Unifying Compositional Verification and Certified Compilation with a Three-Dimensional Refinement Algebra

YU ZHANG, Yale University, USA
JÉRÉMIE KOENIG, Yale University, USA
ZHONG SHAO, Yale University, USA
YUTING WANG, Shanghai Jiao Tong University, China

Formal verification is a gold standard for building reliable computer systems. *Certified* systems in particular come with a formal specification, and a proof of correctness which can easily be checked by a third party.

Unfortunately, verifying large-scale, heterogeneous systems remains out of reach of current techniques. Addressing this challenge will require the use of compositional methods capable of accommodating and interfacing a range of program verification and certified compilation techniques. In principle, compositional semantics could play a role in enabling this kind of flexibility, but in practice existing tools tend to rely on simple and specialized operational models which are difficult to interface with one another.

To tackle this issue, we present a compositional semantics framework which can accommodate a broad range of verification techniques. Its core is a three-dimensional algebra of refinement which operates across program modules, levels of abstraction, and components of the system's state. Our framework is mechanized in the Coq proof assistant and we showcase its capabilities with multiple use cases.

CCS Concepts: • **Software and its engineering** → *Software verification*; **Correctness**; *Compilers*; • **Theory of computation** → **Program semantics**; *Abstraction*; *Program verification*; *Program specifications*.

Additional Key Words and Phrases: Compositional Verification, Compositional Compiler Correctness, Game Semantics, Refinement Convention

ACM Reference Format:
Yu Zhang, Jérémie Koenig, Zhong Shao, and Yuting Wang. 2025. Unifying Compositional Verification and Certified Compilation with a Three-Dimensional Refinement Algebra. *Proc. ACM Program. Lang.* 9, POPL, Article 64 (January 2025), 31 pages. https://doi.org/10.1145/3704900

1 Introduction

Programming language semantics make formal verification possible by providing a mathematical account of program execution. In particular, *operational* semantics are often used as a trusted "ground truth" of program behavior, because they closely mirrors the mechanical process of computation.

However, reasoning about programs directly in terms of their operational semantics is often difficult because traditional operational semantics act on a global state. To reason about a given program, we must examine for every possible program step its effect on every component of the state. Without additional structure, this can rapidly become intractable.

Authors' Contact Information: Yu Zhang, Yale University, Department of Computer Science, New Haven, USA, yu.zhang.yz862@yale.edu; Jérémie Koenig, Yale University, Department of Computer Science, New Haven, USA, jeremie.koenig@yale.edu; Zhong Shao, Yale University, Department of Computer Science, New Haven, USA, zhong.shao@yale.edu; Yuting Wang, John Hopcroft Center for Computer Science, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China, yuting.wang@sjtu.edu.cn.

ACM 2475-1421/2025/1-ART64
https://doi.org/10.1145/3704900

secret.s

```
.globl main
main:   pushl $13
        pushl $msg
        call rot13
        pushl $1
        call write
        addl $12, %esp
        movl $0, %eax
        ret
.data
msg:    .string "hello,␣world!\n"
```

```
$ cc -o secret secret.s rot13.c
$ ./secret
uryyb, jbeyq!
$ cc -o decode decode.c rot13.c
$ ./secret | ./decode
hello, world!
```

rot13.c

```c
void rot13(char *buf, int len)
{
  for (int i = 0; i < len; i++)
    if ('a' <= buf[i] && buf[i] <= 'z')
      buf[i] = (buf[i] - 'a' + 13) % 26 + 'a';
}
```

decode.c

```c
#include <unistd.h>
extern void rot13(char *, int);
int main()
{
        char buf[100];
        int n = read(0, buf, sizeof buf);
        rot13(buf, n);
        write(1, buf, n);
        return 0;
}
```

Fig. 1. Two programs which use a common library are compiled and made to interact through a pipe.

Fortunately, many compositional proof techniques have been developed which break down proofs into localized obligations. For example, program logics can be used to establish correctness against Hoare-style specifications compositionally. Modern logics can deal with complex memory layouts, concurrency, and sophisticated language features while supporting a high degree of automation. This has allowed practitioners to verify increasingly complex algorithms and data structures.

1.1 The Program Logic Paradigm Misses Crucial Aspects of Software Development

Despite its success, the traditional approach to verification discussed above does not account for all aspects of the software development process, nor does it fully describe the operation of a typical software artifact. Concerns outside the scope of a typical program logic include the following:

- To be executed, verified program components must first be compiled and linked, and this *compilation* process may compromise any correctness results obtained at the source level.
- Operational semantics and program logics are typically designed for a single language, but many programs are built from components written in *several different languages.*
- Programs such as network servers and clients are algorithmically simple but conduct complex *external interactions*, which program logics rarely model or take into account.

The following example illustrates some of these limitations.

Example 1.1. The code shown in Figure 1 consists of two different programs which use a common C library and are designed to work together. As illustrated in the usage scenario we have shown, the 32-bit x86 assembly program secret.s outputs a coded message to be deciphered by decode.c. In particular, the programs together satisfy the following informal specification:

Suppose that, after compilation, secret.s *and* decode.c *are each linked with* rot13.c. *If the output of the first program is fed as input to the second, "hello, world!" will be displayed.* (1)

The programs are simple; to verify that property (1) holds, a reader with the right background can mentally execute the code step by step and convince themselves that the expected outcome will occur. However, this task is complex in its own way because it mobilizes implicit knowledge and assumptions regarding the C and x86 assembly languages, the compiler's correctness with respect to the calling convention in use, and some aspects of the Unix execution environment. Likewise, any formal account of property (1) must involve these aspects of the problem as well, encompassing all three of the challenges outlined at the beginning of this section. To our knowledge, there exists no program logic or verification framework which can deal with this example.

A fair amount of work has sought to address the limitations outlined above. For example, the certified compiler CompCert [Leroy 2009] comes with a mechanized proof of correctness. Better yet, the Verified Software Toolchain (VST) [Appel 2011] provides a separation logic which interfaces with the correctness proof of CompCert, ensuring that properties obtained for C programs can be formally transferred to the compiled assembly code. In a further experiment, a network server was verfied by incorporating *interaction trees* into VST to model external interactions [Koh et al. 2019]. Operational semantics [Matthews and Findler 2007] and program logics [Guéneau et al. 2023] have also been developed for multi-language programs. Another line of work uses the Bedrock2 framework to perform *integration verification*. For example, Erbsen et al. [2024] presents the end-to-end verification of a minimalistic but sophisticated embedded system, which mediates access to an external actuator (the opening mechanism for a miniature garage door replica) through cryptographically-authenticated network commands. The top-level correctness statement asserts that a certain model of the complete system satisfies certain constraints on its observable behavior.

These efforts show that overcoming the limitations of the program logic paradigm is possible, but they constitute one-off adaptations to specific settings: a particular specification logic, set of interaction patterns, combination of languages, *etc.* To apply this methodology to Example 1.1, we would need to develop a semantics and logic tailored to the situation at hand. The result would be unlikely to apply directly to another verification task.

By contrast, we envision a situation where future certified systems architects will build complex systems by assembling off-the-shelf certified components, and obtain end-to-end proofs of correctness with little additional effort. The experiments mentioned above represent important progress toward this vision; we seek to build on these successes to deepen our understanding of the underlying principles and distill them into new mathematical tools, which can then serve as a foundation for the next round of ground-breaking challenges and research.

1.2 Compositional Semantics Offer a More Flexible Approach

To be sure, there exist mechanized semantics, program logics and certified compilers which can deal with the C and assembly code used in Example 1.1. However, in our situation, the main story is not what each program *as such* is doing. The programs are part of a larger context where they are interpreted as interacting *processes* and used as building blocks in a larger system. At that level of abstraction, the function calls and shared memory states which a typical program logic deals with are no longer in the picture, having been replaced with forms of inter-process communication.

Indeed, the main difficulty with Example 1.1 is that formalizing property (1) requires adjustments to the *model* within which we consider the behavior of the programs secret.s and decode.c. This is difficult to achieve in many frameworks based on operational semantics because they use a fixed model, of a closed universe, relying on compositional *proof techniques*. As illustrated in Fig. 2, compositional *semantics* can be used to improve this state of affairs. By their nature, compositional semantics focus on the way open components interact with each other. As a result, they are more likely to be suitable building blocks for modeling complex, heterogeneous systems.

Recent work embracing this paradigm shows promising results. For example, whereas prior CompCert research largely focuses on compositional proof techniques, the work on CompCertO

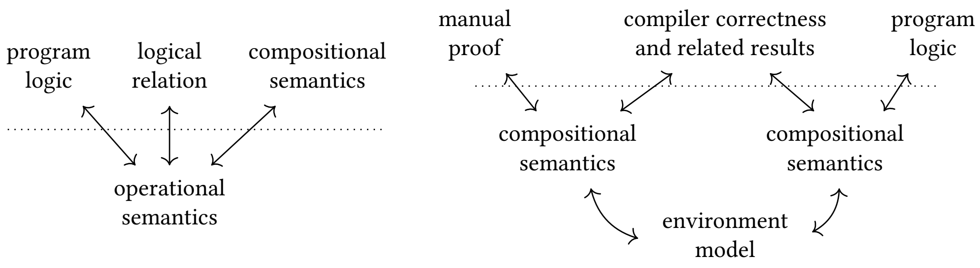


Fig. 2. Approaches to program verification. The system being verified is modeled using the facilities shown below the line, and the techniques shown above are used to reason about its properties. Traditionally (left), the whole universe in which the computation occurs must be modeled in a monolithic and closed operational semantics. By using compositional semantics instead (right), both the model and reasoning techniques can be constructed out of reusable building blocks and adapted to various contexts and situations.

[Koenig and Shao 2021] shows that formulating the compiler's correctness result directly in terms of a compositional semantics is possible with a reasonable proof effort. Likewise, the DimSum framework [Sammler et al. 2023] successfully employs this approach to tackle multi-language semantics and verification: the framework can be used to stitch together independent semantics for individual languages, and to reason about refinement within and across these languages.

At the same time, compositional semantics remains underdeveloped as a practical tool for verification, and lack a proper treatments of many techniques which are routine in the context of operational semantics and program logics.

1.3 Three Dimensions of Compositionality

We will distinguish between several *kinds* of compositionality which semantic models, program logics, refinement frameworks and other formal reasoning tools can exhibit:

- *Horizontal compositionality* refers to the ability to decompose behaviors and proofs along the structure of program. For example, denotational semantics are compositional in this sense. Likewise, the *sequence* rule of Hoare logic is a horizontal composition principle.
- *Vertical compositionality* allows the kind of stepwise reasoning afforded by transitive refinement and data abstraction mechanisms. Compiler correctness proofs make use of vertical compositionality when they combine correctness proofs for individual compilation phases.
- *Spatial compositionality* operates across the system state. This is the kind of compositionality enabled in separation logic by the separating conjunction $*$ and the associated *frame* rule.

One barrier to the use of semantics along the lines of Fig. 2b is that while *horizontally* compositional semantics are a well-developed area of research, there is comparatively less work investigating models which are vertically and spatially compositional, let alone the combination of all three.

1.4 Contributions

We seek to bridge this gap by introducing a generic semantic model—based on effect signatures and formulated in the style of game semantics—which combines horizontal, vertical and spatial composition principles. Our model is mechanized in the Coq proof assistant [Zhang et al. 2024a], and is flexible enough to express the CompCertO semantics of C and assembly programs, and to describe the kind of process interactions required to handle Example 1.1.

$$\begin{tikzcd} E_1 \arrow[r, "L_1"] \arrow[d, leftrightarrow, "\mathbf{R}"'] \arrow[dr, phantom, "\phi"] & F_1 \arrow[d, leftrightarrow, "\mathbf{S}"] \\ E_2 \arrow[r, "L_2"'] & F_2 \end{tikzcd}$$

$$\frac{L_1 : F \twoheadrightarrow G \qquad L_2 : E \twoheadrightarrow F}{L_1 \odot L_2 : E \twoheadrightarrow G}\ \text{ts-}\odot \qquad \frac{\phi : L_1 \leq_{\mathbf{S}\twoheadrightarrow\mathbf{T}} L_1' \qquad \psi : L_2 \leq_{\mathbf{R}\twoheadrightarrow\mathbf{S}} L_2'}{\phi \odot \psi : L_1 \odot L_2 \leq_{\mathbf{R}\twoheadrightarrow\mathbf{T}} L_1' \odot L_2'}\ \text{sim-}\odot$$

$$\frac{\mathbf{R} : E_1 \leftrightarrow E_2 \qquad \mathbf{R}' : E_2 \leftrightarrow E_3}{\mathbf{R} \fatsemi \mathbf{R}' : E_1 \leftrightarrow E_3}\ \text{sc-}\fatsemi \qquad \frac{\phi : L_1 \leq_{\mathbf{R}\twoheadrightarrow\mathbf{S}} L_2 \qquad \psi : L_2 \leq_{\mathbf{R}'\twoheadrightarrow\mathbf{S}'} L_3}{\phi \fatsemi \psi : L_1 \leq_{\mathbf{R}\fatsemi\mathbf{R}'\twoheadrightarrow\mathbf{S}\fatsemi\mathbf{S}'} L_3}\ \text{sim-}\fatsemi$$

Fig. 3. Horizontal ($\odot$) and vertical ($\fatsemi$) composition principles in our model.

The multiple dimensions of compositionality allow us to account for sophisticated reasoning techniques such as data abstraction and memory separation, and to capture—under a uniform notion of refinement—properties as varied as program correctness results, the semantics preservation theorem of CompCertO, the frame property of separation logic, and representation independence for encapsulated state. These properties can then be combined to construct sophisticated refinement proofs of statements such as (1).

We present a high-level overview in §2 and provide a formal description of the model in §3–4. Spatial compositionality is treated separately in §5. We then showcase in §6 several applications:

- We explain in detail how CompCertO semantics and simulation proofs can be embedded, and model the *loading* mechanism which turns an open program into a closed process;
- We use our model to define a framework for *certified abstraction layers* (CAL) [Gu et al. 2015]. Unlike the original work on CAL, our layer framework does not modify the underlying compiler, and its meta-theory requires comparatively negligible effort.
- We define an extension of CompCert's Clight language supporting encapsulated, module-local private variables and provide a correctness proof for the erasure of private annotations.

We discuss related work in §7 and our conclusions in §8.

2 Compositional Semantics for Verification

Our framework consists of four kinds of objects, each subject to some or all of four different composition principles (layered $\odot$, vertical $\fatsemi$, flat $\oplus$, spatial @). We start with a brief overview of how these constructions fit together, then examine each one in more detail.

