all the atomic primitives in both L'_1 and L_2.

After we abstract the memory and get/set operations, we implement palloc on top of L_2, following the *abs-fun* pattern. The previous overlay now becomes the new underlay ("L_1"). Fig. 11 shows both the implementation of palloc in ClightX and the abstract function in Coq. As before, we separately show that $L_1 \vdash_{\mathbf{id}}$ $\mathsf{palloc} \mapsto \kappa_{\mathsf{palloc}} : \mathsf{palloc} \mapsto \sigma'_{\mathsf{palloc}}$, and $\mathsf{palloc} \mapsto \sigma_{\mathsf{palloc}} \leqslant_R$ $\mathsf{palloc} \mapsto \sigma'_{\mathsf{palloc}}$ holds. For the *abs-fun* pattern, the refinement proof is easy. Since we do not introduce any new abstract states in this pattern, the implementation only manipulates the abstract states through the primitive calls of the underlay. Thus, as shown in Fig. 12, the corresponding underlay and overlay specifications are exactly the same, so the relation R here is the identity (**id**) and the proof of refinement is trivial. The main task for the *abs-fun* pattern is to verify the code, which is done using our automation tactic.

The above examples show that for the *getter-setter* pattern, the primary task is to prove data abstraction, while for the *abs-fun* pattern, the main task is to do simple program verification. These two tasks are well understood and manageable, so the decoupling (via these two patterns) makes the layer construction much easier.

5. Layered programming in LAsm

In this section, we describe LAsm, the *Layered Assembly language*, and the extended machine model which LAsm is based on.

The reason we are interested in assembly code and behavior is threefold. First of all, even though we provide ClightX to write most code, we are still interested in the actual assembly code running on the actual machine. In Section 6, we will provide a verified compiler to transport all proofs of code written in ClightX to assembly.

Secondly, there are parts of software that have to be manually written in assembly for various reasons. For example, the standard implementation of kernel context switch modifies the stack pointer register ESP, which does not satisfy the C calling convention and has to be verified in assembly. A linker will be defined in Section 6 to link them with compiled C code.

Last but not least, we are interested not only in the behavior of our code, but also in the behavior of the *context* that will call functions defined in our code. To be as general as possible, we allow the context to include all valid assembly code sequences. To this end, it is necessary to transport per-function refinement proofs to a whole-machine *contextual refinement* proof.

The LAsm assembly language We start from the 32-bit x86 assembly subset specified in CompCert. CompCert x86 assembly is modeled as a state machine with a register set and a memory state. The register set consists of eight 32-bit general-purpose registers and eight XMM registers designated as scalar double-precision floating-point operands. The memory state is same as the one in Clight. In particular, each function executes with its stack frame modeled in its own memory block, so that the stack is not a contiguous piece of memory. Another anomaly regarding function calls in CompCert x86 assembly is that the return address is stored in pseudo-register RA instead of being pushed onto the stack, so that the callee must allocate its own stack frame and store the return address.

Similarly to ClightX, we extend the machine state with an abstract state, which will be modified by primitives. This yields LAsm, whose syntax is the same as that of CompCert x86 assembly, except that the semantics will be parameterized over the type of abstract states and the specifications of primitives. Most notably, primitive calls are syntactically indistinguishable from normal function calls, yet depend on the specifications semantically.

Moreover, in our Coq formalization, the semantics of LAsm is also equipped with memory accessors for address translation in order to handle both kernel memory linear mapping and user space virtual memory. However, for the sake of presentation, we are going to describe a simplified version of LAsm where memory accesses only use the kernel memory.

We define the semantics of LAsm in small-step form. The machine state is (ρ, m, a) where ρ contains the values of registers, m is the concrete memory state and a is the abstract state. Let M be an LAsm module, which is a finite map from identifiers to arrays of LAsm instructions, we write $\Gamma, L, M \vdash (\rho, m, a) \to (\rho', m', a')$

a transition step in the LAsm machine. The full syntax and formal semantics of LAsm is described in the companion technical report.

Assembly layer interfaces The semantics of LAsm is parameterized over a layer interface. Different from C-style primitives (see Def. 1), which are defined using argument list and return value, primitives implemented in LAsm often utilize their full control over the register set and are not restricted to a particular calling convention (e.g. context switch). Therefore, it is necessary to extend the structure of layer interfaces to allow assembly-style primitives modifying the register set.

Definition 5 (Assembly-style primitive)**.** *An assembly-style primitive specification p over the abstract state type A is a predicate on $((\mathsf{preg} \to \mathsf{val}) \times \mathsf{mem} \times A) \times ((\mathsf{preg} \to \mathsf{val}) \times \mathsf{mem} \times A)$. $p(\rho, m, a, \rho', m', a')$ says that the primitive p takes register set ρ, memory state m and abstract state a as arguments, and returns register set ρ', memory state m' and abstract state a' as result.*

By "*style*," we mean the calling convention, not the language in which they are actually implemented. C-style primitives may very well be implemented as hand-written assembly code at underlay.

We can then define assembly layer interfaces by replacing the primitive specification with our assembly-style one in Def. 2. But, to make reasoning simpler, when defining assembly layer interfaces, we distinguish C-style from assembly-style primitives. First, C-style primitives can be refined by other C-style primitives. Second and most importantly, it becomes possible to instantiate the semantics of ClightX with an *assembly* layer interface by just considering C-style primitives and ignoring assembly-style primitives (which might not follow the C calling convention). In this way, ClightX code is only allowed to call C-style primitives, whereas LAsm can actually call both kinds of primitives.

