A Framework for Compositional Verification with Data Abstraction, Encapsulated State and Certified Compilation

Formal verification is the gold standard for building reliable computer systems. *Certified* systems in particular come with a formal specification, and with a proof of correctness which can easily be checked by a third party [17]. Unfortunately, verifying large-scale, heterogeneous systems remains out of reach of current techniques. Addressing this challenge will require the use of compositional methods, making it easier to construct certified systems from off-the-shelf certified components [2].

In principle, compositional semantics could play a key role in enabling this. In practice, simpler operational models have proven more amenable to carrying out actual verification projects, and few compositional semantic models support the combination of features required for large-scale verification tasks.

This paper is concerned with bridging this gap. We present a compositional, refinement-based verification framework which incorporates *data abstraction*, *state encapsulation* and *certified compilation*.

1 INTRODUCTION

Building large-scale certified systems requires the ability to model and specify those systems compositionally, so that verification can be carried out on components of a manageable size. In addition, the verification of large heterogeneous systems—for example, computer systems involving combinations of hardware, software and network components—will require models versatile enough to account for the large variety of operational paradigms and interfaces involved.

1.1 Requirements for Certified Systems Engineering

This paper is concerned with the design of verification frameworks providing the following features.

1.1.1 Compositionality. Complex systems are built by assembling more elementary components. In a compositional verification framework, correctness properties must be compatible with this process. This leads us to favor a *refinement-based* approach, where specifications and implementations are represented by mathematical objects of the same kind. The correctness of a composite system $x \circ y \circ z$ with respect to a specification σ is then stated as the refinement $\sigma \sqsubseteq x \circ y \circ z$.

Under this approach, compositionality is enabled by two key properties. First, the monotonicity of composition with respect to refinement makes it possible to replace a specification by its implementation in any context. Second, the transitivity of refinement allows us to carry out verification in steps. For example, it may be easier to first show $\tau \sqsubseteq x \circ y$, then reason about z in terms of the specification τ to derive $\sigma \sqsubseteq \tau \circ z \sqsubseteq x \circ y \circ z$.

1.1.2 Data Abstraction. It is often desirable to describe a complex system and its constituents in different terms. The transistors x, y, z may operate in terms of continuous voltages, but we would like the specification σ of the logic gate they implement to be formulated in terms of the binary values $\{0, 1\}$. In other words, *data abstraction* is critical to the construction of large systems.

Unfortunately, this means that the truth table σ and the circuit $x \circ y \circ z$ can no longer be directly compared. The correctness property $\sigma \sqsubseteq_R x \circ y \circ z$ is now contingent on a convention R which explains the relationship between the logical and electrical views of the system. To support this kind of abstraction, a verification framework must provide a way to express such conventions and manage their interaction with the composition principles.

1.1.3 State Encapsulation. As stateful systems become larger, so does the number of state variables and the potential for undesirable interactions between them. Reasoning about such systems requires partitioning their state among loosely coupled *objects* and controlling any interference.

State encapsulation achieves this by rendering all or part of the state used by a component inaccessible by its environment. From this restriction, we gain a guarantee that encapsulated state can only be influenced by interaction with the component through its interface.

1.1.4 Certified Compilation. Compilers play a critical role in the construction of modern software systems, by allowing the low-level code executed by the computer to be built and analyzed using high-level languages. However, this also creates a challenge for correctness: if we verify programs at the source level only, the compiler may still introduce bugs into the code that is actually run.

Certified compilers such as CompCert [12] mitigate this issue by establishing a correctness proof for the compiler itself. But ensuring *end-to-end* correctness requires the verification framework to seamlessly integrate with this correctness proof, so that guarantees obtained with respect to source-level components can be formally transferred to the target program.

There exist several verification frameworks implemented in the Coq proof assistant—such as the Verified Software Toolchain [1], Certified Abstraction Layers [5] or CompCertM [18]—which interface with CompCert to yield end-to-end proofs. However, none of them satisfies all of the requirements which we have outlined for supporting large-scale verification.

1.2 Contributions

We present a verification framework, implemented in the Coq proof assistant and based on the certified compiler CompCertO [9], which provides these features:

- We show that, under an alternative definition of horizontal composition, the semantic model of CompCertO exhibits the compositional structure of a *double category* (§3). This formalizes the usual notions of horizontal and vertical composition for compiler correctness proofs.
- We extend the model with a compositional treatment of state (§4), allowing both specifications and data abstraction to act independently on different components of the global state.
- A partial commutative monoid structure defined on the CompCert memory model (§A) bridges the gap between this compositional view and the global memory used by CompCert.
- Conversely, we extend the compiler's source language Clight to take advantage of our treatment of state (§5). The resulting language ClightP allows the definition of component-local *private* variables which are kept separate from the global memory.
- Finally, we extend our model with state encapsulation (§6), and explain how the associated primitives interact with the model's compositional structure.

Identifying the categorical structures underlying our model allows us to make extensive use of *string diagrams* [14] to describe composite specifications, abstraction relations and simulation proofs in an intuitive way. Because our model remains compatible with that of CompCertO, the compiler's correctness theorem can be incorporated into our framework as-is.

2 OVERVIEW OF THE FRAMEWORK

We begin by providing a high-level overview of our framework. Its ingredients are shown in Table 1. Note that as their dimensionality increases, the objects we manipulate support increasingly many composition operations. Our starting point is the semantic model used in CompCertO.

2.1 Semantics in CompCertO

The compositional semantics of CompCertO uses a notion of open transition system $L : A \twoheadrightarrow B$. The type $A \twoheadrightarrow B$ involves two *language interfaces* A and B, which decribe how components interact.

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

Table 1. A summary of the various kinds of mathematical objects we use to model interfaces, component behaviors, abstraction conventions and refinement properties. For each one, we show the horizontal (H), vertical (V) and state (S) composition operations they support. Active components are our main focus, but transition systems can only be @-composed with simpler, passive components. When we make use of a particular string diagram representation for components of a given kind (§2.5), we also list the composition operations which correspond to the horizontal ($\mathbb{H}$) and vertical ($\mathbb{V}$) juxtaposition of diagrams. Note that the two-dimensional structure **TSC** contains **TS** and **SC** as its edges, and likewise **BSR** contains **Bij** and **Rel**.

	Role	Components	Notation	Compose H V S	Diagrams $\mathbb{H}$ $\mathbb{V}$
Active	Interface	Language interfaces	A, B, C	$\otimes$	
	Behavior	Transition systems	$L : A \twoheadrightarrow B \in \mathbf{TS}$	$\odot$ @	$\odot$ @
	Abstraction	Simulation conventions	$\mathbf{R} : A \leftrightarrow B \in \mathbf{SC}$	; $\otimes$	$\otimes$;
	Refinement	Simulations	$\pi : L^\sharp \leq_{\mathbf{R}\twoheadrightarrow\mathbf{S}} L^\flat \in \mathbf{TSC}$	$\odot$; @	$\odot$;
Passive	Interface	Sets	U, V	$\times$	
	Behavior	Bijections	$f : U \cong V \in \mathbf{Bij}$	$\circ$ $\times$	$\circ$ $\times$
	Abstraction	Simulation relations	$R \subseteq U \times V \in \mathbf{Rel}$	$\cdot$ $\times$	$\times$ $\cdot$
	Refinement	Bisimulations	$\phi : f \equiv_{R\cong S} g \in \mathbf{BSR}$	$\circ$ $\cdot$ $\times$	$\circ$ $\cdot$

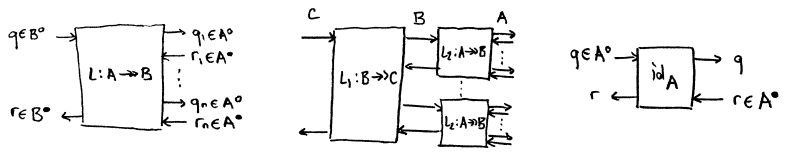


Fig. 1. Informal description of CompCertO's transition systems and their composition

Two important language interfaces are involved in formulating the CompCertO correctness theorem. The language interface C models interactions between C translation units. The language interface $\mathcal{A}$ is used to describe the behavior of assembly-level components.

Example 2.2 (The C language interface). When a C component is invoked, the environment specifies a function to be called and argument values. When the function is done executing, control switches back to the environment and the component specifies a return value. This can be described by the language interface C, defined as follows:

$$C^\circ := \{f(\vec{v}) \mid f \in \mathsf{ident}, \vec{v} \in \mathsf{val}^*\} \qquad C^\bullet := \mathsf{val}$$

In addition, C programs may access and alter the global memory state. This means that the current memory state must be attached to every function call, and an updated state must be communicated back alongside the return value. To this end, CompCertO uses the following language interface:

$$(C \mathbin{@} \mathsf{mem})^\circ := \{f(\vec{v})@m \mid f \in \mathsf{ident}, \vec{v} \in \mathsf{val}^*, m \in \mathsf{mem}\}$$
$$(C \mathbin{@} \mathsf{mem})^\bullet := \{v@m \mid v \in \mathsf{val}, m \in \mathsf{mem}\}\,.$$

Transition Systems. We describe CompCertO's transition systems in detail in §3. For now, it suffices to note that a transition system of type $A \twoheadrightarrow B$ describes a behavior of the shape depicted in Fig. 1a, generating interaction traces of the following form:

$$q \rightarrowtail (q_1 \rightsquigarrow r_1) \rightarrowtail \cdots \rightarrowtail (q_n \rightsquigarrow r_n) \rightarrowtail r$$

,,

```c
/* Encapsulated state */
static int c1, c2;
static V buf[N];

/* Accessors */
int inc1() { int i = c1++; c1 %= N; return i; }
int inc2() { int i = c2++; c2 %= N; return i; }
V get(int i) { return buf[i]; }
void set(int i, V val) { buf[i] = val; }
```

(a) The translation unit rb.c

```c
/* Underlay signature */
extern int inc1(void);
extern int inc2(void);
extern V get(int i);
extern void set(int i, V val);

/* Layer implementation */
void enq(V val) { set(inc2(), val); }
V deq() { return get(inc1()); }
```

(b) The translation unit bq.c

Fig. 2. Running example, consisting of two C components. The component rb.c implements a ring buffer by encapsulating an array and two counters. It is used by the component bq.c to implement a bounded queue.

Here, $\rightarrowtail$ denotes internal execution and $\rightsquigarrow$ denotes a step where the environment is in control: The component is activated by an incoming call, described by a question $q \in B^\circ$. As it executes, the transition system may perform outgoing calls, asking questions $q_1, \ldots, q_n \in A^\circ$ and receiving corresponding answers $r_1, \ldots, r_n \in A^\bullet$. Execution terminates with the top-level answer $r \in B^\bullet$.