2.1 Overview

Our model is built around the notion of *effect signature* ($E, F \ldots$). We use these signatures to describe the interfaces between the components of a software system. Effect signatures serve as horizontal endpoints for *strategies* and vertical endpoints for *refinement conventions.* Strategies ($L : E \twoheadrightarrow F$) describe the behaviors of program components. Refinement conventions ($\mathbf{R} : E_1 \leftrightarrow E_2$) model relationships between views of the system at different levels of abstraction. Finally, *refinement proofs* ($\phi : L_1 \leq_{\mathbf{R}\to\mathbf{S}} L_2$) connect the three kinds of objects above in the shape of a square (Fig. 3).

Composition Principles. Our framework uses refinement squares as the building blocks of compositional proofs. They are assembled in the manner of puzzle pieces alongside matching edges:

- Layered composition ($\odot$) acts horizontally. It connects strategies at a common endpoint (*i.e.* effect signature) over which they are made to interact, and connects refinement squares alongside a common vertical edge (*i.e.* refinement convention), which ensures that the refinement properties are based on compatible abstractions and constraints.
- Vertical composition ($\fatsemi$) combines successive refinement steps, connecting refinement conventions alongside an intermediate signature, and refinement squares along a common strategy, which serves as an intermediate specification.

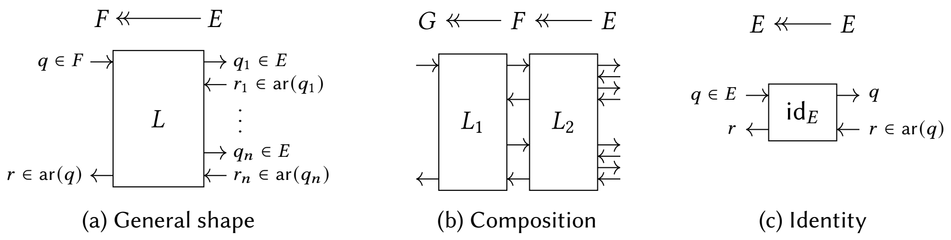


Fig. 4. Informal description of our strategy model under *layered* composition

This basic framework is made more expressive by the introduction of two additional forms of composition, which coherently act on all objects from effect signatures to refinement squares:

- Flat composition ($\oplus$) serves as an alternative form of horizontal composition where components are laid out side by side instead of being made to interact.
- Spatial composition (@) is the core of our infrastructure for compositional state.

2.2 Effect Signatures

Like interaction trees [Koh et al. 2019], our model uses *effect signatures* to describe interfaces between the components of a system. An effect signature enumerates the external operations which a component can invoke or implement, and describes for each one the set of possible outcomes.

Definition 2.1. An *effect signature* is a set E of *questions* together with an assignment $\mathsf{ar} : E \longrightarrow \mathbf{Set}$ associating to each question $m \in E$ a set of *answers* $n \in \mathsf{ar}(m)$. We will often present them together as the set of bindings $\{(m : N) \mid m \in E \wedge N = \mathsf{ar}(m)\}$.

Example 2.2. Consider the execution environment for the programs secret and encode shown in Figure 1 and described in Example 1.1. Since our programs do not use any command-line arguments or environment variables, we can model their invocation with a single question:

$$\mathcal{P} := \{\mathsf{run} : \mathbb{N}\}\,.$$

The answer $x \in \mathbb{N}$ is the exit status of the process. Moreover, in the course of its execution each process can invoke the read and write system calls. We can describe this interface with the signature

$$\mathcal{S} := \{\mathsf{read}_i[n] : \Sigma^*,\ \mathsf{write}_i[s] : \mathbb{N} \ \mid\ i \in \mathbb{N},\ n \in \mathbb{N},\ s \in \Sigma^*\}\,,$$

where $\Sigma := \{0, 1\}^8$ is the alphabet of possible byte values. In this formalism, the program secret will invoke the operation write_1`["uryyb, jbeyq!\n"]` $\in \mathcal{S}$, where $i := 1$ is the file descriptor associated with the standard output; the outcome should be $14 \in \mathbb{N}$ if the operation succeeds.

Example 2.3 (CompCertO language interfaces). The semantic model of CompCertO uses *language interfaces* of the form $A = \langle A^\circ, A^\bullet \rangle$ as the basis for component interactions. These interfaces are similar to effect signatures, but every question $q \in A^\circ$ uses the same set of answers $r \in A^\bullet$.

For the C language, questions are function calls of the form $f(\vec{v})@m$, where f identifies the function to be called, $\vec{v} \in \mathsf{val}^*$ are the actual parameters, and $m \in \mathsf{mem}$ is the memory state at the time of invocation; answers take the form $v'@m'$ where $v' \in \mathsf{val}$ is the value returned by the function f and $m' \in \mathsf{mem}$ is the new state of the memory. This is captured by the effect signature

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

We will see that CompCertO language interfaces can be systematically mapped to effect signatures, and will elucidate the structure of the spatial decomposition $C @ \mathsf{mem}$ below.

2.3 Strategies

We use effect signatures to assign coarse types to program components and to their behaviors. We use a game semantics approach where a *strategy* $L : E \twoheadrightarrow F$ models the behavior of a component which uses operations of the signature E to implement the operations enumerated in F. The strategy can specify actions taken by the component in response to the possible actions of the environment, and is represented as a set of traces.

As depicted in Fig. 4a, the environment can activate L by asking a question $q \in F$, which the component L is expected to eventually answer with a reply $r \in \mathsf{ar}(q)$. In the process, L can perform an arbitrary number of queries $q_i \in E$, which the environment answers with a response $r_i \in \mathsf{ar}(q_i)$. The process can then begin anew with a question $q' \in F$, and so on indefinitely. We will write

$$L \vDash \big(q \rightarrowtail (m_1 \rightsquigarrow n_1) \rightarrowtail (m_2 \rightsquigarrow n_2) \rightarrowtail \cdots \rightarrowtail (m_k \rightsquigarrow n_k) \rightarrowtail r\big) \rightsquigarrow \big(q' \rightarrowtail \cdots \rightarrowtail r'\big) \rightsquigarrow \cdots$$

to mean that L accepts an execution trace of this kind. Note that $\rightarrowtail$ denotes a part of the execution where L is in control, whereas $\rightsquigarrow$ denotes a part of the execution controlled by the environment.

Example 2.4 (Command specifications). We can use strategies $\Gamma_{\text{secret}}, \Gamma_{\text{decode}} : \mathcal{S} \twoheadrightarrow \mathcal{P}$ to formulate specifications for the commands secret and decode. The processes admit the execution traces

$$\Gamma_{\text{secret}} \vDash \mathsf{run} \rightarrowtail (\mathsf{write}_1[\texttt{"uryyb, jbeyq!\textbackslash n"}] \rightsquigarrow 14) \rightarrowtail 0$$

$$\Gamma_{\text{decode}} \vDash \mathsf{run} \rightarrowtail (\mathsf{read}_0[100] \rightsquigarrow \texttt{"uryyb, jbeyq!\textbackslash n"}) \rightarrowtail (\mathsf{write}_1[\texttt{"hello, world!\textbackslash n"}] \rightsquigarrow 14) \rightarrowtail 0\,.$$

Example 2.5 (CompCertO semantics). We explained in Example 2.3 that CompCertO language interfaces can be translated to effect signatures, and described the signature C @ mem used for C-level function calls and returns. By the same token, CompCertO language semantics can be translated to strategies as well. For example, the source language used by CompCertO is a simplified version of C called Clight, and its semantics for a program M can be used to define:

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

The resulting strategies will exhibit traces such as:

$$\begin{aligned}\mathsf{Clight}(\mathsf{decode.c}) \vDash \mathsf{main}()@m &\rightarrowtail (\mathsf{read}(0,b,100)@m[b \mapsto \textit{unspecified}] \rightsquigarrow 14@m[b \mapsto \texttt{"uryyb, jbeyq!\textbackslash n"}])\\ &\rightarrowtail (\mathsf{rot13}(b,14)@m[b \mapsto \texttt{"uryyb, jbeyq!\textbackslash n"}] \rightsquigarrow *@m[b \mapsto \texttt{"hello, world!\textbackslash n"}])\\ &\rightarrowtail (\mathsf{write}(1,b,14)@m[b \mapsto \texttt{"hello, world!\textbackslash n"}] \rightsquigarrow 14@m[b \mapsto \texttt{"hello, world!\textbackslash n"}])\\ &\rightarrowtail 0@m[b \mapsto \textit{deallocated}]\end{aligned}$$

This trace is more complicated than the one shown in Example 2.4; among other things it involves low-level considerations regarding the C memory model. Nevertheless, we will eventually use the CompCertO semantics of C and assembly as a building block to model the scenario in Example 1.1, and connect them to the kind of high-level specifications we have seen so far.

Refinement Ordering. In our model, components can exhibit undefined behavior ($\bot$) and spaces of strategies are equipped with a refinement ordering $\leq$. Refinement means that a strategy L_2 is *more defined* than L_1, and admits at least the same behaviors and desirable properties. We will write

$$\rho : L_1 \leq_{E \twoheadrightarrow F} L_2\,, \qquad \text{or} \qquad \rho : L_1 \leq L_2\,, \qquad \text{or just} \quad L_1 \leq L_2$$

when such a refinement holds, with ρ understood as an elementary *refinement proof term*. While we take a proof irrelevant approach and will treat refinement proofs of the same type as equal, using explicit proof terms will allow us to construct formal refinement proofs from more elementary properties, as a special case of our model's compositional structure.

