CompCertO: Compiling Certified Open C Components*

Jérémie Koenig
Yale University
New Haven, CT, USA
jeremie.koenig@yale.edu

Zhong Shao
Yale University
New Haven, CT, USA
zhong.shao@yale.edu

Abstract

Since the introduction of CompCert, researchers have been refining its language semantics and correctness theorem, and used them as components in software verification efforts. Meanwhile, artifacts ranging from CPU designs to network protocols have been successfully verified, and there is interest in making them interoperable to tackle end-to-end verification at an even larger scale.

Recent work shows that a synthesis of game semantics, refinement-based methods, and abstraction layers has the potential to serve as a common theory of certified components. Integrating certified compilers to such a theory is a critical goal. However, none of the existing variants of CompCert meets the requirements we have identified for this task.

CompCertO extends the correctness theorem of CompCert to characterize compiled program components directly in terms of their interaction with each other. Through a careful and compositional treatment of calling conventions, this is achieved with minimal effort.

Keywords: Compositional Compiler Correctness, Game Semantics, Simulation Convention, Language Interface

ACM Reference Format:
Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3453483.3454097

*The present version, published by the authors as Yale University Technical Report YALEU/DCS/TR-1556 [13] includes supplementary appendices after the References section.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454097

1 Introduction

Over the past decade, researchers have been able to formally verify various key components of computer systems, including compilers [15, 16, 25], operating system kernels [6, 7, 11], file systems [4] and processor designs [3, 5]. Building on these successes, the research community is attempting to construct large-scale, heterogeneous certified systems by using formal specifications as interfaces between the correctness proofs of various components [2]. The ongoing design of suitable semantic frameworks is an important step towards this goal. However, incorporating certified compilers into frameworks of this kind presents a number of difficulties.

1.1 Compositional Compiler Correctness

Compiler correctness is often formulated as a *semantics preservation* property, asserting that the semantics of the compiled program $\mathrm{C}(p)$ are related in some particular way to the semantics of the source program p:

$$[\![p]\!]_{\mathrm{S}} \sim [\![\mathrm{C}(p)]\!]_{\mathrm{T}}\,. \tag{1}$$

For whole-program compilers, semantics preservation is straightforward enough. In CompCert, the semantics of the source and target programs are given as labeled transition systems, and the relation $\sim$ is a simulation property.

However, practical applications involve program *components* which we want to compile and verify separately from each other. In principle, the use of a compositional semantics enables the formulation of (1) at the level of individual components. Unfortunately, traditional approaches to compositional semantics fare poorly in the presence of advanced language features, or of the kind of abstraction involved in the compilation process. For CompCert, early attempts along these lines have proven challenging [21, 23].

As a result, common wisdom holds semantics preservation to be a lost cause for compositional compiler correctness [20]. Instead, research has focused on compositional reasoning methods based on contextual refinement, side-stepping the need for compositional semantics preservation [10, 22].

1.2 Decomposing Heterogeneous Systems

Unfortunately, these methods share an intrinsic limitation: they presuppose the existence of a completed system to be proven correct, and compositionality only operates within its

boundary. This becomes a serious impediment in the context of large-scale heterogeneous systems.

Example 1.1. Consider the problem of verifying a network interface card (NIC) driver. The NIC and its driver are closely coupled, but the details of their interaction are irrelevant to the rest of the system; these details should not leak into our large-scale reasoning. Instead, we wish to treat the NIC and its driver as a unit, and establish a direct relationship between C calls into the driver and network communication. Together, the NIC and driver implement a specification $\sigma : \mathit{Net} \to C$, meaning they *use* the interface *Net* modeling the network, and *provide* the interface C modeling C function calls.

The driver code could be specified (σ_{drv}) and verified at the level of CompCert semantics, whereas device I/O primitives (σ_{io}) and the NIC itself (σ_{NIC}) would be specified as additional components in the context of a richer model:

$$\sigma_{\mathsf{NIC}} : \mathit{Net} \to \mathit{IO} \qquad \sigma_{\mathsf{io}} : \mathit{IO} \to C \qquad \sigma_{\mathsf{drv}} : C \to C$$

By reasoning about their interaction, it would be possible to establish a relationship between the overall specification $\sigma : \mathit{Net} \to C$ and the composition $\sigma_{\mathsf{drv}} \circ \sigma_{\mathsf{io}} \circ \sigma_{\mathsf{NIC}}$. Then a *compiler of certified components* should help us transport specifications and proofs obtained at the C level to the compiled code operating at the level of assembly ($\sigma' : \mathit{Net} \to \mathcal{A}$).

Under existing contextual approaches, the NIC driver can only be specified and verified in terms of its interactions at the boundary of the C program code. Since abstracting away from these interactions is the role of a driver in the first place, this is a serious limitation. To be sure, existing techniques could be extended to address this specific problem. For example, the NIC hardware model could be brought within the scope of the "whole program" being considered, and the exchange of Ethernet packets modeled as part of its observable behavior. However, this approach does not scale.

Example 1.1 is by no means a contrived corner case. In fact, patterns of this kind are pervasive even in more mundane situations. Programmers often use libraries which mediate access to external resources (network services, file systems, user interfaces). Proper high-level specifications for software components of this kind must model these resources. It would rapidly become burdensome to expect the verification framework to fix in advance the dozens or hundreds of kinds of resources which may be involved in the course of verifying a large-scale system.

Fortunately, advances in compositional semantics offer a realistic path to tackling problems of this kind. In particular, game semantics (§2.1) provides a general and expressive framework to model interactions between typed components. Recent work proposes integrating dual nondeterminism and refinement into this framework, extending it with powerful mechanisms to account for abstraction [12]. Establishing a compatible compiler correctness result is an important test of this approach and practical next step.

1.3 Contributions

This paper introduces CompCertO,[1] the first extension of CompCert satisfying the following requirements:

1. A semantics is given for source and target *components*.
2. The correctness theorem relates the behaviors of corresponding source and target components *directly*.
3. The C calling convention is modeled explicitly.
4. A form of certified component *linking* is provided.
5. Changes to existing proofs of CompCert are minimal.

Each of these requirements is fulfilled by some existing CompCert extension, however none satisfies them all.

We generalize CompCert semantics to express interactions between components (§3), using *language interfaces* to describe the form of these interactions and *simulation conventions* to describe the correspondence between the interfaces of source and target languages. The behavior of composite programs is specified by a *horizontal composition* operator (§3.2), which is shown to be correctly implemented by the existing linking operator for assembly programs (§3.3). To facilitate reasoning about CompCert semantics and simulation proofs, we define a notion of *CompCert Kripke logical relation* (§4). We then use a rich simulation convention algebra to derive our main compiler correctness statement (Thm. 3.8) from the simulation properties of individual passes (§5).

2 Main Ideas

2.1 Game Semantics

Game semantics is a form of denotational semantics which incorporates some operational aspects. Typically, game semantics interpret types as two-player games and terms as strategies for these games.

Games describe the form of the interaction between a program component (the *system*) and its execution context (the *environment*). Strategies specify which move the system plays for each possible configuration of the game.

Configurations are usually identified with sequences of moves (*plays*), and strategies with the set of configurations a component can reach. This representation makes game semantics similar to the trace semantics of process algebras, but game semantics is distinguished by a strong polarization between the actions of the system and those of the environment. This confers an inherent "rely-guarantee" flavor to games which facilitates compositional reasoning [1].

Games. A game is defined by a set of moves players will choose from, as well as a stipulation of which sequences of moves are valid. We focus on two-player, alternating games where the environment plays first and where the players each contribute every other move.

[1] This paper discusses CompCertO version 0.1, available in the git repository at https://github.com/CertiKOS/compcert/tree/compcerto.

When typesetting examples, we underline the moves of the system; a valid play in the game of chess may look like:

$$e2e4 \cdot \underline{c7c5} \cdot c2c3 \cdot \underline{d7d5} \cdots$$

The games we use to model low-level components will rely on the following constructions.

Type Structure. Game semantics allows simple games to be combined into more sophisticated ones, which can then be used to interpret compound types. For example, in the game $A \times B$ the environment initially chooses whether to play an instance of A or an instance of B. The game $A \to B$ usually consists of an instance of B played together with instances of A started at the discretion of the system, where the roles of the players are reversed.

The games we start from are particularly simple. We call each one a *language interface*. Their moves are partitioned into questions and answers, where questions correspond to function invocations and answers return control to the caller.

Definition 2.1. A *language interface* is a tuple $A = \langle A^\circ, A^\bullet \rangle$, where A° is a set of *questions* and $A^\bullet$ is a set of *answers*.

We focus on games of the form $A \to B$, where A and B are language interfaces. The valid plays are the sequences

$$q \cdot \underline{m_1} \cdot n_1 \cdots \underline{m_k} \cdot n_k \cdot \underline{r} \in B^\circ (A^\circ A^\bullet)^* B^\bullet$$

and all their prefixes. They describes a program component responding to an incoming call q. The component performs a series of external calls $m_1 \ldots m_k$ which yield the results $n_1 \ldots n_k$. Finally, the component returns from the incoming call with the result r. The arrows show the correspondence between questions and answers but are not part of the model.

Example 2.2. We use a simplified version of C and assembly to illustrate some of the principles behind our model. Consider the program components in Fig. 1. The behavior of B.c as it interacts with A.c is described by plays of the form:

$$\mathsf{sqr}(3) \cdot \underline{\mathsf{mult}(3,3)} \cdot 9 \cdot \underline{9} \tag{2}$$

This corresponds to the game $\tilde{C} \to \tilde{C}$ for a language interface $\tilde{C} := \langle \mathsf{ident} \times \mathsf{val}^*, \mathsf{val} \rangle$. Questions specify the function to invoke and its arguments; answers carry the return value.

To describe the behavior of `A.s` and `B.s`, we use a set of registers $R := \{\mathsf{pc}, \mathsf{eax}, \mathsf{ebx}, \mathsf{ecx}\}$ (pc is the program counter) together with a stack of pending return addresses. The corresponding language interface can be defined as $\tilde{\mathcal{A}} := \langle \mathsf{val}^R \times \mathsf{val}^*, \mathsf{val}^R \times \mathsf{val}^* \rangle$. A possible execution of `B.s` is:

$$\underline{\begin{bmatrix} \mathsf{pc} \mapsto \mathsf{sqr} \\ \mathsf{eax} \mapsto 42 \\ \mathsf{ebx} \mapsto 3 \\ \mathsf{ecx} \mapsto 7 \\ \mathit{stack}: x\vec{k} \end{bmatrix}} \underline{\begin{bmatrix} \mathsf{pc} \mapsto \mathsf{mult} \\ \mathsf{eax} \mapsto 42 \\ \mathsf{ebx} \mapsto 3 \\ \mathsf{ecx} \mapsto 3 \\ \mathit{stack}: \mathsf{L}x\vec{k} \end{bmatrix}} \begin{bmatrix} \mathsf{pc} \mapsto \mathsf{L} \\ \mathsf{eax} \mapsto 9 \\ \mathsf{ebx} \mapsto 3 \\ \mathsf{ecx} \mapsto 3 \\ \mathit{stack}: x\vec{k} \end{bmatrix} \underline{\begin{bmatrix} \mathsf{pc} \mapsto x \\ \mathsf{eax} \mapsto 9 \\ \mathsf{ebx} \mapsto 3 \\ \mathsf{ecx} \mapsto 3 \\ \mathit{stack}: \vec{k} \end{bmatrix}} \tag{3}$$

The correspondence between (2) and (3) is determined by the C calling convention in use. This is discussed in §2.4.

A.c	`int mult(n, p) {`
	`  return n * p;`
	`}`
		A.s	`mult: %eax := %ebx`
			`%eax *= %ecx`
			`ret`

B.c	`int sqr(n) {`
	`  return mult(n, n);`
	`}`
		B.s	`sqr: %ecx := %ebx`
			`call mult`
			`L: ret`

Figure 1. Two simple C compilation units and corresponding assembly code. For this example, the calling convention stores arguments in the registers `%ebx` and `%ecx` and return values in the register `%eax`.

2.2 CompCertO

Under the traditional CompCert semantics, programs are interpreted as transition systems which define strategies for the game $\mathcal{E} \to \mathcal{W}$. They are run without any parameters and produce a single integer denoting their exit status; the corresponding language interface is $\mathcal{W} := \langle \mathbb{1}, \mathsf{int} \rangle$, where $\mathbb{1} = \{*\}$ is the unit set and int is the set of machine integers. Interaction with the environment is captured as a sequence of events from a predefined set. These events, which can be described by a language interface $\mathcal{E}$, correspond mainly to system calls and accesses to volatile variables.

Semantic Model. In CompCertO, to model components and their interactions, a transition system $L : A \twoheadrightarrow B$ will describe a strategy for the game $A \times \mathcal{E} \to B$. The language interface B describes how a component can be activated, and the ways in which it can return control to the caller. The language interface A describes the external calls that the component may perform during its execution.

This flexibility allows us to treat interactions at a level of abstraction adapted to each language. For example, the semantics of the source language Clight has type $C \twoheadrightarrow C$. The questions of C specify a function to call, argument values, and the state of the memory at the time of invocation; the answers specify a return value and an updated memory state. On the other hand, the target language Asm uses $\mathcal{A} \twoheadrightarrow \mathcal{A}$, where $\mathcal{A}$ describes control transfers in terms of processor registers rather than function calls (see §3.2).

Simulations. CompCert uses simulation proofs to establish a correspondence between the externally observable behaviors of the source and target programs of each compilation pass. The internal details of simulation relations have no bearing on this correspondence, so these details can remain hidden to fit a uniform and transitive notion of pass correctness. This makes it easy to derive the correctness of the whole compiler from the correctness of each pass.

Unfortunately, to achieve compositionality across compilation units, our model must reveal details about component interactions which were previously internal. Since many passes transform these interactions in specialized ways, this breaks the uniformity of pass correctness properties.