Example 2.3 (Clight semantics). CompCertO defines the semantics of a C translation unit M as:

$$\mathsf{Clight}(M) : C \ @\ \mathsf{mem} \twoheadrightarrow C \ @\ \mathsf{mem}$$

Consider the translation unit rb.c shown in Fig. 2. If inc1() is invoked with a memory state where the variable c1 has value 5, the transition system Clight(rb.c) will behave as follows:

$$\mathsf{Clight}(\mathsf{rb.c}) \quad \vDash \quad \mathsf{inc1}()@m[\mathsf{c1} \mapsto 5] \ \rightarrowtail\ 5@m[\mathsf{c1} \mapsto 6]$$

Note that the memory is updated to store the new value of the counter c1. By contrast, deq() in bq.c does not directly modify the memory, but it makes outgoing calls which may have that effect:

$$\mathsf{Clight}(\mathsf{bq.c}) \ \vDash\ \mathsf{deq}()@m \rightarrowtail \big(\mathsf{inc1}()@m \rightsquigarrow i@m'\big) \rightarrowtail \big(\mathsf{get}(i)@m' \rightsquigarrow v@m''\big) \rightarrowtail v@m''\,.$$

2.2 Layered Composition

Transition systems can be composed in different ways. To model *linking*, the original work on CompCertO introduces an operator $\oplus_A : (A \twoheadrightarrow A) \times (A \twoheadrightarrow A) \to (A \twoheadrightarrow A)$. This operator allows mutual recursion: in $L_1 \oplus L_2$, both outgoing calls of L_1 to functions of L_2 and outgoing calls of L_2 to functions of L_1 become internal calls and are hidden from the environment. When mutual recursion is not needed, we can use a more traditional and flexible operator

$$\odot_{A,B,C} : (B \twoheadrightarrow C) \times (A \twoheadrightarrow B) \to (A \twoheadrightarrow C)\,.$$

Here, the transition systems interact only over the language interface B, so that in $L_1 \odot L_2$, incoming calls in C activate L_1, the outgoing calls of L_1 in B are handled by L_2, and the outgoing calls of L_2 are directed back to the environment in A. This is depicted in Fig. 1b.

This mode of composition is hinted at in Koenig and Shao [9]. We provide a formal definition in §3.1 and show that it defines a category **TS** of language interfaces and transition systems. In particular, the unit for $\odot$ is the transition system $\mathsf{id}_A : A \twoheadrightarrow A$ depicted in Fig. 1c, which echoes the incoming question as an outgoing one and propagates the answer back to the caller.

Example 2.4. The Clight semantics of bq.c and rb.c compose as:

$$\mathsf{Clight}(\mathsf{bq.c}) \odot \mathsf{Clight}(\mathsf{rb.c}) : C \ @\ \mathsf{mem} \twoheadrightarrow C \ @\ \mathsf{mem}$$

The transition system above generates traces such as

$$\mathsf{deq}()@m[\mathsf{c1} \mapsto 1, \mathsf{buf} \mapsto \{a, b, c, \ldots\}] \ \rightarrowtail\ b@m[\mathsf{c1} \mapsto 2, \mathsf{buf} \mapsto \{a, b, c, \ldots\}]\,.$$

2.3 Adjoining State

We have seen that CompCertO requires the current memory state to be attached to every question and answer, and used the notation C @ mem to describe the resulting language interface. Taking a cue from [8], this can be generalized to arbitrary language interfaces and sets of states:

$$A \mathbin{@} U \;:=\; \langle A^\circ \times U,\ A^\bullet \times U \rangle$$

Example 2.5 (Abstract specifications). CompCertO semantics use the language interface C @ mem, but we can give a simpler description of the code in Figure 2. Using the empty language interface $\top = \langle \varnothing, \varnothing \rangle$ and describing the queue as $\vec{q} \in D_{\mathsf{bq}} := \mathsf{val}^*$, we can give an overall specification

$$L_{\mathsf{bq}} : \top \twoheadrightarrow C \mathbin{@} D_{\mathsf{bq}} \qquad \text{such that} \qquad \begin{array}{l} L_{\mathsf{bq}} \vDash \mathsf{enq}(v)@\vec{q} \rightarrowtail \mathsf{undef}@\vec{q}v\,, \\ L_{\mathsf{bq}} \vDash \mathsf{deq}()@v\vec{q} \rightarrowtail v@\vec{q}\,. \end{array}$$

Likewise, we can specify rb.c in isolation with an abstract state of type $D_{\mathsf{rb}} := \mathsf{val}^N \times \mathbb{N} \times \mathbb{N}$, and define $L_{\mathsf{rb}} : \top \twoheadrightarrow C@D_{\mathsf{rb}}$ such that:

$$\begin{array}{ll} L_{\mathsf{rb}} \vDash \mathsf{set}(i,v)@(b,c_1,c_2) \rightarrowtail \mathsf{undef}@(b[i := v], c_1, c_2) & L_{\mathsf{rb}} \vDash \mathsf{inc1}()@(b,c_1,c_2) \rightarrowtail c_1@(b, c_1{+}1, c_2) \\ L_{\mathsf{rb}} \vDash \mathsf{get}(i)@(b,c_1,c_2) \rightarrowtail b_i@(b,c_1,c_2) & L_{\mathsf{rb}} \vDash \mathsf{inc2}()@(b,c_1,c_2) \rightarrowtail c_2@(b, c_1, c_2{+}1) \end{array}$$

Finally, note that bq.c does not use any state of its own. We can describe it simply as

$$\Sigma_{\mathsf{bq}} : C \twoheadrightarrow C \qquad \text{such that} \qquad \begin{array}{l} \Sigma_{\mathsf{bq}} \vDash \mathsf{enq}(v) \rightarrowtail (\mathsf{inc2}() \rightsquigarrow i) \rightarrowtail (\mathsf{set}(i,v) \rightsquigarrow *) \rightarrowtail *\,, \\ \Sigma_{\mathsf{bq}} \vDash \mathsf{deq}() \rightarrowtail (\mathsf{inc1}() \rightsquigarrow i) \rightarrowtail (\mathsf{get}(i) \rightsquigarrow v) \rightarrowtail v\,. \end{array}$$

Using these specifications, we would hope to decompose a correctness proof into

$$\dfrac{\dfrac{L_{\mathsf{bq}} \sqsubseteq \Sigma_{\mathsf{bq}} \odot L_{\mathsf{rb}} \qquad \Sigma_{\mathsf{bq}} \sqsubseteq \mathsf{Clight(bq.c)}}{L_{\mathsf{bq}} \sqsubseteq \mathsf{Clight(bq.c)} \odot L_{\mathsf{rb}}} \qquad L_{\mathsf{rb}} \sqsubseteq \mathsf{Clight(rb.c)}}{L_{\mathsf{bq}} \sqsubseteq \mathsf{Clight(bq.c)} \odot \mathsf{Clight(rb.c)}}$$

However, note that the different types of states used by the various components prevents them from being composed or compared directly. We address this issue below.

Transition Systems. In our framework, a transition system $L : A \twoheadrightarrow B$ can be made compatible with an extra state component of type U by extending it to $L \mathbin{@} U : A \mathbin{@} U \twoheadrightarrow B \mathbin{@} U$. The transition system $L \mathbin{@} U$ transparently passes along a state component of type U as follows:

$$\dfrac{L \;\vDash\; q \rightarrowtail (q_1 \rightsquigarrow r_1) \rightarrowtail \cdots \rightarrowtail (q_n \rightsquigarrow r_n) \rightarrowtail r}{L \mathbin{@} U \;\vDash\; q@u_0 \rightarrowtail (q_1@u_0 \rightsquigarrow r_1@u_1) \rightarrowtail \cdots \rightarrowtail (q_n@u_{n-1} \rightsquigarrow r_n@u_n) \rightarrowtail r@u_n} \tag{1}$$

Here, the value $u_0 \in U$ is initially received from the environment as part of the incoming question. $L \mathbin{@} U$ then mirrors the execution of L but keeps track of this additional state component. The state is attached to any outgoing question in A and updated when the corresponding answer is received. When L terminates, the final value of the state is returned with the answer in B.

Example 2.6. Following up on Example 2.5, This allows us to state $\Sigma_{\mathsf{bq}} \mathbin{@} \mathsf{mem} \equiv \mathsf{Clight(bq.c)}$ and to compose $(\Sigma_{\mathsf{bq}} \mathbin{@} D_{\mathsf{rb}}) \odot L_{\mathsf{rb}} : \top \twoheadrightarrow C \mathbin{@} D_{\mathsf{rb}}$. Note however that the different data representations used by the various components remain an obstacle when handling other parts of the proof.

, ,

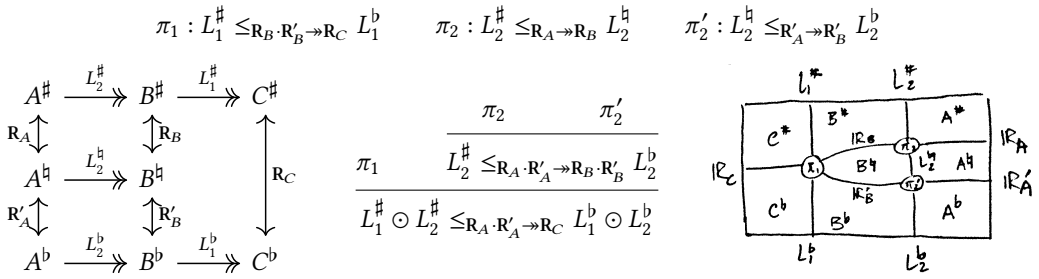


Fig. 3. A composite simulation proof and associated graphical representations

2.4 Simulations

CompCert establishes compiler correctness as a simulation property between closed semantics

$$\mathsf{CompCert}(M) = M' \quad \Rightarrow \quad \mathsf{Clight}[M] \le \mathsf{Asm}[M'] .$$

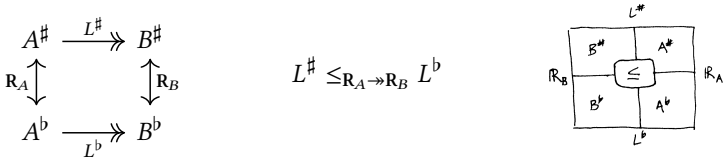
This means that when M compiles to M', the behavior of the source program M is simulated by the behavior of M'. The transitivity of simulations enables *vertical compositionality*: the overall correctness of the compiler can be derived from that of individual compilation passes.

In CompCertO, correctness is formulated at the level of individual components. When M is compiled to M', we must establish a relationship between the open transition systems

$$\mathsf{Clight}(M) : C \ @ \ \mathsf{mem} \twoheadrightarrow C \ @ \ \mathsf{mem} \qquad \text{and} \qquad \mathsf{Asm}(M') : \mathcal{A} \ @ \ \mathsf{mem} \twoheadrightarrow \mathcal{A} \ @ \ \mathsf{mem} .$$

This raises the question of the relationship between the source-level interactions in C @ mem and their corresponding target-level interactions in $\mathcal{A}$ @ mem. In other words, compositional compiler correctness only makes sense with respect to a particular calling convention.

Simulation Conventions. CompCertO makes this explicit: simulations operate in the context of specified *simulation conventions*. This introduces a form of two-dimensional typing. To establish a simulation of a transition system $L^\sharp : A^\sharp \twoheadrightarrow B^\sharp$ by a transition system $L^\flat : A^\flat \twoheadrightarrow B^\flat$, we must first specify a simulation convention $\mathbf{R}_B : B^\sharp \leftrightarrow B^\flat$ for their incoming calls, and a simulation convention $\mathbf{R}_A : A^\sharp \leftrightarrow A^\flat$ for their outgoing calls. We can depict this situation in the following ways:



Simulation conventions are relational in nature. Given $\mathbf{R} : A \leftrightarrow B$ and $\mathbf{R}' : B \leftrightarrow C$, we can derive their sequential composition $\mathbf{R} \cdot \mathbf{R}' : A \leftrightarrow C$. We will write $\epsilon_A : A \leftrightarrow A$ for the identity simulation convention, and call $\mathbf{SC}$ the associated category of language interfaces and simulation conventions.

Composition Properties. As illustrated in Fig. 3, when their boundaries are compatible, simulation squares of the kind depicted above compose both vertically and horizontally. Vertical composition produces a simulation with composite incoming and outgoing simulation conventions. Horizontal composition produces a simulation between composite source and target components.

As suggested in Koenig and Shao [9], given our use of $\odot$ as horizontal composition, the composition properties outlined above equip language interfaces, transition systems and simulation conventions with the structure of a double category. Compared with CompCertO, we also make

simulation conventions more expressive by allowing them to retain a form of *state*, so that they can relate successive interactions in different ways. Finally, based on the @ construction introduced in §2.3, we add one more dimension to the compositional structure.