Definition 6 (Assembly layer interface)**.** *An assembly layer interface L is a tuple $L = (A, P_{\mathrm{ClightX}}, P_{\mathrm{LAsm}})$ where:*

- $(A, P_{\mathrm{ClightX}})$ *is a C layer interface (see Def. 2)*
- P_{LAsm} *is a finite map from identifiers to assembly-style primitive specifications over the abstract state A. The domains of P_{ClightX} and P_{LAsm} shall be disjoint.*

Whole-machine semantics and contextual refinement Based on the relational transition system which we just defined for LAsm, we can define the whole-machine semantics including not only the code that we wrote by hand or that we compile, but also the *context* code that shall call our functions. To this end, it suffices to equip the semantics with a notion of initial and final state, in a way similar to the CompCert x86 whole-program assembly semantics.

In CompCert, the initial state consists of an empty register set with only EIP (instruction pointer register) pointing to the `main` function of the module, and the memory state is constructed by allocating a memory block for each global variable of the program. We follow the same approach for LAsm, except that we also need an initial abstract state, provided by the layer interface, so we need to extend its definition:

Definition 7 (Whole-machine layer interface)**.** *A whole-machine layer interface L is a tuple $L = (A, P_{\mathrm{ClightX}}, P_{\mathrm{LAsm}}, a_0)$ where:*

- $(A, P_{\mathrm{ClightX}}, P_{\mathrm{LAsm}})$ *is an assembly layer interface*
- $a_0 : A$ *is the initial abstract state.*

Definition 8 (Whole-machine initial state)**.** *The whole-machine LAsm initial state for layer interface L and module M is the LAsm state (ρ_0, m_0, a_0) defined as follows:*

- $\rho_0(r) = \begin{cases} (\Gamma(\mathtt{main}), 0) & \textit{if } r = \mathsf{EIP} \\ 0 & \textit{if } r = \mathsf{RA} \\ \bot & \textit{otherwise} \end{cases}$
- m_0 *is constructed from the global variables of* Γ, L, M
- a_0 *is the whole-machine initial state specified in* L

Definition 9 (Whole-machine final state)**.** *A whole-machine LAsm state (ρ, m, a) is final with return code n if, and only if, $\rho(\mathsf{EAX}) = n$ and $\rho(\mathsf{EIP}) = 0$, where EAX is the accumulator register.*

Notice that $\rho(\mathsf{EIP})$ contains the *integer* 0, which is also the initial return address and is not a valid pointer. This ensures that executions do not go beyond a final state, following the CompCert x86 whole-program semantics: `main` has returned to its "caller", which does not exist. Thus, the final state is uniquely determined (there can be no other possible behavior once such a state is reached), so the whole-machine semantics is deterministic once the primitives are.

Definition 10 (Whole-machine behavior)**.** *Let Γ be a mapping of global variables to memory blocks. Then, we say that*

- $LAsm(\Gamma, L, M)$ diverges *if there is an infinite execution sequence from the whole-machine initial state for L*
- $LAsm(\Gamma, L, M)$ terminates with return code n *if there is a finite execution sequence from the whole-machine initial state for L to a whole-machine final state with return code n*
- $LAsm(\Gamma, L, M)$ goes wrong *if there is a finite execution sequence from the whole-machine initial state for L to a non-final state that can take no step.*

Then, we are interested in *refinement* between whole machines:

Definition 11 (Whole-machine refinement)**.** *Let L_{high}, L_{low} be two whole-machine assembly layer interfaces, and M_{high}, M_{low} be two LAsm modules. Then, we say that $M_{low}@L_{low}$ refines $M_{high}@L_{high}$, and write $M_{low}@L_{low} \sqsubseteq M_{high}@L_{high}$ if, and only if, for any Γ such that $\mathrm{dom}(L_{high}) \cup \mathrm{dom}(M_{high}) \cup \mathrm{dom}(L_{low}) \cup \mathrm{dom}(M_{low}) \subseteq \mathrm{dom}(\Gamma)$ and $LAsm(\Gamma, L_{high}, M_{high})$ does not go wrong, then (1) $LAsm(\Gamma, L_{low}, M_{low})$ does not go wrong; (2) if $LAsm(\Gamma, L_{low}, M_{low})$ terminates with return code n, then so does $LAsm(\Gamma, L_{high}, M_{high})$; (3) if $LAsm(\Gamma, L_{low}, M_{low})$ diverges, so does $LAsm(\Gamma, L_{high}, M_{high})$.*

In our Coq implementation, we actually formalized the semantics of LAsm with a richer notion of observable behaviors involving CompCert-style events such as I/O. Thus, we define the whole-machine behaviors and refinement using event traces *a la* CompCert [20, 3.5 sqq.]: if the higher machine does not go wrong, then every valid behavior of the lower machine is a valid behavior of the higher.

Finally, we can define *contextual refinement* between layer interfaces through a module M:

Definition 12 (Contextual refinement)**.** *We say a module M implements an overlay L_{high} on top of an underlay L_{low}, and write $L_{low} \models M : L_{high}$ if, and only if, for any module (*context*) M' disjoint from M, L_{low}, L_{high}, we have $(M \oplus M')@L_{low} \sqsubseteq M'@L_{high}$.*

Per-module semantics As for ClightX, we can also specify the semantics of an LAsm module as a layer interface. However, a major difference between ClightX and LAsm is that it is not possible to uniquely characterize the "per-function final state" at which function execution should stop. Indeed, as in LAsm there is no control stack, when considering the per-function semantics of a function f, it is not possible to distinguish f exiting and returning control to its caller, from a callee g returning to f.

Thus, even though both the step relation of the LAsm semantics and the primitive specifications (of a layer interface) are deterministic, the semantics of a function could still be non-deterministic.