Existing work attempts to recover this uniformity by using more general notions of correctness covering all passes [22, 23] or by delaying pass composition so that it operates on closed semantics only [10, 22]. Unfortunately, these techniques either conflict with our requirement #2, make proofs more complex, or cascade into subtle "impedance mismatch" problems requiring their own solutions (see §6).

By contrast, we capture the particularities of each simulation proof by introducing a notion of *simulation convention* expressing the correspondence between source- and target-level interactions. To describe simulation conventions and reason about them, we use logical relations.

2.3 Logical Relations

Logical relations are structure-preserving relations in the way homomorphisms are structure-preserving maps. However, logical relations are more compositional than homomorphisms, because they do not suffer from the same problems in the presence of mixed-variance constructions like the function arrow [9]. In the context of typed languages, this means that type-indexed logical relations can be defined by recursion over the structure of types.

Logical relations can be of any arity, but we restrict our attention to binary logical relations. Given an algebraic structure $\mathcal{S}$, a *logical relation* between two instances S_1, S_2 of $\mathcal{S}$ is a relation R between their carrier sets, such that the corresponding operations of S_1 and S_2 take related arguments to related results. We write $R \in \mathcal{R}(S_1, S_2)$.

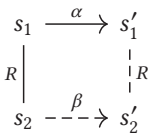
Example 2.3. A monoid is a set with an associative operation $\cdot$ and a unit ϵ. A *logical relation of monoids* between $\langle A, \cdot_A, \epsilon_A \rangle$ and $\langle B, \cdot_B, \epsilon_B \rangle$ is a relation $R \subseteq A \times B$ such that:

$$(u\ R\ u' \wedge v\ R\ v' \implies u \cdot_A v\ \ R\ \ u' \cdot_B v') \wedge \epsilon_A\ R\ \epsilon_B \qquad (4)$$

Logical relations between multisorted structures consist of one relation for each sort, between the corresponding carrier sets. In the case of structures which include type operators, we can associate to each base type A a relation over its carrier set $[\![A]\!]$, and to each type operator $T(A_1, \ldots, A_n)$ a corresponding *relator*: given relations $R_1, \ldots, R_n$ over the carrier sets $[\![A_1]\!], \ldots, [\![A_n]\!]$, the relator for T will construct a relation $T(R_1, \ldots, R_n)$ over $[\![T(A_1, \ldots, A_n)]\!]$. Relators for some common constructions are shown in Fig. 2. In this framework, the proposition (4) can be reformulated as:

$$\cdot_A\ [R \times R \to R]\ \cdot_B \wedge \epsilon_A\ R\ \epsilon_B\,.$$

Example 2.4. A simulation relation between the transition systems $\alpha : A \to \mathcal{P}(A)$ and $\beta : B \to \mathcal{P}(B)$ is a relation $R \subseteq A \times B$ satisfying the following property:



$$\forall s_1\ s_2\ s_1'\,.\ \alpha(s_1) \ni s_1' \wedge s_1\ R\ s_2 \implies \exists s_2'\,.\ \beta(s_2) \ni s_2' \wedge s_1'\ R\ s_2'$$

$$\begin{aligned}
x\ [R_1 \times R_2]\ y &\iff \pi_1(x)\ [R_1]\ \pi_1(y) \wedge \pi_2(x)\ [R_2]\ \pi_2(y) \\
x\ [R_1 + R_2]\ y &\iff (\exists\, x_1\ y_1\,.\, x_1\ [R_1]\ y_1 \wedge x = i_1(x_1) \wedge y = i_1(y_1)) \\
&\quad \vee (\exists\, x_2\ y_2\,.\, x_2\ [R_2]\ y_2 \wedge x = i_2(x_2) \wedge y = i_2(y_2)) \\
f\ [R_1 \to R_2]\ g &\iff \forall\, x\ y\,.\, x\ [R_1]\ y \Rightarrow f(x)\ [R_2]\ g(y) \\
A\ [\mathcal{P}^{\leq}(R)]\ B &\iff \forall\, x \in A\,.\, \exists\, y \in B\,.\, x\ [R]\ y
\end{aligned}$$

Figure 2. A selection of relators

Using the relators in Fig. 2, we can express the same property concisely and compositionally as $\alpha\ [R \to \mathcal{P}^{\leq}(R)]\ \beta$.

Kripke Relations. Since relations for stateful languages often depend on the current state, Kripke logical relations are parametrized over a set of state-dependent *worlds*. Components related at the same world are guaranteed to be related in compatible ways. We use the following notations.

Definition 2.5. A *Kripke* relation is a family of relations $(R_w)_{w \in W}$. We write $R \in \mathcal{R}_W(A, B)$ for a W-indexed Kripke relation between the sets A and B. For $w \in W$ we write:

$$[w \Vdash R] \coloneqq R_w \qquad [\Vdash R] \coloneqq \bigcap\nolimits_w R_w$$

A simple relation $R \in \mathcal{R}(A, B)$ can be promoted to a Kripke relation $\lceil R \rceil \in \mathcal{R}_W(A, B)$ by defining $\lceil R \rceil_w \coloneqq R$ for all $w \in W$. More generally, for an n-ary relator F we have:

$$\frac{F : \mathcal{R}(A_1, B_1) \times \cdots \times \mathcal{R}(A_n, B_n) \to \mathcal{R}(A, B)}{\lceil F \rceil : \mathcal{R}_W(A_1, B_1) \times \cdots \times \mathcal{R}_W(A_n, B_n) \to \mathcal{R}_W(A, B)}$$

where for the Kripke relations $R_i \in \mathcal{R}_W(A_i, B_i)$,

$$[w \Vdash \lceil F \rceil(R_1, \ldots, R_n)] \coloneqq F(w \Vdash R_1, \ldots, w \Vdash R_n)\,.$$

We use $\lceil - \rceil$ implicitly when a relator appears in a context where a Kripke logical relation is expected. Since reasoning with logical relations often involves self-relatedness, we use the notation $x :: R$ to denote $x\ R\ x$. For legibility, we will also write $w \Vdash x\ R\ y$ for $x\ [w \Vdash R]\ y$ and $\Vdash x\ R\ y$ for $x\ [\Vdash R]\ y$.

2.4 Simulation Conventions

Kripke relations are used to define simulation conventions. The worlds ensure that corresponding pairs of questions and answers are related consistently.

Definition 2.6. A *simulation convention* between the language interfaces $A_1 = \langle A_1^\circ, A_1^\bullet \rangle$ and $A_2 = \langle A_2^\circ, A_2^\bullet \rangle$ is a tuple $\mathbb{R} = \langle W, \mathbb{R}^\circ, \mathbb{R}^\bullet \rangle$, where W is a set, $\mathbb{R}^\circ \in \mathcal{R}_W(A_1^\circ, A_2^\circ)$ and $\mathbb{R}^\bullet \in \mathcal{R}_W(A_1^\bullet, A_2^\bullet)$. We will write $\mathbb{R} : A_1 \Leftrightarrow A_2$. The *identity* simulation convention for a language interface A is defined as $\mathrm{id}_A \coloneqq \langle \mathbb{1}, =, = \rangle : A \Leftrightarrow A$. We usually omit the subscript A.

A simulation between the transition systems $L_1 : A_1 \twoheadrightarrow B_1$ and $L_2 : A_2 \twoheadrightarrow B_2$ is then assigned a type $\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B$, where $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$ relate the corresponding language interfaces; we write $L_1 \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$. These notations are summarized in Table 1 and will be used extensively throughout the paper.

Table 1. Summary of notations

Notation	Examples	Description
$R \in \mathcal{R}(S_1, S_2)$	$\leq_{\mathsf{v}}$	Simple relation
$R \in \mathcal{R}_W(S_1, S_2)$	$\hookrightarrow_{\mathsf{m}}$	Kripke relation (Def. 2.5)
$w \Vdash R$		Kripke relation at world w
$w \Vdash x\ R\ y$		x and y related at world w
$\mathbf{R} \in \mathsf{CKLR}$	injp	CompCert KLR (§4.4)
A, B, C	$C, \mathcal{A}, \mathbf{1}$	Language interface (Def. 2.1)
$\mathbb{R} : A_1 \Leftrightarrow A_2$	CL	Simulation convention (Def. 2.6)
$L : A \twoheadrightarrow B$	$\mathsf{Clight}(p)$	LTS for $A \twoheadrightarrow B$ (Def. 3.1)
$L_1 \oplus L_2$		Horizontal composition (Def. 3.2)
$L_1 \leq_{\mathbb{R} \twoheadrightarrow \mathbb{S}} L_2$	Thm. 3.8	Simulation property (Def. 3.3)

Example 2.7. The calling convention used in Example 2.2 can be formalized as $\tilde{\mathbb{C}} := \langle \mathsf{val}^*, \tilde{\mathbb{C}}^\circ, \tilde{\mathbb{C}}^\bullet \rangle : \tilde{C} \Leftrightarrow \tilde{\mathcal{A}}$. We use the set of worlds val^* to relate the stack component of assembly questions to that of the corresponding answers. The relations $\tilde{\mathbb{C}}^\circ, \tilde{\mathbb{C}}^\bullet$ are defined by:

$$\frac{rs[\mathsf{pc}] = f \qquad \vec{v} \sqsubseteq rs[\mathsf{ebx}, \mathsf{ecx}]}{x\vec{k} \Vdash f(\vec{v})\ \tilde{\mathbb{C}}^\circ\ rs@x\vec{k}} \qquad \frac{rs[\mathsf{eax}] = v' \qquad rs[\mathsf{pc}] = x}{x\vec{k} \Vdash v'\ \tilde{\mathbb{C}}^\bullet\ rs@\vec{k}}$$

For a C-level function invocation $f(\vec{v})$, we expect the register pc to point to the beginning of the function f, and the registers ebx and ecx to contain the first and second arguments (if applicable). The register eax can contain an arbitrary value. The stack $x\vec{k}$ has no relationship to the C question, however the assembly answer is expected to pop the return address x and branch to it, setting the program counter pc accordingly. In addition, the register eax must store the return value v'.

2.5 Simulation Convention Algebra

Simulation conventions simplify the adaptation of the pass correctness proofs of CompCert. Instead of forcing all passes into the same mold, we can choose conventions matching the simulation relation and invariants used in each pass. The proofs can then be composed as shown in Fig. 3.

Unfortunately, the simulation convention obtained when we vertically compose the updated simulation properties suffers two serious problems. First, it is overly specific to the construction of CompCert and the exact sequence of passes included in the compiler. Second, because the correctness proofs in CompCert sometimes assume more guarantees on outgoing calls than they provide for incoming calls, outgoing and incoming calls use different simulation conventions. This asymmetry breaks horizontal compositionality (Thm. 3.4).

In CompCertO, we rectify this imbalance *outside* of the simulation proof itself. The requirements of most passes on their outgoing calls are met using the properties of the source language Clight, encoded as self-simulations and inserted as a pseudo-pass. We can then perform algebraic manipulations on simulation statements to rewrite the overall simulation convention used by the compiler.

$$\mathsf{id} : A \Leftrightarrow A \qquad L \leq_{\mathsf{id} \twoheadrightarrow \mathsf{id}} L$$

$$\frac{\mathbb{R} : A_1 \Leftrightarrow A_2 \qquad \mathbb{S} : A_2 \Leftrightarrow A_3}{\mathbb{R} \cdot \mathbb{S} : A_1 \Leftrightarrow A_3}$$

$$\frac{L_1 \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2 \qquad L_2 \leq_{\mathbb{S}_A \twoheadrightarrow \mathbb{S}_B} L_3}{L_1 \leq_{\mathbb{R}_A \cdot \mathbb{S}_A \twoheadrightarrow \mathbb{R}_B \cdot \mathbb{S}_B} L_3}$$

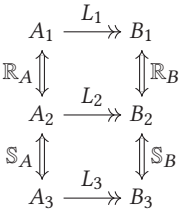


Figure 3. Simulation identity and vertical composition

This is achieved using a notion of simulation convention refinement ($\sqsubseteq$) allowing a simulation convention to replace another in all simulation statements. We construct a typed Kleene algebra [14] based on this ordering, and use it to ensure that our compiler correctness statement uses a simple, compositional simulation convention (§5).

3 Operational Semantics

This section describes CompCertO's semantic infrastructure. We start by reviewing the techniques used in CompCert.

3.1 Whole-Program Semantics in CompCert

The semantics of CompCert languages are given in terms of a simple notion of process behavior. By *process*, we mean a self-contained computation which can be characterized by the sequence of system calls it performs. For a C program to be executed as a process, its translation units must be compiled to object files, then linked together into an executable binary loaded by the system.

The model used for verifying CompCert accounts for this in the following way. Linking is approximated by merging programs, seen as sets of global definitions. The execution of a program composed of the translation units `M1.c` ... `Mn.c` which compile to `M1.s` ... `Mn.s` is modeled as:

$$L_{\mathsf{tgt}} := \mathsf{Asm}(\mathtt{M1.s} + \cdots + \mathtt{Mn.s})\,.$$

Here, + denotes CompCert's linking operator and Asm maps an assembly program to its semantics. Note that the loading process and the conventional invocation of `main` are encoded as part of the definition of Asm.

To formulate compiler correctness, we must also specify the behavior of the source program. To this end, CompCert defines a linking operator and semantics for the language Clight,[2] allowing the desired behavior to be specified as:

$$L_{\mathsf{src}} := \mathsf{Clight}(\mathtt{M1.c} + \cdots + \mathtt{Mn.c})\,.$$

Compiler correctness can then be stated as a refinement property of some kind between L_{src} and L_{tgt}.