2.5 Monoidal Structures

We introduced in §2.3 the constructions $A \mathbin{@} U$ and $L \mathbin{@} U$ which extend a language interface A and a transition system L with an additional state component taken in the set U. This construction can be decomposed further and extended to simulation conventions.

Definition 2.7 (Composite language interfaces). Given two language interfaces A and B, the language interface $A \otimes B$ is defined as: $A \otimes B := \langle A^\circ \times B^\circ,\ A^\bullet \times B^\bullet \rangle$. The language interface $\mathbf{I} = \langle \mathbb{1}, \mathbb{1} \rangle$ is a unit for $\otimes$. In addition, for a set U we define the language interface $[U] := \langle U, U \rangle$.

We can recover $A \mathbin{@} U := A \otimes [U]$. Note also that $\otimes$ is associative and commutative, and that:

$$[U \times V] = [U] \otimes [V] \qquad [\mathbb{1}] = \mathbf{I} \qquad [\varnothing] = \top$$

Moreover, these structures are mirrored at the level of simulation conventions.

Simulation Conventions. Two simulation conventions $\mathbf{R} : A^\sharp \leftrightarrow A^\flat$ and $\mathbf{S} : B^\sharp \leftrightarrow B^\flat$ can be combined into $\mathbf{R} \otimes \mathbf{S} : A^\sharp \otimes B^\sharp \leftrightarrow A^\flat \otimes B^\flat$. This simulation convention (Def. 4.4) requires the A and B components of questions and answers to be related independently by $\mathbf{R}$ and $\mathbf{S}$. In addition, a relation $R \subseteq U^\sharp \times U^\flat$ can be promoted to a simulation convention $[R] : [U^\sharp] \leftrightarrow [U^\flat]$ which uses R as the underlying relation for both questions and answers.

Example 2.8 (Correctness of bq.c*).* Building on Example 2.6, consider the refinement between

$$L_{\mathrm{bq}} : \top \twoheadrightarrow C @ D_{\mathrm{bq}} \qquad \text{and} \qquad (\Sigma_{\mathrm{bq}} \mathbin{@} D_{\mathrm{rb}}) \odot L_{\mathrm{rb}} : \top \twoheadrightarrow C @ D_{\mathrm{rb}}\,.$$

To establish a simulation between them, we use the abstraction relation $R \subseteq D_{\mathrm{bq}} \times D_{\mathrm{rb}}$ defined by

$$\begin{aligned} \vec{q}\ R\ (b, c_1, c_2) \ \Leftrightarrow\ & (c_1 \le c_2 < N \ \wedge\ \vec{q} = b_{c_1} \cdots b_{c_2 - 1}) \vee \\ & (c_2 \le c_1 < N \ \wedge\ \vec{q} = b_{c_1} \cdots b_{N-1} b_0 \cdots b_{c_2 - 1})\,. \end{aligned}$$

to formulate the correctness property $\pi_{\mathrm{bq}} : L_{\mathrm{bq}} \ \le_{\top \twoheadrightarrow C \otimes [R]}\ (\Sigma_{\mathrm{bq}} \mathbin{@} D_{\mathrm{rb}}) \odot L_{\mathrm{rb}}$.

Categorical Structure. These constructions satisfy many properties which are well-understood in the context of category theory. For example, the properties

$$\epsilon_A \otimes \epsilon_B = \epsilon_{A \otimes B} \qquad \text{and} \qquad (\mathbf{R}_1 \otimes \mathbf{R}_2) \cdot (\mathbf{S}_1 \otimes \mathbf{S}_2) = (\mathbf{R}_1 \cdot \mathbf{S}_1) \otimes (\mathbf{R}_2 \cdot \mathbf{S}_2)\,,$$

and various properties of the invertible simulation conventions:

$$\lambda_A : A \otimes \mathbf{I} \cong A\,, \qquad \alpha_{ABC} : (A \otimes B) \otimes C \cong A \otimes (B \otimes C)\,, \qquad \gamma_{AB} : A \otimes B \cong B \otimes A\,,$$

equip $\mathbf{SC}$ with the structure of a *symmetric monoidal category*. Likewise, the properties

$$[=_U] = \epsilon_{[U]}\,, \qquad [R \cdot S] = [R]\ ;\ [S]\,, \qquad [R \times S] = [R] \otimes [S]$$

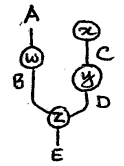
can be captured by describing $[-] : \mathbf{Rel} \to \mathbf{SC}$ as a *monoidal functor* from the symmetric monoidal category $\mathbf{Rel}$ of sets and relations to the symmetric monoidal category $\mathbf{SC}$.

Symmetric monoidal categories capture the algebra of systems or processes which compose both in series and parallel [3]. In the case of simulation conventions, the process is one of concretization from a high-level, abstract representation of component interactions to a more concrete and low-level one. Series composition ($\cdot$) allows us to carry out this process in a stepwise manner, while parallel composition ($\otimes$) allows us to operate independently on various components of questions and answers. This intuition is backed by the formal language of string diagrams.

, ,

String Diagrams. As implied by the properties above, a composite morphism in a symmetric monoidal category can often be written in a variety of equivalent ways. String diagrams provide a more economical representation, where these equivalences are captured by simple geometric intuition. For example, consider the following situation:

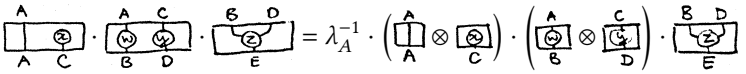
$$\frac{\begin{array}{c} w : A \leftrightarrow B \\ x : \mathbf{I} \leftrightarrow C \\ y : C \leftrightarrow D \\ z : B \otimes D \leftrightarrow E \end{array}}{\mathbf{R} : A \leftrightarrow E} \qquad \begin{aligned} \mathbf{R} &:= \lambda_A^{-1} \cdot (A \otimes x) \cdot (w \otimes y) \cdot z \\ &= \lambda_A^{-1} \cdot (w \otimes (x \cdot y)) \cdot z \\ &= w \cdot \lambda_B^{-1} \cdot (B \otimes (x \cdot y)) \cdot z \\ &\vdots \end{aligned} \qquad \mathbf{R} :=$$



Here, we define a simulation convention **R** from various components using categorical operations. On the left, we show the type of every variable, and give several equivalent definitions for **R**. The string diagram on the right captures the same information. Note that string diagrams are *formal* diagrams which denote a particular morphism with the same rigor as traditional notation.

The string diagrams we use to represent simulation conventions can be read from top to bottom. Vertical lines denote language interfaces, and horizontal juxtaposition represent tensor products. Since it is the unit for $\otimes$, the language interface **I** is not explicitly represented. Nodes connect a group of lines above to a group of lines below and denote elementary simulation conventions, and are connected vertically to denote sequential composition. Like the language interface **I**, the identity simulation convention ϵ is omitted, and may appear as a vertical line without an intervening node. Based on these conventions, the string diagram above can be read as:

$$\mathbf{R} = \quad = \lambda_A^{-1} \cdot \left(\quad \otimes \quad \right) \cdot \left(\quad \otimes \quad \right) \cdot \quad$$



$$= \lambda_A^{-1} \cdot (\epsilon_A \otimes x) \cdot (w \otimes y) \cdot z\,.$$

Transition Systems. Unfortunately, it is not possible in general to form the tensor product of transition systems. To see why, consider a hypothetical product

$$L_1 \otimes L_2 : A_1 \otimes A_2 \twoheadrightarrow B_1 \otimes B_2\,.$$

When a question is received in $B_1 \otimes B_2$, its components in B_1 and B_2 can be used to activate the underlying transition systems L_1 and L_2. However, during their execution, each may ask an arbitrary number of questions in A_1 and A_2. In general, there is no reason to expect that these questions will synchronize or that there will be a way to combine them meaningfully into questions of $A_1 \otimes A_2$.

Although **TS** cannot be made into a symmetric monoidal category, the state threading construction $L@U$ can be generalized to allow the added state component to use different representations in incoming and outgoing questions. Formally, we define a functor $@ : \mathbf{TS} \times \mathbf{Bij} \to \mathbf{TS}$ which from a transition system $L : A \twoheadrightarrow B$ and a bijection $f : U \cong V$ constructs a transition system

$$L @ f \;:\; A \otimes [U] \;\twoheadrightarrow\; B \otimes [V]\,.$$

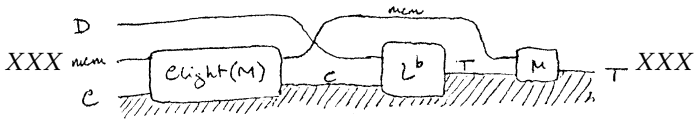
The behavior of this transition system is similar to that of $L@U$, but the bijection f is used to translate the state component between incoming and outgoing questions.

Since the right-hand side argument of $@$ is limited to sets and bijections, it cannot be used to define a proper monoidal structure on **TS**. Nevertheless, there are transition systems

$$\lambda_A^{\mathsf{L}} : \mathbf{I} @ U \cong [U] \qquad \lambda_A^{\mathsf{R}} : A @ \mathbb{1} \cong A \qquad \alpha_{AUV} : A @ (U \times V) \cong (A @ U) @ V$$

which satisfy a version of the expected properties. This makes it possible to construct for transition systems a string diagram language similar to that of simulation conventions.

Example 2.9 (Client code for L_{rb}). To compose Clight(bq.c) with L_{rb}, each must be extended with an additional state component. This can be achieved in the following way:



$$(\mathsf{Clight}(\mathsf{bq.c}) \mathbin{@} D_{\mathsf{rb}}) \odot (C \mathbin{@} \gamma_{\mathsf{mem},D_{\mathsf{rb}}}) \odot (L_{\mathsf{bq}} \mathbin{@} \mathsf{mem}) \odot \mu_{\mathsf{mem}}$$

Composition runs from left to right and the product $C \mathbin{@} \mathsf{mem} \mathbin{@} D_{\mathsf{rb}}$ run from bottom to top. Lines and nodes connecting the shaded area to the rest of the diagram can be arbitrary language interfaces and transition systems, but those appearing in the upper part are restricted to sets and bijections. We have drawn $\gamma_{UV} : U \times V \cong V \times U \in \mathbf{Bij}$ as a twist and it is trivial to define $\mu_U : \top \twoheadrightarrow \top \mathbin{@} U$.

2.6 Memory Separation

The constructions we have outlined so far make it possible to describe the state of complex systems compositionally. State can be separated into different fields; we can then describe and reason about individual components independently of the fields which they do not access, and use the @ operator to connect these components with the rest of the system.

This compositional description facilitates local reasoning, but it must eventually be refined into a concrete implementation: a program acting on a global memory state into which all state components have been merged. To achieve this in a way which preserves compositionality, we use a *partial commutative monoid* over the CompCert memory model. This provides an operation $\bullet$ which can be used to decompose a memory state m into a number of *shares* $m_1 \bullet \cdots \bullet m_n$.