Example 2.6. We wish to show that the program decode satisfies the specification Σ_{decode} given in Example 2.4. This requires modeling the way in which decode.c and rot13.c behave together *as a process.* Assuming that $[\![\mathsf{decode}]\!] : \mathcal{S} \twoheadrightarrow \mathcal{P}$, models the response of the combined program to

```c
static int c1, c2;
static V buf[N];

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
extern int inc1(void);
extern int inc2(void);
extern V get(int i);
extern void set(int i, V val);

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

(b) The translation unit bq.c

Fig. 5. Running example, adapted from Koenig and Shao [2020]. The component rb.c implements a ring buffer of capacity N by encapsulating an array and two counters. It is used by bq.c to implement a bounded queue.

the trigger run $\in \mathcal{P}$ in terms of system calls performed over the interface $\mathcal{S}$, our goal will be to establish a refinement $\Sigma_{\mathsf{decode}} \leq [\![\mathsf{decode}]\!]$. The model $[\![-]\!]$ will involve CompCertO semantics and take into account the way the program is compiled, linked and loaded.

2.4 Layered Composition

When a component $L_1 : F \twoheadrightarrow G$ uses an interface F implemented by another component $L_2 : E \twoheadrightarrow F$, we can direct the questions asked by L_1 in F to L_2 (Fig. 4b). The result is the composite strategy $L_1 \odot L_2 : E \twoheadrightarrow G$. This is the main form of horizontal composition which we will be using.

Example 2.7 (Verifying a bounded queue). The code shown in Fig. 5 implements a bounded queue with at most N values of type V. This is done in two steps. The translation unit rb.c provides access to a ring buffer in the form of an array as well as two counters which wrap around to stay in the interval $[0, N)$. The translation unit bq.c then uses that interface to implement the queue. We can describe this situation using high-level specifications with the following types:

$$\begin{aligned} \Gamma_{\mathsf{rb}} &: 0 \twoheadrightarrow E_{\mathsf{rb}}, \quad && \text{where } E_{\mathsf{rb}} := \{\mathsf{inc}_1 : \mathbb{N},\ \mathsf{inc}_2 : \mathbb{N},\ \mathsf{get}[i] : V,\ \mathsf{set}[i,v] : \mathbb{1} \mid i \in \mathbb{N},\ v \in V\} \\ \Sigma_{\mathsf{bq}} &: E_{\mathsf{rb}} \twoheadrightarrow E_{\mathsf{bq}}, \quad && \text{where } E_{\mathsf{bq}} := \{\mathsf{enq}[v] : \mathbb{1},\ \mathsf{deq} : V \mid v \in V\}, \end{aligned}$$

and where 0 is the empty signature. The specifications Γ_{rb} and Σ_{bq} admit interaction traces such as:

$$\begin{aligned} \Gamma_{\mathsf{rb}} \vDash{} & (\mathsf{inc}_2 \rightarrowtail 0) \rightsquigarrow (\mathsf{set}[0,v] \rightarrowtail *) \rightsquigarrow \\ & (\mathsf{inc}_2 \rightarrowtail 1) \rightsquigarrow (\mathsf{set}[1,v'] \rightarrowtail *) \rightsquigarrow \\ & (\mathsf{inc}_1 \rightarrowtail 0) \rightsquigarrow (\mathsf{get}[0] \rightarrowtail v) \end{aligned} \qquad \begin{aligned} \Sigma_{\mathsf{bq}} \vDash{} & (\mathsf{enq}[v] \rightarrowtail (\mathsf{inc}_2 \rightsquigarrow 0) \rightarrowtail (\mathsf{set}[0,v] \rightsquigarrow *) \rightarrowtail *) \rightsquigarrow \\ & (\mathsf{enq}[v'] \rightarrowtail (\mathsf{inc}_2 \rightsquigarrow 1) \rightarrowtail (\mathsf{set}[1,v'] \rightsquigarrow *) \rightarrowtail *) \rightsquigarrow \\ & (\mathsf{deq} \rightarrowtail (\mathsf{inc}_1 \rightsquigarrow 0) \rightarrowtail (\mathsf{get}[0] \rightsquigarrow v) \rightarrowtail v) \end{aligned}$$

Layered composition allows us to compute their behavior as they interact over E_{rb}. The resulting strategy $\Sigma_{\mathsf{bq}} \odot \Gamma_{\mathsf{rb}} : 0 \to E_{\mathsf{bq}}$ will admit traces like the following one:

$$\Sigma_{\mathsf{bq}} \odot \Gamma_{\mathsf{rb}} \ \vDash\ (\mathsf{enq}[v] \rightarrowtail *) \rightsquigarrow (\mathsf{enq}[v'] \rightarrowtail *) \rightsquigarrow (\mathsf{deq} \rightarrowtail v)\,.$$

2.5 Data Abstraction and Vertical Composition

The functionality implemented in Example 2.7 can be described at different levels of abstraction. The user may rely on a specification $\Gamma_{\mathsf{bq}} : 0 \twoheadrightarrow E_{\mathsf{bq}}$ defined in terms of a queue state $\vec{q} \in V^*$. However, the refinement $\Sigma_{\mathsf{bq}} \odot \Gamma_{\mathsf{rb}} : 0 \twoheadrightarrow E_{\mathsf{bq}}$ might use a buffer state $(c_1, c_2, \vec{b}) \in \mathbb{N} \times \mathbb{N} \times V^N$ more closely related to the in-memory representation used by the actual code.

Data abstraction techniques can be used to connect these two views. For example, simulation relations are a simple form of data abstraction which express the relationship between state representations used in two different transition systems. If Γ_{bq} and $\Sigma_{\mathsf{bq}} \odot \Gamma_{\mathsf{rb}}$ were defined as transition systems, a simulation relation $\rho_{\mathsf{bq}} \subseteq V^* \times (\mathbb{N} \times \mathbb{N} \times V^N)$ could spell out the correspondence between high- and low-level views. Since the interface E_{bq} reveals no details about internal state of either component, this would be enough to prove a refinement.

The situation is more complicated when a component's *interactions* change across levels of abstraction. For example, consider the code of decode.c in Fig. 1. Seen as a *process*, the program is invoked with the question run $\in \mathcal{P}$ and relies on system calls such as $\text{read}_i[n] \in \mathcal{S}$. However, at a lower level of abstraction, the action run will take the form of a call to the main function in the context of a carefully prepared initial memory state, and likewise calls to read and write will take the form of C-level calls into the C standard library.

Refinement Conventions. To address this challenge, we adapt to the setting of effect signatures and game semantics the notion of *simulation convention* used in CompCertO. A simulation convention connects the ways an interface is viewed at different levels of abstraction. In CompCertO, the compiler's correctness theorem involves a simulation convention $\mathbb{C} : \mathcal{C} \leftrightarrow \mathcal{A}$, which is used to express the way in which C-level function calls ($\mathcal{C}$) are encoded as assembly-level interactions ($\mathcal{A}$).

We build on this idea and define a richer notion of *refinement convention* between effect signatures. Then a refinement property $\phi : L_1 \leq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} L_2$ between $L_1 : E_1 \twoheadrightarrow F_1$ and $L_2 : E_2 \twoheadrightarrow F_2$ is parameterized by two simulation conventions $\mathbf{R} : E_1 \leftrightarrow E_2$ and $\mathbf{S} : F_1 \leftrightarrow F_2$. The corresponding refinement property assumes that incoming source- and target-level questions in F_1 and F_2 will be related according to the convention $\mathbf{S}$, and guarantees that outgoing questions in E_1 and E_2 will be related according to $\mathbf{R}$. Conversely, it assumes that the environment's answers in E will be related according to $\mathbf{R}$ and guarantees that the components' answers in F will be related according to $\mathbf{S}$.

Example 2.8 (Semantics preservation of CompCert). We will see in §6 that the correctness proof of CompCertO can be put in the form of a refinement square:

$$\mathsf{CompCert}(p) = p' \quad \Longrightarrow \quad \phi_p^{\text{cc}} : \mathsf{Clight}(p) \leq_{\mathbb{C} \twoheadrightarrow \mathbb{C}} \mathsf{Asm}(p')$$

where $\mathbb{C} : \mathcal{C} \leftrightarrow \mathcal{A}$ captures the calling convention used to represent C calls at the level of assembly.

2.6 Combining Effect Signatures

We now introduce a composition operation $\oplus$ operating on the effect signatures themselves, which will act on all higher-dimensional objects as well.

Definition 2.9 (Sum of signatures). A family $(E_i)_{i \in I}$ of effect signatures can be combined into

$$\bigoplus_{i \in I} E_i := \{\iota_i(m) : N \mid i \in I,\ (m : N) \in E_i\},$$

which uses the set of operations $\iota_i(m) \in \sum_i E_i$ and uses for each one the arity assigned to it in its signature of origin E_i. The binary case where $i \in \{1, 2\}$ will be written as $E_1 \oplus E_2$.

The signature $E \oplus F$ contains the combined questions of E and F. Each question retains the same set of answers. Many of the signatures we have seen can be decomposed using $\oplus$.

Example 2.10 (Per-file interfaces). We have seen that processes can be modeled as strategies of type $P : \mathcal{S} \twoheadrightarrow \mathcal{P}$, where the signature $\mathcal{S}$ contains questions for each file descriptor $i \in \mathbb{N}$. We can decompose this signature as $\mathcal{S} = \bigoplus_{i \in \mathbb{N}} \mathcal{F}$, where $\mathcal{F} := \{\text{read}[n] : \Sigma^*, \text{write}[s] : \mathbb{N} \mid n \in \mathbb{N}, s \in \Sigma^*\}$. Since our examples focus on standard input ($i = 0$) and output ($i = 1$), we will simplify $\mathcal{S} := \mathcal{F} \oplus \mathcal{F}$.

The compositional properties of $\oplus$ are summarized in Fig. 6 and discussed below. The strategy $L_1 \oplus L_2 : E_1 \oplus E_2 \to F_1 \oplus F_2$ is straightforward and lets L_1 and L_2 operate independently. When a question $q \in F_1$ is asked in the left-hand side component of $F_1 \oplus F_2$, it is used to activate L_1 which

$$\frac{L_1 : E_1 \twoheadrightarrow F_1 \qquad L_2 : E_2 \twoheadrightarrow F_2}{L_1 \oplus L_2 : E_1 \oplus E_2 \twoheadrightarrow F_1 \oplus F_2}\ \textsf{ts-}\oplus \qquad\qquad \frac{\mathbf{R} : E_1 \leftrightarrow F_1 \qquad \mathbf{S} : E_2 \leftrightarrow F_2}{\mathbf{R} \oplus \mathbf{S} : E_1 \oplus E_2 \leftrightarrow F_1 \oplus F_2}\ \textsf{sc-}\oplus$$

$$(L_1 \odot L_2) \oplus (L_1' \odot L_2') \equiv (L_1 \oplus L_1') \odot (L_2 \oplus L_2') \qquad (\mathbf{R}_1 \mathbin{⨾} \mathbf{R}_2) \oplus (\mathbf{S}_1 \mathbin{⨾} \mathbf{S}_2) \equiv (\mathbf{R}_1 \oplus \mathbf{S}_1) \mathbin{⨾} (\mathbf{R}_2 \oplus \mathbf{S}_2)$$

$$\mathsf{id}_E \oplus \mathsf{id}_F \equiv \mathsf{id}_{E\oplus F} \qquad \mathbf{id}_E \oplus \mathbf{id}_F \equiv \mathbf{id}_{E\oplus F}$$

$$\frac{\phi : L_1 \leq_{\mathbf{R}_1 \twoheadrightarrow \mathbf{S}_1} L_1' \qquad \psi : L_2 \leq_{\mathbf{R}_2 \twoheadrightarrow \mathbf{S}_2} L_2'}{\phi \oplus \psi : L_1 \oplus L_2 \leq_{\mathbf{R}_1 \oplus \mathbf{R}_2 \twoheadrightarrow \mathbf{S}_1 \oplus \mathbf{S}_2} L_1' \oplus L_2'}\ \textsf{sim-}\oplus$$

Fig. 6. Signature composition ($\oplus$) for strategies, refinement conventions and simulation proofs.

executes until the question is answered. L_2 handles the questions of F_2 in a similar way. Additional strategies can be defined in relation to $\oplus$, namely

$$\Delta_E : E \twoheadrightarrow E \oplus E, \qquad \gamma_{E,F} : E \oplus F \cong F \oplus E, \qquad \pi_1^{E,F} : E \oplus F \twoheadrightarrow E, \qquad \pi_2^{E,F} : E \oplus F \twoheadrightarrow F\,.$$

The strategy Δ_E passes along questions received in two independent copies of E but consolidates them into a single copy. The projections $\pi_i^{E,F}$ can be used to "forget" the unused summand of the signature $E \oplus F$. These constructions are illustrated in the following example.

Example 2.11 (Composing processes). We can define shell-like operators for composing processes. Two processes $P, Q : \mathcal{S} \twoheadrightarrow \mathcal{P}$ can be combined into $(P \,;\, Q) : \mathcal{S} \twoheadrightarrow \mathcal{P}$. To this end, we define the scheduling component $\mathsf{seq} : \mathcal{P} \oplus \mathcal{P} \twoheadrightarrow \mathcal{P}$ which invokes one process, then the other:

$$\mathsf{seq} \vDash \mathsf{run} \cdot (\mathsf{run}_1 \cdot n) \cdot (\mathsf{run}_2 \cdot m) \cdot m$$

This component can be used to define:

$$P \,;\, Q \ :=\ \mathsf{seq} \odot (P \oplus Q) \odot (\mathcal{F} \oplus \gamma \oplus \mathcal{F}) \odot (\Delta \oplus \Delta)$$

We could likewise model the shell operators && and || by replacing seq with different scheduling policies. In addition, we can use a component $\mathsf{fifo} : 0 \longrightarrow \mathcal{F}$ with behaviors such as:

$$\mathsf{fifo} \vDash (\mathsf{write}[\texttt{"hello, "}] \rightarrowtail 7) \rightsquigarrow (\mathsf{write}[\texttt{"world!\textbackslash n"}] \rightarrowtail 7) \rightsquigarrow (\mathsf{read}[100] \rightarrowtail \texttt{"hello, world!\textbackslash n"})$$

With fifo to model a buffer, we can define:

$$P \mid Q \ :=\ \mathsf{seq} \odot (P \oplus Q) \odot (\mathcal{F} \oplus (\Delta \odot \mathsf{fifo}) \oplus \mathcal{F})\,.$$

Using this construction, we can express the relationship between the behaviors Γ_{secret} and Γ_{encode} to formulate a partial account of property (1). Specifically, we expect the behavior

$$\Gamma_{(1)} \vDash \mathsf{run} \rightarrowtail (\mathsf{write}_1[\texttt{"hello, world!\textbackslash n"}] \rightsquigarrow 14) \rightarrowtail 0$$

to admit the refinement square $\phi_{(1)} : \Gamma_{(1)} \leq \Gamma_{\text{secret}} \mid \Gamma_{\text{decode}}$.

Note that since our model does not support concurrency, the construction $P \mid Q$ above can only offer a sequential approximation of the corresponding Unix shell operator. We intend the example shown in Figure 1 to illustrate the issues that come up when the horizon of verification is pushed beyond the boundary of a fixed language or model, but providing a realistic account of Unix processes remains beyond the scope of the present work.

Remark 2.12 (Morphisms in Context). Above we rely on the category theory convention by which the same notation is used for a functor's action on objects and morphisms. When functors are combined and specialized, objects and morphisms may appear together in certain expressions. For example, applying the functor $U \times - + V : \mathbf{Set} \to \mathbf{Set}$ to a function $f : X \to Y$ yields

$$U \times f + V \;:\; U \times X + V \;\to\; U \times Y + V \qquad \text{(also known as } \mathsf{id}_U \times f + \mathsf{id}_V\text{)}$$

Seeing id_A as the morphism part of the nullary functor A, another interpretation is that objects can simply denote their identity morphism. In any case, this idea generalizes to higher dimensions. For example, given $L_1 : A \twoheadrightarrow B$, $L_2 : B \twoheadrightarrow C$, $\mathbf{R} : A \leftrightarrow B$, $\mathbf{S} : B \leftrightarrow C$ and $\phi : L_1 \leq_{\mathbf{R} \twoheadrightarrow B} B$, we can write

$$L_2 \odot \phi \;:\; L_2 \odot L_1 \leq_{\mathbf{R} \twoheadrightarrow C} L_2 \qquad \text{and} \qquad \phi \,;\, \mathbf{S} \;:\; L_1 \leq_{\mathbf{R};\mathbf{S} \twoheadrightarrow \mathbf{S}} C\,.$$