Definition 13. *Let $L = (A, \_, \_)$ be an assembly layer interface, and M be an LAsm module. The module semantics $[\![M]\!]L$ is then the assembly layer interface $[\![M]\!]L = (A, \varnothing, P)$, where the assembly-style primitive specification P is defined for each $f \in \mathrm{dom}(M)$*

using the small-step semantics of LAsm as follows:

$$\begin{aligned} & P(f)(\rho, m, a, \rho', m', a') \\ \Leftrightarrow \quad & \Gamma(f) = b \wedge \rho(\mathsf{EIP}) = (b, 0) \\ & \wedge \Gamma, L, M \vdash (\rho, m, a) \to^{+} (\rho', m', a') \end{aligned}$$

Soundness of per-module refinement In this paper, we aim at showing that the layer calculus given in Section 3 is a powerful device to prove contextual refinement: instead of proving the whole-machine contextual refinement directly, we only need to prove the *downward simulation* relations about individual modules, notated as $L_{\text{low}} \vdash_R M : L_{\text{high}}$, and apply the soundness theorem to get the contextual refinement properties at the whole-machine level.

Lemma 1 (Downward simulation diagram)**.** *Let (L_{low}, M, L_{high}) be a certified layer, such that $L_{low} \vdash_R M : L_{high}$. Then, for any module M', we have the following downward simulation diagram:*

$$\begin{tikzcd}
s_{high} \arrow[r, "1", "{\Gamma, L_{high}, M'}"'] \arrow[d, no head, "R"] & s'_{high} \arrow[d, dotted, no head, "R"] \\
s_{low} \arrow[r, dotted, "+", "{\Gamma, L_{low}, M \oplus M'}"'] & s'_{low}
\end{tikzcd}$$

Theorem 1 (Soundness)**.** *Let (L_{low}, M, L_{high}) be a certified layer. If the primitive specifications of L_{low} are deterministic and if $L_{low} \vdash_R M : L_{high}$, then $L_{low} \models M : L_{high}$.*

Proof. Since the whole machine $LAsm(\Gamma, L_{\text{low}}, M)$ is deterministic, we can flip the downward simulation given by Lemma 1 to an upward one, hence the whole-machine refinement. □

Since the per-function semantics is non-deterministic due to its final state not being uniquely defined, we can only flip the downward simulation to contextual refinement at the whole-machine level.

6. Certified compilation and linking

We would like to write most parts of our kernel in ClightX rather than in LAsm for easier verification. This means that, for each layer interface L, we have to compile our ClightX(L) source code to the corresponding LAsm(L) assembly language in such a way that all proofs at the ClightX level are preserved at the LAsm level.

This section describes how we have modified the CompCert compiler to compile certified C layers into certified assembly layers. It also talks about how we link compiled certified C layers with other certified assembly layers.

6.1 The CompCertX verified compiler

To transport the proofs at ClightX down to LAsm, we adapt the CompCert verified compiler to parameterize all its intermediate languages over the layer interface L similarly to how we defined ClightX(L), including the assembly language. This gives rise to CompCertX(L) (for "CompCert eXtended", where external functions are instantiated with layer interface L).

CompCertX goes from ClightX to the similarly parameterized AsmX and then to LAsm. We retain all features and optimizations of CompCert, including function inlining, dead code elimination, common subexpression elimination, and tail call recognition.

Compiler correctness for CompCertX Because CompCert only proves semantics preservation for whole programs, the major challenge is to adapt the semantics preservation statements of all compilation passes (from Clight to assembly) to per-function semantics.

The operational semantics of all CompCert languages are given through small-step transition relations equipped with sets of whole-program initial and final states, so we have to redesign those states to per-function setting. For the initial state, whereas CompCert constructs an initial memory and calls `main` with no arguments, we take the function pointer to call, the initial memory, and the list of arguments as parameters. For the final state, we take not only the return value, but also the memory state when we exit the function.

Consequently, the compiler correctness proofs have to change. Currently, CompCert uses a downward simulation diagram [20, 2.1] for each pass from Clight, then, thanks to the fact that the CompCert assembly language is deterministic (up to input values given by the environment), CompCert composes all of them together before turning them to a single upward simulation which actually entails that the compiled code refines the source code.

In this work, we follow a similar approach: for each individual pass, we prove per-function semantics preservation in a downward simulation flavor. We do not, however, turn it into an upward simulation, because the whole layer refinement proof is based on downward simulation, which is in turn turned into an upward simulation at whole-machine contextual refinement thanks to the determinism (up to the environment) of LAsm(L).

Memory state during compilation The main difference between CompCert and CompCertX lies in the memory given at the beginning of a function call.

In the whole-program setting, the initial state is the same across all languages, because it is uniquely determined by the global variables (which are preserved by compilation). On the other hand, in the middle of the execution when entering an arbitrary function, the memory in Clight is different from its assembly counterpart because CompCert introduces *memory transformations* such as memory injections or extensions [21, 5.4] to manage the callees' stack frames. This is actually advantageous for compilation of handling arguments and the return address.

For CompCertX, within the module being compiled, the same memory state mismatch also exists. At module entry, however, we cannot assume much about the memory state because it is given as a parameter to the semantics of each function in the module. In fact, this memory state is determined by the caller, so it may very well come from non-ClightX code (e.g., arbitrary assembly user code), thus we have to take the same memory as initial state across all the languages of CompCertX. It follows then that the arguments of the function already have to be present in the memory, following the calling convention imposed by the assembly language, even though ClightX does not read the arguments from memory.