The properties of $\bullet$ and its interaction with memory operations ensure that CompCert semantics satisfy a *frame* property, meaning that they are insensitive to additional memory shares:

$$\frac{L \vDash q@m_0 \rightarrowtail (q_1@m_1 \rightsquigarrow r_1@m_1') \rightarrowtail \cdots \rightarrowtail (q_n@m_n \rightsquigarrow r_n@m_n') \rightarrowtail r@m'}{\begin{array}{l} L \vDash q@(m_0 \bullet w_0) \rightarrowtail \big(q_1@(m_1 \bullet w_0) \rightsquigarrow r_1@(m_1' \bullet w_1)\big) \rightarrowtail \cdots \\ \qquad \cdots \rightarrowtail \big(q_n@(m_n \bullet w_{n-1}) \rightsquigarrow r_n@(m_n' \bullet w_n)\big) \rightarrowtail r@(m' \bullet w_n) \end{array}} \tag{2}$$

The similarity of (2) with the behavior (1) of the transition system $L \mathbin{@} U$ (§2.3) is no coincidence. In fact, using the separation algebra as a relation $\bullet \subseteq (\mathsf{mem} \times \mathsf{mem}) \times \mathsf{mem}$, the frame property for a transition system $L : A \mathbin{@} \mathsf{mem} \twoheadrightarrow B \mathbin{@} \mathsf{mem}$ can be described as $L \mathbin{@} \mathsf{mem} \leq_{A \otimes [\bullet] \twoheadrightarrow B \otimes [\bullet]} L$.

The partial commutative monoid we use is similar in spirit to the *algebraic memory model* of Gu et al. [6]; its construction is explained in Appendix A.

2.7 ClightP

To further facilitate the implementation of abstract state, we introduce the language

$$\mathsf{ClightP}(M) : C \mathbin{@} \mathsf{mem} \twoheadrightarrow C \mathbin{@} \mathsf{mem} \mathbin{@} \mathsf{penv},$$

a variant of Clight where global variables can be declared *private*. Private variables cannot be accessed from other translation units and are stored in a separate *private environment* $p \in \mathsf{penv}$. The program M can easily be compiled to a regular Clight program $M' := \mathsf{ClightUnP}(M)$ by erasing the private annotations from all variables. We will see in §5 that the techniques presented in §2.6 play a key role in formulating the corresponding correctness property for this transformation.

Example 2.10 (Verifying rb.c*).* Using ClightP, we can declare the variables `c1`, `c2` and `buf` as private. We can then define a relation $R_{\mathsf{rb}} \subseteq D_{\mathsf{rb}} \times \mathsf{penv}$ in order to verify

$$(C \mathbin{@} \gamma) \odot (L_{\mathsf{rb}} \mathbin{@} \mathsf{mem}) \leq_{\varnothing \twoheadrightarrow C@\mathsf{mem}@R_{\mathsf{rb}}} \mathsf{ClightP}(\mathsf{rb.c}).$$

, ,

2.8 Encapsulated State

In the semantic model of CompCertO, state exists in two forms:

- The memory state is global and persistent. It is shared across all components and must be carried along every time a question or answer transfers control between components.
- The states used to define transition systems are local to each component. They can only be observed indirectly, through the component's interaction with the environment.

Since C does not offer objects, closures or similar abstractions, component-local state has a lifetime limited to a particular function activation. In the semantics of CompCert languages, maintaining persistent local state across successive activations of a given component would serve no purpose. Nevertheless, persistent local state is important in many applications.

The constructions we have introduced so far make it possible to manage global state and control interference between components, but do not support true encapsulation. In §6, we introduce a construction $\mathbf{C}^\dagger$ which can be applied to **TS** and **Bij** to obtain a model with state encapsulation.

Encapsulation Primitive. The double category structure and the constructions we have described embed into the extended model. In addition, the extended model supports a component:

$$\langle u \rangle : [U] \to \mathbf{I} \qquad \langle u \rangle := (u \in U \mid \mathsf{id}_U) = \quad \textcircled{u} \text{---} \cup$$

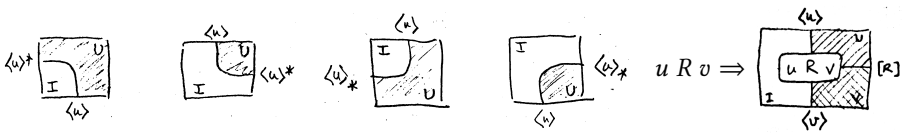
When this component is activated by an incoming question $* \in \mathbf{I}^\circ$, it uses the state $u \in U$ as an outgoing question. When an answer $u' \in U$ is received, it stores u' as the next state and returns control to the environment. This allows $\langle u \rangle$ to act as a state encapsulation primitive.

Example 2.11 (Encapsulated ClightP). We define a version $\mathsf{ClightP}\langle M \rangle : C \mathbin{@} \mathsf{mem} \to C \mathbin{@} \mathsf{mem}$ of the ClightP semantics which encapsulates the private environment, as

$$\mathsf{ClightP}\langle M \rangle := \quad \text{mem } C \text{ — } (p_0) \text{ — } \boxed{\mathsf{ClightP}(M)} \text{ — } \text{mem } C \quad = (C \mathbin{@} \mathsf{mem} \mathbin{@} \langle p_0 \rangle) \odot \mathsf{ClightP}(M)\,,$$

where $p_0 := \mathsf{init\_penv}(M)$ is the initial state of the private environment. Note that the resulting type means that ClightP semantics in this form can be composed directly.

Simulations. The component $\langle u \rangle$ has a *companion* $\langle u \rangle^* : [U] \leftrightarrow \mathbf{I}$ and a *conjoint* $\langle u \rangle_* : \mathbf{I} \leftrightarrow [U]$, which are simulation conventions satisfying the simulation properties:



(3)

As shown above, the simulation conventions $\langle u \rangle^*$ and $\langle u \rangle_*$ make it possible to relate components where different amounts of state have been encapsulated. The last property establishes a form of representation independence for encapsulated state.

3 A DOUBLE CATEGORY OF TRANSITION SYSTEMS

We begin our technical exposition with our extensions to the semantic model of CompCertO. While we reuse CompCertO's notion of transition system as-is, we introduce our own layered composition operator. Moreover, our model makes simulation conventions and simulations more general, so that they can account for history-sensitive relationships between source and target interactions. After providing a formal description of this model, we discuss its relationship with the original.

3.1 Transition Systems

As in the original CompCert semantics, a transition system in CompCertO executes by updating an internal state, which is not directly observable in its interactions with the environment.

Definition 3.1. A *transition system* $L : A \twoheadrightarrow B$ is a tuple $L = \langle S, \to, I, X, Y, F \rangle$ consisting of:

- a set S of states;
- a transition relation $\to \subseteq S \times S$;
- a relation $I \subseteq B^\circ \times S$ which assigns possible *initial states* to each question of B;
- a relation $F \subseteq S \times B^\bullet$ which specifies *final states* together with corresponding answers in B;
- a relation $X \subseteq S \times A^\circ$ which identifies *external states* and corresponding questions of A;
- a relation $Y \subseteq S \times A^\bullet \times S$, which identifies *resumption states*.

We use infix notation for the binary relations I, F, X. We also write $r\ Y^s\ s'$ when $(s, r, s') \in Y$; this means that after an external state s triggers a question in A° and the environment replies with an answer $r \in A^\bullet$, the execution resumes with state s'. In other words, the trace

$$q \rightarrowtail (q_1 \rightsquigarrow r_1) \rightarrowtail \cdots \rightarrowtail (q_n \rightsquigarrow r_n) \rightarrowtail r$$

is generated by a transition sequence

$$q\ I\ s_0 \to^* s_1\ X\ q_1 \rightsquigarrow r_1\ Y^{s_1}\ s_1' \to^* s_2 \cdots s_n\ X\ q_n \rightsquigarrow r_n\ Y^{s_n}\ s_n' \to^* s_f\ F\ r$$

Note that when $L : A \twoheadrightarrow B$ performs an outgoing call in A, the internal state s is preserved until an answer resumes the execution. By contrast, in B no state is preserved across invocations of L. Each question $q \in B^\circ$ initializes a fresh state s such that $q\ I\ s$ regardless of any previous calls.

Definition 3.2 (Transition system composition). The transition system $\mathrm{id}_A : A \twoheadrightarrow A$ is defined as

$$\mathrm{id}_A := \left\langle A^\circ + A^\bullet,\ \varnothing,\ \iota_1,\ \iota_1^{-1},\ \iota_2,\ \iota_2^{-1} \right\rangle.$$

The composition of $L_1 = \langle S_1, \to_1, I_1, X_1, Y_1, T_1 \rangle : B \twoheadrightarrow C$ and $L_2 = \langle S_2, \to_2, I_2, X_2, Y_2, T_2 \rangle : A \twoheadrightarrow B$, is the transition system $L_1 \odot L_2 := \langle S, \to, I, X, Y, F \rangle : A \twoheadrightarrow C$ defined as follows. States are taken in the set $S := S_1 + (S_2 \times S_1)$. When an call in C activates L_1, the left summand is used:

$$\frac{q_C\ I_1\ s_1}{q_C\ I\ \iota_1(s_1)} \qquad \frac{s_1 \to_1 s_1'}{\iota_1(s_1) \to \iota_1(s_1')} \qquad \frac{s_1\ F_1\ r_C}{\iota_1(s_1)\ F\ r_C}$$

When L_1 makes an outgoing call in B, its current state is saved and the question activates L_2. The execution then operates on the state of L_2 until a final state of L_2 is reached and L_1 is resumed:

$$\frac{s_1\ X_1\ q_B \qquad q_B\ I_2\ s_2}{\iota_1(s_1) \to \iota_2(s_2, s_1)} \qquad \frac{s_2 \to_2 s_2'}{\iota_2(s_2, s_1) \to \iota_2(s_2', s_1)} \qquad \frac{s_2\ X_2\ q_A}{\iota_2(s_2, s_1)\ X\ q_A} \qquad \frac{r_A\ Y_2^{s_2}\ s_2'}{r_A\ Y^{\iota_2(s_2, s_1)}\ \iota_2(s_2', s_1)} \qquad \frac{s_2\ F_2\ r_B \qquad r_B\ Y_1^{s_1}\ s_1'}{\iota_2(s_2, s_1) \to \iota_1(s_1')}$$

3.2 Kripke Relators

To formulate our simulation infrastructure, we will rely on the Kripke relator framework used in Koenig and Shao [9], which we summarize below. Relators are operations on relations which mirror operations on types. They are similar to functors on **Rel** but satisfy weaker properties.

Basic Relators for Simulations. Given two relations $R \subseteq A \times B$ and $S \subseteq U \times V$, the relation $(R \to S) \subseteq (A \to U) \times (B \to V)$ is defined in the usual way:

$$f\ [R \to S]\ g \quad :\Leftrightarrow \quad \forall (a, b) \in A \times B\ .\ a\ R\ b \Rightarrow f(a)\ S\ g(b)\ .$$

The more unusual powerset relator $\mathcal{P}^\leq$ is used to express simulation diagrams. Given $R \subseteq A \times B$, the relation $\mathcal{P}^\leq(R) \subseteq \mathcal{P}(A) \times \mathcal{P}(B)$ is defined as:

$$x\ [\mathcal{P}^\leq(R)]\ y \quad :\Leftrightarrow \quad \forall a \in x\ .\ \exists b \in y\ .\ a\ R\ b$$

, ,

To see how this works, consider the simple transition systems $\alpha : A \to \mathcal{P}(A)$ and $\beta : B \to \mathcal{P}(B)$. The relation R is a simulation relation between them when the following property holds:

$$\alpha \ \ [R \to \mathcal{P}^{\leq}(R)] \ \ \beta \qquad\qquad \begin{array}{ccc} a & \xrightarrow{\alpha} & a' \\ {\scriptstyle R}\big| & & \big|{\scriptstyle R} \\ b & \xrightarrow[\beta]{} & b' \end{array}$$

Kripke Relations. Components of complex data structures must often be related in ways which vary depending on the context in which they appear, or the history of the computation. In such cases, component relations can be indexed over a set of *worlds*, which capture the relevant contextual information. Formally, a *Kripke relation* over a set of worlds W is a ternary relation $R \subseteq W \times A \times B$. We will write $R \in \mathcal{R}_W(A, B)$, and use the notation $w \Vdash a \ R \ b$ to mean that $(w, a, b) \in R$. We will also write $\Vdash a \ R \ b$ to mean that a and b are related at all worlds.