Transition Systems. In the original CompCert, language semantics are given as labeled transition systems (LTS),

[2] CompCert features a richer version of the C language, but the dialect Clight is usually used as the source language when building certified artifacts.

$$v \in \mathsf{val} ::= \mathsf{undef} \mid \mathsf{int}(n) \mid \mathsf{long}(n) \mid \mathsf{float}(x) \mid \mathsf{single}(x) \mid \mathsf{vptr}(b, o)$$
$$(b, o) \in \mathsf{ptr} = \mathsf{block} \times \mathbb{Z} \qquad (b, l, h) \in \mathsf{ptrrange} = \mathsf{block} \times \mathbb{Z} \times \mathbb{Z}$$
$$\mathsf{alloc} : \mathsf{mem} \to \mathbb{Z} \to \mathbb{Z} \to \mathsf{mem} \times \mathsf{block}$$
$$\mathsf{free} : \mathsf{mem} \to \mathsf{ptrrange} \to \mathsf{option}(\mathsf{mem})$$
$$\mathsf{load} : \mathsf{mem} \to \mathsf{ptr} \to \mathsf{option}(\mathsf{val})$$
$$\mathsf{store} : \mathsf{mem} \to \mathsf{ptr} \to \mathsf{val} \to \mathsf{option}(\mathsf{mem})$$

Figure 4. Outline of the CompCert memory model

which characterize a program's behavior in terms of sequences of observable events. Schematically, a CompCert LTS is a tuple $L = \langle S, \to, I, F \rangle$ consisting of a set of states S, a labeled transition relation $\to \subseteq S \times \mathbb{E}^* \times S$, a subset $I \subseteq S$ of initial states, and a set $F \subseteq S \times \mathsf{int}$ of final states with exit statuses. The relation $s \xrightarrow{t} s'$ indicates that the state s may transition to the state s' through an interaction $t \in \mathbb{E}^*$.

The construction of states in CompCert language semantics follows common patterns. In particular, all languages start with the same notion of *memory state*.

Memory Model. The CompCert memory model [17, 18] is the core algebraic structure underlying the semantics of CompCert languages. Some of its operations are shown in Fig. 4. The idealized version presented here involves the type of memory states mem, the type of runtime values val, and the types of pointers ptr and address ranges ptrrange.

The memory is organized into a finite number of *blocks*. Each memory block has a unique identifier $b \in \mathsf{block}$ and is equipped with its own linear address space. Block identifiers and offsets are often manipulated together as pointers $(b, o) \in \mathsf{ptr}$. New blocks are created with prescribed boundaries using the primitive alloc. A runtime value $v \in \mathsf{val}$ can be stored at a given address using the primitive store, and retrieved using the primitive load. Values can be integers (int, long) and floating point numbers (float, single) of different sizes, as well as pointers (vptr). The special value undef represents an undefined value. Simulation relations often allow undef to be refined into a more concrete value; we write value refinement as $\leq_{\mathsf{v}} := \{(\mathsf{undef}, v), (v, v) \mid v \in \mathsf{val}\}$.

The memory model is shared by all of the languages in CompCert. States always consist of a memory component $m \in \mathsf{mem}$, alongside language-specific components which may contain additional values (val).

3.2 Open Semantics in CompCertO

The memory model also plays a central role when describing interactions between program components. In our approach, the memory state is passed alongside all control transfers.

Language Interfaces. Our models of cross-component interactions in CompCert languages are shown in Table 2. At the source level (C), questions consist of the address of the function to invoke ($vf \in \mathsf{val}$), its signature ($sg \in \mathsf{signature}$), the values of its arguments ($\vec{v} \in \mathsf{val}^*$), and the state of the memory at the point of entry ($m \in \mathsf{mem}$); answers consist of the function's return value and the state of the memory at the point of exit. This language interface is used for Clight and most of CompCert's intermediate languages.

Table 2. Language interfaces used in CompCertO

Name	Question	Answer	Description
C	$vf[sg](\vec{v})@m$	$v'@m'$	C calls
$\mathcal{L}$	$vf[sg](ls)@m$	$ls'@m'$	Abstract locations
$\mathcal{M}$	$vf(sp, ra, rs)@m$	$rs'@m'$	Machine registers
$\mathcal{A}$	$rs@m$	$rs'@m'$	Arch-specific
$\mathbf{1}$	n/a	n/a	Empty interface
$\mathcal{W}$	$*$	r	Whole-program

As we move towards lower-level languages, this is reflected in the language interfaces we use: function arguments are mapped into abstract locations alongside local temporary variables ($\mathcal{L}$, used by LTL and Linear). These locations are eventually concretized into stack slots and machine registers ($\mathcal{M}$, used by Mach). Finally, the target assembly language stores the program counter, stack pointer, and return address into their own registers ($\mathcal{A}$, used by Asm). Note that the actual stack contents are part of the memory state m.

The interface of whole-program execution can also be described in this setting: the language interface $\mathbf{1}$ contains no move; per §2.2, the interface $\mathcal{W}$ has a single trivial question $*$, and the answers $r \in \mathsf{int}$ give the exit status of a process. Hence the original CompCert semantics described in §3.1 can be seen to define strategies for $\mathbf{1} \twoheadrightarrow \mathcal{W}$: the process can only be started in a single way, cannot perform any external calls, and indicates an exit status upon termination.

Transition Systems. To account for the cross-component interactions described by language interfaces, CompCertO extends the transition systems described in §3.1 as follows.

Definition 3.1. Given an *incoming* language interface B and an *outgoing* language interface A, a *labeled transition system for the game* $A \twoheadrightarrow B$ is a tuple $L = \langle S, \to, D, I, X, Y, F \rangle$. The relation $\to \subseteq S \times \mathbb{E}^* \times S$ is a *transition relation* on the set of states S. The set $D \subseteq B^\circ$ specifies which questions the component accepts; $I \subseteq D \times S$ then assigns to each one a set of *initial states*. $F \subseteq S \times B^\bullet$ designates *final states* together with corresponding answers. External calls are specified by $X \subseteq S \times A^\circ$, which designates *external states* together with a question of A, and $Y \subseteq S \times A^\bullet \times S$, which is used to select a *resumption state* to follow an external state based on the answer provided by the environment. We write $L : A \twoheadrightarrow B$ when L is a labeled transition system for $A \twoheadrightarrow B$.

We use infix notation for the various transition relations I, X, Y, F. In particular, $n\ Y^s\ s'$ denotes that $n \in A^\bullet$ resumes the suspended external state s to continue with state s'.

$$\frac{q \in D_i \quad q\ I_i\ s}{q\ I\ (i,s)}\ \mathsf{i}^\circ \qquad \frac{s \xrightarrow{t}_i s'}{(i,s)\,\vec{k} \xrightarrow{t} (i,s')\,\vec{k}}\ \mathsf{run} \qquad \frac{s\ F_i\ r}{(i,s)\ F\ r}\ \mathsf{i}^\bullet$$

$$\frac{s\ X_i\ q \quad q \in D_j \quad q\ I_j\ s'}{(i,s)\,\vec{k} \xrightarrow{\epsilon} (j,s')\,(i,s)\,\vec{k}}\ \mathsf{push} \qquad \frac{s'\ F_j\ r \qquad r\ Y_i^s\ s''}{(j,s')\,(i,s)\,\vec{k} \xrightarrow{\epsilon} (i,s'')\,\vec{k}}\ \mathsf{pop}$$

$$\frac{s\ X_i\ q \qquad \forall j\,.\,q \notin D_j}{(i,s)\,\vec{k}\ X\ q}\ \mathsf{x}^\circ \qquad \frac{r\ Y_i^s\ s'}{r\ Y^{(i,s)\,\vec{k}}\ (i,s')\,\vec{k}}\ \mathsf{x}^\bullet$$

Figure 5. Horizontal composition of open semantics.

Horizontal Composition. To model linking, the following definition expresses the behavior of a collection of components in terms of the behavior of each one.

Definition 3.2 (Horizontal composition)**.** For two transition systems $L_1, L_2 : A \twoheadrightarrow A$ with $L_i = \langle S_i, \rightarrow_i, D_i, I_i, X_i, Y_i, F_i \rangle$, the *horizontal composition* of L_1 and L_2 is

$$L_1 \oplus L_2 := \left\langle (S_1 + S_2)^*,\ \rightarrow,\ D_1 \cup D_2,\ I,\ X,\ Y,\ F \right\rangle,$$

where $\rightarrow, I, X, Y, F$ are defined by the rules shown in Fig. 5.

When the composite transition system receives an incoming question, an appropriate component is chosen based on the domains D_1 and D_2 (i°). This component becomes active and the composite transition system proceeds accordingly (run, x°, $\mathsf{x}^\bullet$, $\mathsf{i}^\bullet$). To enable mutual recursion between L_1 and L_2, the composite system maintains an alternating stack of suspended states of L_1 and L_2. When L_1 is active and performs an external call to L_2, the current state is suspended and the question of L_1 is used to initialize a new instance of L_2 (push). When that instance terminates, the suspended state of L_1 is resumed by the corresponding answer (pop). In between, L_2 may itself perform cross-component calls, creating *new* instances of L_1, and so on to an arbitrary depth.

3.3 Open Simulations

CompCert is proved correct using a simulation between the transition semantics of the source and target programs. This *forward* simulation is used to establish a *backward* simulation. Backward simulations are in turn proved to be sound with respect to trace containment. We have adapted forward and backward simulations to the semantic model of CompCertO. In this section we present forward simulations, which are used as our primary notion of refinement.

Forward Simulations. A forward simulation asserts that any transition in the source program has a corresponding transition sequence in the target. The sequence may be empty, but to ensure the preservation of silent divergence this can only happen for finitely many consecutive source transitions. This is enforced by indexing simulation relations over well-founded orders, and requiring the index to decrease whenever an empty transition sequence is used.

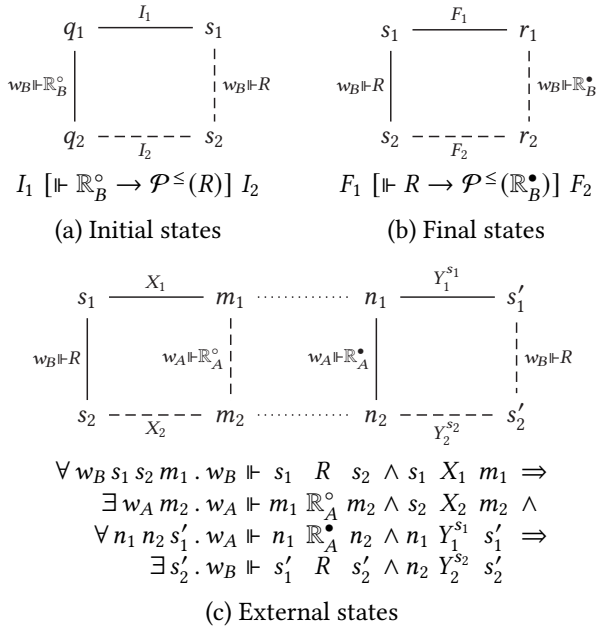


Figure 6. Selected forward simulation properties

This mechanism is unchanged in CompCertO and is largely orthogonal to the techniques we introduce, so we omit this aspect of forward simulations in our exposition.

This omission aside, our forward simulations are outlined in Fig. 6. To take external interactions into account, a simulation of $L_1 : A_1 \twoheadrightarrow B_1$ by $L_2 : A_2 \twoheadrightarrow B_2$ operates in the context of the simulation conventions $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$. If incoming questions are related by $\mathbb{R}_B^\circ$ at a world w_B, these properties guarantee that the corresponding answers will be related by $\mathbb{R}_B^\bullet$ at the same world.

Conversely, the simulation convention $\mathbb{R}_A$ determines the correspondence between outgoing questions triggered by the transition systems' external states. Compared with the treatment of incoming questions, the roles of the system and environment are reversed: the simulation proof can choose w_A to relate the outgoing questions, and can assume that any corresponding answers will be related at that world.

Definition 3.3 (Forward simulation)**.** Given the simulation conventions $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$, and given the transition systems $L_1 : A_1 \twoheadrightarrow B_1 = \langle S_1, \rightarrow_1, D_1, I_1, X_1, Y_1, F_1 \rangle$ and $L_2 : A_2 \twoheadrightarrow B_2 = \langle S_2, \rightarrow_2, D_2, I_2, X_2, Y_2, F_2 \rangle$, a *forward simulation* between L_1 and L_2 is a relation $R \in \mathcal{R}_{W_B}(S_1, S_2)$ satisfying the properties shown in Fig. 6 as well as:

$$(\lambda q_1\,.\,(q_1 \in D_1))\ [\Vdash \mathbb{R}_B^\circ \to \Leftrightarrow]\ (\lambda q_2\,.\,(q_2 \in D_2))$$
$$\to_1\ [\Vdash R \to = \to \mathcal{P}^\leq(R)]\ \to_2^*$$

We will write $L_1 \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ when such a relation exists.

Horizontal Composition. The horizontal composition operator described by Def. 3.2 preserves simulations. Roughly

speaking, whenever a new component instance is created by a cross-component call, the simulation property for the new instance can be stitched in-between the two halves of the callers' simulation property as described in Fig. 6(c).

Theorem 3.4 (Horizontal composition of simulations)**.** *For a simulation convention* $\mathbb{R} : A_1 \Leftrightarrow A_2$ *and transition systems* $L_1, L_1' : A_1 \twoheadrightarrow A_1$ *and* $L_2, L_2' : A_2 \twoheadrightarrow A_2$*, the following holds:*

$$\frac{L_1 \leq_{\mathbb{R}\to\mathbb{R}} L_2 \qquad L_1' \leq_{\mathbb{R}\to\mathbb{R}} L_2'}{L_1 \oplus L_1' \leq_{\mathbb{R}\to\mathbb{R}} L_2 \oplus L_2'}$$

One interesting and novel aspect of the proof is the way worlds are managed. Externally, only the worlds corresponding to incoming and outgoing questions and answers are observed. Internally, the proof of Thm. 3.4 maintains a *stack* of worlds to relate the corresponding stack of activations in the source and target composite semantics. See also §4.6.