Another difference between CompCert and CompCertX is the treatment of final memory states. In CompCert, only the return value of a program is observable at the end; the final memory state is not. By contrast, in CompCertX, the final memory state is passed back to the caller hence observable. Thus, it is necessary to account for memory transformations when relating the final states in the simulation diagrams.

Compilation refinement relation Finally, the per-function compiler correctness statement of CompCertX can be roughly summarized as this commutative diagram and formally defined below.

$$\begin{tikzcd}
 & v \arrow[d, hook, dashed, "j"] & m' \arrow[d, hook, dashed, "j"] & a' \arrow[d, equal] \\
\begin{matrix} l \\ \rho \end{matrix}, m, a \arrow[ur, "L_{\mathrm{C}}(f)"] \arrow[r, dashed, "L_{\mathrm{Asm}}(f)"] & \rho' & m'' & a' \\
l \approx m(\rho(\mathsf{ESP})) & & &
\end{tikzcd}$$

Definition 14. *Let L_C be a C layer interface, and L_{Asm} be an assembly layer interface. We say that L_C is simulated by L_{Asm} by compilation, written $L_C \leqslant^{comp} L_{Asm}$, if and only if, for any Γ, and for any execution $L_C(f)(l, m, a, v, m', a')$ of a primitive f of L_C for some list l of arguments and some return value v, from memory state m and abstract state a to m' and a', and for any register map ρ such that the following requirements hold:*

1. *the memory m contains the arguments l in the stack pointed to by $\rho(\mathsf{ESP})$*
2. EIP *points to the function f being called:* $\rho(\mathsf{EIP}) = (\Gamma(f), 0)$

Then, there is a primitive execution $L_{Asm}(f)(\rho, m, a, \rho', m'', a')$ and a memory injection j from m' to m'' preserving the addresses of m such that the following holds:

- *the values of callee-save registers in ρ are preserved in ρ';*
- $\rho'(\mathsf{EIP})$ *points to return address* $\rho(\mathsf{RA})$*;*
- *the return value contained in* $\rho'(\mathsf{EAX})$ *(for integers/pointers) or* $\rho'(\mathsf{FP0})$ *(for floating-points) is related to v by j;*

Theorem 2. *Let L be an assembly layer interface with all C-style primitives preserving memory transformations. Then, for any M:*

$$\llbracket M \rrbracket L \leqslant^{comp} \llbracket \mathrm{CompCertX}(M) \rrbracket L$$

More details can be found in the companion technical report.

6.2 Linking compiled code with assembly code

Contrary to traditional separate compilation, we target compiling ClightX functions that may be called by LAsm assembly code. Since the caller may be arbitrary LAsm code, not necessarily well-behaved code written in or compiled from ClightX, we have to assume that the memory we are given follows LAsm layout. When reasoning about memory states that involve compiled code, we then have to accommodate memory injections introduced by the compiler.

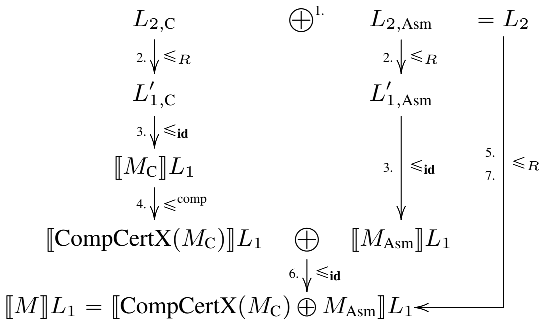
During a whole-machine refinement proof, the two memory states of the overlay and the underlay are related with a simulation relation R. However, consider when the higher (LAsm) code calls an overlay primitive, that, in the underlay, is compiled from ClightX. Because during the per-primitive simulation proofs we ignored the effects of the compiler, the memory injection introduced by the compiler may become a source of discrepancy. That is why we encapsulate, in R, a memory injection between the higher memory state and the lower memory state. This injection is identity until the lower state calls a compiled ClightX function. Then, at every such call, the layer simulation relation R can "absorb" compilation refinement on its right-hand side:

Lemma 2. *If L' and L_C are C overlays and L_{Asm} is an assembly underlay, with $L' \leqslant_R L_C$ and $L_C \leqslant^{comp} L_{Asm}$, then $L' \leqslant_R L_{Asm}$.*

Proof. If R encapsulates a memory injection j_0, and compilation introduces a memory injection j, then, the simulation relation R will still hold with the composed memory injection $j \circ j_0$. $\square$

Summary of the refinement proof with compilation and linking Finally, the outline of proving layer refinement $L_1 \vdash M : L_2$, where $M = \mathrm{CompCertX}(M_\mathrm{C}) \oplus M_\mathrm{Asm}$ is the union of a compiled ClightX module and an LAsm module, is summarized in the following steps, also shown in Fig. 13:

1. Split the overlay L_2 into two layer interfaces $L_{2,\mathrm{C}}$ and $L_{2,\mathrm{Asm}}$ where $L_{2,\mathrm{C}}$ is a C layer interface containing primitive specifications to be implemented by ClightX code (necessarily C-style) and $L_{2,\mathrm{Asm}}$ is an assembly layer interface containing all other primitives (implemented in LAsm), so that $L_2 = L_{2,\mathrm{C}} \oplus L_{2,\mathrm{Asm}}$.
2. For each such part of the overlay, design an intermediate layer interface $L'_{1,\mathrm{C}}$ and $L'_{1,\mathrm{Asm}}$ with the same abstract state type as L_1 (see Section 4.3), and prove $L_{2,\mathrm{C}} \leqslant_R L'_{1,\mathrm{C}}$ and $L_{2,\mathrm{Asm}} \leqslant_R L'_{1,\mathrm{Asm}}$ independently of the implementation.
3. For both intermediate layer interfaces, prove that they are implemented by modules M_C and M_Asm on top of L_1 respectively, i.e. $L'_{1,\mathrm{C}} \leqslant_{\mathbf{id}} \llbracket M_\mathrm{C} \rrbracket L_1$ and $L'_{1,\mathrm{Asm}} \leqslant_{\mathbf{id}} \llbracket M_\mathrm{Asm} \rrbracket L_1$.
4. Then, compile M_C: $\llbracket M_\mathrm{C} \rrbracket L_1 \leqslant^{\mathrm{comp}} \llbracket \mathrm{CompCertX}(M_\mathrm{C}) \rrbracket L_1$.