It is often useful to let the world evolve by endowing W with an *accessibility* relation $\rightsquigarrow \subseteq W \times W$. World transitions are then captured by the modal relator $\Diamond$, which associates to a Kripke relation $R \in \mathcal{R}_W(A, B)$ the Kripke relation $\Diamond R$ of the same type, defined by:

$$w \Vdash a \ [\Diamond R] \ b \quad :\Leftrightarrow \quad \exists w' \,.\, w \rightsquigarrow w' \wedge w' \Vdash a \ R \ b$$

For example, a Kripke simulation between $\alpha : A \to \mathcal{P}(A)$ and $\beta : B \to \mathcal{P}(B)$ operating in the context of a Kripke frame $\langle W, \rightsquigarrow \rangle$ and a Kripke simulation relation $R \in \mathcal{R}_W(A, B)$ can be formulated in the following way. The relators $\to$ and $\mathcal{P}^{\leq}$ can be promoted to Kripke relators by pointwise extension over the set of worlds. The complex Kripke simulation property:

$$\forall w \in W \,.\, \forall a \in A \,.\, \forall b \in B \,.\, w \Vdash a \ R \ b \Rightarrow \forall a' \in \alpha(a) \,.\, \exists b' \in \beta(b) \,.\, \exists w' \in W \,.\, w \rightsquigarrow w' \wedge w' \Vdash a' \ R \ b'$$

can be stated simply as $\Vdash \alpha \ [R \to \mathcal{P}^{\leq}(\Diamond R)] \ \beta$.

3.3 Simulation Conventions

Simulation conventions characterize the relationship between source- and target-level questions and answers. In CompCertO, every pair of calls is related in isolation, independently of any past or future calls. Our notion of simulation convention is more general, and maintains state across calls.

Remark 3.3 (Motivating stateful simulation conventions). *This will be useful in §6 when we introduce encapsulation. Calls to a specification with encapsulated state may produce traces like:*

$$\mathsf{inc}() \cdot 0 \cdot \mathsf{inc}() \cdot 1 \cdots \tag{4}$$

However, a more concrete version (and eventually, the implementation) may use explicit state:

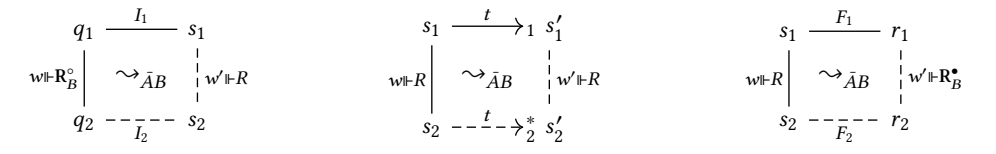
$$\mathsf{inc}()@[\mathsf{c} \mapsto 0] \ \cdot \ 0@[\mathsf{c} \mapsto 1] \ \cdot \ \mathsf{inc}()@[\mathsf{c} \mapsto 1] \ \cdot \ 1@[\mathsf{c} \mapsto 2] \cdots \tag{5}$$

The correspondence between the questions and answers in (4) and those in (5) cannot be formulated on a call-by-call basis but must take into account the history of the computation.

State is maintained using Kripke worlds. In CompCertO's version, Kripke worlds are only used to ensure that questions and answers for a given call are related consistently. We extend the definition to incorporate caller and callee world transitions as well as an initial world.

Definition 3.4. A *simulation convention* $\mathbf{R} = \langle W, \rightsquigarrow, \mapsto, R^\circ, R^\bullet \rangle$ between $A^\sharp$ and $A^\flat$ is specified by:

- a pointed set of worlds W, which store the current state of the simulation convention;
- a *caller* accessibility relation $\mapsto \subseteq W \times W$;
- a *callee* accessibility relation $\rightsquigarrow \subseteq W \times W$;
- a Kripke relation $R^\circ \in \mathcal{R}_W\left(A^\circ_\sharp, A^\circ_\flat\right)$ between the language interfaces' questions, and



(a) Initial states (b) Internal states (c) Final states

Fig. 4. Stateful simulation properties for internal steps

- a Kripke relation $R^\bullet \in \mathcal{R}_W\left(A_\sharp^\bullet,\ B_\flat^\bullet\right)$ between their answers.

The accessibility relations are required to be reflexive and transitive. We write $\mathbf{R} : A^\sharp \leftrightarrow A^\flat$.

Example 3.5. Referring to Remark 3.3, we can formulate a simulation convention whose worlds capture the counter's value. The callee may update it but the caller must leave it unchanged.

$$\mathbf{R} := \langle \mathbb{N}, \top, =, R^\circ, R^\bullet \rangle \qquad \frac{}{n \Vdash \mathsf{inc}()\ R^\circ\ \mathsf{inc}()@[\mathsf{c} \mapsto n]} \qquad \frac{}{n+1 \Vdash n\ R^\bullet\ n@[\mathsf{c} \mapsto n+1]}$$

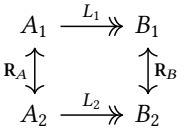
The simulation convention $\mathbf{R}$ defined above relates the sequences (4) and (5).

Definition 3.6 (Composition of Simulation Conventions). The identity simulation convention $\mathsf{id}_A : A \leftrightarrow A$ is given by $\mathsf{id}_A := \langle \mathbb{1}, =_{\mathbb{1}}, =_{\mathbb{1}}, =_{A^\circ}, =_{A^\bullet} \rangle$. The simulation conventions $\mathbf{R}_1 : A^\sharp \leftrightarrow A^\natural$ and $\mathbf{R}_2 : A^\natural \leftrightarrow A^\flat$, compose into $\mathbf{R}_1 ; \mathbf{R}_2 : A^\sharp \leftrightarrow A^\flat$, which is defined in the following way:

$$W := W_1 \times W_2 \qquad R^\circ := R_1^\circ \cdot R_2^\circ \qquad (w_1, w_2) \mapsto (w_1', w_2') :\Leftrightarrow w_1 \mapsto_1 w_1' \wedge w_2 \mapsto_2 w_2'$$
$$R^\bullet := R_1^\bullet \cdot R_2^\bullet \qquad (w_1, w_2) \rightsquigarrow (w_1', w_2') :\Leftrightarrow w_1 \rightsquigarrow_1 w_1' \wedge w_2 \rightsquigarrow_2 w_2' .$$

3.4 Simulations

We are now ready to define our generalized notion of simulation. Consider a simulation of type:



The simulation simultaneously plays the role of the caller with respect to the simulation convention $\mathbf{R}_A$ and the role of the callee with respect to $\mathbf{R}_B$. Hence, it will operate in the context of a Kripke frame constructed from both W_A and W_B. The possible states of a simulation will be a subset $W \subseteq W_A \times W_B$, which must contain the initial state $\underline{w} := (\underline{w}_A, \underline{w}_B) \in W$. Between successive activations, the environment may update the W_B component. Hence we require:

$$(w_A, w_B) \in W \ \wedge\ w_B \mapsto_B w_B' \ \Rightarrow\ (w_A, w_B') \in W$$

When the components execute, the worlds will evolve according to the accessibility relation:

$$(w_A, w_B) \rightsquigarrow_{\bar{A}\bar{B}} (w_A', w_B') :\Leftrightarrow w_A \mapsto_A w_A' \ \wedge\ w_B \rightsquigarrow_B w_B'$$

Reading the constituent transition relations within L_1, L_2 as functions of type:

$$I_1 : B_1^\circ \to \mathcal{P}(S_1) \qquad \to_1 : S_1 \to \mathcal{P}(\mathbb{E}^* \times S_1) \qquad F_1 : S_1 \to \mathcal{P}(B_1^\bullet)$$
$$I_2 : B_2^\circ \to \mathcal{P}(S_2) \qquad \to_2 : S_2 \to \mathcal{P}(\mathbb{E}^* \times S_2) \qquad F_2 : S_2 \to \mathcal{P}(B_2^\bullet),$$

we can formulate the simulation properties for internal steps as shown in Figure 4.

, ,

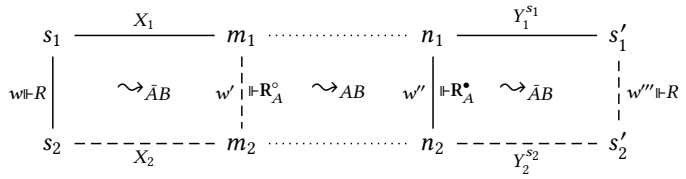


Fig. 5. Simulation property for external states

Conversely, for external calls, the simulation plays the role of the environment. We expect that:

$$(w_A, w_B) \in W \ \wedge \ w_A \rightsquigarrow_A w_A' \ \Rightarrow \ (w_A', w_B) \in W$$

Moreover, from the point of view of the simulation, an external call will make a transition according to the following accessibility relation:

$$(w_A, w_B) \rightsquigarrow_{AB} (w_A', w_B') \ :\Leftrightarrow \ w_A \rightsquigarrow_A w_A' \ \wedge \ w_B = w_B'$$

By reading the action of transition systems at external calls in terms of the functions:

$$Z_1 : S_1 \to \mathcal{P}(A_1^\circ \times (A_1^\bullet \to \mathcal{P}(S_1))) \qquad Z_1(s_1) := \{(q_1, Y_1^{s_1}) \mid s_1 \ X_1 \ q_1\}$$
$$Z_2 : S_2 \to \mathcal{P}(A_2^\circ \times (A_2^\bullet \to \mathcal{P}(S_2))) \qquad Z_2(s_2) := \{(q_2, Y_2^{s_2}) \mid s_2 \ X_2 \ q_2\},$$

we can then formulate the simulation condition for external calls as presented in Figure 5.

Definition 3.7 (Open simulation). There is a simulation of $L_1 : A_1 \twoheadrightarrow B_1$ by $L_2 : A_2 \twoheadrightarrow B_2$ under the simulation conventions $\mathbf{R}_A : A_1 \leftrightarrow A_2$ and $\mathbf{R}_B : B_1 \leftrightarrow B_2$, if there exists

- a set of worlds W, closed under $\rightsquigarrow_A \times \mapsto_B$ and such that $(\underline{w}_A, \underline{w}_B) \in W \subseteq W_A \times W_B$; and
- a Kripke relation $R \in \mathcal{R}_W(S_1, S_2)$ between the states of L_1 and L_2;

satisfying the properties given in Figure 4 and Figure 5. We will write $L_1 \leq_{\mathbf{R}_A \twoheadrightarrow \mathbf{R}_B} L_2$.

3.5 Compositional Structure

The composition of transitions systems and simulation conventions define the respective categories **TS** and **SC**. In addition, simulations compose both horizontally and vertically, namely:

$$\frac{L_1^\sharp \leq_{\mathbf{R}_B \twoheadrightarrow \mathbf{R}_C} L_1^\flat \qquad L_2^\sharp \leq_{\mathbf{R}_A \twoheadrightarrow \mathbf{R}_B} L_2^\flat}{L_1^\sharp \odot L_2^\sharp \leq_{\mathbf{R}_A \twoheadrightarrow \mathbf{R}_C} L_1^\flat \odot L_2^\flat} \qquad \frac{L^\sharp \leq_{\mathbf{R}_A \twoheadrightarrow \mathbf{R}_B} L^\natural \qquad L^\natural \leq_{\mathbf{S}_A \twoheadrightarrow \mathbf{S}_B} L^\flat}{L^\sharp \leq_{\mathbf{R}_A ; \mathbf{S}_A \twoheadrightarrow \mathbf{R}_B ; \mathbf{S}_B} L^\flat}$$

Overall, the compositional structure of our model can be stated concisely in the following way.

Theorem 3.8. *Language interfaces, transition systems, simulation conventions and simulation properties form a thin double category* **TSC**.