3 Strategy Model

We now turn to the task of formalizing the constructions we have outlined in §2.

3.1 Strategies

We have already informally described many strategies using interaction traces of the form

$$q \rightarrowtail (m_1 \rightsquigarrow n_1) \rightarrowtail (m_2 \rightsquigarrow n_2) \rightarrowtail \cdots \rightarrowtail (m_k \rightsquigarrow n_k) \rightarrowtail r \rightsquigarrow q' \rightarrowtail \cdots .$$

We will often write such traces more compactly as $q\underline{m_1}n_1\underline{m_2}n_2 \cdots \underline{m_k}n_k\underline{r}q' \cdots$, where actions of the component have been underlined and alternate with environment actions.

Definition 3.1 (Strategy). Consider an *outgoing* effect signature E and an *incoming* effect signature F. A *play* in the game $E \twoheadrightarrow F$ is an element $s \in P_{E,F}$ in the set generated by the grammar:

$$\frac{q \in F \quad s \in P^q_{E,F}}{qs \in P_{E,F}} \qquad \frac{m \in E \quad s \in P^{qm}_{E,F}}{\underline{m}s \in P^q_{E,F}} \qquad \frac{n \in \mathsf{ar}(m) \quad s \in P^q_{E,F}}{ns \in P^{qm}_{E,F}}$$

$$\frac{}{\epsilon \in P_{E,F}} \qquad \frac{r \in \mathsf{ar}(q) \quad s \in P_{E,F}}{\underline{r}s \in P^q_{E,F}} \qquad \frac{}{\epsilon \in P^{qm}_{E,F}}$$

$$\begin{tikzcd} P_{E,F} \arrow[r, bend left, "q \in F"] & P^q_{E,F} \arrow[l, bend left, "r \in \mathsf{ar}(q)"] \arrow[r, bend left, "m \in E"] & P^{qm}_{E,F} \arrow[l, bend left, "n \in \mathsf{ar}(m)"] \end{tikzcd}$$

and ordered by the prefix relation $\sqsubseteq$. Moreover, the *coherence* relation $\frown \subseteq P_{E,F} \times P_{E,F}$ is defined by:

$$\frac{}{\epsilon \frown s} \quad \frac{}{s \frown \epsilon} \qquad \frac{q_1 = q_2 \Rightarrow s_1 \frown^{q_1} s_2}{q_1 s_1 \frown q_2 s_2} \qquad \frac{s_1 \frown s_2}{\underline{r}s_1 \frown^q \underline{r}s_2} \qquad \frac{s_1 \frown^{qm} s_2}{\underline{m}s_1 \frown^q \underline{m}s_2} \qquad \frac{n_1 = n_2 \Rightarrow s_1 \frown^q s_2}{n_1 s_1 \frown^{qm} n_2 s_2} \qquad \frac{}{\epsilon \frown^{qm} s} \quad \frac{}{s \frown^{qm} \epsilon}$$

Then a *strategy* $\sigma : E \twoheadrightarrow F$ is a prefix-closed subset of $P_{E,F}$ where any two $s_1, s_2 \in \sigma$ satisfy $s_1 \frown s_2$.

Example 3.2. The behavior of a queue with infinite capacity can be modeled as follows. For a starting state $\vec{q} \in V^*$, the strategy $\sigma_{\vec{q}} : 0 \twoheadrightarrow E_{\mathrm{bq}}$ is defined by the following rules:

$$\epsilon \in \sigma_{\vec{q}} \qquad s \in \sigma_{\vec{q}} \;\Rightarrow\; \mathsf{deq} \cdot \underline{v} \cdot s \;\in\; \sigma_{v\vec{q}} \qquad s \in \sigma_{\vec{q}v} \;\Rightarrow\; \mathsf{enq}[v] \cdot \underline{*} \cdot s \;\in\; \sigma_{\vec{q}}$$

Note that as expected, the strategy never performs any outgoing calls but only interacts over E_{bq}. The behavior of a queue which is initially empty is described by $\sigma_\epsilon : 0 \twoheadrightarrow E_{\mathrm{bq}}$.

Unlike $\sigma_{\vec{q}}$ above, many strategies of interest are stateless in the sense that every incoming question is handled in the same way regardless of any previous history.

Definition 3.3 (Regular Strategy). Consider a strategy $\sigma : E \twoheadrightarrow F$. Given two plays $s, t \in P_{E,F}$ the play $s \rhd t$ initially proceeds as s but goes on to proceed as t if s ends with $\epsilon \in P_{E,F}$ when a question $q \in F$ is expected. Formally, we can define $\rhd^x : P^x_{E,F} \times P_{E,F} \to P^x_{E,F}$ as follows:

$$qs \rhd t := q(s \rhd^q t) \qquad \underline{m}s \rhd^q t := \underline{m}(s \rhd^{qm} t) \qquad ns \rhd^{qm} t := n(s \rhd^q t)$$
$$\epsilon \rhd t := t \qquad \underline{r}s \rhd^q t := \underline{r}(s \rhd t) \qquad \epsilon \rhd^{qm} t := \epsilon$$

The *regular closure* $\sigma^* : E \twoheadrightarrow F$ allows the strategy σ to start over with each new incoming question:

$$\epsilon \in \sigma^* \qquad\qquad s \in \sigma \wedge t \in \sigma^* \Rightarrow s \rhd t \in \sigma^*$$

Moreover, a strategy is *single-use* when its plays are of the form $q\underline{m_1}n_1 \cdots \underline{m_k}n_k\underline{r}$ or prefixes thereof. We say that σ is a *regular strategy* when it is the regular closure $\sigma = \tau^*$ of a single-use strategy τ.

In the previous section we described the behavior of various components by writing down execution traces. We can use the constructions above to turn such descriptions into formal strategies.

3.2 Layered Composition

The layered composition of $\sigma : F \twoheadrightarrow G$ with $\tau : E \twoheadrightarrow F$ allows the strategies to synchronize over the signature F. Their interaction over the intermediate signature is then hidden from the composite strategy $\sigma \odot \tau : E \twoheadrightarrow G$. Layered composition can be defined at the level of individual plays.

Definition 3.4 (Layered Composition of Strategies). The *identity* strategy $\mathsf{id}_E : E \twoheadrightarrow E$ is defined as:

$$\mathsf{id}_E := \big(\{\epsilon\} \cup \{m\underline{m} \mid m \in E\} \cup \{m\underline{m}n\underline{n} \mid m \in E, n \in \mathsf{ar}(m)\}\big)^* .$$

In addition, two strategies $\sigma : F \twoheadrightarrow G$ and $\tau : E \twoheadrightarrow F$ compose to yield a strategy $\sigma \odot \tau : E \twoheadrightarrow G$. Individual plays compose according to the relations:

$$\begin{aligned}
\odot &: P_{F,G} \times P_{E,F} \to \mathcal{P}(P_{E,G}) & \epsilon \odot t &:= \{\epsilon\} \\
& & qs \odot t &:= \{qw \mid w \in s \odot^q t\} \\
\odot^q &: P^q_{F,G} \times P_{E,F} \to \mathcal{P}(P^q_{E,G}) & \underline{r}s \odot^q t &:= \{\underline{r}w \mid w \in s \odot t\} \\
& & \underline{m}s \odot^q t &:= \{w \mid \exists t' \cdot t = mt' \wedge w \in s \odot^{qm} t'\} \\
\odot^{qm} &: P^{qm}_{F,G} \times P^{m}_{E,F} \to \mathcal{P}(P^q_{E,G}) & s \odot^{qm} \underline{u}t &:= \{\underline{u}w \mid w \in s \odot^{qmu} t\} \\
& & s \odot^{qm} \underline{n}t &:= \{w \mid \exists s' \cdot s = ns' \wedge w \in s' \odot^q t\} \\
\odot^{qmu} &: P^{qm}_{F,G} \times P^{mu}_{E,F} \to \mathcal{P}(P^{qu}_{E,G}) & s \odot^{qmu} \epsilon &:= \{\epsilon\} \\
& & s \odot^{qmu} vt &:= \{vw \mid w \in s \odot^{qm} t\}
\end{aligned}$$

We can then define the *layered composition* of σ and τ as $\sigma \odot \tau := \bigcup_{(s,t) \in \sigma \times \tau} s \odot t$.

Theorem 3.5. *Layered composition is associative and admits identity strategies as units.*

3.3 Flat Composition

In addition to layered composition, strategies can also be combined side-by-side. Specifically, two strategies $\sigma_1 : E \twoheadrightarrow F_1$ and $\sigma_2 : E \twoheadrightarrow F_2$ can be used to independently handle the two components of an incoming effect signature $F_1 \oplus F_2$.

Definition 3.6 (Flat Composition). The strategy $\pi_i : E_1 \oplus E_2 \twoheadrightarrow E_i$ can be defined as:

$$\pi_i := \big(\{\epsilon\} \cup \{m\,\underline{\iota_i(m)} \mid m \in E_i\} \cup \{m\,\underline{\iota_i(m)}\,n\,\underline{n} \mid m \in E_i, n \in \mathsf{ar}(m)\}\big)^* .$$

Moreover, two strategies $\sigma_1 : E \twoheadrightarrow F_1$ and $\sigma_2 : E \twoheadrightarrow F_2$ can be combined into $\langle \sigma_1, \sigma_2 \rangle : E \twoheadrightarrow F_1 \oplus F_2$. Individual plays combine as follows:

$$\begin{aligned}
\langle qs_1, s_2 \rangle &:= \{\iota_1(q)\,w \mid w \in \langle s_1, s_2 \rangle^q_1\} & \langle s_1, qs_2 \rangle &:= \{\iota_2(q)\,w \mid w \in \langle s_1, s_2 \rangle^q_2\} \\
\langle \underline{r}s_1, s_2 \rangle^q_1 &:= \{\underline{r}w \mid w \in \langle s_1, s_2 \rangle\} & \langle s_1, \underline{r}s_2 \rangle^q_2 &:= \{\underline{r}w \mid w \in \langle s_1, s_2 \rangle\} \\
\langle \underline{m}s_1, s_2 \rangle^q_1 &:= \{\underline{m}w \mid w \in \langle s_1, s_2 \rangle^{qm}_1\} & \langle s_1, \underline{m}s_2 \rangle^q_2 &:= \{\underline{m}w \mid w \in \langle s_1, s_2 \rangle^{qm}_2\} \\
\langle ns_1, s_2 \rangle^{qm}_1 &:= \{nw \mid w \in \langle s_1, s_2 \rangle^q_1\} & \langle s_1, ns_2 \rangle^{qm}_2 &:= \{nw \mid w \in \langle s_1, s_2 \rangle^q_2\}
\end{aligned}$$

Then $\langle \sigma_1, \sigma_2 \rangle := \bigcup_{(s_1,s_2) \in \sigma_1 \times \sigma_2} \langle s_1, s_2 \rangle$. In addition, for $\sigma_1 : E_1 \twoheadrightarrow F_1$ and $\sigma_2 : E_2 \twoheadrightarrow F_2$ we define

$$\sigma_1 \oplus \sigma_2 \ := \ \langle \sigma_1 \odot \pi_1, \, \sigma_2 \odot \pi_2 \rangle \ : \ E_1 \oplus E_2 \twoheadrightarrow F_1 \oplus F_2 \, .$$

Theorem 3.7 (Properties of $\oplus$). *The definitions above satisfy the rules and properties in Fig. 6.*

4 Refinement Conventions

The inclusion order induces a simple notion of strategy refinement. For example, consider the strategies $\sigma \subseteq \tau : E \twoheadrightarrow \{* : \varnothing\}$. Ignoring the initial move $*$, plays of σ and τ take the form $\underline{m_1} n_1 \underline{m_2} n_2 \cdots \underline{m_k}$. Operationally, inclusion induces the following coinductive simulation property, where we write $\underline{m}n \backslash \sigma$ for the residual strategy $\{s \mid \underline{m}ns \in \sigma\}$:

$$\begin{aligned} \sigma \leq \tau :\Leftrightarrow \ & \forall m \cdot \underline{m} \in \sigma \Rightarrow \underline{m} \in \tau \, \land \\ & \forall n \cdot (\underline{m}n \backslash \sigma) \leq (\underline{m}n \backslash \tau) \, . \end{aligned} \qquad \begin{tikzcd} \sigma \arrow[r, no head] \arrow[d, no head, "\leq"'] & m \arrow[r, dotted, no head] \arrow[d, equal] & n \arrow[r] \arrow[d, equal] & (\underline{m}n \backslash \sigma) \arrow[d, dashed, no head, "\leq"] \\ \tau \arrow[r, dashed, no head] & m \arrow[r, dotted, no head] & n \arrow[r, dashed] & (\underline{m}n \backslash \tau) \end{tikzcd} \tag{2}$$

In other words, any behavior prescribed by the specification σ must be mirrored by the refinement τ.

Refinement conventions and refinement squares generalize this notion of refinement to cover situations where the source σ and target τ differ in their interactions with the environment.

4.1 Overview

Building on the example above, suppose $\sigma : E_1 \twoheadrightarrow \{* : \varnothing\}$ and $\tau : E_2 \twoheadrightarrow \{* : \varnothing\}$ now differ in the type of their outgoing interactions. To relate them, we will define a notion of *refinement convention* $\mathbf{R} : E_1 \leftrightarrow E_2$ establishing a correspondence between the questions and answers of E_1 and E_2. A refinement *up to* $\mathbf{R}$, written in this case $\sigma \leq_{\mathbf{R} \twoheadrightarrow \{*:\varnothing\}} \tau$, will correspond to the property

$$\begin{aligned} & \forall m_1 \cdot \underline{m_1} \in \sigma \Rightarrow \exists m_2 \cdot \underline{m_2} \in \tau \ \land \ m_1 \ \mathbf{R}^\circ \ m_2 \ \land \\ & \forall \, n_1 \, n_2 \cdot \ n_1 \ \mathbf{R}^\bullet_{m_1,m_2} \ n_2 \Rightarrow \\ & \quad \big(\underline{m_1} n_1 \backslash \sigma\big) \leq_{\mathbf{R}^{n_1,n_2}_{m_1,m_2} \twoheadrightarrow \{*:\varnothing\}} \big(\underline{m_2} n_2 \backslash \tau\big) \end{aligned} \qquad \begin{tikzcd} \sigma \arrow[r, no head] \arrow[d, no head, "\leq_{\mathbf{R}}"'] & m_1 \arrow[r, dotted, no head] \arrow[d, dashed, no head, "\mathbf{R}^\circ"'] & n_1 \arrow[r] \arrow[d, no head, "\mathbf{R}^\bullet_{m_1 m_2}"'] & \big(\underline{m_1} n_1 \backslash \sigma\big) \arrow[d, dashed, no head, "\leq_{\mathbf{R}^{n_1 n_2}_{m_1 m_2}}"'] \\ \tau \arrow[r, dashed, no head] & m_2 \arrow[r, dotted, no head] & n_2 \arrow[r, dashed] & \big(\underline{m_2} n_2 \backslash \tau\big) \end{tikzcd} \tag{3}$$

Here, the refinement convention provides a relation $\mathbf{R}^\circ \subseteq E \times F$ between the questions of E and the questions of F; furthermore, for related question $m_1 \ \mathbf{R}^\circ \ m_2$ the refinement convention provides a relation on answers $\mathbf{R}^\bullet_{m_1,m_2} \subseteq \mathsf{ar}(m_1) \times \mathsf{ar}(m_2)$ and an updated refinement convention $\mathbf{R}^{n_1 n_2}_{m_1 m_2}$ to be used for the next question whenever the answers $n_1 \ \mathbf{R}^\bullet_{m_1 m_2} \ n_2$ are received.

As this example illustrates, one source of complexity is the *alternating* character of (3). While the *client* is free to choose matching questions m_1 and m_2, it must be ready to accept for every answer n_1 any related n_2 which the handler could return. In other words, the kind of data abstraction realized by refinement conventions involves *demonic* as well as angelic choices. While $\leq_{\mathbf{R}}$ becomes larger when $\mathbf{R}^\circ$ relates more questions, the opposite is true of $\mathbf{R}^\bullet$, which introduces additional constraints. Moreover, since our strategies simultaneously play the roles of a client and a handler on their outgoing and incoming sides, general refinement squares involve two different refinement conventions, again with opposite variances.

4.2 Refinement Conventions

Our construction of refinement conventions is similar in spirit to that of strategies. However, to tackle the challenges outlined above, we must introduce an important technical novelty. Specifically, to handle the alternating angelic and demonic choices which a refinement convention can perform, we must go beyond the usual prefix ordering of plays.

Definition 4.1. Refinement conventions of type $\mathbf{R} : E \leftrightarrow F$ are constructed using plays of the form

$$s \in P_{E \leftrightarrow F} \ ::= \ (m_1, m_2)\bot \ \mid \ (m_1, m_2)(n_1, n_2)\, s \ \mid \ (m_1, m_2)(n_1, n_2)\top \qquad \begin{pmatrix} m_1 \in E, n_1 \in \mathsf{ar}(m_1) \\ m_2 \in F, n_2 \in \mathsf{ar}(m_2) \end{pmatrix}.$$

As suggested by the notation, the plays are ordered by the smallest relation $\preceq$ such that

$$s_1 \preceq s_2 \implies (m_1, m_2)\bot \ \preceq \ (m_1, m_2)(n_1, n_2)s_1 \ \preceq \ (m_1, m_2)(n_1, n_2)s_2 \ \preceq \ (m_1, m_2)(n_1, n_2)\top\,.$$

Then refinement conventions are elements of

$$S_{E \leftrightarrow F} \ := \ \mathcal{D}\big(P_{E \leftrightarrow F}, \preceq\big) \ = \ \{\mathbf{R} \subseteq P_{E \leftrightarrow F} \mid \forall s\, t \cdot s \preceq t \land t \in \mathbf{R} \Rightarrow s \in \mathbf{R}\}\,.$$

The plays of $P_{E \leftrightarrow F}$, interpreted as follows, allow more simulations to succeed as more and larger plays are added to the convention:

- The play $(m_1, m_2)\bot$ allows the questions m_1 and m_2 to be related by $\mathbf{R}^\circ$. By default, *all possible pairs of answers* (n_1, n_2) are permitted by $\mathbf{R}^\bullet_{m_1 m_2}$. However, no questions are allowed beyond that point until plays of the following kind are added to the refinement convention.