$$\begin{tikzcd}
L_{2,\mathrm{C}} \arrow[d, "2.\ \leqslant_R"] & \bigoplus^{1.} & L_{2,\mathrm{Asm}} \arrow[d, "2.\ \leqslant_R"] & = L_2 \arrow[ddddll, "5.\ 7.\ \leqslant_R"] \\
L'_{1,\mathrm{C}} \arrow[d, "3.\ \leqslant_{\mathbf{id}}"] & & L'_{1,\mathrm{Asm}} \arrow[dd, "3.\ \leqslant_{\mathbf{id}}"] & \\
\llbracket M_\mathrm{C} \rrbracket L_1 \arrow[d, "4.\ \leqslant^{\mathrm{comp}}"] & & & \\
\llbracket \mathrm{CompCertX}(M_\mathrm{C}) \rrbracket L_1 & \bigoplus \arrow[d, "6.\ \leqslant_{\mathbf{id}}"] & \llbracket M_\mathrm{Asm} \rrbracket L_1 & \\
& \llbracket M \rrbracket L_1 = \llbracket \mathrm{CompCertX}(M_\mathrm{C}) \oplus M_\mathrm{Asm} \rrbracket L_1 & &
\end{tikzcd}$$

Figure 13. Proof steps of layer refinement $L_1 \vdash_R M : L_2$

5. Using LLE-TRANS and LLE-MON to combine 2. and 3., we have: $L_{2,\mathrm{C}} \oplus L_{2,\mathrm{Asm}} \leqslant_R L'_{1,\mathrm{C}} \oplus L'_{1,\mathrm{Asm}} \leqslant_{\mathbf{id}} \llbracket M_\mathrm{C} \rrbracket L_1 \oplus \llbracket M_\mathrm{Asm} \rrbracket L_1$
   On the C side (left of $\oplus$), Lemma 2 shows that $\leqslant_R$ absorbs $\leqslant^{\mathrm{comp}}$. By 4.: $L_{2,\mathrm{C}} \oplus L_{2,\mathrm{Asm}} \leqslant_R \llbracket \mathrm{CompCertX}(M_\mathrm{C}) \rrbracket L_1 \oplus \llbracket M_\mathrm{Asm} \rrbracket L_1$
6. From the soundness of HCOMP (proof in TR [13]), and because $M = \mathrm{CompCertX}(M_\mathrm{C}) \oplus M_\mathrm{Asm}$, we have:
   $$\llbracket \mathrm{CompCertX}(M_\mathrm{C}) \rrbracket L_1 \oplus \llbracket M_\mathrm{Asm} \rrbracket L_1 \leqslant_{\mathbf{id}} \llbracket M \rrbracket L_1$$
7. Finally, by combining 5. and 6., we have $L_{2,\mathrm{C}} \oplus L_{2,\mathrm{Asm}} \leqslant_R \llbracket M \rrbracket L_1$. Since $L_2 = L_{2,\mathrm{C}} \oplus L_{2,\mathrm{Asm}}$, by using LLE-UB-LEFT and LLE-COMM, we have: $L_2 \leqslant_R \llbracket M \rrbracket L_1 \leqslant_{\mathbf{id}} \llbracket M \rrbracket L_1 \oplus L_1 \leqslant_{\mathbf{id}} L_1 \oplus \llbracket M \rrbracket L_1$, thus we get $L_1 \vdash_R M : L_2$.

7. Case study: certified OS kernels

To demonstrate the power of our new languages and tools, we have applied our new layered approach to specify and verify four variants of mCertiKOS kernels in the Coq proof assistant. This section describes these kernels and the benefits of the approach.

The mCertiKOS base kernel is a simplified uniprocessor version of the CertiKOS kernel [12] designed for the 32 bit x86 architecture. It provides a multi-process environment for user-space applications using separate virtual address space, where the communications between different applications are established by message passing. The mCertiKOS-hyp kernel, built on top of the base kernel, is a realistic hypervisor kernel that can boot recent versions of unmodified Linux operating systems (Debian 6.0.6 and Ubuntu 12.04.2). The mCertiKOS-rz kernel extends the hypervisor supporting "ring 0" processes, hosting "certifiably safe" services and application programs inside the kernel address space. Finally, we strip the last kernel down to the mCertiKOS-emb kernel, removing virtualization, virtual memory, and user-space interrupt handling. This results in a minimal operating system suitable for embedded environments.

The layer structures of these kernels are shown in the top half of Fig. 14; each block in the top half represents a collection of sub-layers shown in the bottom half (as we zoom in on mCertiKOS-hyp).

mCertiKOS The layered approach is the key to our success in fully certifying a kernel. In Sec. 4.3, we have shown how to define getters and setters for abstract data types like those in Fig. 8, allowing higher layers to manipulate abstract states. Furthermore, layering is also crucial to certification of thread queues as discussed in Sec. 2. Instead of directly proving that a C linked-list implements a functional list, we insert an intermediate layer as shown in Fig. 1 to divide the difficult task into two steps.