This characterization gives a formal underpinning to the usual notions of horizontal and vertical composition found in existing work on compositional certified compilers.

3.6 Relationship with CompCertO

The simulation conventions used in CompCertO constitute a subset of the ones we have defined. In CompCertO, the caller may specify an arbitrary world with each new call. The callee must relate the answers at the same world. Hence, within the context of our definition, CompCertO's simulation conventions take the form $\langle W, =, \top, R^\circ, R^\bullet \rangle$.

When simulation conventions of this form are used, CompCertO's simulations coincide with our own, making them possible to reuse within our framework. In particular, CompCertO's correctness theorem can be reused as-is, and combined with any correctness result obtained for Clight programs.

Moreover, CompCertO defines a notion of simulation convention *refinement*, whereby a simulation convention can replace another in all simulation statments. Having defined a proper categorical structure for **TS**, in our setting it will be possible to encode simulation convention refinement as:

$$\mathbf{R} \sqsubseteq \mathbf{S} \quad :\Leftrightarrow \quad \mathsf{id} \leq_{\mathbf{R}\to\mathbf{S}} \mathsf{id}$$

The interaction of $\sqsubseteq$ with simulations properties is then just an instance of horizontal composition.

Finally, our layered composition operator $\odot$ *under-approximates* CompCertO's semantic linking operator $\oplus$. Since CompCert's syntactic linking of assembly programs is known to implement $\oplus$, this shows that linking is also a correct implementation of the layered composition $\odot$.

Theorem 3.9 (Linking implements layered composition). *For two assembly programs M_1, M_2,*

$$\mathsf{Asm}(M_1) \odot \mathsf{Asm}(M_2) \leq \mathsf{Asm}(M_1) \oplus \mathsf{Asm}(M_2)\,.$$

This means that when a system is compositionally specified and verified at the Clight level, and an overall correctness property is derived in terms of $\odot$, we can combine it with the compiler's correctness theorem to obtain guarantees about the linked assembly program.

4 COMPOSITIONAL STATE

The constructions described in §3 equip CompCertO semantics with the two-dimensional structure of a double category. In this section, we add one more dimension to our compositional infrastructure, spanning different components of a system's state.

4.1 Bijections and Relations

As we have noted in §2.5, transition systems acting on independent components of language interfaces cannot in general be composed directly. However, transition systems can be extended to pass along an additional state component, possibly transformed between their incoming and outgoing interfaces using a bijective function. In fact, bijections and relations form their own double category, which embeds into **TSC**. The associated simulation property is defined as follows.

Definition 4.1. Given two bijective functions $f^\sharp : U^\sharp \to V^\sharp$ and $f^\flat : U^\flat \to V^\flat$, we say that there is a bisimulation of $f^\sharp$ by $f^\flat$ with respect to $R_U \subseteq U^\sharp \times U^\flat$ and $R_V \subseteq V^\sharp \times V^\flat$ when:

$$\forall u^\sharp u^\flat\,.\,\left(u^\sharp\ R_U\ u^\flat \Leftrightarrow f^\sharp(u^\sharp)\ R_V\ f^\flat(u^\flat)\right)$$

We will write $f^\sharp \equiv_{R_U \cong R_V} f^\flat$.

Simulations of this kind compose both horizontally and vertically so that sets, bijective functions and relations, under their usual composition algebras, constitute a thin double category **SBR**. We will see that there is a double functor $[-] : \mathbf{SBR} \hookrightarrow \mathbf{TSC}$ identifying **SBR** as a subcategory of **TSC**.

4.2 Transition Systems

As explained in §2.3, transition systems can be extended to pass along an extra state component. In addition, a bijection can be used to translate this extra state between incoming and outgoing calls.

, ,

Definition 4.2 (Lifting). Given a transition system $L = \langle S, \to, I, X, Y, F \rangle : A \twoheadrightarrow B$ and a bijective function $f : U \to V$, the transition system $L \mathbin{@} f : A \mathbin{@} U \twoheadrightarrow B \mathbin{@} V$ is defined as:

$$L \mathbin{@} f := \langle S \times U, \to_U, I_U, X_U, Y_U, F_U \rangle$$

$$\frac{q\ I\ s \qquad f^{-1}(v) = u}{q@v\ I_U\ s@u} \qquad \frac{s \to s'}{s@u \to_U s'@u} \qquad \frac{s\ X\ m}{s@u\ X_U\ m@u} \qquad \frac{n\ Y^s\ s'}{n@u'\ Y_U^{s@u}\ s'@u'} \qquad \frac{s\ F\ r \qquad f(u) = v}{s@u\ F_U\ r@v}$$

Theorem 4.3. *The operation defined above satisfies defines a functor* $@ : \mathbf{TS} \times \mathbf{Bij} \to \mathbf{TS}$ *since*

$$(L_1 \odot L_2) \mathbin{@} (f \circ g) \equiv (L_1 \mathbin{@} f) \odot (L_2 \mathbin{@} g), \qquad \mathsf{id}_A \mathbin{@} \mathsf{id}_U \equiv \mathsf{id}_{A@U}.$$

This functor can be specialized to $[-] : \mathbf{Bij} \to \mathbf{TS}$ *by defining* $[f] := \mathsf{id}_{\mathbf{I}} \mathbin{@} f$.

4.3 Simulation Conventions

Compared with transition systems, simulation conventions compose more easily. They admit a fine-grained composition structure which follows the constructions introduced in Def. 2.7.

Definition 4.4 (Tensor product of simulation conventions). Given $R_A : A^\sharp \leftrightarrow A^\flat$ and $R_B : B^\sharp \leftrightarrow B^\flat$, we can define the simulation convention $R_A \otimes R_B : A^\sharp \otimes B^\sharp \leftrightarrow A^\flat \otimes B^\flat$ as follows:

$$R_A \otimes R_B := \left\langle W_A \times W_B,\ R_A^\circ \times R_B^\circ,\ R_A^\bullet \times R_B^\bullet \right\rangle$$

In addition, given a relation $R \subseteq U \times V$, the simulation convention $[R] : [U] \leftrightarrow [V]$ is defined as

$$[R] := \langle \mathbb{1}, \top, \top, R, R \rangle.$$

We will also write $\mathbf{R} \mathbin{@} R := \mathbf{R} \otimes [R]$.

Theorem 4.5. *The constructions above constitute* functors $\otimes : \mathbf{SC} \times \mathbf{SC} \to \mathbf{SC}$ *and* $[-] : \mathbf{Rel} \to \mathbf{SC}$. $\mathbf{SC}$ *under* $\otimes$ *is a symmetric monoidal category, and* $[-]$ *is a monoidal functor from* $\langle \mathbf{Rel}, \times \rangle$ *to* $\langle \mathbf{SC}, \otimes \rangle$.

4.4 Simulations

Finally, the compositional structures we have introduced above are compatible with simulations:

$$\frac{L^\sharp \leq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} L^\flat \qquad f \equiv_{R \cong S} g}{L^\sharp \mathbin{@} f \ \leq_{\mathbf{R}@R \to \mathbf{S}@S}\ L^\flat \mathbin{@} g}$$

This can be stated as follows.

Theorem 4.6. *The constructions in this section define a double functor* $@ : \mathbf{TSC} \times \mathbf{SBR} \to \mathbf{TSC}$ *and a double functor* $[-] : \mathbf{SBR} \hookrightarrow \mathbf{TSC}$.

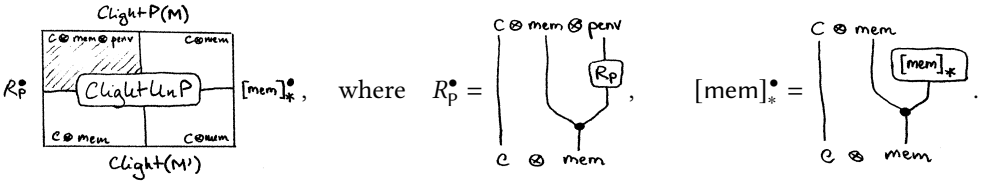
5 CLIGHT WITH MODULE-LOCAL STATE

We have briefly introduced in §2.7 the language ClightP which allows *private* variables to be stored in a separate environment. We now discuss the correctness of the transformation ClightUnP which converts private variables into regular variables stored in the global memory state. The corresponding correctness property is formulated using the techniques outlined in §2.6.

5.1 Correctness Property

Given a simulation convention $\mathbf{R} : A \leftrightarrow \mathsf{mem}$, we write $\mathbf{R}^\bullet := (\mathsf{mem} \otimes \mathbf{R})\ ;\ [\bullet] : \mathsf{mem} \otimes A \leftrightarrow \mathsf{mem}$ for the simulation convention which merges the target memory share of $\mathbf{R}$ into another memory component. In addition, the simulation convention $[U]_* : \mathbf{I} \leftrightarrow U$ allows the caller to use an arbitrary

target question u but requires the callee to return it unchanged. The correctness of ClightUnP can then be stated as:



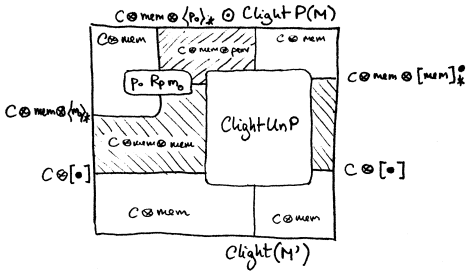
The Clight program stores formerly private variables in the main memory. For incoming calls, the private environment is refined into a memory share according to a relation R_P, and is then merged with the main memory. For outgoing calls, the private environment is not included in the source question, but the variables are still stored in the target memory. The use of $[\mathsf{mem}]^\bullet_*$ requires the callee to return this region of the memory unchanged.

5.2 Encapsulating Private Environments

In order to give the semantics a more tractable type, we can encapsulate the private environment:

$$\mathsf{ClightP}\langle M\rangle := (C \mathbin{@} \mathsf{mem} \mathbin{@} \langle p_0\rangle_*) \odot \mathsf{ClightP}(M) : C \mathbin{@} \mathsf{mem} \twoheadrightarrow C \mathbin{@} \mathsf{mem}$$

Here $p_0 := \mathsf{init\_penv}(M)$ is the initial private environment for M. The facilities discussed in §2.8 can then be used to derive a correctness property stated in terms of the encapsulated semantics:



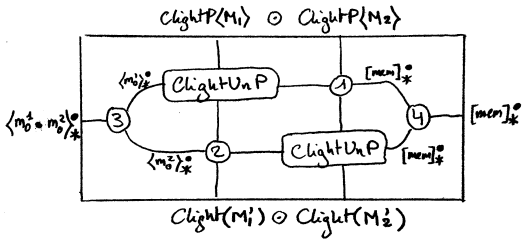
Here, $m_0 := \mathsf{init\_mem}(M'_{\mathsf{priv}})$ is the initial memory fragment associated with the global variable definitions introduced in M' in order to implement the private variables of M. Note that private environments no longer appear in the type of the ClightP semantics or in the correctness property.

5.3 Composition

One challenge is that the correctness property depicted above is not compositional, because the incoming and outgoing simulation conventions are different:

$$\mathsf{ClightP}\langle M\rangle \leq_{C\otimes\langle m_0\rangle^\bullet_* \twoheadrightarrow C\otimes[\mathsf{mem}]^\bullet_*} \mathsf{Clight}(M')$$

Two ClightP modules $\mathsf{ClightP}\langle M_1\rangle \odot \mathsf{ClightP}\langle M_2\rangle$ can be composed, but the associated ClightUnP correctness properties do not directly compose alongside them. We will use the following strategy:



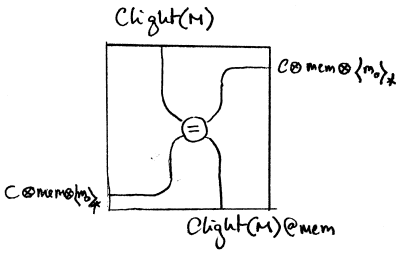
, ,

First, we will establish self-simulation properties for ClightP (1) and Clight (2) which show that they are compatible with the simulation conventions $\langle m \rangle_*^\bullet$ and $[\mathsf{mem}]_*^\bullet$. These properties follow mainly from the frame rule of our memory separation primitives. Second, we show that the incoming (3) and outgoing (4) simulation conventions compose with themselves.