- The play $(m_1, m_2)(n_1, n_2)s$ extends the "next" convention $\mathbf{R}^{n_1, n_2}_{m_1, m_2}$ with the play s. Importantly, it does *not* modify the "answers" relation $\mathbf{R}^\circ$. As explained above, the pair $(n_1, n_2) \in \mathbf{R}^\circ_{m_1 m_2}$ was already—and remains—permitted. However,
- the play $(m_1, m_2)(n_1, n_2)\top$ *disallows* the pair $(n_1, n_2) \notin \mathbf{R}^\circ_{m_1 m_2}$ as related answers. Since this restricts the *handler*, simulations between client computations become easier to prove, which is why plays of the form $(m_1, m_2)(n_1, n_2)\top$ are the "largest".

Based on this interpretation, we could define the components R°, $R^\bullet$ and $R^{n_1 n_2}_{m_1 m_2}$ as follows:

$$m_1 \ \mathbf{R}^\circ \ m_2 \ :\Leftrightarrow \ (m_1, m_2)\bot \in \mathbf{R}, \qquad n_1 \ \mathbf{R}^\bullet_{m_1 m_2} \ n_2 \ :\Leftrightarrow \ (m_1, m_2)(n_1, n_2)\top \notin \mathbf{R},$$
$$\mathbf{R}^{n_1 n_2}_{m_1 m_2} \ := \ (m_1, m_2)(n_1, n_2)\backslash\mathbf{R} \ = \ \{s \mid (m_1, m_2)(n_1, n_2)s \in \mathbf{R}\}\,.$$

Note the negative involvement of $(m_1, m_2)(n_1, n_2)\top$ in the definition of $\mathbf{R}^\bullet$. When this play appears in $\mathbf{R}$, then by construction $\mathbf{R}$ must contain all plays of the form $(m_1, m_2)(n_1, n_2)s$ as well. However, they become meaningless as a simulation can never proceed in a way that they could influence.

4.3 Refinement Squares

We now use refinement conventions to express our general notion of a refinement square $\sigma \leq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \tau$. Specifically, we will use $\mathbf{R}$ and $\mathbf{S}$ to translate each play of σ into a *challenge* for the strategy τ. Because of the alternating nature of refinement, this challenge will involve nested $\forall$ and $\exists$ quantifiers over the possible choices of questions and answers offered by the refinement conventions.

Definition 4.2 (Refinement Square). Consider two strategies $\sigma : E_1 \twoheadrightarrow F_1$ and $\tau : E_2 \twoheadrightarrow F_2$ as well as two refinement conventions $\mathbf{R} : E_1 \leftrightarrow E_2$ and $\mathbf{S} : F_1 \leftrightarrow F_2$. We say that there is a refinement square when the proposition $\sigma \leq_{R \twoheadrightarrow S} \tau$ defined below holds. To this end, we recursively define a family of relations $\trianglelefteq^x_{\mathbf{R} \twoheadrightarrow \mathbf{S}}$ between the possible plays of σ and the possible residuals of τ. Using the short-hands $\mathbf{R}' := (m_1, m_2)(n_1, n_2)\backslash\mathbf{R}$ and $\mathbf{S}' := (q_1, q_2)(r_1, r_2)\backslash\mathbf{S}$, we can write:

$$\begin{array}{rll}
\epsilon \ \trianglelefteq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \ \tau & :\Leftrightarrow & \epsilon \in \tau \\
q_1 s \ \trianglelefteq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \ \tau & :\Leftrightarrow & \forall q_2 \cdot (q_1, q_2)\bot \in \mathbf{S} \Rightarrow s \trianglelefteq^{q_1, q_2}_{\mathbf{R} \twoheadrightarrow \mathbf{S}} (q_2\backslash\tau) \\
\underline{r}_1 s \ \trianglelefteq^{q_1, q_2}_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \ \tau & :\Leftrightarrow & \exists r_2 \cdot (q_1, q_2)(r_1, r_2)\top \notin \mathbf{S} \ \land \ s \trianglelefteq_{\mathbf{R} \twoheadrightarrow \mathbf{S}'} (\underline{r}_2\backslash\tau) \\
\underline{m_1} s \ \trianglelefteq^{q_1, q_2}_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \ \tau & :\Leftrightarrow & \exists m_2 \cdot (m_1, m_2)\bot \in \mathbf{R} \ \land \ s \trianglelefteq^{q_1 m_1, q_2 m_2}_{\mathbf{R} \twoheadrightarrow \mathbf{S}} (\underline{m_2}\backslash\tau) \\
\epsilon \ \trianglelefteq^{q_1 m_1, q_2 m_2}_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \ \tau & :\Leftrightarrow & \epsilon \in \tau \\
n_1 s \ \trianglelefteq^{q_1 m_1, q_2 m_2}_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \ \tau & :\Leftrightarrow & \forall n_2 \cdot (m_1, m_2)(n_1, n_2)\top \notin \mathbf{R} \Rightarrow s \trianglelefteq^{q_1, q_2}_{\mathbf{R}' \twoheadrightarrow \mathbf{S}} (n_2\backslash\tau)
\end{array}$$

Then we can formulate the existence of a refinement square as:

$$\sigma \leq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \tau \;:\Leftrightarrow\; \forall s \in \sigma \cdot s \trianglelefteq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \tau \,.$$

Refinement squares are compatible with strategy composition in the following sense.

Theorem 4.3. *Refinement squares compose horizontally as described by the rule* sim-$\odot$ *in Fig. 3.*

Refinement squares are also connected to the inclusion ordering on both strategies and refinement conventions. The relationship is formulated using identities.

Definition 4.4. The *identity refinement convention* $\mathbf{id}_E$ associated with a signature E is defined by:

$$\begin{aligned}
(m_1, m_2)\bot \in \mathbf{id}_E \;&:\Leftrightarrow\; m_1 = m_2 \\
(m_1, m_2)(n_1, n_2)\top \in \mathbf{id}_E \;&:\Leftrightarrow\; m_1 = m_2 \wedge n_1 \neq n_2 \\
(m_1, m_2)(n_1, n_2)s \in \mathbf{id}_E \;&:\Leftrightarrow\; m_1 = m_2 \wedge (n_1 = n_2 \Rightarrow s \in \mathbf{id}_E)
\end{aligned}$$

Theorem 4.5. *For all* $\sigma, \tau : E \twoheadrightarrow F$ *and for all* $\mathbf{R}, \mathbf{S} : E \leftrightarrow F$, *the following relationships hold:*

$$\sigma \subseteq \tau \Rightarrow \sigma \leq_{\mathbf{id}_E \twoheadrightarrow \mathbf{id}_F} \tau \,, \qquad \mathbf{R} \supseteq \mathbf{S} \Rightarrow \mathrm{id}_E \leq_{\mathbf{R} \twoheadrightarrow \mathbf{S}} \mathrm{id}_F \,.$$

Remark 4.6. Refinement conventions enforce a 1-to-1 mapping between the moves of the source- and target-level strategies, and require that their plays have similar structures. However, in some cases the relationship between events in the high-level view of the system and their realization in low-level terms is more complex; for example, the high-level view of a TCP/IP connection as a stream of bytes could model the transmission of a block of data as a single event, whereas its realization in terms of low-level packets may involve a complex interaction.

While the strict mapping enforced by refinement conventions is a limitation, situations like the one described above can still be modeled within our formalism. Suppose $\sigma : \mathcal{B} \twoheadrightarrow E$ uses the "byte stream" interface $\mathcal{B}$ while its refinement $\tau : \mathcal{K} \twoheadrightarrow E$ is implemented in terms of a network packet interface $\mathcal{K}$. It remains possible to express their relationship as a refinement square $\sigma \odot x \leq_{\mathbf{R} \twoheadrightarrow E} \tau$ with the help of auxiliary constructions $x : \mathcal{X} \twoheadrightarrow \mathcal{B}$ and $\mathbf{R} : \mathcal{X} \leftrightarrow \mathcal{K}$, proceeding in two steps:

- the effect signature $\mathcal{X}$ can provide a high-level, abstract representation of the packet interaction, and the strategy $x : \mathcal{X} \twoheadrightarrow \mathcal{B}$ explains how byte stream operations are expanded into abstract packet interactions with more complex shapes;
- the refinement convention $\mathbf{R} : \mathcal{X} \leftrightarrow \mathcal{K}$ can then be used to express the data abstraction component of the relationship, refining high-level abstract packets into their low-level actual representations, and encapsulating details such as TCP sequence numbers.

4.4 Vertical Composition

Refinement conventions compose similarly to relations, in that $R \mathbin{⨟} S$ relates two incoming questions m_1 and m_2 when there exists an intermediate m such that $(m_1, m)\bot \in R$ and $(m, m_2)\bot \in S$. However, we take into account the history of the interaction and the mixed variance of questions vs. answers.

Definition 4.7 (Vertical composition of refinement conventions). For the refinement conventions $\mathbf{R} : E_1 \leftrightarrow E_2$ and $\mathbf{S} : E_2 \leftrightarrow E_3$, the refinement convention $\mathbf{R} \mathbin{⨟} \mathbf{S} : E_1 \leftrightarrow E_3$ is defined as follows:

$$\begin{aligned}
(m_1, m_3)\bot \in \mathbf{R} \mathbin{⨟} \mathbf{S} \;&:\Leftrightarrow\; \exists m_2 \cdot (m_1, m_2)\bot \in \mathbf{R} \;\wedge\; (m_2, m_3)\bot \in \mathbf{S} \\
(m_1, m_3)(n_1, n_3)\top \in \mathbf{R} \mathbin{⨟} \mathbf{S} \;&:\Leftrightarrow\; \exists m_2 \cdot (m_1, m_2)\bot \in \mathbf{R} \;\wedge\; (m_2, m_3)\bot \in \mathbf{S} \;\wedge \\
&\qquad\; \forall n_2 \cdot (m_1, m_2)(n_1, n_2)\top \in \mathbf{R} \;\vee\; (m_2, m_3)(n_2, n_3)\top \in \mathbf{S} \\
(m_1, m_3)(n_1, n_3)\, s \in \mathbf{R} \mathbin{⨟} \mathbf{S} \;&:\Leftrightarrow\; \exists m_2 \cdot (m_1, m_2)\bot \in \mathbf{R} \;\wedge\; (m_2, m_3)\bot \in \mathbf{S} \;\wedge \\
&\qquad\; \forall n_2 \cdot (m_1, m_2)(n_1, n_2)\top \in \mathbf{R} \;\vee\; (m_2, m_3)(n_2, n_3)\top \in \mathbf{S} \;\vee \\
&\qquad\qquad s \;\in\; \big((m_1, m_2)(n_1, n_2)\backslash\mathbf{R}\big) \mathbin{⨟} \big((m_2, m_3)(n_2, n_3)\backslash\mathbf{S}\big) \,.
\end{aligned}$$

This allows us to express the vertical composition property for refinement squares.

Theorem 4.8 (Vertical composition of refinement squares). *Refinement squares compose vertically as described by the rule* sim-$\fatsemi$ *shown in Fig. 3.*

Remark 4.9 (Associativity of vertical composition). It should be noted that the vertical composition of refinement conventions is not associative in general (although associativity holds in most practical cases that we have encountered). We discuss this phenomenon and a counter-example in Appendix A [Zhang et al. 2024b].

4.5 Flat Composition

Finally, we show that the flat composition operation $\oplus$ which we have defined for effect signatures and strategies can be extended to refinement conventions and refinement squares as well.

Definition 4.10 (Flat composition of refinement conventions). The conventions $\mathbf{R}_1 : E_1 \leftrightarrow F_1$ and $\mathbf{R}_2 : E_2 \leftrightarrow F_2$ compose into $\mathbf{R}_1 \oplus \mathbf{R}_2 : E_1 \oplus E_2 \leftrightarrow F_1 \oplus F_2$, defined by:

$$\begin{aligned}
\big(\iota_i(m_1), \iota_i(m_2)\big)\bot \in \mathbf{R}_1 \oplus \mathbf{R}_2 \;&:\Leftrightarrow\; (m_1, m_2)\bot \in \mathbf{R}_i \\
\big(\iota_i(m_1), \iota_i(m_2)\big)(n_1, n_2)\top \in \mathbf{R}_1 \oplus \mathbf{R}_2 \;&:\Leftrightarrow\; (m_1, m_2)(n_1, n_2)\top \in \mathbf{R}_i \\
\big(\iota_1(m_1), \iota_1(m_2)\big)(n_1, n_2)\, s \in \mathbf{R}_1 \oplus \mathbf{R}_2 \;&:\Leftrightarrow\; s \in \big((m_1, m_2)(n_1, n_2)\backslash \mathbf{R}_1\big) \oplus \mathbf{R}_2 \\
\big(\iota_2(m_1), \iota_2(m_2)\big)(n_1, n_2)\, s \in \mathbf{R}_1 \oplus \mathbf{R}_2 \;&:\Leftrightarrow\; s \in \mathbf{R}_1 \oplus \big((m_1, m_2)(n_1, n_2)\backslash \mathbf{R}_2\big)
\end{aligned}$$

Theorem 4.11. *Flat composition of refinement conventions and squares obeys the rules in Fig. 6.*

5 Compositional State

The model described so far adds a vertical dimension to the usual horizontal dimension of compositional semantics. We now discuss how the model can be extended further by introducing a *spatial* dimension, which serves as a foundation of our compositional treatment of state. We omit many formal definitions in the interest of space and readability, but they can be found in Appendix B [Zhang et al. 2024b].

5.1 Explicit State

Like the sum used by flat composition, the tensor product is another well-known operation on effect signatures, which expects the client to *simultaneously* ask a question in each component:

$$\bigotimes_{i \in I} E_i \;:=\; \Big\{ \langle m_i \rangle_{i \in I} : \textstyle\prod_{i \in I} N_i \;\Big|\; \forall i \,.\, (m_i : N_i) \in E_i \Big\}$$

Unfortunately, while the simulation convention $\mathbf{R} \otimes \mathbf{S}$ is straightforward to define, the tensor product unlike $\oplus$ does not generalize easily to strategies. Defining $L_1 \otimes L_2$ we have no reason to expect that outgoing questions of L_1 and L_2 will synchronize to combine into questions of $E_1 \otimes E_2$.

Although a general form of $\otimes$ does not apply in our framework, by restricting the right-hand side to a form of *passive* components we obtain a form of *spatial* composition and a way to approach compositional state. Specifically, for a set U we start from the effect signature construction

$$E \mathbin{@} U \;:=\; E \otimes \{u : U \mid u \in U\} \;=\; \{m@u : N \times U \;\mid\; (m : N) \in E,\, u \in U\},$$

where $m@u$ is a stylized version of the pair (m, u). This construction will play an important role in our treatment of spatial composition and state encapsulation.

Example 5.1. In CompCertO language interfaces, every question and answer includes a global memory state $m \in \mathsf{mem}$ (Example 2.3). The decomposition $C \mathbin{@} \mathsf{mem}$ allows us to separate

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

$L_{\mathrm{bq}} : 0 \twoheadrightarrow E_{\mathrm{bq}} \mathbin{@} D_{\mathrm{bq}}$	$L_{\mathrm{bq}} \vDash \mathsf{enq}[v]@\vec{q} \rightarrowtail *@\vec{q}v$	$\vec{q} \in D_{\mathrm{bq}} := V^*,\ v \in V$
	$L_{\mathrm{bq}} \vDash \mathsf{deq}@v\vec{q} \rightarrowtail v@\vec{q}$	
$\Sigma_{\mathrm{bq}} : E_{\mathrm{rb}} \twoheadrightarrow E_{\mathrm{bq}}$	$\Sigma_{\mathrm{bq}} \vDash \mathsf{enq}[v] \rightarrowtail (\mathsf{inc}_2 \rightsquigarrow i) \rightarrowtail (\mathsf{set}[i,v] \rightsquigarrow *) \rightarrowtail *$	$i \in \mathbb{N},\ v \in V$
	$\Sigma_{\mathrm{bq}} \vDash \mathsf{deq} \rightarrowtail (\mathsf{inc1} \rightsquigarrow i) \rightarrowtail (\mathsf{get}[i] \rightsquigarrow v) \rightarrowtail v$	
$L_{\mathrm{rb}} : 0 \twoheadrightarrow E_{\mathrm{rb}} \mathbin{@} D_{\mathrm{rb}}$	$L_{\mathrm{rb}} \vDash \mathsf{inc}_1@(b,c_1,c_2) \rightarrowtail c_1@(b,(c_1{+}1)\%N, c_2)$	$(b,c_1,c_2) \in D_{\mathrm{rb}} := V^N \times \mathbb{N} \times \mathbb{N},$
	$L_{\mathrm{rb}} \vDash \mathsf{inc}_2@(b,c_1,c_2) \rightarrowtail c_2@(b,c_1,(c_2{+}1)\%N)$	$i \in \mathbb{N},\ v \in V$
	$L_{\mathrm{rb}} \vDash \mathsf{set}[i,v]@(b,c_1,c_2) \rightarrowtail *@(b[i := v], c_1, c_2)$	
	$L_{\mathrm{rb}} \vDash \mathsf{get}[i]@(b,c_1,c_2) \rightarrowtail b_i@(b,c_1,c_2)$	
$R_{\mathrm{bq}} \subseteq D_{\mathrm{bq}} \times D_{\mathrm{rb}}$	$\vec{q}\ R_{\mathrm{bq}}\ (b,c_1,c_2) \Leftrightarrow (c_1 \le c_2 < N \wedge \vec{q} = b_{c_1} \cdots b_{c_2-1}) \vee$	$(b,c_1,c_2) \in D_{\mathrm{rb}},$