These may look like mere proof techniques for enabling abstract states or reducing proof effort, but they echo the following mantra which makes our certification more efficient and scalable:

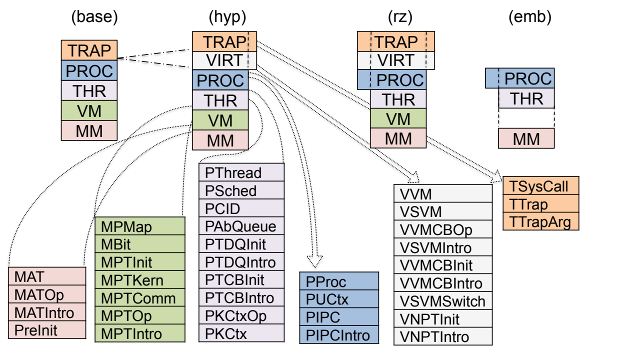


Figure 14. Various mCertiKOS layer structures. Layer short-hands: TRAP: interrupt handling; VIRT: virtualization; PROC: process management; THR: thread management; VM: virtual memory; MM: physical memory management.

Abstract in minimal steps, specify full *behavior, and hide* all *underlying details.*

This is also how we prove the overall contextual correctness guarantees for all system calls and interrupt handlers. Fig. 15 shows the call graph of the page fault handler, including all functions called both directly and indirectly. Circles indicate functions, solid arrows mean primitive invocations, and faint dashed lines are primitives that are translated by all the layers they pass through.

Defined in TSysCall layer interface, the page fault handler makes use of proc_exit and proc_start, both defined in PProcd layer interface. Since the invocations of them are separated by other primitive calls, one may expect that the invariants need to be re-established or the effects of the in-between calls re-interpreted. Fortunately, as our mantra suggests, when the in-between layers translate the two primitives to TTrap layer interface, the behaviors of them are *fully* specified in terms of TTrap's abstract states, and the invariants of PProc layer interface are considered the underlying details and have *all* been hidden. This is especially important for calls like proc_exit to ikern_set which span over 20 layers with the abstract states so different that direct translation is not feasible.

Finally, kernel initialization is another difficult task that has been missing from other kernel verification projects. The traditional kernel initialization process is not compatible with *"specify full behavior and hide all underlying details."* For example, start_kernel in Linux kernel makes a sequence of calls to module initializations. mCertiKOS's initialization (see its call graph in Fig. 16) is a *chain* of calls to layer initializations; this pattern complies with the guideline that initializing one layer should hide the detail about initializing the lower layers. Without layering, the specifications of *all* functions will be populated with initialization flags for each module they depend on. This makes encapsulation harder and could also lead to a quadratic blowup in size and proving effort.

mCertiKOS-hyp The mCertiKOS-hyp kernel provides core primitives to build full-fledged user-level hypervisors by supporting one of the two popular hardware virtualization technologies – AMD SVM. The primitives include the operations for manipulating the virtual machine status, handling VMEXITs, starting or stopping a virtual machine, *etc*. The details of virtualization, e.g., the virtual machine control block and the nested page table, are hidden from the guest applications. The hypervisor functionalities are implemented in nine layers and then inserted in between process management and interrupt handling layers. The layered approach allows us to do so while (1) only modeling virtualization-specific structures when needed; (2) retaining primitives in the layer interface PProc by systematic lifting; and (3) adding new primitives (including a new initialization function) guaranteed not to interfere with existing primitives.

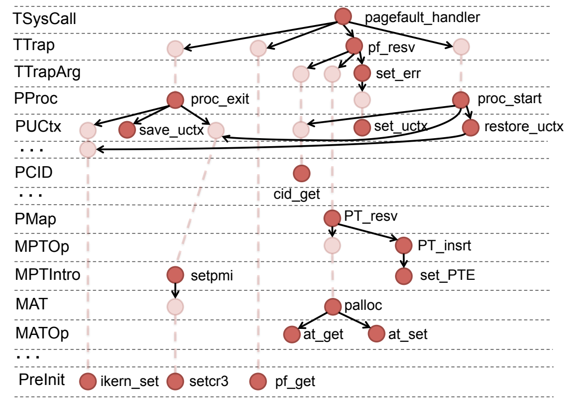


Figure 15. Call graph of the page fault handler

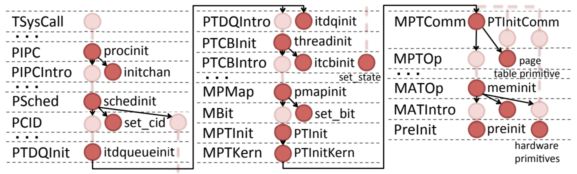


Figure 16. Call graph of mCertiKOS initializer

mCertiKOS-rz The mCertiKOS-rz kernel explores a different dimension—instead of adding intermediate layers, we augmented a few existing layers (in mCertiKOS-hyp) with support of ring 0 processes. The main modification is at PProc, where an additional kind of threads is defined. However, all the layers between PProc and TSysCall also need to be extended to expose the functionality as system calls. Thankfully, since all the new primitives are already described in deep specifications, lifting them to system calls only requires equality reasoning in Coq.

mCertiKOS-emb The mCertiKOS-emb kernel cuts features down to a bare minimum: it does not switch to user mode, hence does not require memory protection and does not provide system call interfaces. This requires *removing* features instead of adding them. Since the layered structure minimizes entanglements by eliminating unnecessary dependencies and code coupling, the removal process was relatively easy and straightforward. Moreover, removing the top 12 layers requires no additional specifications for those now top-level primitives—deep specifications are suitable for both internal reasonings and external descriptions. Thread and process management layers now sit directly on top of physical memory management; virtual memory is never enabled. The layers remain largely the same barring the removal of primitives mentioning page tables.