Theorem 3.4 allows us to decompose the verification of a complex program into the verification of its parts. To establish the correctness of the linked assembly program, we can use the following result.

Theorem 3.5. *Linking two* Asm *programs* p_1 *and* p_2 *yields a correct implementation of horizontal composition:*

$$\mathsf{Asm}(p_1) \oplus \mathsf{Asm}(p_2) \leq_{\mathsf{id}\to\mathsf{id}} \mathsf{Asm}(p_1 + p_2)$$

Vertical Composition. Simulations also compose *vertically*, combining the simulation properties for successive compilation passes into a single one. The convention used by the resulting simulation can be described as follows.

Definition 3.6 (Composition of simulation conventions)**.** The composition of two Kripke relations $R \in \mathcal{R}_{W_R}(X, Y)$ and $S \in \mathcal{R}_{W_S}(Y, Z)$ is $R \cdot S \in \mathcal{R}_{W_R \times W_S}(X, Z)$, defined by:

$$(w_R, w_S) \Vdash x\ [R \cdot S]\ z \Leftrightarrow \exists y \in Y\,.\, w_R \Vdash x\ R\ y \wedge w_S \Vdash y\ S\ z\,.$$

For the simulation conventions $\mathbb{R} : A \Leftrightarrow B$ and $\mathbb{S} : B \Leftrightarrow C$, we define $\mathbb{R} \cdot \mathbb{S} : A \Leftrightarrow C$ as:

$$\mathbb{R} \cdot \mathbb{S} := \langle W_{\mathbb{R}} \times W_{\mathbb{S}},\ \mathbb{R}^\circ \cdot \mathbb{S}^\circ,\ \mathbb{R}^\bullet \cdot \mathbb{S}^\bullet \rangle$$

Theorem 3.7. *Open forward simulations compose vertically as depicted in Fig. 3.*

3.4 Compiler Correctness

The passes of CompCertO are shown in Table 3. They can be composed using the mechanisms we have just described. Through techniques developed in §4 and §5, we can then formulate a uniform simulation convention $\mathbb{C} : C \Leftrightarrow \mathcal{A}$ for the whole compiler and establish our main results.

Theorem 3.8 (Compositional Correctness of CompCertO)**.** *For a* Clight *program* p *and an* Asm *program* p' *such that* $\mathsf{CompCert}(p) = p'$*, the following simulation holds:*

$$\mathsf{Clight}(p) \leq_{\mathbb{C}\to\mathbb{C}} \mathsf{Asm}(p')\,.$$

Table 3. Passes of CompCertO grouped by source language. † indicates optional optimizations. Significant lines of code (SLOC) measured by `coqwc`, compared to CompCert v3.6.

Language/Pass	Outgoing ↠ Incoming	SLOC	
Clight	$C \twoheadrightarrow C$	+17	(+3%)
SimplLocals	inj ↠ inj	-3	(-0%)
Cshmgen	id ↠ id	+0	(+0%)
Csharpminor	$C \twoheadrightarrow C$	+15	(+4%)
Cminorgen	inj ↠ inj	-15	(-1%)
Cminor	$C \twoheadrightarrow C$	+15	(+3%)
Selection	wt · ext ↠ wt · ext	+46	(+1%)
CminorSel	$C \twoheadrightarrow C$	+15	(+3%)
RTLgen	ext ↠ ext	+12	(+1%)
RTL	$C \twoheadrightarrow C$	+11	(+3%)
Tailcall†	ext ↠ ext	+4	(+1%)
Inlining	inj ↠ inj	+62	(+3%)
Renumber	id ↠ id	-14	(-7%)
Constprop†	va · ext ↠ va · ext	-15	(-1%)
CSE†	va · ext ↠ va · ext	+6	(+0%)
Deadcode†	va · ext ↠ va · ext	-5	(-0%)
Allocation	wt · ext · CL ↠ wt · ext · CL	+46	(+2%)
LTL	$\mathcal{L} \twoheadrightarrow \mathcal{L}$	+18	(+8%)
Tunneling	ext ↠ ext	+15	(+3%)
Linearize	id ↠ id	-15	(-3%)
Linear	$\mathcal{L} \twoheadrightarrow \mathcal{L}$	+18	(+8%)
CleanupLabels	id ↠ id	-10	(-3%)
Debugvar	id ↠ id	-12	(-2%)
Stacking	inj · LM ↠ LM · inj	+268	(+10%)
Mach	$\mathcal{M} \twoheadrightarrow \mathcal{M}$	+184	(+49%)
Asmgen	ext · MA ↠ ext · MA	+277	(+9%)
Asm	$\mathcal{A} \twoheadrightarrow \mathcal{A}$	+566	(+10%)
	Total	**+1,136**	**(+3%)**

The simulation convention $\mathbb{C}$ formalizes the correspondence between C and assembly interactions. By necessity, it follows Clight and Asm in their use of the CompCert memory model, but is otherwise independent of the compiler. In particular, $\mathbb{C}$ is not sensitive to the inclusion of optional optimization passes. The details are discussed in §5.

Corollary 3.9. *If* `M1.c`, ..., `Mn.c` *are compiled and linked to* `M1.s` + ... + `Mn.s` = `M.s`*, we can use Thms. 3.4, 3.5 and 3.8 to establish the following* separate compilation *property:*

$$\mathsf{Clight}(\texttt{M1.c}) \oplus \cdots \oplus \mathsf{Clight}(\texttt{Mn.c}) \leq_{\mathbb{C}\to\mathbb{C}} \mathsf{Asm}(\texttt{M.s})$$

In other words, the horizontal composition of the source modules' behaviors is faithfully implemented by the compiled and linked Asm program `M.s`.

3.5 Towards Heterogeneous Verification

Although we focus in this paper on a symmetric form of horizontal composition which models linking and mutual

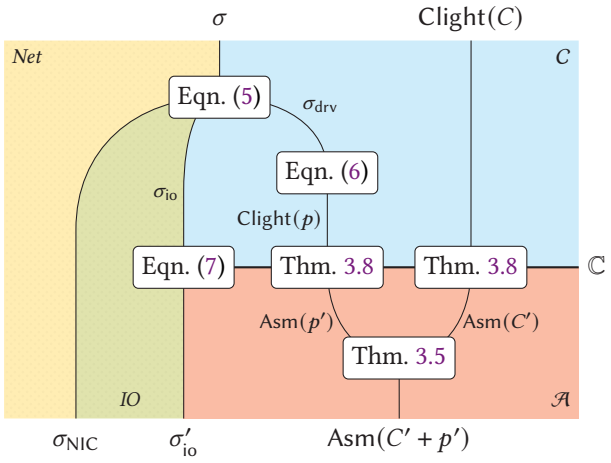


Figure 7. Heterogeneous scenario outlined in Example 3.10, depicted as a string diagram. Thin vertical lines represent transition systems; their horizontal juxtaposition denotes composition. Colored regions correspond to language interfaces and rectangles to simulation properties.

recursion, it is possible to define a more traditional operator

$$\circ_{A,B,C} : (B \twoheadrightarrow C) \times (A \twoheadrightarrow B) \longrightarrow (A \twoheadrightarrow C)\,.$$

In $L_1 \circ L_2$, calls propagate from the environment to L_1, then to L_2, then back to the environment, but L_2 cannot call directly into L_1. The homogeneous case $\circ_{A,A,A}$ is an underapproximation of $\oplus$ and therefore Thm. 3.5 applies:

$$\mathsf{Asm}(p_1) \circ \mathsf{Asm}(p_2) \leq_{\mathsf{id}\twoheadrightarrow\mathsf{id}} \mathsf{Asm}(p_1 + p_2)$$

These constructions would enrich our framework with the structure of a double category, and make it possible to use CompCertO in a heterogeneous context.

Example 3.10. Revisiting Example 1.1 we consider:

- a network interface card model $\sigma_{\mathsf{NIC}} : \mathit{Net} \twoheadrightarrow \mathit{IO}$,
- device I/O primitives modeled as $\sigma_{\mathsf{io}} : \mathit{IO} \twoheadrightarrow \mathcal{C}$,
- a network card driver specified by $\sigma_{\mathsf{drv}} : \mathcal{C} \twoheadrightarrow \mathcal{C}$,

Together, they implement a specification $\sigma : \mathit{Net} \twoheadrightarrow \mathcal{C}$ as:

$$\sigma \leq_{\mathsf{id}\twoheadrightarrow\mathsf{id}} \sigma_{\mathsf{drv}} \circ \sigma_{\mathsf{io}} \circ \sigma_{\mathsf{NIC}} \tag{5}$$

The interface *Net* models the flow of ethernet packets sent and received by the network adapter, whereas *IO* models its interaction with the CPU. Device I/O primitives are implemented in unverified code and axiomatized in σ_{io}. The driver itself can be implemented as a Clight program p:

$$\sigma_{\mathsf{drv}} \leq_{\mathsf{id}\twoheadrightarrow\mathsf{id}} \mathsf{Clight}(p) \tag{6}$$

To characterize the compiled driver, we must use an assembly-level specification for I/O primitives $\sigma'_{\mathsf{io}} : \mathit{IO} \twoheadrightarrow \mathcal{A}$ such that:

$$\sigma_{\mathsf{io}} \leq_{\mathsf{id}\twoheadrightarrow\mathbb{C}} \sigma'_{\mathsf{io}} \tag{7}$$

Then per Fig. 7, the compiled driver p' satisfies

$$\sigma \leq_{\mathsf{id}\twoheadrightarrow\mathbb{C}} \mathsf{Asm}(p') \circ \sigma'_{\mathsf{io}} \circ \sigma_{\mathsf{NIC}}$$

and a client program C interacting with σ can be soundly compiled into C' and linked with the driver p':

$$\mathsf{Clight}(C) \circ \sigma \leq_{\mathsf{id}\twoheadrightarrow\mathbb{C}} \mathsf{Asm}(C' + p') \circ \sigma'_{\mathsf{io}} \circ \sigma_{\mathsf{NIC}}\,.$$

It is admittedly unclear whether the model presented in this section, designed specifically for the verification of CompCertO, could adequately capture the complexity of hardware interfaces or the concurrency inherent in this kind of composite system. However, our semantics and proofs can be embedded into richer game models, where the approach outlined in Example 3.10 could be realistically carried out.

4 Logical Relations for CompCert

The questions, answers and states used in the semantics of CompCert languages are each composed of a memory state surrounded by runtime values. Likewise, simulation conventions and relations are constructed around *memory transformations* and relate the surrounding values in ways that are compatible with the chosen memory transformation.

4.1 Memory Extensions

For passes where strict equality is too restrictive, but the source and target programs use similar memory layouts, CompCert uses the *memory extension* relation, which allows the values stored in the target memory state to refine the values stored in the source memory at the same location.

By analogy with the relation $v_1 \leq_{\mathsf{v}} v_2$ introduced in §3.1, we write $m_1 \leq_{\mathsf{m}} m_2$ to signify that the source memory m_1 is extended by the target memory m_2. Together, the relations $\leq_{\mathsf{v}}$ and $\leq_{\mathsf{m}}$ constitute a logical relation for the memory model: loads from memory states related by extension yield values related by refinement, storing values related by refinement preserves memory extensions, and similarly for the remaining memory operations.

4.2 Memory Injections

The most complex simulation relations of CompCert allow memory blocks to be dropped, added, or mapped at a given offset within a larger block. These transformations of the memory structure are specified by partial functions of type:

$$\mathsf{meminj} := \mathsf{block} \rightharpoonup \mathsf{block} \times \mathbb{Z}$$

We will call $f \in \mathsf{meminj}$ an *injection mapping*. An entry $f(b) = (b', o)$ means that the source memory block with identifier b is mapped into the target block b' at offset o.

As with refinement and extension, an injection mapping determines both a relation on values and a relation on memory states, which work together as a logical relation for the CompCert memory model. The value relation $f \Vdash v_1 \hookrightarrow_{\mathsf{v}} v_2$ allows v_2 to refine v_1, but also requires any pointer present in v_1 to be transformed according to f. The memory relation $f \Vdash m_1 \hookrightarrow_{\mathsf{m}} m_2$ requires that any addresses of m_1 and m_2 related by f hold values related by $f \Vdash \hookrightarrow_{\mathsf{v}}$.

$$\rightsquigarrow \text{ is reflexive and transitive}$$

$$\begin{aligned}
\text{alloc} &:: \Vdash \mathbf{R}^{\text{mem}} \to = \to = \to \Diamond(\mathbf{R}^{\text{mem}} \times \mathbf{R}^{\text{block}}) \\
\text{free} &:: \Vdash \mathbf{R}^{\text{mem}} \to \mathbf{R}^{\text{ptrrange}} \to \text{option}^{\leq}(\Diamond\mathbf{R}^{\text{mem}}) \\
\text{load} &:: \Vdash \mathbf{R}^{\text{mem}} \to \mathbf{R}^{\text{ptr}} \to \text{option}^{\leq}(\mathbf{R}^{\text{val}}) \\
\text{store} &:: \Vdash \mathbf{R}^{\text{mem}} \to \mathbf{R}^{\text{ptr}} \to \mathbf{R}^{\text{val}} \to \text{option}^{\leq}(\Diamond\mathbf{R}^{\text{mem}})
\end{aligned}$$

Figure 8. Defining properties of CKLRs. Note the correspondence with the types of operations in Fig. 4.

Since memory allocations create new block identifiers, corresponding allocations in the source and target memory states cause f to evolve into a more defined mapping $f \subseteq f'$. This is handled by the following constructions.

4.3 Modal Kripke Relators

We defined general Kripke relations in §2.3. We add structure to sets of Kripke worlds, specifying how they can evolve.