First, the property $\mathsf{Clight}(M) \leq_{\langle m \rangle_*^\bullet \twoheadrightarrow \langle m \rangle_*^\bullet} \mathsf{Clight}(M)$, stems from the fact that

$$L \odot (A \mathbin{@} \langle u \rangle) = L \mathbin{@} \langle u \rangle = (B \mathbin{@} \langle u \rangle) \odot (L \mathbin{@} U) \quad \text{for any} \quad L : A \twoheadrightarrow B\,.$$

This can be combined with the frame rule and properties of $\langle m_0 \rangle_*$ to show:



A similar argument can be used to show the analogous property of ClightP.

Second, the self-composition of the simulation conventions involved follows mostly from properties of the partial commutative monoid.

6 ENCAPSULATED STATE

We now construct the version of our model which allows state components to be encapsulated. The underlying transition systems are unchanged, and encapsulated state is communicated from one activation to the next in the same way global state would be. However, marking state as encapsulated removes it from a component's type, and prevents the environment from accessing it.

6.1 Components

The compositional treatment of state presented in §4 allows a component $L_1 : B \to C \in \mathbf{TS}$ to interact with $L_2 : A \to B \mathbin{@} U$ as the composite $(L_1 \mathbin{@} U) \odot L_2$. This guarantees that L_1 will not interfere with the state in U. Our approach to state encapsulation is to force the environment to interact with components in this manner. In the extended model, a component $L : A \to B \in \mathbf{TS}^\dagger$ is defined using an underlying transition system of type $A \to B \mathbin{@} U$ with an additional state component $U \in \mathbf{Bij}$.

Definition 6.1. Consider a category $\mathbf{C}$ equipped with a functor $\star : \mathbf{C} \times \mathbf{Bij} \to \mathbf{C}$ and a family of isomorphisms $\alpha_{A,U,V} : A \star (U \times V) \cong A \star U \star V$. We define a category $\mathbf{C}^\dagger$ with the same objects, where morphism take the form $(u \in U \mid L) : A \to B \in \mathbf{C}^\dagger$ and consist of:

- a set of states U with a distinguished initial state $u \in U$;
- a morphism $L : A \to B \star U \in \mathbf{C}$ of the underlying category.

Composition in $\mathbf{C}^\dagger$ is defined as:

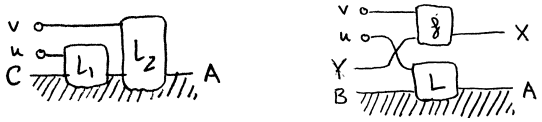
$$(u \in U \mid L_1) \circ (v \in V \mid L_2) := \big((u, v) \in U \times V \mid \alpha^{-1} \circ (L_1 \star V) \circ L_2\big)$$

An embedding of $\mathbf{C}$ into $\mathbf{C}^\dagger$ is defined by $(* \in \mathbb{1} \mid -)$.

The construction above can be applied both to the active components in $\mathbf{TS}$ (with $\star := @$) and the passive components in $\mathbf{Bij}$ (with $\star := \otimes$). The first time a component $(u \in U \mid L) : A \to B \in \mathbf{TS}^\dagger$ is activated by a question $q \in B^\circ$, the transition system L is initialized using $q@u \in (B \mathbin{@} U)^\circ$.

When L terminates with an answer $r@u' \in (B \mathbin{@} U)^\bullet$, the component passes $r \in B^\bullet$ on to the environment and the new state u' is set aside to be used with the next activation.

In addition to composition, the functor $\star : \mathbf{C}\times\mathbf{Bij} \to \mathbf{C}$ can also be extended to $\star : \mathbf{C}^\dagger\times\mathbf{Bij}^\dagger \to \mathbf{C}^\dagger$. This can be shown using string diagrams of the following form:



where the beads on the left-hand side represent the encapsulated state components. This representation stems from the fact that any morphism in $\mathbf{C}^\dagger$ can be defined using the embedded morphisms of $\mathbf{C}$ together with the following state encapsulation primitive:

$$\langle u \in U\rangle : U \to \mathbb{1} \in \mathbf{Bij}^\dagger \qquad\qquad \langle u \in U\rangle := (u \in U \mid \mathrm{id}_U) = \quad u \circ\!\!-$$

6.2 Simulations

We must now extend the double category structure of **TSC** to the encapsulated state model. The same notion of simulation convention can be reused as-is. In addition, the simulation property $L^\sharp \leq_{\mathbf{R}\to\mathbf{S}} L^\flat \in \mathbf{TSC}^\dagger$ can be defined using a simulation in the original model **TSC** between the underlying transition systems. However, to take into account the components' encapsulated states, we must first incorporate those states into the incoming simulation convention **S**. To this end, we will use the elementary simulation conventions $\langle u\rangle^*$ and $\langle u\rangle_*$ mentioned in §2.8.

Definition 6.2. For $u \in U$, the simulation conventions $\langle u\rangle^*$ and $\langle u\rangle^*$ are defined as:

$$\langle u\rangle^* := \left\langle U,\, u,\, \top,\, =,\, \Gamma_U,\, \Gamma_U \right\rangle : U \leftrightarrow \mathbf{I} \qquad\qquad \langle u\rangle^* := \left\langle U,\, u,\, \top,\, =,\, \Gamma_U^{\mathrm{op}},\, \Gamma_U^{\mathrm{op}} \right\rangle : \mathbf{I} \leftrightarrow U$$

where $\Gamma_U \in \mathcal{R}(U, \mathbb{1})$ is the smallest Kripke relation containing $u \Vdash u\ \Gamma_U\ *$ for all $u \in U$.

These elementary simulation conventions internalize a state component of type U in their Kripke worlds, and express the guarantees provided by the environment for encapsulated state: on the first call, the source (resp. target) component must be passed the initial state $u \in U$. It may then modify the state, performing a callee world transition in $\langle u\rangle^*$ (resp. $\langle u\rangle_*$). However, between successive calls, the caller may not update the world. With the ability to express these constraints, we can define our notion of simulation for components with encapsulated states.

Definition 6.3. Simulations of components with encapsulated state are defined by

$$(u \in U \mid L^\sharp) \leq_{\mathbf{R}\to\mathbf{S}} (v \in V \mid L^\flat) \in \mathbf{TSC}^\dagger \quad :\Leftrightarrow \quad L^\sharp \leq_{\mathbf{R}\to\mathbf{S}\otimes(\langle u\rangle^*;\langle v\rangle_*)} L^\flat \in \mathbf{TSC}\,.$$

As in **TSC**, simulations in $\mathbf{TSC}^\dagger$ compose horizontally and vertically.

Theorem 6.4. *Language interfaces, encapsulated transition systems, simulation conventions and simulations in the sense of Def. 6.3 constitute a double category* $\mathbf{TSC}^\dagger$. *There is an embedding double functor* $\mathbf{TSC} \hookrightarrow \mathbf{TSC}^\dagger$ *which preserves the compositional structure of* @ *and* $\otimes$.

7 RELATED WORK

7.1 CompCert-based Vertification Frameworks

CompCert [12] is the first certified C compiler offering industrial-grade performance. As a result, there has long been interest in extending the compiler to offer verification frameworks with formal guarantees going all the way to assembly code. For example, VST [1] provides a separation logic for Clight whose soundness proof interfaces with CompCert's correctness proof.

Compositional Semantics. Compositional semantics were introduced by Compositional CompCert [19] in part to support the verification of programs with multiple translation units, potentially written in different languages, but used a fixed language interface and simulation convention. CompCertM [18] refined this approach with a better model of the interaction between C and assembly programs, and more flexibility in simulation conventions, making it possible for example to introduce of module-local state by expressing constraints on the environment. However, neither Compositional CompCert nor CompCertM support data abstraction or state encapsulation of the kinds we have presented, and the general notions of language interface and simulation convention were first introduced in CompCertO. We refer interested readers to Koenig and Shao [9] for a detailed comparison between Compositional CompCert, CompCertM and CompCertO.

Interaction trees [10, 20] provide another framework for compositional semantics formalized in the Coq proof assistant which presents similarities with our own, though their interface with CompCert is less comprehensive.

Certified Abstraction Layers. Data abstraction in a form closer to our own is present in certified abstraction layers (CAL), originally implemented by CompCertX [5] to verify the certified operating CertiKOS. The semantic model of CompCertX can be summarized as

$$\forall D \in \mathbf{Set}\ .\ L^\flat : \top \twoheadrightarrow C \mathbin{@} \mathsf{mem} \mathbin{@} D^\flat \ \vdash\ \mathsf{Clight}_{L^\flat}[M] : \top \twoheadrightarrow C \mathbin{@} \mathsf{mem} \mathbin{@} D^\flat\ .$$

Here $L^\flat$ is an *underlay interface*. The goal is to implement an *overlay* $L^\sharp \leq_{\top \twoheadrightarrow C@R} \mathsf{Clight}_{L^\flat}[M]$, in a way which allows to abstraction relation $R \subseteq (\mathsf{mem} \times L^\sharp) \times (\mathsf{mem} \times L^\flat)$ to refine and augment the abstract data component. Layers verified in this way can then be combined sequentially.

Subsequent work has extended CAL to support concurrency [6]. The original model is tied to the CompCert semantics and further work explores more abstract versions [8, 15]. In particular a version of the @ construction appears in Koenig and Shao [8]. However, these more abstract frameworks have not been implemented.

7.2 Categorical Structures

To our knowledge, the work described in this paper constitutes the first formal and explicit use of double categories in the context of compositional verification, although the approach was initially suggested in Koenig and Shao [9]. String diagrams for double categories, which use for simulation proofs, were developed and shown to be sound in Myers [14]. Monoidal categories and their string diagrams are much more ubiquitous—a good introduction is provided by Baez and Stay [3].

The construction $\mathbf{C}^\dagger$ which we used in §6 bears some resemblance and was inspired by the *free category with feedback* construction [7, 11]. Indeed, traced monoidal categories could provide a basis for encapsulation in a version of our framework supporting reentrancy and mutual recursion, as in interaction trees. The present version based on an elementary encapsulation primitive $\langle u \rangle$ is simpler but less powerful.

8 CONCLUSION

It is argued in Koenig and Shao [9] that the use of compositional semantics and the formalization of abstraction provided by simulation conventions are important steps towards the construction of large-scale systems certified end-to-end. We have sought to take a further step in this direction by incorporating data abstraction and compositional state encapsulation into CompCertO's framework.

We believe in particular that the underlying algebraic structures that we have uncovered in this process provide an elegant structure with applications beyond the present work, and may constitute an important facet of future certified systems engineering work.

REFERENCES

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Symposium on Programming (ESOP 2011)*. Springer, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[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] John Baez and Mike Stay. 2010. Physics, topology, logic and computation: a Rosetta Stone. In *New structures for physics*. Springer, Berlin, Heidelberg, 95–172. https://doi.org/10.1007/978-3-642-12821-9_2

[4] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Zhenjiang Hu (Ed.). Springer, Berlin, Heidelberg, 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

[5] 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

[6] 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

[7] Piergiulio Katis, Nicoletta Sabadini, and Robert FC Walters. 2002. Feedback, trace and fixed-point semantics. *RAIRO-Theoretical Informatics and Applications* 36, 2 (2002), 181–194.