Evaluation and limitations The planning and development of mCertiKOS took 9.5 person months plus 2 person months on linking and code extraction. With the infrastructure in place, mCertiKOS-hyp only took 1.5 person months to finish, and mCertiKOS-rz and mCertiKOS-emb take half a person month each. The kernels are written, layer by layer, in LAsm and ClightX abstract syntaxes along with driver functions specifying how to compose (link) them. All of those are in Coq for the proofs to refer to. We utilize Coq's code extraction to get an OCaml program which contains CompCertX, the abstract syntax trees of the kernels, and the driver functions, which invoke CompCertX on pieces of ClightX code and generate the full assembly file. The output of the OCaml program is then fed to an assembler to produce the kernel executable.

With the device drivers (running as user processes) and a cooperative scheduler, most of the benchmarks in lmbench are under 2x slowdown running in mCertiKOS-hyp, well within expected over-head. Ring 0 processes, not used in the above experiment, can easily lower the number as we measured one to two orders of magnitude reduction in the number of cycles needed to serve system calls.

Because the proof was originally developed directly in terms of abstract machines and program transformations, the current code base does not yet reflect the calculus presented in Sec. 3 in its entirety. Notably, vertical composition is done at the level of the whole-machine contextual refinements obtained by applying the soundness theorem to each individual abstraction layer.

Outside our verified kernels (mCertiKOS-hyp consists of about 3000 lines of C and assembly), there are 300 lines of C and 170 lines of x86 assembly code that are not verified yet: the preinit procedure, the ELF loader used by user process creation, and functions such as memcpy which currently cannot be verified because of a limitation arising from the CompCert memory model. Device drivers are not verified because LAsm lacks device models for expressing the correctness statement. Finally, the CompCert assembler for converting LAsm into machine code remains unverified.

8. Related work

Hoare-style program verification Hoare logic [14] and its modern variants [32, 2, 26] were introduced to prove strong (partial or total) correctness properties of programs annotated with pre- and postconditions. A total-correctness Hoare triple $[P]C[Q]$ often means a refinement between the implementation C and the specification $[P, Q]$: given any state S, if the precondition $P(S)$ holds, then the command C can run safely and terminate with a state that satisfies Q. Though not often done, it is also possible to introduce auxiliary/ghost states to serve as "abstract states" and prove that a program implements a specification via a simulation.

Our layer language can be viewed as a novel way of imposing a module system over Hoare-style program verification. We insist on using interfaces with deep specifications and we address the "conflicting abstract states" problem mentioned in Sec. 2. Traditional program verification does not always use deep specification (for pre- and post-conditions) so the module interfaces (e.g., $[P, Q]$) may allow some safe but unwanted behaviors. Such gap is fine if the goal is to just prove safety (as in static type-checking), but if we want to prove the strong contextual correctness property across module boundaries, it is important that each interface accurately describes the functionality and scope of the underlying implementation.

In addition to the obvious benefits on compositionality, our layered approach also enables a new powerful way of combining programming- and specification languages in a single setting. Each layer interface enables a new programming language at a specific abstraction level, which is then used to implement layers at even higher levels. As we move up the layer hierarchy, our programming language gets closer and closer to the specification language—it can call primitives at higher abstraction levels but it still supports general-purpose programming (e.g., in ClightX).

Interestingly, we did not need to introduce any program logic to verify our OS kernel code. Instead, we verify it directly using the ClightX (or LAsm) language semantics (which is already conveniently parameterized over a layer interface). In fact, unlike Hoare logic which shows that a program (e.g., C) refines a specification (e.g., $[P, Q]$), we instead show there is a downward simulation from the specification to the program. As in CompCert, we found this easier to prove and we can do this because both our specification and language semantics are deterministic relative to external events.

Stepwise program refinement Dijkstra [9] proposed to "realize" a complex program by decomposing it into a hierarchy of linearly ordered "abstract machines." Based on this idea, the PSOS team at SRI [27] developed the Hierarchical Development Methodology (HDM) and applied HDM to design and specify an OS using 20 hierarchically organized modules. HDM was difficult to be rigorously applied in practice, probably because of the lack of powerful specification and proof tools. In this paper, we advance the HDM paradigm by using a new formal layer language to connect multiple layers and by implementing all certified layers and proofs in a modern proof assistant. We also pursued decomposition more aggressively since it made our verification task much easier.

Morgan's refinement calculus [25] is a formalized approach to Dijkstra's stepwise refinement. Using this calculus, a high-level specification can be refined through a series of correctness-preserving transformations and eventually turned into an efficient executable. Our work imposes a new layer language to enhance compositional reasoning. We use ClightX (or LAsm) and the Coq logic as our "refinement" language, and use a certified layer (with deep specification) to represent each such correctness-preserving transformation. All our ClightX and LAsm instances have executable semantics and can be compiled and linked using our new CompCertX compiler.

Separate compilation for CompCert Compositional compiler correctness is an extremely challenging problem [3, 15], especially when it involves an open compiler with multiple languages [29]. In the context of CompCert, a recent proposal [4] aims to tackle the full Clight language but it has not been fully implemented in the CompCert compiler. While our CompCertX compiler proves a stronger correctness theorem for each ClightX layer, the ClightX language is subtly different from the original full-featured Clight language. Within each ClightX layer, all locally allocated memory blocks (e.g., stack frames) cannot be updated by functions defined in another layer. This means that ClightX does not support the same general "stack-allocated data structures" as in Clight. This is fine for our OS kernels since they do not allocate any data structures on stack, but it means that CompCertX can not be regarded as a full featured separate compiler for CompCert.