Definition 4.1. A *Kripke frame* is a tuple $\langle W, \rightsquigarrow \rangle$, where W is a set of *possible worlds* and $\rightsquigarrow$ is a binary *accessibility relation* over W. Then the Kripke relator $\Diamond$ is defined by:

$$w \Vdash x \ [\Diamond R] \ y \iff \exists w' . w \rightsquigarrow w' \wedge w' \Vdash x \ R \ y$$

Example 4.2 (Injection simulation diagrams)**.** Following up on Example 2.4, Consider once again the simple transition systems $\alpha : A \to \mathcal{P}(A)$ and $\beta : B \to \mathcal{P}(B)$. An injection-based simulation relation between them will be a Kripke relation $R \in \mathcal{R}_{\text{meminj}}(A, B)$ satisfying the property:

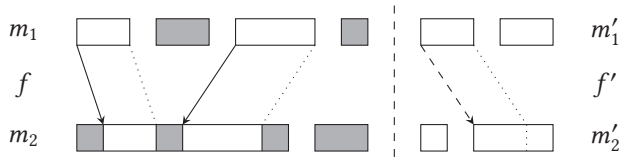
$$\begin{array}{ccc} s_1 & \xrightarrow{\alpha} & s_1' \\ {\scriptstyle f}\,\vert & & \vert\,{\scriptstyle f'} \\ s_2 & \xrightarrow[\beta]{} & s_2' \end{array} \qquad \begin{array}{l} \forall f\, s_1\, s_2\, s_1' . f \Vdash s_1 \ R \ s_2 \wedge \alpha(s_1) \ni s_1' \Rightarrow \\ \exists f' s_2' . f \subseteq f' \wedge \beta(s_2) \ni s_2' \wedge f' \Vdash s_1' \ R \ s_2' \end{array} \tag{8}$$

The new states may be related according to a new injection mapping f'. To preserve existing relationships between any surrounding source and target pointers, the new mapping must include the original one ($f \subseteq f'$). This pattern is common in CompCert and appears in a variety of contexts. By using $\langle \text{meminj}, \subseteq \rangle$ as a Kripke frame, we can express (8) as:

$$\alpha \ [\Vdash R \to \mathcal{P}^{\leq}(\Diamond R)] \ \beta .$$

4.4 CompCert Kripke Logical Relations

A *CompCert Kripke logical relation* (CKLR) must provides a Kripke frame $\langle W, \rightsquigarrow \rangle$, as well as a component relation for every type involved in the CompCert memory model. These relations must satisfy the properties given in Fig. 8. Note that R^{mem} is the central component driving world transitions, as witnessed by the uses of $\Diamond$ in Fig. 8. The surrounding relations must be monotonic in w, so that any extra state constructed from pointers and runtime values will be able to "follow along" when world transitions occur.



$$(f, m_1, m_2) \rightsquigarrow_{\text{injp}} (f', m_1', m_2')$$

Figure 9. External calls and memory injections. The source and target memory states are depicted at the top and bottom of the figure. Arrows describe the injection mapping. The memory block on the left of the dashed line are present at the beginning of the call. Memory blocks on the right are allocated during the call, adding a new entry to the injection mapping. The shaded areas must not be modified by the call.

Among other constructions, we can define the CKLRs ext and inj, which correspond to CompCert's memory extensions and memory injections. Their component relations for values and memory states coincide with the usual ones:

$$\begin{array}{lll} W_{\text{ext}} := \mathbb{1} & \text{ext}^{\text{val}} := \leq_{\text{v}} & \text{ext}^{\text{mem}} := \leq_{\text{m}} \\ W_{\text{inj}} := \text{meminj} & \text{inj}^{\text{val}} := \hookrightarrow_{\text{v}} & \text{inj}^{\text{mem}} := \hookrightarrow_{\text{m}} \end{array}$$

Simulation Conventions. Given a language interface $\mathcal{X}$, we can use the components of $\mathbf{R} \in \text{CKLR}$ to build a simulation convention $\mathbf{R}_{\mathcal{X}} : \mathcal{X} \Leftrightarrow \mathcal{X}$. For instance:

$$\mathbf{R}_C := \left\langle W, \ (\mathbf{R}^{\text{val}} \times = \times \vec{\mathbf{R}}^{\text{val}} \times \mathbf{R}^{\text{mem}}), \ \Diamond(\mathbf{R}^{\text{val}} \times \mathbf{R}^{\text{mem}}) \right\rangle .$$

We will often implicitly promote $\mathbf{R}$ to $\mathbf{R}_{\mathcal{X}}$.

Note that in our model, accessibility relations are not directly involved in the interface of simulations. Instead, a single world is used to formulate the 4-way relationship between pairs of questions and answers. In the definition above, we allow world transitions in simulations by qualifying the relation $\mathbf{R}_C^{\bullet}$ with the modality $\Diamond$.

Parametricity. Since CompCert language semantics are built out of the operations of the memory model, they are well-behaved with respect to CKLRs.

Theorem 4.3. *For the languages* $L \in \{\text{Clight}, \text{RTL}, \text{Asm}\}$,

$$\forall \mathbf{R} \in \text{CKLR} . L(p) \leq_{\mathbf{R} \twoheadrightarrow \mathbf{R}} L(p)$$

4.5 External Calls in Injection Passes

Passes which alter the block structure of the memory use *memory injections* (§4.2). The CKLR inj can be used for incoming calls, but it is insufficient for outgoing calls.

Example 4.4. The SimplLocals pass removes some local variables from the memory. The corresponding values are instead stored as temporaries in the target function's local environment, and the correspondence between the two is enforced by the simulation relation. To maintain it, we need to know that external calls do not modify source memory blocks with no counterpart in the target memory.

More generally, as depicted in Fig. 9, injection passes expect external calls to leave regions outside of the injection's footprint untouched. This expectation is reasonable because external calls should behave uniformly between the source and target executions. These requirements can be formalized as a CKLR inj$p$ which includes the following components:

$$W_{\mathsf{injp}} := \mathsf{meminj} \times \mathsf{mem} \times \mathsf{mem}$$

$$\frac{f \Vdash v_1 \hookrightarrow_{\mathsf{v}} v_2}{(f, m_1, m_2) \Vdash v_1 \; R^{\mathsf{val}}_{\mathsf{injp}} \; v_2} \qquad \frac{f \Vdash m_1 \hookrightarrow_{\mathsf{m}} m_2}{(f, m_1, m_2) \Vdash m_1 \; R^{\mathsf{mem}}_{\mathsf{injp}} \; m_2}$$

Then, $(f, m_1, m_2) \rightsquigarrow_{\mathsf{injp}} (f', m_1', m_2')$ ensures that $f \subseteq f'$ and that the memory states satisfy the constraints in Fig. 9.

4.6 World Transitions and Compositionality

The CKLR inj$p$ illustrates a key novelty in the granularity at which we deploy Kripke world transitions. Consider two related executions with the shape:

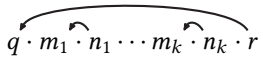
$$q \cdot s_1 \cdot s_2 \cdot m \cdot s_1' \cdot s_2' \cdot n \cdot s_3 \cdots s_k \cdot r$$

Each element of the sequence denotes a pair of related moves or states. Here q, m, n, r each denote a pair of related moves transferring control between components, whereas s_i and s_i' denote pairs of internal states of two different components. We draw an arrow from a pair n to an earlier pair m when the world used to relate the source and target constituents of n is accessible from the world used for m.

Traditionally, Kripke worlds evolve linearly with time:

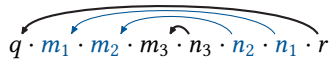
To enable horizontal compositionality, the challenge is then to construct worlds, accessibility relations, and simulation relations which are sophisticated enough to express ownership constraints like the ones discussed in §4.5, which shift as the execution switches between components.

In our open simulations, worlds can be deployed independently for incoming and outgoing calls, in a way which follows the structure of plays, as depicted here:

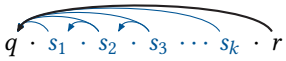


Internal states are not part of a component's observable behavior, and individual simulation proofs can relate them in arbitrary ways, as long as the simulation relation is compatible with the simulation convention at interaction sites.

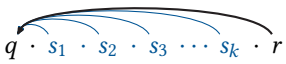
Two examples illustrate this flexibility. First, as explained in §3.3, to handle nested calls between its components, a composite simulation uses an internal stack of worlds. A situation where m_1 and m_2 are internal calls and m_3 is an external call can be described as:



Second, in simulations which use CKLRs, the simulation relation is qualified as $w_B \Vdash \Diamond R$ to allow the world to evolve as the execution progresses. Since $\Diamond\Diamond R = \Diamond R$, per-step world transitions ($\Vdash R \longrightarrow \Diamond R$) maintain the overall constraint:



Moreover, this allows steps which *do not* individually conform to world transitions but *do* maintain $\Diamond R$ with respect to the initial world ($w_B \Vdash \Diamond R \longrightarrow \Diamond R$):



For instance, recall that in the SimplLocals pass, the source program may write to local variables which have been removed from the memory in the target program (Example 4.4). At a granular level, the corresponding steps break $\rightsquigarrow_{\mathsf{injp}}$ by writing to a memory block outside of the injection's footprint (see Fig. 9). But since the block in question is allocated after the function is invoked, $\rightsquigarrow_{\mathsf{injp}}$ is satisfied with respect to the initial world.

In combination, these two examples show how our framework handles ownership constraints using conditions like $\rightsquigarrow_{\mathsf{injp}}$ already present in CompCert. It does not require sophisticated permission maps as in other approaches [22, 23].

5 Calling Convention

The simulation convention $\mathbb{C} : C \Leftrightarrow \mathcal{A}$ used in the formulation of Thm. 3.8 is constructed as follows:

$$\mathbb{C} := \mathcal{R}^* \cdot \mathsf{wt} \cdot \mathsf{CA} \cdot \mathsf{vainj}_{\mathcal{A}}$$

The structure of $\mathbb{C}$ is explained below:

- The first component allows the caller to use CKLRs in the set $\mathcal{R} := \mathsf{injp} + \mathsf{inj} + \mathsf{ext} + \mathsf{vainj} + \mathsf{vaext}$. In particular, it is used to satisfy the requirements placed on external calls by the passes of CompCert.
- The component wt ensures that the arguments and return values of the C-level calls are well-typed.
- The component $\mathsf{CA} \equiv \mathsf{CL} \cdot \mathsf{LM} \cdot \mathsf{MA}$ formalizes the structural aspects of the C calling convention, in particular the marshaling of C function arguments into assembly registers and onto the stack. It guarantees the preservation of callee-save registers and ensures that the source-level execution does not have access to the arguments region of the stack.
- The component $\mathsf{vainj}_{\mathcal{A}}$ ensures that calls are compatible with memory injections and that global constants have their prescribed values in the source memory.

In the remainder of this section, we explain how the passes of CompCertO (Table 3) can be composed to build a compiler which obeys the simulation convention $\mathbb{C}$.

5.1 Refinement of Simulation Conventions

A refinement $\mathbb{R} \sqsubseteq \mathbb{S}$ captures the idea that the simulation convention $\mathbb{R}$ is more general than $\mathbb{S}$, so that any simulation accepting $\mathbb{R}$ as its incoming convention can accept $\mathbb{S}$ as well. The shape of the symbol illustrates its meaning: when $\mathbb{R} \sqsubseteq \mathbb{S}$,

related questions of $\mathbb{S}$ can be transported to related question of $\mathbb{R}$; when we get a response, the related answers of $\mathbb{R}$ can be transported back to related answers of $\mathbb{S}$.

Definition 5.1 (Simulation convention refinement)**.** Given two simulation conventions $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$, we say that $\mathbb{R}$ *is refined by* $\mathbb{S}$ and write $\mathbb{R} \sqsubseteq \mathbb{S}$ when the following holds:

$$\forall w\, m_1\, m_2\, .\, w \Vdash m_1\ \mathbb{S}^\circ\ m_2 \Rightarrow \exists v\, .\, (v \Vdash m_1\ \mathbb{R}^\circ\ m_2 \wedge$$
$$\forall n_1\, n_2\, .\, v \Vdash n_1\ \mathbb{R}^\bullet\ n_2 \Rightarrow w \Vdash n_1\ \mathbb{S}^\bullet\ n_2)\, .$$

We write $\mathbb{R} \equiv \mathbb{S}$ when both $\mathbb{R} \sqsubseteq \mathbb{S}$ and $\mathbb{S} \sqsubseteq \mathbb{R}$.

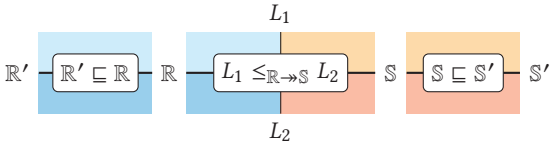
Theorem 5.2. *For* $\mathbb{R} : A_1 \Leftrightarrow A_2$, $\mathbb{S} : A_2 \Leftrightarrow A_3$, $\mathbb{T} : A_3 \Leftrightarrow A_4$, *the following properties hold:*

$$(\mathbb{R} \cdot \mathbb{S}) \cdot \mathbb{T} \equiv \mathbb{R} \cdot (\mathbb{S} \cdot \mathbb{T}) \qquad \mathbb{R} \cdot \mathsf{id} \equiv \mathsf{id} \cdot \mathbb{R} \equiv \mathbb{R}$$
$$(\cdot) :: \sqsubseteq \times \sqsubseteq \to \sqsubseteq$$

In addition, when $\mathbb{R}' \sqsubseteq \mathbb{R} : A_1 \Leftrightarrow A_2$ *and* $\mathbb{S} \sqsubseteq \mathbb{S}' : B_1 \Leftrightarrow B_2$, *for all* $L_1 : A_1 \twoheadrightarrow B_1$ *and* $L_2 : A_2 \twoheadrightarrow B_2$*:*

$$L_1 \le_{\mathbb{R} \to \mathbb{S}} L_2 \Rightarrow L_1 \le_{\mathbb{R}' \to \mathbb{S}'} L_2\, .$$

Building on the graphical language used in Fig. 7, we can represent simulation convention refinements as boxes on horizontal lines. For example, Thm. 5.2 enables the horizontal gluing of the following tiles:



The composition of simulation conventions can be depicted as the vertical juxtaposition of the corresponding lines. Refinement boxes can be replaced by distinctive shapes. This is illustrated in Fig. 10, which outlines a proof of Thm. 3.8.