[8] 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

[9] 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*. 1095–1109.

[10] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 234–248.

[11] Elena Di Lavore, Alessandro Gianola, Mario Román, Nicoletta Sabadini, and Paweł Sobociński. 2021. A canonical algebra of open transition systems. In *International Conference on Formal Aspects of Component Software*. Springer, 63–81.

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

[13] 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

[14] David Jaz Myers. 2016. String diagrams for double categories and equipments. *arXiv preprint arXiv:1612.02762* (2016).

[15] Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. 2022. Layered and Object-Based Game Semantics. *Proc. ACM Program. Lang.* 6, POPL, Article 42 (jan 2022), 32 pages. https://doi.org/10.1145/3498703

[16] Uday S. Reddy. 1997. *Global State Considered Unnecessary: Introduction to Object-Based Semantics*. Birkhäuser Boston, Boston, MA, 227–295. https://doi.org/10.1007/978-1-4757-3851-3_9

[17] Zhong Shao. 2010. Certified Software. *Commun. ACM* 53, 12 (Dec. 2010), 56–66. https://doi.org/10.1145/1859204.1859226

[18] 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

[19] 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

[20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119

A MEMORY SEPARATION IN COMPCERT

The constructions we have introduced so far make it possible to manage and encapsulate persistent state in the context of CompCertO, with certified abstraction layers as one key application. The global memory state used in the semantics of CompCert languages can then be regarded as one possible kind of state among others, and replaced in specifications by more abstract data representations.

Unfortunately, because of the monolithic nature of CompCert's memory, abstracting only part of it is challenging and yields complex simulation conventions. In Example **??**, the abstraction relation had to involve not only the whole target memory state, but also the residual source memory state not subject to abstraction, and used complex properties to express their relationships. In other words, instead of focusing on the particular fragment of the memory which we seek to abstract away, we are to forced to characterize the effect of partial abstraction on the context as well, even though the remaining areas of the memory are not meaningfully involved in the task at hand.

In this section, we use techniques from separation logic to address this problem. We propose to equip the CompCert memory model with a structure akin to separation algebra [**?**] and incorporate the resulting construction within the framework of simulation conventions, CompCert Kripke logical relations, and state encapsulation.

A.1 The CompCert Memory Model

In essence, a CompCert memory state assign to each possible memory address $i \in \mathrm{block} \times \mathbb{Z}$:

- a permission level $p \in \mathrm{option\,perm}$;
- a memory value $v \in \mathrm{memval}$.

In addition, a memory state contains a nextblock counter which keeps track of the next block identifier to be allocated. We discuss these various components in more detail below.

A.1.1 Memory Addresses. The CompCert memory is divided in a number of *blocks*. As new blocks are allocated, they are assigned a positive identifier $b \in \mathbb{N}_*$ in sequential order. As mentioned above, the nextblock counter within each memory state keeps track of the smallest unallocated block identifier. When a new block identifier is needed, nextblock is incremented and its previous value is used for the new block.

Memory blocks represent independent address spaces. Within each block, a byte can be addressed using an offset $o \in \mathbb{Z}$. When a new block is allocated, a range of addresses $[lo, hi)$ must be provided; this range determines which addresses within the block are valid. However, rather than storing the range directly within the memory state, the allocation operation uses it to assign initial permissions for each address within the new block.

A.1.2 Permissions. Each memory address within a memory state is assigned a permission level among the following:

$$p \in \mathrm{option\,perm} ::= \bot \mid \mathsf{nonempty} \mid \mathsf{readable} \mid \mathsf{writable} \mid \mathsf{freeable}$$

The permissions are listed above in increasing order, so that for example the permission level writable represents the set of permissions $\{\mathsf{nonempty}, \mathsf{readable}, \mathsf{writable}\}$. Permissions play an important role in the memory separation relation we define.

When a block is first allocated, addresses within the provided range are assigned the permission level freeable; all remaining addresses are assigned empty permissions $\bot$. Further memory operations may then decrease the permission level, but can never increase it. Memory operations which access a particular address will first check that this address has sufficient permissions, and fail if that is not the case.

$$m_1 \bullet m_2 \equiv m \ \wedge \ m_1 \bullet m_2 \equiv m' \ \Rightarrow \ m = m'$$
$$m_1 \bullet m_2 \equiv m \ \wedge \ m_1' \bullet m_2 \equiv m \ \Rightarrow \ m_1 = m_1' \ \text{XXX}$$
$$m_1 \bullet m_2 \equiv m \ \Rightarrow \ m_2 \bullet m_1 \equiv m$$
$$m_1 \bullet m_2 \equiv m_{12} \ \wedge \ m_{12} \bullet m_3 \equiv m \ \Rightarrow \ \exists m_{23} \,.\, m_2 \bullet m_3 \equiv m_{23} \ \wedge \ m_1 \bullet m_{23} \equiv m$$
$$m \bullet \mathsf{empty} \equiv m$$

Fig. 6. Properties of separation algebras in relational form. See also Dockins et al. [4].

A.1.3 Memory Values. Each memory value represents the contents of exactly one byte of memory. It may be stored as a concrete byte, or may be identified as a particular one-byte fragment within a larger, more abstract value (for instance, the third byte of a given pointer).

The exact representation of memory values is not essential to the work discussed in this section. Therefore we will not discuss the specifics further, but refer the interested reader to Leroy et al. [13] for more background on this topic.

A.1.4 Memory Transformations. The compilation passes of CompCert often transform the structure of the memory state: multiple blocks can merged into one; new blocks may be introduced in the target memory and blocks may be dropped from the source memory. To express these transformations, CompCert introduces *memory extensions* and *memory injections* as possible relations between source- and target-level memory states.

In CompCertO, these memory transformations are generalized and consolidated into a notion of *CompCert Kripke Logical Relations* (CLKRs), which play an important role in defining simulation conventions. The underlying idea is that if two memory states are related by a CKLR, then memory operations which succeed at the source level should also succeed on at the target level, and their outcomes should in turn be related by the CKLR.

Unfortunately, these memory transformations are difficult to use to express the relationships between different *fragments* of a single memory state. The notion of *separation relation* introduced below seeks to fill this gap.

A.2 Separation Relations

To express memory separation in CompCert, and define a *join* relation $J \subseteq (\mathsf{mem} \times \mathsf{mem}) \times \mathsf{mem}$. We will write $J(m_1, m_2, m)$ as:

$$m_1 \bullet m_2 \equiv m \,,$$

understood to mean that the memory states m_1 and m_2 can be merged into m. This relation satisfies the properties listed in Fig. 6 and defines a separation algebra in the sense of Dockins et al. [4].

In addition to these structural properties, the join relation must be compatible with CompCert's memory operations. If an operation which reads from the memory succeeds on a fragment, it should succeed with the same result on a larger memory state:

$$\frac{\mathsf{op}(m_1) = \mathsf{Some}\, v \qquad m_1 \bullet m_2 \equiv m}{\mathsf{op}(m) = \mathsf{Some}\, v}$$

Likewise, operations which updates the memory should be insensitive to additional fragments:

$$\frac{\mathsf{op}(m_1) = \mathsf{Some}\, m_1' \qquad m_1 \bullet m_2 \equiv m}{\exists m' \,.\, m_1' \bullet m_2 \equiv m' \wedge \mathsf{op}(m) = \mathsf{Some}\, m'}$$

, ,

$\boxed{J_{\text{contents}}}$

$(p, v) \in \text{option perm} \times \text{memval}$

$(\bot, \text{undef}) \bullet (p, v) \equiv (p, v)$

$(p, v) \bullet (\bot, \text{undef}) \equiv (p, v)$

(a) Memory contents

$\boxed{J_{\text{nextblock}}}$

$(nb, a) \in \text{block} \times \text{bool}$

$$\frac{\max(nb_1, nb_2) = nb \qquad \neg(a_1 \wedge a_2)}{(nb_1, a_1) \bullet (nb_2, a_2) \equiv (nb,\ a_1 \vee a_2)}$$

(b) Fresh blocks

Fig. 7. Basic ingredients for separation algebras of the CompCert memory model.

Together, these properties allow us to derive versions of the *frame rule* for CompCert languages: if a program can successfully execute on m_1 alone to yield a new memory fragment m_1', then executing it on a larger memory state $m_1 \bullet m_2$ will succeed as well, and yield a memory state $m_1' \bullet m_2$ where the irrelevant portion m_2 has not been modified.

Moreover, executions which affect disjoint parts of the memory can be considered independently. Specifically, from the rules above we can derive the property:

$$\frac{\text{op}_1(m_1) = \text{Some}\, m_1' \qquad \text{op}_2(m_2) = \text{Some}\, m_2' \qquad m_1 \bullet m_2 \equiv m}{\exists m' \,.\, \text{op}_1(\text{op}_2(m)) = m' \ \wedge\ m_1' \bullet m_2' \equiv m'}$$

As in separation logic, this facilitates reasoning about program components which affect the memory state in independent ways.

Below we explain how a separation relation can be defined for the CompCert memory model.

A.2.1 Memory Contents. A CompCert memory state essentially defines a map of type

$$\text{ptr} \longrightarrow \text{option perm} \times \text{memval}\,,$$

which assigns to every possible address a permission level and a memory value. Figure 7 shows the definition of a simple separation relation for the contents of individual memory cells. This relation can then be extended to the whole map in the obvious way.

A.2.2 Block Validity. A more challenging issue is the treatment of nextblock. When a memory state m is separated into $m_1 \bullet m_2 \equiv m$, the fragments m_1 and m_2 will share a common view of the address space. However, they each carry their own copy of the nextblock counter. As a result, performing independent allocations in each fragment will break the separation property, because the new blocks will be assigned conflicting names.

As a starting point, we solve this problem by making sure that new blocks can only be allocated in one of the fragments. In addition to the nextblock counter, memory states carry a boolean flag indicating whether allocations are permitted. When memory fragments are joined, this flag can only be set in one of the fragments. Figure 7b shows the corresponding separation algebra for the nextblock counter.

A.3 Frame rule

The compatibility of memory operations with our separation algebras can be used to show that more complex ways to manipulate memory states enjoy similar properties. Ultimately this allows us to derive a kind of *frame rule* for the Clight semantics. We can state this informally as follows:

$$\frac{\text{Clight}(p) : m_1 \rightsquigarrow m_2}{\text{Clight}(p) : m_1 \bullet m \rightsquigarrow m_2 \bullet m}$$

In other words, if the program p safely acts on a memory state m_1 to transform it into a memory state m_2, then we can frame a memory fragment m onto m_1 and expect the program to leave that fragment intact. Intuitively, this holds because if p ever needed or affected any of the memory present in fragment m, it would have gone wrong on m_1 alone.

To formalize this property in the context of CompCertO, we can promote the memory separation relation to a simulation convention:

$$\forall A\,. \quad A@\bullet : A@(\mathsf{mem} \times \mathsf{mem}) \leftrightarrow A@\mathsf{mem}$$

We will then compare the "source"-level semantics

$$\mathsf{Clight}(p)@\mathsf{mem} : C@(\mathsf{mem} \times \mathsf{mem}) \twoheadrightarrow C@(\mathsf{mem} \times \mathsf{mem})\,,$$

which acts on one of the memory fragments but leaves the other one unchanged, to the concrete semantics of p acting on the total memory state:

$$\mathsf{Clight}(p) : C@\mathsf{mem} \leftrightarrow C@\mathsf{mem}\,.$$

This yields the following property.

Lemma A.1 (Frame rule for Clight).

$$\mathsf{Clight}(p)@\mathsf{mem} \leq_{C@\bullet \twoheadrightarrow C@\bullet} \mathsf{Clight}(p)$$