OS kernel verification The seL4 team [17] were the first to build a proof of functional correctness for a realistic microkernel. The seL4 work is impressive in that all the proofs were done inside a modern mechanized proof assistant. They have shown that the behaviors of 7500 lines of their C code always follow an abstract specification of their kernel. To make verification easier, they introduced an intermediate executable specification to hide C specifics. Both their abstract and executable specifications are "monolithic" as they are not divided into layers to support abstraction among different kernel modules. These kernel interdependencies led to more complex invariants which may explain why their effort took 11 person years.

The initial seL4 effort was done completely at the C level so it does not support many assembly level features such as address translation. This also made verification of assembly code and kernel initialization difficult (1200 lines of C and 500 lines of assembly are still unverified). It is also unclear how to use their verified kernel to reason about user-level programs since they would be running in a different address space. Our certified kernels, on the other hand, directly model assembly-level machines that support all kernel/user and host/guest programs. Memory access to a user-level address space must go through a page table, and memory access in a guest virtual machine must go through a nested page table. We thus had no problem verifying our kernel initialization or assembly code.

Modular verification of low-level code Vaynberg and Shao [36] also used a layered approach to verify a small virtual memory manager. Their layers are not linearly ordered; instead, their seven abstract machines form a DAG with potential upcalls (i.e., calls from a lower layer to upper ones). As a result, their initialization function (an upcall) was much harder to verify. Their refinement proofs between layers are insensitive to termination, from which they can only prove partial correctness but not the strong contextual correctness property which we prove in our current work.

Feng *et al.* [11] developed OCAP, an open framework for linking components verified in different domain-specific program logics. They verified a thread library with hardware interrupts and preemption [10] using a variant of concurrent separation logic [28]. They decomposed the thread implementation into a sequential layer (with interrupts disabled) and a concurrent layer (with interrupts enabled). Chlipala [8] developed Bedrock, an automated Coq library to support verified low-level programming. All these systems aimed to prove *partial correctness* only, so they are quite different from the layered simulation proofs given in this paper.

9. Conclusions

Abstraction layers are key techniques used in building large-scale computer software and hardware. In this paper, we have presented a novel language-based account of abstraction layers and shown that they are particularly suitable for supporting abstraction over deep specifications, which is essential for compositional verification of strong correctness properties. We have designed a new layer language and imposed it on two different core languages (ClightX and LAsm). We have also built a verified compiler from ClightX to LAsm. By aggressively decomposing each complex abstraction into smaller abstraction steps, we have successfully developed several certified OS kernels that prove deeper properties (contextual correctness), contain smaller trusted computing bases (all code verified at the assembly level), require significantly less effort (3000 lines of C and assembly code proved in less than 1 person year), and demonstrate strong support for extensibility (layers are heavily reused in different certified kernels). We expect that both deep specifications and certified abstraction layers will become critical technologies and important building blocks for developing large-scale certified system infrastructures in the future.

Acknowledgments We thank Quentin Carbonneaux, David Costanzo, Rance DeLong, Xinyu Feng, Bryan Ford, Liang Gu, Jan Hoffmann, Hongjin Liang, Joshua Lockerman, Peter Neumann, David Pichardie, members of the CertiKOS team at Yale, and anonymous referees for helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, NSF grants 1065451 and 0915888, and ONR Grant N00014-12-1-0478. It is also supported in part by China Scholarship Council and National Natural Science Foundation of China (NSFC grants 61229201 and 61202052). Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] C. Y. Baldwin and K. B. Clark. *Design Rules: Volume 1, The Power of Modularity*. MIT Press, March 2000.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. 4th Symp on Formal Methods for Components and Objects*, 2005.

[3] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP'09*, pages 97–108, 2009.

[4] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *ESOP'14*, pages 107–127, 2014.

[5] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3):263–288, 2009.

[6] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *PLDI'14*.

[7] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL'10*, pages 57–69, 2010.

[8] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245, 2011.

[9] E. W. Dijkstra. Notes on structured programming. In *Structured programming*, pages 1–82. Academic Press, 1972.

[10] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI'08*, pages 170–182, June 2008.

[11] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *VSTTE'08*, pages 54–69, 2008.

[12] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: a certified kernel for secure cloud computing. In *APSys '11*, 2011.

[13] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. Yale Univ. Technical Report YALEU/DCS/TR-1500; `http://flint.cs.yale.edu/publications/dscal.html`, Oct. 2014.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[15] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL'12*, pages 59–72.

[16] D. Jackson. *Software abstractions: logic, languages, and analysis*. The MIT Press, 2012.

[17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, et al. seL4: Formal verification of an OS kernel. In *SOSP'09*, pages 207–220, October 2009.

[18] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[19] X. Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2014.

[20] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[21] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformation. *J. Automated Reasoning*, 41(1):1–31, 2008.

[22] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[24] J. C. Mitchell. Representation independence and data abstraction. In *POPL'86*, pages 263–276, January 1986.

[25] C. C. Morgan. *Programming from specifications, 2nd Edition*. Prentice-Hall, 1994.

[26] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP'06*, pages 62–73, Sept. 2006.

[27] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: its system, its applications, and proofs. Technical Report CSL-116, SRI, May 1980.

[28] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04*, pages 49–67, 2004.

[29] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP'14*, pages 128–148, 2014.

[30] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[31] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[32] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[33] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), 2013.

[34] M. Spivey. *The Z Notation: A reference manual*. Prentice Hall, 1992.

[35] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2014.

[36] A. Vaynberg and Z. Shao. Compositional verification of a baby virtual memory manager. In *CPP'12*, pages 143–159, Dec 2012.