5.2 Composing CKLRs

The simulation convention obtained by composing the passes of CompCertO (Table 3) involves a multitude of CKLRs. Thankfully, ext and inj compose nicely.

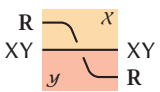
Lemma 5.3. *The CKLRs* ext *and* inj *compose as follows:*

$$\mathsf{ext} \cdot \mathsf{inj} \equiv \mathsf{inj} \cdot \mathsf{ext} \equiv \mathsf{inj} \cdot \mathsf{inj} \equiv \mathsf{inj} \qquad \mathsf{ext} \cdot \mathsf{ext} \equiv \mathsf{ext}$$

The corresponding refinements can be depicted as variations on forks of the following kind:

However, intervening calling conventions (XY) and invariants (wt, va) prevent us from using them to their full extent. This is addressed in part by the following properties.

Lemma 5.4. *For* $\mathsf{XY} \in \{\mathsf{CL}, \mathsf{LM}, \mathsf{MA}\}$ *and* $\mathbf{R} \in \mathsf{CKLR}$*:*



$$\mathbf{R}_x \cdot \mathsf{XY} \sqsubseteq \mathsf{XY} \cdot \mathbf{R}_y$$

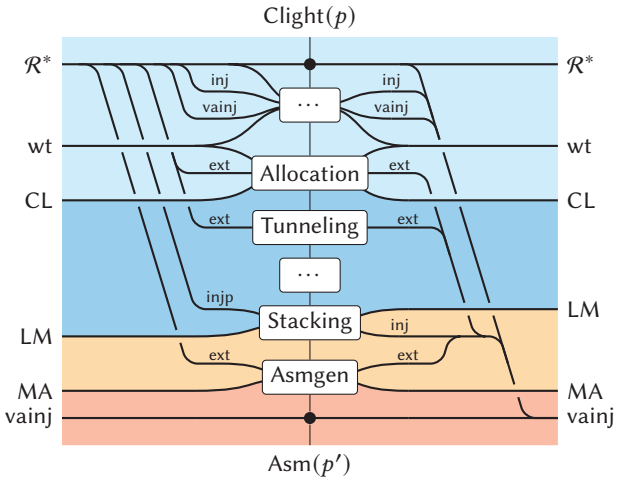


Figure 10. Overall structure of the proof of Thm. 3.8. The intersections marked with • denote parametricity properties of Clight and Asm (Thm. 4.3). The simulation proof for C passes, depicted as "…" in the top part of the figure, can be constructed incrementally (see Fig. 11). The other ellipsis denotes identity passes which can be composed transparently.

Using Lemmas 5.3 and 5.4 together, an incoming memory injection can be fanned out across the compiler to satisfy the requirements of all passes. Conversely, outgoing memory transformations can be regrouped at the source level, where we can require all components to be compatible with the CKLRs in $\mathcal{R}$. To express this requirement, we rely on the Kleene algebra described in the next section.

5.3 Kleene Algebra

The sum of a family of simulation conventions allows the caller to choose any one of them. The Kleene star allows a choice of finite iterations.

Definition 5.5. Consider $(\mathbb{R}_i)_{i \in I}$ a family of conventions with $\mathbb{R}_i = \langle W_i, \mathbb{R}_i^\circ, \mathbb{R}_i^\bullet \rangle : A_1 \Leftrightarrow A_2$ for all $i \in I$. We define the simulation convention $\sum_{i \in I} \mathbb{R}_i := \langle W, \mathbb{R}^\circ, \mathbb{R}^\bullet \rangle$, where:

$$W := \sum_{i \in I} W_i \qquad \begin{aligned} (i, w) \Vdash \mathbb{R}^\circ &:= w \Vdash \mathbb{R}_i^\circ \\ (i, w) \Vdash \mathbb{R}^\bullet &:= w \Vdash \mathbb{R}_i^\bullet\, . \end{aligned}$$

We will write $\mathbb{R}_1 + \cdots + \mathbb{R}_n$ for the finitary case $\sum_{i=1}^n \mathbb{R}_i$. Finally, for $\mathbb{R} : A \Leftrightarrow A$ we define $\mathbb{R}^* := \sum_{n \in \mathbb{N}} \mathbb{R}^n$, where $\mathbb{R}^0 := \mathsf{id}$ and $\mathbb{R}^{n+1} := \mathbb{R}^n \cdot \mathbb{R}$.

Theorem 5.6. *The constructions* $\sqsupseteq, \cdot, +, *$ *work together as a typed Kleene algebra. Moreover, the following properties hold:*

$$\frac{\forall i\, .\, L_1 \le_{\mathbb{R} \to \mathbb{S}_i} L_2}{L_1 \le_{\mathbb{R} \to \sum_i \mathbb{S}_i} L_2} \qquad \frac{L \le_{\mathbb{R} \to \mathbb{S}} L}{L \le_{\mathbb{R}^* \to \mathbb{S}^*} L}$$

Writing $\mathcal{R} = \sum_i \mathbf{R}_i$, this enables refinements such as:

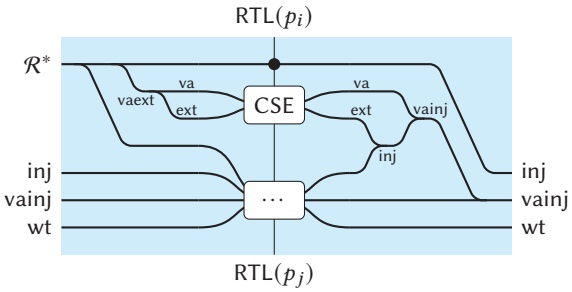


Figure 11. Incremental composition of C passes. Here the correctness proof of CSE is pre-composed without modifying the overall simulation convention. This process is iterated for the passes Deadcode through SimplLocals, and initiated using the compatibility of RTL with CKLRs and wt.

5.4 Dealing with Invariants

With some care, we can make sure the typing invariant wt is fairly well-behaved. The invariant va is more recalcitrant to commutation, but can be built into two new CKLRs.

Lemma 5.7. *If $\mathbb{R}$ is built from CKLRs, $\cdot$, $+$ and $*$,*

Lemma 5.8. *The CKLRs* vainj *and* vaext *satisfy:*

$$\mathsf{vaext} \equiv \mathsf{va} \cdot \mathsf{ext} \qquad \mathsf{vainj} \equiv \mathsf{va} \cdot \mathsf{inj} \equiv \mathsf{vainj} \cdot \mathsf{vainj}$$

Using these properties, correctness proofs for the C passes can be composed incrementally and selectively. This is illustrated in Fig. 11.

6 Related Work and Evaluation

A general survey, discussion and synthesis of various compositional compiler correctness results is provided by Patterson and Ahmed [20]. We focus on CompCert extensions. Our conceptual framework can be used to establish the taxonomy presented in Table 4.

CompCert and SepCompCert. The original correctness theorem of CompCert was stated as $\mathsf{C}_{\mathsf{wp}}(p) \preceq \mathsf{Asm}_{\mathsf{wp}}(p')$, where C_{wp} and $\mathsf{Asm}_{\mathsf{wp}}$ denote the source and target whole-program semantics. SepCompCert [10] later introduced the linking operator + and generalized the correctness theorem to the form discussed in §3.1.

Since external calls are not accounted for explicitly in this model, their behavior is specified by a parameter χ shared across all language semantics. The correctness proof assumes that χ is deterministic and that it satisfies a number of healthiness requirements, which roughly correspond to the $\mathcal{R}^*$ component of our simulation convention (§5).

Contextual Compilation. CompCertX [6], later followed by Stack-Aware CompCert [24], generalizes the incoming interface of programs from $\mathcal{W}$ to C, characterizing the behavior not only of `main` but of any function of the program, called with any argument values. This allows CompCertX and its correctness theorem to be used in the layer-based verification of the CertiKOS kernel: once the code of an abstraction layer has been verified, that layer's specification can be used as the new χ when the next layer is verified. However, this approach does not support mutually recursive components, and requires the healthiness conditions on χ to be proved before the next layer is added.

Table 4. Taxonomy of CompCert extensions in terms of the corresponding game models. The parameter $\chi : \mathbf{1} \twoheadrightarrow C$ fixes the behavior of external functions, whereas games on the left of arrows correspond to dynamic interactions. CompCertO's model is parametrized a language interface $A \in \mathbb{L} \supseteq \{C, \mathcal{A}\}$.

Variant	Semantic model
(Sep)CompCert [10, 16]	$\chi : \mathbf{1} \twoheadrightarrow C \vdash \mathbf{1} \twoheadrightarrow \mathcal{W}$
CompCertX [6]	$\chi : \mathbf{1} \twoheadrightarrow C \times \mathcal{A} \vdash \mathbf{1} \twoheadrightarrow C \times \mathcal{A}$
Comp. CompCert [23]	$C \twoheadrightarrow C$
CompCertM [22]	$C \times \mathcal{A} \twoheadrightarrow C \times \mathcal{A}$
CompCertO	$A \twoheadrightarrow A$

Compositional CompCert. The *interaction semantics* of Compositional CompCert [23] are closer to our own model but are limited to the language interface C. Likewise, the simulations used in Compositional CompCert correspond to our notion of forward simulation for a single convention called *structured injections*, which we will write $\mathbb{SI}$. Simulation proofs are updated to follow this model, and the *transitivity* of $\mathbb{SI}$ is established ($\mathbb{SI} \cdot \mathbb{SI} \equiv \mathbb{SI}$), so that passes can be composed to obtain a simulation for the whole compiler.

Compositional CompCert also introduced a notion of *semantic linking* similar to our horizontal composition (§3.2). As in our case, semantic linking is shown to preserve simulations (Thm. 3.4), however semantic linking is not related to syntactic linking of assembly programs, and this was later shown to present difficulties [22].

Another limitation of Compositional CompCert is the complexity of the theory and the proof effort required. In particular, many assumptions naturally expressed as relational invariants in the simulation relations of CompCert must be either captured by $\mathbb{SI}$ or handled at the level of language semantics, and many simulation proofs had to be largely rewritten to adapt them to structured injections.

CompCertM. The most recent extension of CompCert is CompCertM [22], which shares common themes and was developed concurrently with our work. While its correctness is ultimately stated in terms of closed semantics, CompCertM uses a notion of open semantics as an intermediate construction to enable compositional compilation and verification.

Table 5. Significant lines of code in CompCertO relative to CompCert v3.6. See Table 3 for a per-pass breakdown of the increase in size of pass correctness proofs, and `overhead.py` in the development for the list of files included in each group.

Component	SLOC	
Semantic framework (§3)	+566	(+10%)
Horizontal composition (§3.2)	698	
Simulation convention algebra (§2.5)	1,209	
CKLR theory and instances (§4)	2,756	
Parametricity theorems (§4)	3,314	
Invariant preservation proofs	+604	(+8%)
Simulation convention refinements (§5)	1,837	
Pass correctness proofs (Table 3)	+1,136	(+3%)
Total	**12,120**	

The open semantics used in CompCertM builds on interaction semantics by incorporating an assembly language interface. The resulting model can be characterized roughly as $C \times \mathcal{A} \twoheadrightarrow C \times \mathcal{A}$. Simulations are parametrized by Kripke relations similar to CKLRs (§4). While simulations do not directly compose, a new technique called *refinement under self-related context* (RUSC) can nonetheless be used to derive a contextual refinement theorem for the whole compiler with minimal overhead.

This approach has many advantages. As in CompCertO, CompCertM avoids much of the complexity of Compositional CompCert when it comes to composing passes, and the flexibility of simulations makes updating the correctness proofs of passes much easier. CompCertM also charts new ground, and goes beyond CompCertO in several directions. The RUSC relation used to state the final theorem is shown to be adequate with respect to the trace semantics of closed programs. CompCertM has improved support for static variables and module-local state, and the verification of the assembly runtime function `utod` is demonstrated.

In other aspects, CompCertM inherits limitations of previous approaches whereas CompCertO goes further. Because the compiler correctness theorem is not itself expressed as a simulation, it fails requirement #2 laid out in §1.3. Moreover, the parametrization of simulations does not offer the same flexibility as our notion of simulation convention. As a consequence, a cascade of techniques (repaired interaction semantics, enriched memory injections, the mixed simulations of Neis et al. [19]) are deployed to enforce invariants which find a natural relational expression under our approach.

An interesting question for future work will be to determine to what extent the techniques used by CompCertM and CompCertO could be integrated to combine the strengths of both developments. As a first step we present a detailed comparison of the two developments in Appendix A [13].

CompCertO. To give a sense of the overall complexity of CompCertO, we list in Table 5 the increase in significant lines of code it introduces compared to CompCert v3.6. As shown in Table 3, our methodology comes with a negligible increase in the complexity of most simulation proofs. Although SLOC is an imperfect measure, and a 1:1 comparison between developments which prove different things is difficult, our numbers represent a drastic improvement over Compositional CompCert, and compare favorably or are on par with the corresponding sections of CompCertM.

Our use of the simulation conventions `injp`, `CL`, and `LM` in particular underscores the benefits of our approach. The corresponding passes are the root of much complexity in Compositional CompCert, CompCertX and CompCertM. For instance, to express the requirement on the areas protected by `injp`, both Compositional CompCert and CompCertM introduce general mechanisms for tracking ownership of different regions of memory as part of an extended notion of memory injection. By contrast, our framework is expressive enough to capture a compositional version of the healthiness conditions imposed in CompCert on external functions. Consequently, very few changes were needed to update most injection passes.

Likewise, the preservation of callee-save registers ensured by the Allocation pass, and the subtle issues associated with argument-passing in the Stacking pass have been the cause of much pain in previous CompCert extensions, but they were fairly straightforward to address in our framework. This demonstrates the power of an explicit treatment of abstraction, made possible by our notions of language interface and simulation convention.

7 Conclusion

The distinguishing feature of CompCertO is the expressivity of our model, which allowed us to formulate a more precise correctness theorem and offered flexibility in the formalization and deployment of our composition techniques. Its design builds on our experience with certified abstraction layers [6, 8, 12], and aims to address some of the limitations associated with their use of CompCertX.

Looking forward, we hope this work will not only provide a compiler to be used in the verification of large-scale heterogeneous systems, but also represent a contribution to the conceptual apparatus required for this challenging task.

Acknowledgments

We would like to thank Arthur Olivera Vale, Yuting Wang, our shepherd Magnus Myreen, and anonymous referees for helpful feedback that improved this paper significantly. We also want to thank Yuyang Sang, Pierre Wilke, and Yu Zhang for their help with the Coq development. This research is based on work supported in part by NSF grants 1521523, 1763399, and 2019285.

References

[1] Samson Abramsky. 2010. From CSP to Game Semantics. In *Reflections on the Work of C.A.R. Hoare*. Springer, London, 33–45. https://doi.org/10.1007/978-1-84882-912-1_2

[2] Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Phil. Trans. R. Soc. A* 375, 2104 (2017), 20160331. https://doi.org/10.1098/rsta.2016.0331

[3] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. 2014. A verified information-flow architecture. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, New York, NY, USA, 165–178. https://doi.org/10.1145/2535838.2535839

[4] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

[5] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[6] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

[7] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. https://dl.acm.org/doi/10.5555/3026877.3026928

[8] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

[9] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. 2014. Logical Relations and Parametricity - A Reynolds Programme for Category Theory and Programming Languages. *Electron. Notes Theor. Comput. Sci.* 303 (March 2014), 149–180. https://doi.org/10.1016/j.entcs.2014.02.008

[10] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL'16)*. ACM, New York, NY, USA, 178–190. https://doi.org/10.1145/2837614.2837642

[11] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[12] Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. ACM, New York, NY, USA, 633–647. https://doi.org/10.1145/3373718.3394799

[13] Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. Yale Univ. Technical Report YALEU/DCS/TR-1556; https://flint.cs.yale.edu/publications/compcerto.html.

[14] Dexter Kozen. 1998. *Typed Kleene algebra*. Technical Report TR98-1669. Cornell University. https://hdl.handle.net/1813/7323

[15] Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2578855.2535841

[16] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[17] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

[18] Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.* 41, 1 (2008), 1–31.

[19] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2784731.2784764

[20] Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (July 2019), 29 pages. https://doi.org/10.1145/3341689

[21] Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. 2015. A Compositional Semantics for Verified Separate Compilation and Linking. In *Proceedings of the 2015 Conference on Certified Programs and Proofs* (Mumbai, India) *(CPP '15)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2676724.2693167

[22] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

[23] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 275–287. https://doi.org/10.1145/2676726.2676985

[24] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290375

[25] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, New York, NY, USA, 427–440. https://doi.org/10.1145/2103621.2103709

A Detailed Comparison with CompCertM

Since CompCertM is the state of the art when it comes to compositional extensions of CompCert, we provide in this appendix a discussion of the similarities and differences between CompCertM and CompCertO.

In some ways, CompCertM and CompCertO aim to solve different problems. CompCertM is a complete platform for the modular verification of composite C–assembly programs. By contrast, the goal of CompCertO is only to provide a precise formulation of CompCert's correctness theorem, with the aim of integrating it into a more general framework. We expect this framework to provide a verification infrastructure, in the same way certified abstraction layers provide a verification infrastructure around CompCertX.

Nevertheless, there is some amount of overlap between the techniques used in both projects, and we hope that the detailed comparison below can provide a useful discussion of the design space explored by both approaches.

A.1 Semantics in CompCertM

The CompCertM verification framework involves both the original closed semantics of CompCert ($\mathbf{1} \twoheadrightarrow \mathcal{W}$), and a more compositional semantics modeled after the interaction semantics of Compositional CompCert ($C \twoheadrightarrow C$).

To support hand-written assembly functions which do not follow the C calling convention, some functions can be specifically designated to be called using the assembly language interface $\mathcal{A}$ instead of C. In particular, this feature is used to demonstrate the verification of the builtin function utod used by compiled programs, whereas its behavior was previously axiomatized by CompCert. However, this facility is not used for assembly functions invoked in the usual way or compiled from C code, and does not otherwise have any bearing on CompCertM's approach to compiler correctness, so we elide it in our discussion below.

CompCertM does not directly define horizontal composition of open semantics (Definition 3.2), but uses a similar construction to give a closed semantics to a collection of open components, modeling both the conventional invocation of main and cross-component interactions:

$$\forall i \,.\, L_i : C \twoheadrightarrow C \;\vdash\; [L_1 \oplus \cdots \oplus L_n] : \mathbf{1} \twoheadrightarrow \mathcal{W}$$

Adequacy of the compositional semantics $\mathsf{Asm}_C(-)$ with respect to the original closed semantics $\mathsf{Asm}[-]$ is then established as the trace containment property

$$[\mathsf{Asm}_C(p_1) \oplus \cdots \oplus \mathsf{Asm}_C(p_n)] \supseteq \mathsf{Asm}[p_1 + \cdots + p_n], \quad (9)$$

similar in its role to Thm. 3.5.

A.2 Compiler Correctness

The correctness of each pass of CompCertM is established using a notion of *open simulation* similar in principle to the ones described in §3.3. These open simulations are parameterized by a notion of memory relation $\mathbf{R}$ which mirrors our use of CKLRs, so that they correspond roughly to $\leq_{\mathbf{R}_C \twoheadrightarrow \mathbf{R}_C}$.

The challenge is then to achieve horizontal and vertical compositionality, while making it possible for different compiler passes across different compilation units to use different memory relations. To this end, CompCertM eschews direct composition of open simulations in favor of a novel contextual technique dubbed *refinement under self-related contexts* (RUSC). This technique works in the following way.

First, every possible form of the simulation relation is shown to be adequate with respect to the closed semantics. More precisely, the following property is established for every memory relation $\mathbf{R} \in \mathcal{R}$ used in CompCertM:

$$\frac{\forall i \,.\, L_i \leq_{\mathbf{R}_C \twoheadrightarrow \mathbf{R}_C} L_i'}{[L_1 \oplus \cdots \oplus L_n] \supseteq [L_1' \oplus \cdots \oplus L_n']} \quad (10)$$

Next, consider a particular $\mathbf{R} \in \mathcal{R}$ and suppose that $L_i = L_i'$ for all but a single component L_k. Then as long as the fixed components are self-simulating under $\mathbf{R}$, a simulation of the form $L_k \leq_{\mathbf{R}_C \twoheadrightarrow \mathbf{R}_C} L_k'$ can be used to replace L_k by L_k' while preserving the observable behavior of the whole program. Iterating this strategy for the components $k = 1, \ldots, n$, we are free to use for each of the components any vertical succession of passes, as long as the source and target components are self-simulating under every possible $\mathbf{R} \in \mathcal{R}$.

The result is a very flexible approach which achieves horizontal and vertical compositionality with only minimal requirements. Every pass of CompCert can be easily extended to an open simulation by using the corresponding memory relation. CompCertM even demonstrates a memory relation which allows reasoning about module-local state and the introduction of a new Unreadglob optimization.

A.3 Symbol Tables

Compared with our work, CompCertM also has a more sophisticated handling of symbol tables.

CompCertO relies on a global symbol table which is used as-is by every module. This prevents us from verifying the Unusedglob pass of CompCert: when components access locally undeclared symbols, these accesses may become valid in in the context of a larger symbol table. These "rogue" accesses may in turn be invalidated by compilation if the component providing the symbols do not actually access or export them, causing their removal by Unusedglob.

By contrast, in addition to the global symbol table which results from linking, CompCertM assigns to each component a more restricted local symbol table, which is used for all the component's accesses. This prevents "rogue" accesses from occurring in the first place.

A.4 Expressing the Calling Convention

On the other hand, the use of RUSC in CompCertM imposes a series of design choices introducing their own complexity, which CompCertO's more direct approach can avoid.

Most immediately, RUSC imposes a homogeneous treatment of the semantics of source and target programs. Like that of Compositional CompCert, the compositional semantics $\mathsf{Asm}_C(-)$ of CompCertM is formulated in terms of the C language interface, with the the C calling convention built into its definition. Consequently, $\mathsf{Asm}_C(-)$ is by design an under-approximation: it must model as undefined any assembly execution which does not realize a C-level behavior. Unfortunately, the simple definition used in Compositional CompCert actually fails in this regard: it can assign incorrect C behaviors to assembly executions which violate the calling convention, breaking property (9).

To an extent, this reflects the inherent complexity of the kind of data abstraction performed by calling conventions; specifically, the correspondence between C and assembly behaviors involves *dual nondeterminism*. On one hand, the *environment* is free to choose among many valid assembly representations of a given C query. In particular, registers which do not encode relevant information can be assigned arbitrary values. On the other hand, the *system* can choose among the valid assembly representations of the C reply: caller-save registers may be left in unspecified states, and the assembly execution is free to allocate additional or larger memory blocks.

As a result, in the context of the transition systems and forward simulations used in most of CompCert, the connection between the C interface and the mechanics of assembly semantics is difficult to express. Nevertheless, this was successfully achieved in CompCertM at the cost of some complexity:

- For incoming calls, in order to simulate the way arguments are passed across assembly functions, the assembly semantics allocates a stack block and stores the arguments there. It also assigns opaque handles to the registers unconstrained by the C calling convention, in the form of pointers to newly allocated empty memory blocks, preventing their use by the assembly program. When the call returns, the semantics checks that callee-save registers maintain their original values and frees the simulated caller stack block.
- For outgoing calls, a complementary process occurs. Permissions are cleared on the arguments region of the stack, ensuring that any access by the context will result in an undefined behavior. After the call returns, permissions are restored and the new register state is constructed by copying the original values of callee-save register and setting caller-save registers to undef.

Similar constructions must be introduced in the semantics of the intermediate languages LTL, Linear and Mach, and the proof of (9) must show that they give a sound account of the execution of the linked assembly program.

A.5 Mixed Forward-Backward Simulations

The changes described in the previous section introduce nondeterminism in the initial states of $\mathsf{Asm}_C(-)$. In turn, this propagates to those internal states of $[L_1 \oplus \cdots \oplus L_n]$ which correspond to cross-component interactions. As a consequence, the forward simulations used in most of CompCert are no longer a satisfactory reasoning principle.

To address this issue, CompCertM uses a notion of *mixed* forward-backward simulation [19]. These mixed simulations allow a fine-grained use of forward simulations for deterministic states, but requires a more complex backward simulation when nondeterministic states are encountered.

A.6 CompCertO's Simplifications

This series of complications is avoided in CompCertO in two ways. First, our semantics of assembly is formulated exclusively in terms of the language interface $\mathcal{A}$. As such, it characterizes the behavior of assembly components more precisely and independently of the C calling convention, rendering the proof of Thm. 3.5 straightforward. Secondly, by formulating the C calling convention as a simulation convention, we are able to circumscribe its inherent nondeterminism and avoid the use of nondeterministic transition systems and sophisticated simulation properties.

Another aspect where CompCertO is simpler concerns our treatment of injection passes. In the context of RUSC, the per-pass simulations used by CompCertM must use the same memory relation for incoming and outgoing calls. This means that the guarantees provided by injections proofs on incoming calls must be strengthened to correspond to their own requirements on outgoing calls. The approach used in CompCertO is more flexible, allowing us to assign the convention $\mathsf{injp} \twoheadrightarrow \mathsf{inj}$ for the passes SimplLocals, Inlining and Stacking and to avoid the corresponding modifications of proofs needed in CompCertM.

B Invariants

Several passes of CompCert rely on the preservation of invariants by their source program: when the semantics of a language preserves an invariant, the preservation properties can assist in proving forward simulations which use the language as their source. This allows us to decompose the simulation proof, and in the case of RTL the preservation proofs can be reused for multiple passes.

In CompCert, this technique is deployed in an ad-hoc manner: for each pass using an invariant, the simulation relation is strengthened to assert that the invariant holds on the source state, and the preservation properties for the source language are used explicitly in the simulation proof to maintain this invariant. In CompCertO, this becomes more

involved, because the simulation convention must be altered to ensure that invariants are preserved by external calls.

On the other hand, our simulation infrastructure offers the opportunity to capture and reason about invariants explicitly, and to further decouple preservation and simulation proofs. In this section, we give an overview of our treatment of invariants. For details, see `common/Invariant.v` in the Coq development.

B.1 Invariants and Language Interfaces

First, we define a sort of "invariant convention", which describes how a given invariant impacts the questions and answers of the language under consideration.

Definition B.1. An *invariant for a language interface* A is a tuple $\mathbb{P} = \langle W, \mathbb{P}^\circ, \mathbb{P}^\bullet \rangle$, where W is a set of worlds and $\mathbb{P}^\circ, \mathbb{P}^\bullet$ are families of predicates on $A^\circ, A^\bullet$ indexed by W.

Example B.2. Typing constraints for the language interface C can be expressed as the invariant:

$$\mathsf{wt} := \left\langle \mathsf{sig},\ \mathbb{P}^\circ_{\mathsf{wt}},\ \mathbb{P}^\bullet_{\mathsf{wt}} \right\rangle$$

$$\frac{\vec{v} <: sg.\mathsf{args}}{sg \Vdash vf[sg](\vec{v})@m \in \mathbb{P}^\circ_{\mathsf{wt}}} \qquad \frac{v' <: sg.\mathsf{res}}{sg \Vdash v'@m' \in \mathbb{P}^\bullet_{\mathsf{wt}}}$$

The proposition $\vec{v} <: sg.\mathsf{args}$ asserts that the types of the arguments $\vec{v}$ match those specified by the signature sg. The proposition $v' <: sg.\mathsf{res}$ asserts a similar property for the return value v'.

Invariants can be seen as a special case of simulation convention which constrain the source and target questions and answers to be equal. This can be formalized as follows.

Definition B.3 (Simulation conventions for invariants)**.** A W-indexed predicate P on a set X can be promoted to a Kripke relation $\hat{P} \in \mathcal{R}_W(X, X)$ defined by the rule:

$$\frac{w \Vdash x \in P}{w \Vdash x\ \hat{P}\ x}$$

Then an invariant $\mathbb{P} = \langle W, \mathbb{P}^\circ, \mathbb{P}^\bullet \rangle$ can be promoted to a simulation convention: $\hat{\mathbb{P}} := \langle W, \hat{\mathbb{P}}^\circ, \hat{\mathbb{P}}^\bullet \rangle$.

B.2 Typing Invariants

The typing invariant described in Ex. B.2 is used by the Selection and Allocation passes. We have updated their correctness proofs as well as the preservation proofs found in `Cminortyping.v` and `RTLtyping.v` to use our framework.

The invariant wt satisfies one key property: when a simulation convention $\mathbb{R}$ consists of a sequence of CKLRs and other invariants, the following property holds:

$$\mathsf{wt} \cdot \mathbb{R} \cdot \mathsf{wt} \equiv \mathbb{R} \cdot \mathsf{wt}$$

This means CompCertO's overall simulation convention can eliminate the typing invariant for the Selection pass, retaining only that used for Allocation. In turn, this facilitates the simplification of the convention for the passes from Clight to Inlining.

B.3 Value Analysis

The passes Constprop, CSE and Deadcode use CompCert's value analysis framework. Abstract interpretation is performed on their source program, and the resulting information is used to carry out the optimizations. The correctness proofs for these passes then rely on the invariant va, which asserts that the concrete runtime states satisfy the constraints encoded in the corresponding abstract states.

We have updated the value analysis framework and the associated pass correctness proofs to fit the invariant infrastructure described in this section. Value analysis passes use the convention $\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$. Unfortunately, because it combines constraints with mixed variance, the invariant va does not propagate in the same way as wt. However, as discussed in §5.4, we can define CKLRs which embed va, allowing it to propagate across language boundaries.

B.4 Simulations Modulo Invariants

The top row in Fig. 12 illustrates the preservation of invariants by transition systems. In the context of a transition system $L_1 : A_1 \to B_1$, we consider three invariants working together:

- an invariant $\mathbb{P}_A$ for the language interface A;
- an invariant $\mathbb{P}_B$ for the language interface B;
- a W_B-indexed predicate P on the states of L_1.

The preservation of $\langle \mathbb{P}_A, \mathbb{P}_B, P \rangle$ is then analogous to a unary simulation property, where $\mathbb{P}_A \twoheadrightarrow \mathbb{P}_B$ play the roles of the simulation conventions, and P plays the role of the simulation relation. In fact, when L_1 preserves these invariants, the following property holds:

$$L_1 \leq_{\hat{\mathbb{P}}_A \twoheadrightarrow \hat{\mathbb{P}}_B} L_1$$

Once we have established that the source language preserves the invariants, we wish to use this fact to help prove the forward simulation for a given pass. To this end, we define a *strengthened* transition system $L_1^{\mathbb{P}} : A_1 \twoheadrightarrow B_1$, with the property that $L_1 \leq_{\hat{\mathbb{P}}_A \twoheadrightarrow \hat{\mathbb{P}}_B} L_1^{\mathbb{P}}$. For a target transition system $L_2 : A_2 \twoheadrightarrow B_2$, it then suffices to show that $L_1^{\mathbb{P}} \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ to establish:

$$L_1 \leq_{\hat{\mathbb{P}}_A \cdot \mathbb{R}_A \twoheadrightarrow \hat{\mathbb{P}}_B \cdot \mathbb{R}_B} L_2\,.$$

Simulations from $L_1^{\mathbb{P}}$ are easier to prove, because $L_1^{\mathbb{P}}$ provides assumption that the invariants hold on all source questions, answers and states. The simulation diagrams reduce to those shown in the bottom row of Fig. 12. However, since they are formulated in terms of Def. 3.3, the standard forward simulation techniques defined by CompCert in `Smallstep.v` remain available.

C Specialized Simulation Conventions

For the Alloc, Stacking and Asmgen passes, we the simulation conventions CL, LM and MA to bridge the gap between the language interfaces of the source and target programs. These simulation conventions express the correspondence

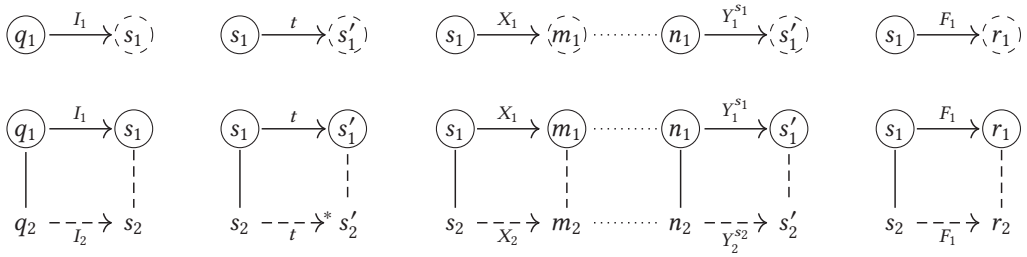


Figure 12. Simulation with invariants. Circles indicate questions, answers and states which satisfy the appropriate invariants. When the transition system L_1 preserves the invariants in the way shown in the top row, a simulation of L_1 by L_2 can be established through the weakened diagrams shown in the bottom row. The resulting simulation uses the convention $\mathbb{P}_A \cdot \mathbb{R}_A \twoheadrightarrow \mathbb{P}_B \cdot \mathbb{R}_B$, ensuring that the environment establishes and preserves the appropriate invariants on questions and answers. The simulation relation $P \cdot R$ then ensures that the strengthened assumptions used by the weakened simulation diagrams can be satisfied.

between the higher-level and lower-level representations of function calls and returns.

We briefly discuss each one below. In the interest of clarity, some technical details have been elided. We refer the reader to the code for details, especially `backend/Conventions.v`, `backend/Mach.v`, `x86/Asm.v` and `driver/CallConv.v`.

C.1 The Allocation Pass

The Allocation pass from RTL to LTL is the first pass to modify the interface of function calls. LTL uses *abstract locations* which represent the stack slots and machine registers eventually used in the target assembly program. Abstract locations contain arguments, temporaries and return values. The contents are stored in a *location map*, passed across components by the interface $\mathcal{L}$ alongside memory states. Up to RTL, arguments are passed as standalone values, but in LTL they are mapped to abstract locations.

To express the simulation convention used by Allocation, we will use the following notations. For a signature sg and a location map ls, we write $\mathsf{args}(sg, ls)$ to represent the argument values stored in ls. Likewise, $\mathsf{retval}(sg, ls)$ extracts the contents of locations used to store the return value.

The simulation convention $\mathsf{CL} : C \Leftrightarrow \mathcal{L}$ uses its world to remember the signature associated with a call, and can then be defined by:

$$\mathsf{CL} := \langle \mathsf{signature},\ R^{\circ}_{\mathsf{CL}},\ R^{\bullet}_{\mathsf{CL}} \rangle$$

$$\frac{\vec{v} = \mathsf{args}(sg, ls)}{sg \Vdash vf[sg](\vec{v})@m\ R^{\circ}_{\mathsf{CL}}\ vf[sg](ls)@m}$$

$$\frac{v' = \mathsf{retval}(sg, ls')}{sg \Vdash v'@m'\ R^{\bullet}_{\mathsf{CL}}\ ls'@m'}$$

To incorporate the typing invariant and memory extension used by Allocation, we can then use the convention:

$$\mathsf{wt} \cdot \mathsf{ext} \cdot \mathsf{CL} \twoheadrightarrow \mathsf{wt} \cdot \mathsf{ext} \cdot \mathsf{CL}$$

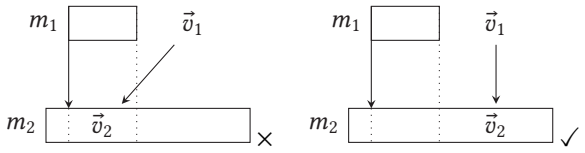


Figure 13. Separation of arguments in $\mathsf{LM} : \mathcal{L} \Leftrightarrow \mathcal{M}$.

C.2 The Stacking Pass

The information which LTL and Linear store in abstract stack locations is consolidated by the Stacking pass into in-memory stack frames and machine registers. The simulation proof uses a memory injection, and involves maintaining separation properties ensuring that the source memory and the regions of stack frames introduced by Stacking occupy disjoint areas of the target memory.

With regards to the memory state, Stacking uses the CKLR injp for outgoing calls. Since the new regions of stack frames are outside the image of the source memory, and most of them are local to function activations, the properties of injp are largely sufficient (see also §4.6).

However, argument passing creates a particular challenge. Most stack locations used by a given function activation are allocated by the function itself, ensuring their protection by injp. By contrast, in order to read incoming arguments, the function accesses the caller's stack frame. If the area used to store arguments overlaps with the injected source memory state, then the source program and external calls may alter them in unexpected ways (Fig. 13).

In previous CompCert extensions, sophisticated techniques were required to prevent this from happening. In our model, we can simply encode the required separation condition in the simulation convention $\mathsf{LM} : \mathcal{L} \Leftrightarrow \mathcal{M}$. To achieve this, LM requires the source memory state to be identical to the

target memory state, with the arguments region removed:

$$\mathsf{LM} := \left\langle \mathsf{signature} \times \mathsf{regset} \times \mathsf{mem} \times \mathsf{val},\ \mathsf{LM}^\circ,\ \mathsf{LM}^\bullet \right\rangle$$

$$\frac{ls = \mathsf{make\_locset}(rs, m, sp) \qquad \bar{m} = \mathsf{free\_args}(sg, m, sp)}{(sg, rs, m, sp) \Vdash vf[sg](ls)@\bar{m}\ \mathsf{LM}^\circ\ vf(sp, ra, rs)@m}$$

The operation make_locset synthesizes a location map by accessing the target-level machine registers and stack slots corresponding to arbitrary abstract locations. The operation free_args removes the argument region from memory.

For answers, the various data saved as (sg, rs, m, sp) can be used to encode various constraints:

$$\frac{rs' \equiv_{\mathsf{R}} ls' \qquad rs' \equiv_{\mathsf{CS}} rs \qquad m' = \mathsf{mix}(sg, sp, m, \bar{m}')}{(sg, rs, m, sp) \Vdash ls'@\bar{m}'\ \mathsf{LM}^\bullet\ rs'@m'}$$

Here, $\equiv_{\mathsf{R}}$ ensures the ls' and rs' have identical contents for machine registers, $\equiv_{\mathsf{CS}}$ enforces the preservation of callee-save registers, and mix copies back the arguments region of m into $\bar{m}'$ to obtain m'.

Note that because of the complexity of the Stacking pass, we use an intermediate simulation convention stacking[**R**] to characterize the correctness proof, using **R** := injp for outgoing calls and **R** := inj for incoming calls. This simulation convention combines the CKLR and structural changes together, making it easier to connect to the simulation relation used in the Stacking proof. To derive the more flexible convention, we then use the following refinements:

$$\mathsf{injp} \cdot \mathsf{LM} \sqsubseteq \mathsf{stacking}[\mathsf{injp}] \twoheadrightarrow \mathsf{stacking}[\mathsf{inj}] \sqsubseteq \mathsf{LM} \cdot \mathsf{inj}$$

See `driver/CallConv.v` for details.

Remark C.1. *Operations like* free_args *are used for similar purposes in CompCertX and CompCertM, in different ways.*

In CompCertX, the source and target program always share the same initial memory state m. To deal with the issues surrounding argument passing, the correctness theorem evaluates the source program on $\bar{m}$ as well as m, and checks that it doesn't go wrong. The required Stacking *invariants can be established from this side-condition, however this significantly increases the size and complexity of the proof.*

In CompCertM, a free_args *operation is used prior to external calls in the semantics of* Mach *and* Asm*: the arguments to use for the external call are read from the arguments region, then region is removed. After the call returns, the permissions on the region are restored and the execution resumes. This follows the more general pattern in CompCertM where the calling convention is modeled as part of the semantics of low-level languages.*

C.3 The Asmgen Pass

Asm introduces explicit registers for the program counter, stack pointer and return address. The simulation convention $\mathsf{MA} : \mathcal{M} \Leftrightarrow \mathcal{A}$ ensures that the appropriate components of Mach-level queries are mapped to the new registers. In addition, we must ensure that the call returns the stack pointer to its original value, and sets the program counter to the return address specified by the caller. This is captured by:

$$\mathsf{MA} := \left\langle \mathsf{val} \times \mathsf{val},\ \mathsf{MA}^\circ,\ \mathsf{MA}^\bullet \right\rangle$$

$$\frac{rs_2 = rs_1 \uplus [\mathsf{sp} := sp, \mathsf{ra} := ra, \mathsf{pc} := vf]}{(sp, ra) \Vdash vf(sp, ra, rs_1)@m\ \mathsf{MA}^\circ\ rs_2@m}$$

$$\frac{rs_2' = rs_1' \uplus [\mathsf{sp} := sp, \mathsf{pc} := ra]}{(sp, ra) \Vdash rs_1'@m'\ \mathsf{MA}^\bullet\ rs_2'@m'}$$