	$(c_2 \le c_1 < N \wedge \vec{q} = b_{c_1} \cdots b_{N-1} b_0 \cdots b_{c_2-1})$	$\vec{q} \in D_{\mathrm{bq}}$

Fig. 7. Abstract specifications for bq.c and rb.c, formulated as regular strategies using explicit state. The overall specification Γ_{bq} describes the queue operations in terms of a sequence of values $\vec{q} \in D_{\mathrm{bq}} := V^*$. Verification can be decomposed using the intermediate specifications Σ_{bq} and Γ_{rb} for bq.c and rb.c.

which does not mention the memory state. This affords us more flexibility when describing the ways in which C code can affect both the global memory and other forms of more abstract state.

Example 5.2 (Abstract specifications). The specification L_{bq} shown in Fig. 7 gives an abstract description of the code in Fig. 5 by representing the queue state as a sequence $\vec{q}$. Likewise L_{rb} uses the data (b, c_1, c_2) to represent the contents of the buffer and the counter values. Finally, bq.c does not use any state of its own and can be described by the simple specification $\Sigma_{\mathrm{bq}} : E_{\mathrm{rb}} \twoheadrightarrow E_{\mathrm{bq}}$. We hope to decompose a correctness proof along the following lines:

$$\phi_1 : L_{\mathrm{bq}} \le_{0 \twoheadrightarrow ?} \Sigma_{\mathrm{bq}} \text{“}\odot\text{”}\, \Gamma_{\mathrm{rb}} \qquad \phi_2 : \Sigma_{\mathrm{bq}} \le_{? \twoheadrightarrow ?} \mathsf{Clight}(\mathsf{bq.c}) \qquad \phi_{\mathrm{rb}} : L_{\mathrm{rb}} \le_{\varnothing \twoheadrightarrow ?} \mathsf{Clight}(\mathsf{rb.c})$$

However, the different types of states prevent the components from being composed directly.

To make the approach outlined above practical, we must turn @ into a proper composition principle and establish its action on strategies, refinement conventions and refinement squares.

5.2 Passing State Through

We start by outlining how the construction $- \mathbin{@} U$ acts on strategies in the case of a fixed set U. Namely, given $L : A \twoheadrightarrow B$, the strategy $L \mathbin{@} U : A \mathbin{@} U \twoheadrightarrow B \mathbin{@} U$ transparently passes along a state component of type U as follows:

$$\frac{L \ \vDash\ q \rightarrowtail (q_1 \rightsquigarrow r_1) \rightarrowtail \cdots \rightarrowtail (q_n \rightsquigarrow r_n) \rightarrowtail r}{L \mathbin{@} U \ \vDash\ q@u_0 \rightarrowtail (q_1@u_0 \rightsquigarrow r_1@u_1) \rightarrowtail \cdots \rightarrowtail (q_n@u_{n-1} \rightsquigarrow r_n@u_n) \rightarrowtail r@u_n} \tag{4}$$

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 5.3. We can use @ to interface $\Sigma_{\mathrm{bq}} : E_{\mathrm{rb}} \twoheadrightarrow E_{\mathrm{bq}}$ with the specification $L_{\mathrm{rb}} : 0 \twoheadrightarrow E_{\mathrm{rb}} @ D_{\mathrm{rb}}$. The result $(\Sigma_{\mathrm{bq}} @ D_{\mathrm{rb}}) \odot L_{\mathrm{rb}} : 0 \twoheadrightarrow E_{\mathrm{bq}} @ D_{\mathrm{rb}}$ uses the construction $\Sigma_{\mathrm{bq}} @ D_{\mathrm{rb}} : E_{\mathrm{rb}} @ D_{\mathrm{rb}} \twoheadrightarrow E_{\mathrm{bq}} @ D_{\mathrm{rb}}$ which allows Σ_{bq} to “pass through” the abstract data D_{rb} on which L_{rb} operates.

$$\frac{L : A \twoheadrightarrow B \qquad f : U \leftrightarrows V}{L @ f : A @ U \twoheadrightarrow B @ V}\ \textsf{ts-@} \qquad\qquad \frac{\mathbf{R} : A \leftrightarrow B \qquad \mathbf{S} : U \leftrightarrow V}{\mathbf{R} @ \mathbf{S} : A @ U \leftrightarrow B @ V}\ \textsf{sc-@}$$

$$(L_1 \odot L_2) @ (f \circ g) \equiv (L_1 @ f) \odot (L_2 @ g) \qquad (\mathbf{R}_1 \fatsemi \mathbf{R}_2) @ (\mathbf{S}_1 \fatsemi \mathbf{S}_2) \equiv (\mathbf{R}_1 @ \mathbf{S}_1) \fatsemi (\mathbf{R}_2 @ \mathbf{S}_2)$$
$$\mathrm{id}_A @ \mathrm{id}_U \equiv \mathrm{id}_{A@U} \qquad \mathbf{id}_A @ \mathbf{id}_U \equiv \mathbf{id}_{A@U}$$

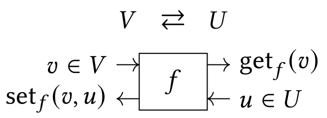
$$\frac{\phi : L \leq_{\mathbf{R}_1 \twoheadrightarrow \mathbf{S}_1} L' \qquad \psi : f \leq_{\mathbf{R}_2 \twoheadrightarrow \mathbf{S}_2} f'}{\phi @ \psi : L @ f \leq_{\mathbf{R}_1 @ \mathbf{R}_2 \twoheadrightarrow \mathbf{S}_1 @ \mathbf{S}_2} L' @ f'}\ \textsf{sim-@}$$

Fig. 8. Spatial composition (@) for strategies, simulation conventions and simulation proofs.

5.3 Transforming State

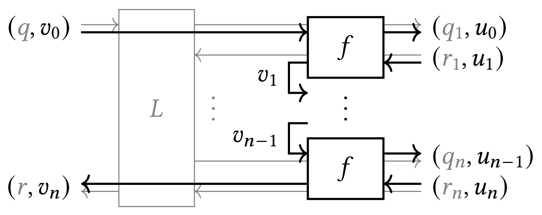
It is possible to generalize the construction $L @ U$ to incorporate a *lens* $f : U \leftrightarrows V$ with a more sophisticated action on the state component than a simple pass-through. Lenses [Bohannon et al. 2008] provides access to a field of type U within V through functions:

$$\mathrm{get}_f : V \to U \qquad \mathrm{get}_f(\mathrm{set}_f(v, u)) = u$$
$$\mathrm{set}_f : V \times U \to V \qquad \mathrm{set}_f(v, \mathrm{get}_f(v)) = v$$
$$\mathrm{set}_f(\mathrm{set}_f(v, u_1), u_2) = \mathrm{set}_f(v, u_2)$$



Operationally, as illustrated above, we think of a lens as a component which behaves somewhat like the identity strategy (Fig. 4c). When an incoming question $v \in V$ activates the components, the view $\mathrm{get}_f(v) \in U$ is extracted and forwarded as an outgoing question. When this outgoing question is answered with an update $u \in U$, the updated value $\mathrm{set}_f(v, u)$ is returned to the caller.

As with $L @ U$, in the strategy $L @ f : A @ U \twoheadrightarrow B @ V$, every question and answer consists of a pair, with one component from A or B and one component from the sets U or V; the first component is handled by L while the second one is just carried along. But now, when L makes an outgoing call, the second component first passes through the lens f to be projected into U:



In practice, two kinds of lens turn out to be especially useful. First, every bijection is a lens, and this can be used to define structural isomorphisms such as $\gamma_{U,V} : U \times V \cong V \times U$. Secondly, the trivial lens $\langle V] : \mathbb{1} \leftrightarrows V$ where $\mathrm{get}_{\langle V]}(v) = *$ and $\mathrm{set}_{\langle V]}(v, *) = v$ can act as a "terminator", which does not propagate any part of the state in U but instead returns it unchanged to the caller.

The @ construction can be further extended to act on refinement conventions and refinement squares to obtain the compositional structure shown in Fig. 8. The composite refinement convention $\mathbf{R}@\mathbf{S}$ simply requires that the two fields within the questions and answers of the composite language interfaces be independently related by the corresponding simulation conventions. Moreover, a relation $R \subseteq U \times V$ can be promoted to a simple simulation convention. See the appendix for details.

Example 5.4. Building on Example 5.3, consider the relationship between the overall specification $L_{\mathrm{bq}} : 0 \twoheadrightarrow E_{\mathrm{bq}} @ D_{\mathrm{bq}}$ and its partial refinement $(\Sigma_{\mathrm{bq}} @ D_{\mathrm{rb}}) \odot L_{\mathrm{rb}} : 0 \twoheadrightarrow E_{\mathrm{bq}} @ D_{\mathrm{rb}}$. To establish a simulation between them, we use the abstraction relation $R_{\mathrm{bq}} \subseteq D_{\mathrm{bq}} \times D_{\mathrm{rb}}$ given in Fig. 7. The refinement property can then be formulated as $\phi_1 : L_{\mathrm{bq}} \ \leq_{0 \twoheadrightarrow E_{\mathrm{bq}} @ R_{\mathrm{bq}}} \ (\Sigma_{\mathrm{bq}} \ @ \ D_{\mathrm{rb}}) \odot \Gamma_{\mathrm{rb}}$.

5.4 State Encapsulation

To go beyond the realm of regular strategies, in our model a component $f : U \leftrightarrows V$ allows a hidden *persistent state* component P to be incorporated into the incoming data $V \times P$ used by the lens f.

Encapsulation Primitive. Hidden state allows us to define an *encapsulation* primitive $[u\rangle : U \leftrightarrows \mathbb{1}$, which can be used to turn an explicit state component into a private one. This primitive uses $P := U$ for its persistent state. When it is activated by $* \in \mathbb{1}$, the current state (initially $u \in U$) is used as an outgoing question; the answer $u' \in U$ is then used as an updated state for the next activation.

Example 5.5. The component $\Gamma_{\mathrm{bq}} := \big(E_{\mathrm{bq}} \ @ \ [\epsilon\rangle\big) \odot L_{\mathrm{bq}} : 0 \twoheadrightarrow E_{\mathrm{bq}}$ describes the behavior of an initially empty bounded queue. The set of abstract states D_{bq} is used to define it, but is not exposed as part of its interface, so that client code will only observe call traces where state is implicit:

$$\Gamma_{\mathrm{bq}} \vDash (\mathsf{enq}[v_1] \rightarrowtail *) \rightsquigarrow (\mathsf{enq}[v_2] \rightarrowtail *) \rightsquigarrow (\mathsf{deq} \rightarrowtail v_1) \rightsquigarrow (\mathsf{enq}[v_3] \rightarrowtail *) \rightsquigarrow (\mathsf{deq} \rightarrowtail v_2) \rightsquigarrow \cdots$$

Likewise, we can use $d_0 := (\{\ldots\}, 0, 0) \in D_{\mathrm{rb}}$ to define $\Gamma_{\mathrm{rb}} := \big(E_{\mathrm{rb}} \ @ \ [d_0\rangle\big) \odot L_{\mathrm{rb}} : 0 \twoheadrightarrow E_{\mathrm{rb}}$ as an encapsulated specification for the ring buffer data structure.

Representation Independence. Two components may use different representations for their explicit state, but otherwise exhibit identical behaviors. In this case, encapsulating their state will yield identical strategies. Within our framework, this follows from the property:

$$\zeta : u \, R \, v \quad \Longrightarrow \quad [\zeta\rangle : [u\rangle \leq_{R \twoheadrightarrow \mathbb{1}} [v\rangle \tag{5}$$

Indeed, to establish that $L_1 : E \twoheadrightarrow F \ @ \ U$ and $L_2 : E \twoheadrightarrow F \ @ \ V$ exhibit similar behaviors, we can define a relation $R \subseteq U \times V$ between their explicit states and prove the simulation

$$\phi : L_1 \leq_{E \twoheadrightarrow F @ R} L_2 \, .$$

This shows that when invoked in related states, L_1 and L_2 behave similarly and the updated states they eventually return are related as well. Per (5), the primitives $[u\rangle$ and $[v\rangle$ establish this invariant for the initial states and preserve it across successive calls. This allows us to show that:

$$(F \ @ \ [\zeta\rangle) \odot \phi \ : \ (F \ @ \ [u\rangle) \odot L_1 \ \leq \ (F \ @ \ [v\rangle) \odot L_2 \, .$$

Proving the simulation in both directions would allow us to conclude that the behaviors are equal.

Example 5.6. Following up on Example 5.5, we can use the fact $\zeta_{\mathrm{bq}} : \epsilon \, R_{\mathrm{bq}} \, d_0$ that the initial states are related to prove the following property:

$$\phi_1' \ := \ (E_{\mathrm{bq}} \ @ \ [\zeta_{\mathrm{bq}}\rangle) \odot \phi_1 \ : \ \Gamma_{\mathrm{bq}} \ \leq \ \Sigma_{\mathrm{bq}} \odot \Gamma_{\mathrm{rb}} \, .$$

That is, encapsulation not only makes it easier to interface $\Sigma_{\mathrm{bq}} : E_{\mathrm{rb}} \twoheadrightarrow E_{\mathrm{bq}}$ with $\Gamma_{\mathrm{rb}} : 0 \twoheadrightarrow E_{\mathrm{rb}}$, but it also means the simulation ϕ_1' can be stated in terms of the identity refinement convention.

5.5 Implementing Encapsulated State

Ultimately, our goal is to connect a high-level specification such as Γ_{bq} which uses encapsulated state to a low-level implementation like the one shown in Fig. 5b where state is explicit and stored as part of the concrete C memory. To construct refinement conventions which can capture this concretization process, we can use *vertical* versions of our encapsulation primitives.

Definition 5.7. We say that a strategy $L : A \twoheadrightarrow B$ or lens $L : A \leftrightarrows B$ has:

- a *companion* $L^* : A \leftrightarrow B$ when $L^{\vartriangle} : A \leq_{A \twoheadrightarrow L^*} L$ and $L^{\triangledown} : L \leq_{L^* \twoheadrightarrow B} B$;
- a *conjoint* $L_* : B \leftrightarrow A$ when $L_{\vartriangle} : B \leq_{L_* \twoheadrightarrow B} L$ and $L_{\triangledown} : L \leq_{A \twoheadrightarrow L_*} A$.

Table 1. Components of our Coq artifact, with corresponding lines of code counted by coqwc.

Component	Definitions	Proofs	Application	Definitions	Proofs
coqrel library	2,382	959	CompCertO embedding (§6.2)	1,000	1,743
CompCertO	124,217	95,187	Bounded queue example (§6.3)	1,572	2,606
Other support code	271	491	Process example (§6.4)	1,414	2,621
Our framework (§2–5, §6.1)	2,198	3,252	CAL (§6.5)	262	667
			ClightP (§6.6)	1,656	2,126

Concretely, these properties mean that for certain refinement squares, we can choose whether a particular component should appear horizontally or vertically. This makes it possible to decompose proofs along non-rectangular boundaries, and generally affords us additional flexibility. In practice, companions and conjoints are especially useful for lenses, which satisfy the following property.

Theorem 5.8. *Every lens $f : U \leftrightarrows V$ has a companion $f^* : U \leftrightarrow V$ and a conjoint $f_* : V \leftrightarrow U$.*

In particular, the conjoint $[u\rangle_* : \mathbb{1} \leftrightarrow U$ can be used to formalize state "deencapsulation". Concretely, $[u\rangle_*$ requires the first target question to carry the value u. When the question is answered with a new state u', this new state replaces u. The next question is expected to carry the value u', and so on. In other words, $[u\rangle_*$ requires the target system to be provided with a state component of type U, maintained across successive activations and initially set to the value u.

6 Evaluation and Applications

Having described our formalism, we discuss its mechanization in the Coq proof assistant and several possible applications. This is demonstrated in the companion artifact [Zhang et al. 2024a], whose components are outlined in Table 1 and discussed below.

6.1 Mechanization in the Coq Proof Assistant

Our code uses a library called coqrel [Koenig 2024] for relational reasoning and the relevant parts interface with CompCertO as well. In addition, our goal is eventually to incorporate our model into a broader library for compositional semantics and heterogeneous system verification[1], and we rely ("other support code") on this library's formalization of downward-closed sets.

With these dependencies, the definitions and theorems given in §2–5 can be mechanized in 2,198 lines of Coq definitions and 3,252 lines of proofs, as counted by coqwc. The mechanization is straightforward and closely follows the definitions we have given.

Use of Dependent Types. One interesting aspect of our development is its use of dependent types to capture combinatorial aspects of strategy interaction.

As suggested by the inductive grammar given in Definition 3.1, we use dependent types to enforce the structure of plays, defining play $E\,F$: position $E\,F \longrightarrow$ Type and strat $E\,F$: position $E\,F \longrightarrow$ Type as families indexed by the type:

```
Variant position E F := ready | running (q : op F) | suspended (q : op F) (m : op E).
```

This way, rather than defining plays as simple lists of moves and separately demanding that they satisfy some validity criterion, we can use the type system to enforce their expected shape and avoid having to deal with a proliferation of side-conditions.

At the same time, under this approach, definitions and proofs which involve plays and strategies in different positions require a way to express the combinatorial constraints which tie them together.

[1] https://github.com/CertiKOS/rbgs

Our solution is to define a new *position* type for each of these constructions, with fully unbundled projections onto the corresponding positions of their components. For example, our formalization of layered composition (Definition 3.4) involves the type:

```
Variant cpos : position F G -> position E F -> position E G -> Type :=
  | cpos_ready : cpos ready ready ready
  | cpos_left q : cpos (running q) ready (running q)
  | cpos_right q m : cpos (suspended q m) (running m) (running q)
  | cpos_suspended q m u : cpos (suspended q m) (suspended m u) (suspended q u).
```

where the first two parameters specify compatible positions for the strategies or plays being composed, and the third specifies the position for the result. The same principle is applied for higher-level constructions; for example the proof of

$$(\sigma_1 \oplus \sigma_2) \odot (\tau_1 \oplus \tau_2) = (\sigma_1 \odot \tau_1) \oplus (\sigma_2 \odot \tau_2) \tag{6}$$

involves a position type whose parameters project onto the four simple positions for the strategies involved, as well as the $\odot$ and $\oplus$ composite positions for each intermediate expression. The fully unbundled approach allows us to use the type system to encode the complex synchronization constraints involved. In fact, for many proofs of that nature, laying down those constraints was the most complex part of the job, and once the cases were enumerated the proof itself became more or less self-evident. It would be interesting to compare them with the definitions and proofs that would be obtained under a more traditional approach.

Note that for high-level reasoning, the user will usually only manipulate strategies and use properties such as (6) above in the context of trivial ready positions, so that they are not exposed to the internal complexity associated with the combinatorial constraints.

6.2 CompCertO semantics

As mentioned in Examples 2.3 and 2.5, the language semantics and correctness properties defined by the certified compiler CompCertO can be used within our model.

Open Transition System. CompCertO uses a notion of *open* transition system to describe interactions across component boundaries. These boundaries are specified using language interfaces of the form $A := \langle A^\circ, A^\bullet \rangle$, which translate to effect signatures $[\![A]\!] := \{q : A^\bullet \mid q \in A^\circ\}$.

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

- a set S of states and a transition relation $\rightarrow \subseteq S \times S$;
- 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.*

Writing $(s, r, s') \in Y$ as $r\ Y^s\ s'$, executions take the form

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

corresponding to an interaction trace $q \rightarrowtail (q_1 \rightsquigarrow r_1) \rightarrowtail \cdots \rightarrowtail (q_n \rightsquigarrow r_n) \rightarrowtail r$.

To describe the strategy associated with a CompCertO transition system, we first formalize the set of plays generated by an internal state $s \in S$ as follows:

$$\frac{s \rightarrow^* s'\ X\ m \rightsquigarrow n\ Y^{s'}\ s'' \qquad s'' \Vdash w}{s \Vdash \underline{m} n w} \qquad \frac{s \rightarrow^* s'\ F\ r}{s \Vdash \underline{r}}$$

For an invocation on the transition system, the play qw will then result when $q\ I\ s \vDash w$. To handle subsequent invocations, the process is iterated using the regular closure operator defined in §3.1:

$$[\![L]\!] := \Big(\bigcup_{q \in B^\circ} \{ qw \mid \exists s \cdot q\ I\ s \land s \Vdash w \} \Big)^* .$$

Simulation Convention. The simulation conventions used in CompCertO can likewise be translated to our richer notion of refinement convention.

Definition 6.1. A *simulation convention* $\mathbb{R} : A \Leftrightarrow B := \langle W, R^\circ, R^\bullet \rangle$ between the CompCertO language interfaces A and B is specified by a set W of worlds, a Kripke relation $R^\circ \subseteq W \times A^\circ \times B^\circ$ between questions and a Kripke relation $R^\bullet \subseteq W \times A^\bullet \times B^\bullet$ between answers.

Kripke worlds are used to ensure that questions and answers for a given call are related consistently. However, every pair of calls is related in isolation, independently of any past or future calls. Thus, the following refinement convention embeds the simulation convention $\mathbb{R} : A \Leftrightarrow B$:

$$\begin{aligned}
(m_1, m_2)\bot \in [\![\mathbb{R}]\!] &:\Leftrightarrow \exists w \cdot m_1 \mathbb{R}^\circ_w m_2 \\
(m_1, m_2)(n_1, n_2)\top \in [\![\mathbb{R}]\!] &:\Leftrightarrow \exists w \cdot m_1 \mathbb{R}^\circ_w m_2 \ \land\ \neg n_1 \mathbb{R}^\bullet_w n_2 \\
(m_1, m_2)(n_1, n_2)s \in [\![\mathbb{R}]\!] &:\Leftrightarrow \exists w \cdot m_1 \mathbb{R}^\circ_w m_2 \ \land\ (n_1 \mathbb{R}^\bullet_w n_2 \Rightarrow s \in [\![\mathbb{R}]\!]) .
\end{aligned}$$

Simulations. Using the embedding above, simulations between CompCertO transition systems and simulation conventions induce refinement squares between the corresponding strategies and refinement conventions within our model. In particular, CompCertO's compiler correctness corresponds to the following refinement:

$$[\![\phi^{\mathrm{cc}}_p]\!] : [\![\mathsf{Clight(p.c)}]\!] \leq_{[\![\mathbb{C}]\!] \twoheadrightarrow [\![\mathbb{C}]\!]} [\![\mathsf{Asm(p.s)}]\!] .$$

Composition. To model *linking* CompCertO introduces an operator $\oplus_A : (A \twoheadrightarrow A) \times (A \twoheadrightarrow A) \rightarrow (A \twoheadrightarrow A)$. 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. It is known that the syntactic linking of assembly programs implements $\oplus$:

$$\ell : \mathsf{Asm}(p_1) \oplus \mathsf{Asm}(p_2) \leq \mathsf{Asm}(p_1 + p_2)$$

The embedding is compatible with the $\oplus$ operator in CompCertO in the following sense:

$$e : [\![L_1]\!] \odot [\![L_2]\!] \leq [\![L_1 \oplus L_2]\!] \qquad \forall\, L_1, L_2 \in A \twoheadrightarrow A$$

This is no surprise because $\odot$ only permits calls from one direction, and therefore *under-approximates* the $\oplus$ combinator where mutually recursive calls can happen. In the rest of this section, we will omit $[\![-]\!]$ for brevity when the context is clear.

6.3 Memory Separation

Spatial composition allows us to separate complex states into different fields; we can then reason about components independently of the fields which they do not access, and use @ to connect these components with the rest of the system. However, eventually this abstract description must be refined into a concrete program acting on a global memory, where all state has been consolidated.

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* $m_1 \bullet \cdots \bullet m_n$. This construction is similar in spirit to the *algebraic memory model* of Gu et al. [2018]; its construction is explained in Appendix C [Zhang et al. 2024b].

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:

$$\frac{L \vDash q@m_0 \rightarrowtail (q_1@m_1 \rightsquigarrow r_1@m_1') \rightarrowtail \cdots \rightarrowtail (q_n@m_n \rightsquigarrow r_n@m_n') \rightarrowtail r@m'}{\begin{array}{l} L \vDash q@(m_0 \bullet w_0) \rightarrowtail \big(q_1@(m_1 \bullet w_0) \rightsquigarrow r_1@(m_1' \bullet w_1)\big) \rightarrowtail \cdots \\ \qquad\qquad \cdots \rightarrowtail \big(q_n@(m_n \bullet w_{n-1}) \rightsquigarrow r_n@(m_n' \bullet w_n)\big) \rightarrowtail r@(m' \bullet w_n) \end{array}} \tag{7}$$

The similarity of (7) with the behavior (4) of the transition system $L \mathbin{@} U$ (§5.2) is no coincidence. Reading $\bullet$ as a *join* relation $\curlyvee \subseteq (\mathsf{mem} \times \mathsf{mem}) \times \mathsf{mem}$, we can state one in terms of the other.

Theorem 6.2 (Frame property for Clight). *The Clight semantics satisfies*

$$\mathsf{FP}(M) : \mathsf{Clight}(M) \mathbin{@} \mathsf{mem} \leq_{A@\curlyvee \twoheadrightarrow B@\curlyvee} \mathsf{Clight}(M)\,, \ \textit{where}\ (m_1, m_2)\ \curlyvee\ m :\Leftrightarrow m_1 \bullet m_2 = m\,.$$

It will often be the case that the join relation is applied to the target of simulation convention components $\mathbf{R} : U \leftrightarrow \mathsf{mem}$ and $\mathbf{S} : V \leftrightarrow \mathsf{mem}$. In this case, we will use the notation:

$$\mathbf{R} \circledast \mathbf{S} : U \mathbin{@} V \leftrightarrow \mathsf{mem} \qquad \mathbf{R} \circledast \mathbf{S} := (\mathbf{R} \mathbin{@} \mathbf{S}) \mathbin{⨟} \curlyvee\,.$$

Example 6.3. To show that rb.c faithfully implements Γ_{rb}, we establish a correspondence between the operations of the signature E_{rb} and their representation as C calls by defining a refinement convention $\mathbf{E}_{\mathrm{rb}} : E_{\mathrm{rb}} \leftrightarrow C$. In addition, we explain how the abstract states of D_{rb} are realized in the concrete memory using the relation $R_{\mathrm{rb}} \subseteq D_{\mathrm{rb}} \times \mathsf{mem}$ defined by:

$$(b, c_1, c_2)\ \ R_{\mathrm{rb}}\ \ [\mathsf{buf} \mapsto \{b_0, \ldots, b_{N-1}\},\ \mathsf{c1} \mapsto c_1,\ \mathsf{c2} \mapsto c_2]\,.$$

At the implementation level, the memory state passed to rb.c will contain buf, c1 and c2, whose values must match the high-level abstract state and will be updated according to the specification. Also, the initial memory share $m_0 := \mathsf{init\_mem}(\mathsf{rb.c})$ associated with rb.c satisfies $\zeta_{\mathrm{rb}} : d_0\ R_{\mathrm{rb}}\ m_0$. The remaining part of the memory should not be changed by rb.c. This can be expressed as

$$\phi_{\mathrm{rb}} : \Gamma_{\mathrm{rb}} \leq_{\varnothing \twoheadrightarrow \mathbf{E}_{\mathrm{rb}}@\langle m_0 \rangle} \mathsf{Clight}(\mathsf{rb.c}) \tag{8}$$

where the refinement convention component $\langle m_0 \rangle := \langle \mathsf{mem}]^* \circledast [m_0\rangle_* : \mathbb{1} \leftrightarrow \mathsf{mem}$ expresses the idea that the memory state introduced at the target level is split into two halves. One half will contain buf, c1 and c2; it must be initialized to m_0 and preserved by the environment from one call to the next. The other half is unconstrained but is guaranteed to be left unchanged by rb.c.

Verifying rb.c. To establish (8) above, it suffices to show $\phi_{\mathrm{rb}}^{\mathsf{min}} : L_{\mathrm{rb}} \leq_{\varnothing \twoheadrightarrow \mathbf{E}_{\mathrm{rb}}@R_{\mathrm{rb}}} \mathsf{Clight}(\mathsf{rb.c})$. In other words, we can prove the correctness of rb.c in the context of a minimal memory share which contains only the variables buf, c1 and c2.

On one end, the program must manage all the memory shares passed from the client. To achieve this, we can use the Clight frame property for rb.c and the absorption property $z : \varnothing \sqsubseteq \varnothing \circledast \langle \mathsf{mem}]^*$ to derive $\phi_{\mathrm{rb}}' : L_{\mathrm{rb}} \leq_{\varnothing \twoheadrightarrow \mathbf{E}_{\mathrm{rb}}@\langle \mathsf{mem}]^* \circledast R_{\mathrm{rb}}} \mathsf{Clight}(\mathsf{rb.c})$ as follows:

$$\phi_{\mathrm{rb}}' := \big(\phi_{\mathrm{rb}}^{\mathsf{min}} \mathbin{@} \langle \mathsf{mem}]^* \mathbin{⨟} \mathsf{FP}(\mathsf{rb.c})\big) \odot z$$

On the other end, we transform the explicit state passing specification L_{rb} to its encapsulated counterpart Γ_{rb} using the auxiliary property ψ_{rb}, where

$$\psi_{\mathrm{rb}} := \Big(\mathbf{E}_{\mathrm{rb}} \mathbin{@} \langle \mathsf{mem}]^* \circledast \big([\zeta_{\mathrm{rb}}\rangle \mathbin{⨟} [m_0\rangle_{\triangledown}\big)\Big) : E_{\mathrm{rb}} \mathbin{@} [d_0\rangle \leq_{\mathbf{E}_{\mathrm{rb}}@\langle \mathsf{mem}]^* \twoheadrightarrow \mathbf{E}_{\mathrm{rb}}@\langle m_0 \rangle} C \mathbin{@} \mathsf{mem}\,.$$

With these ingredients, the desired property ϕ_{rb} can be derived as:

$$\phi_{\mathrm{rb}} := \psi_{\mathrm{rb}} \odot \phi_{\mathrm{rb}}' : \Gamma_{\mathrm{rb}} \leq_{\varnothing \twoheadrightarrow \mathbf{E}_{\mathrm{rb}}@\langle m_0 \rangle} \mathsf{Clight}(\mathsf{rb.c})\,.$$

This process of deriving the full-blown property from a minimal one can be easily streamlined.

Verifying bq.c. In Example 5.2, we were unable to state the relationship between the specification $\Sigma_{\text{bq}} : E_{\text{rb}} \twoheadrightarrow E_{\text{bq}}$ and the corresponding implementation $\mathsf{Clight}(\text{bq.c}) : \mathcal{C} \mathbin{@} \mathsf{mem} \twoheadrightarrow \mathcal{C} \mathbin{@} \mathsf{mem}$ due to their difference in type. We can now formulate the requirement

$$\phi'_{\text{bq}} : \Sigma_{\text{bq}} \mathbin{@} \mathsf{mem} \leq_{\mathbf{E}_{\text{rb}}@\mathsf{mem} \twoheadrightarrow \mathbf{E}_{\text{bq}}@\mathsf{mem}} \mathsf{Clight}(\text{bq.c})$$

which expresses that bq.c makes the outgoing calls prescribed by Σ_{bq} but does not modify the global memory state. To interface with the property ϕ_{rb}, its incoming simulation convention $\langle m_0 \rangle$ can easily be incorporated into the property as follows:

$$\phi_{\text{bq}} \;:=\; (\Sigma_{\text{bq}} \mathbin{@} \langle m_0 \rangle) \mathbin{⨟} \phi'_{\text{bq}} \;:\; \Sigma_{\text{bq}} \leq_{\mathbf{E}_{\text{rb}}@\langle m_0 \rangle \twoheadrightarrow \mathbf{E}_{\text{bq}}@\langle m_0 \rangle} \mathsf{Clight}(\text{bq.c})$$

Revisiting the challenge articulated in Example 5.2, we now give the complete proof:

$$\phi'_1 \mathbin{⨟} \big((\phi_{\text{bq}} \mathbin{⨟} \phi^{\text{cc}}_{\text{bq}}) \odot (\phi_{\text{rb}} \mathbin{⨟} \phi^{\text{cc}}_{\text{rb}}) \odot z\big) \mathbin{⨟} e \mathbin{⨟} \ell \quad : \quad \Gamma_{\text{bq}} \leq_{\varnothing \twoheadrightarrow (\mathbf{E}_{\text{bq}}@\langle m_0 \rangle) ⨟ \mathbb{C}} \mathsf{Asm}(\text{bq.s} + \text{rb.s})\,.$$

Remark 6.4 (Allocation permission). Defining a partial commutative monoid • which satisfies the properties above is largely straightforward, but some subtleties arise when it comes to memory allocation. In a real-world scenario, memory is a finite resource and calls to malloc can fail contingent upon the amount of memory available. Under an accurate model of memory as a finite resource, this would have to be taken into account by our notion of memory share and the definition of •. For example, memory states (m, k) could incorporate a number of bytes $k \in \mathbb{N}$ which remain to be allocated (or perhaps the size of the largest contiguous memory region available, to take fragmentation into account), and malloc would trigger an undefined behavior when the program attempts to allocate a region of size greater than k. In this case the partial monoid • could be defined along the lines

$$(m_1, k_1) \bullet (m_2, k_2) := (m_1 \circ m_2,\; k_1 + k_2)\,.$$

In this setting, incorporating an additional memory share could only increase the amount of memory available; the behavior of malloc on the larger, composite share would refine that of the original one, validating the frame property.

Since in CompCert, memory is modeled as an infinite resource and malloc always succeeds, this was not an issue in our implementation. However, CompCert memory states maintain a nextblock counter which is used to assign identifiers to newly allocated memory blocks. Since this counter increases as the program executes, in order to enable the frame property we must allow the nextblock counters of the two shares $m_1 \bullet m_2$ to become out of sync. But in this case, allocating a new block in the "stale" share would result in a naming conflict.

To work around this, we model memory allocation as a permission, such that at any given time, only one of the two shares is able to allocate new blocks and carries the up-to-date nextblock counter. This requires a slight modification to the CompCert memory model to incorporate this permission flag and allow Mem.alloc to fail when the flag is not set. This flag is subject to ownership transfer reasoning and similar techniques used in the context of separation logic—in our framework this can be accommodated by the refinement convention $\mathbf{Y}$, which allows partial memory shares to migrate between the two source-level branches at any time.

6.4 Modeling Loading and the Execution Environments

Verifying functionalities of library code substantially benefits from CompCertO's open semantics. However, the openness hinders reasoning on the behavior of executables. For the executables, it is desirable to model them in terms of the *process behavior*; the behaviors are self-contained, and can be characterized by the sequence of system calls they perform. To bridge the gap between the open semantics of the process behavior, we introduce the notion of a *loader*.

On one end, the loader launches the component as a process by using the $\mathsf{entry}_{\mathcal{A}} : \mathcal{A} \to \mathcal{P}$ to invoke its main function:

$$\mathsf{entry}_{\mathcal{A}} \vDash \mathsf{run} \rightarrowtail (\vec{rs}_0[\mathsf{PC} \mapsto \mathsf{main}, \mathsf{RA} \mapsto \mathsf{null}, \mathsf{RSP} \mapsto \mathsf{null}]@m_0 \rightsquigarrow \vec{rs}[\mathsf{RAX} \mapsto r]@m) \rightarrowtail r\,.$$

The registers $\vec{rs}_0$ and the memory m_0 are initialized that the program counter PC holds a pointer value that points to the main function, and the static variables are properly initialized in the memory. The return address RA and the stack pointer RSP are initialized to null according to CompCertO's simulation convention. At the end, the value stored in RAX is returned.

On the other end, the $\mathsf{runtime} : \mathcal{S} \to \mathcal{A}$ acts as the conduit for runtime libraries to interface the program with the operating system. In our scenario, the programs only use read and write functions from unistd.h to perform I/O operations. Thus, we implement the minimalist runtime:

$$\begin{aligned}
\mathsf{runtime}_{\mathcal{A}} \vDash{} & \vec{rs}[\mathsf{PC} \mapsto \mathsf{read}, \mathsf{RDI} \mapsto 0, \mathsf{RSI} \mapsto b, \mathsf{RDX} \mapsto n]@m[b \mapsto \mathit{unspecified}] \\
& \rightarrowtail (\mathsf{read}_0[n] \rightsquigarrow s) \rightarrowtail \vec{rs}'[\mathsf{RAX} \mapsto \mathsf{len}(s)]@m[b \mapsto s] \\
\mathsf{runtime}_{\mathcal{A}} \vDash{} & \vec{rs}[\mathsf{PC} \mapsto \mathsf{write}, \mathsf{RDI} \mapsto 1, \mathsf{RSI} \mapsto b, \mathsf{RDX} \mapsto n]@m[b \mapsto s] \\
& \rightarrowtail (\mathsf{write}_1[s[0:n]] \rightsquigarrow n') \rightarrowtail \vec{rs}'[\mathsf{RAX} \mapsto n']@m[b \mapsto s]
\end{aligned}$$

Following the x86 conventions, arguments are passed via the RDI, RSI, and RDX registers. The read function loads a sequence of bytes from the standard input, stores them into the memory where the pointer value b points to, and returns the length of the byte sequence. Conversely, the write function writes the first n bytes of the byte sequence s to the standard output, and the return value n' indicates the number of bytes that are successfully written.

The assembly loader can be obtained from:

$$\mathsf{load}_{\mathcal{A}}(L) := \mathsf{entry}_{\mathcal{A}} \odot [\![L]\!] \odot \mathsf{runtime}_{\mathcal{A}}\,.$$

With the assembly loader, we then formally formulate the property (1) as follows:

$$\Gamma_{(1)} \leq \mathsf{load}_{\mathcal{A}}(\mathsf{secret.s} + \mathsf{rot13.s}) \mid \mathsf{load}_{\mathcal{A}}(\mathsf{decode.s} + \mathsf{rot13.s})\,.$$

Verifying Loaded Programs. Reasoning about the process behavior directly at the level of assembly programs is intricate because of the large abstraction gap between the strategy-level specifications and the assembly semantics. Therefore, we also introduce a loader for the Clight semantics to divide the proof into manageable pieces. Furthermore, the loaders must transport the $\mathbb{C}$-related CompCertO simulations into simulation relations between process behaviors:

$$\mathsf{load}_C(-) : (C@\mathsf{mem} \twoheadrightarrow C@\mathsf{mem}) \to (\mathcal{S} \twoheadrightarrow \mathcal{P}) \qquad \frac{\phi : L_1 \leq_{\mathbb{C} \to \mathbb{C}} L_2}{\phi^{\ell} : \mathsf{load}_C(L_1) \leq \mathsf{load}_{\mathcal{A}}(L_2)}$$

Example 6.5. Revisiting the task articulated in property (1), we first define the program-level specifications $\Sigma_{\mathsf{secret}}, \Sigma_{\mathsf{decode}} : C@\mathsf{mem} \twoheadrightarrow C@\mathsf{mem}$, and prove they meet the strategy-level specifications via the loader:

$$\phi_{\mathsf{decode}} : \Gamma_{\mathsf{decode}} \leq \mathsf{load}_C(\Sigma_{\mathsf{decode}}) \qquad \phi_{\mathsf{secret}} : \Gamma_{\mathsf{secret}} \leq \mathsf{load}_C(\Sigma_{\mathsf{secret}})\,.$$

Then, the rest of the proof only involves the CompCertO semantics. In particular, the following properties state that the programs correctly implement their corresponding specifications:

$$\pi_{\mathsf{secret}} : \Sigma_{\mathsf{secret}} \leq L_{\mathsf{secret}} \oplus \mathsf{Clight}(\mathsf{rot13.c}) \qquad \pi'_{\mathsf{secret}} : L_{\mathsf{secret}} \leq_{\mathbb{C} \to \mathbb{C}} \mathsf{Asm}(\mathsf{secret.s})$$
$$\pi_{\mathsf{decode}} : \Sigma_{\mathsf{decode}} \leq \mathsf{Clight}(\mathsf{decode.c}) \oplus \mathsf{Clight}(\mathsf{rot13.c})\,.$$

where L_{secret} is a transition system defined in terms of the C language interface that captures the behavior of the assembly program secret.s. Combining the above simulations with CompCertO's compiler correctness, we obtain:

$$\begin{aligned}\psi_{\text{secret}} &:= \pi_{\text{secret}} \fatsemi (\pi'_{\text{secret}} \oplus \phi^{cc}_{\text{rot13}}) \fatsemi \ell \quad &:& \quad \Sigma_{\text{secret}} \leq_{\mathbb{C}\to\mathbb{C}} \mathsf{Asm}(\text{secret.s} + \text{rot13.s}) \\ \psi_{\text{decode}} &:= \pi_{\text{decode}} \fatsemi (\phi^{cc}_{\text{decode}} \oplus \phi^{cc}_{\text{rot13}}) \fatsemi \ell \quad &:& \quad \Sigma_{\text{decode}} \leq_{\mathbb{C}\to\mathbb{C}} \mathsf{Asm}(\text{decode.s} + \text{rot13.s})\,.\end{aligned}$$

Note the $\oplus$ operator here is CompCertO's linking operator, which should not be confused with the flat composition on strategies. Eventually, the property (1) is witnessed by the following proof:

$$\phi_{(1)} \fatsemi (\phi_{\text{secret}} \fatsemi \psi^{\ell}_{\text{secret}} \mid \phi_{\text{decode}} \fatsemi \psi^{\ell}_{\text{decode}}) : \Gamma_{(1)} \leq \mathsf{load}_{\mathcal{A}}(\text{secret.s+rot13.s}) \mid \mathsf{load}_{\mathcal{A}}(\text{decode.s+rot13.s})$$

The definitions of the Clight loader and the Clight level specifications, and the detailed proof of directly proving simulation between L_{secret} and $\mathsf{Asm}(\text{secret.s})$ can be found in Appendix G [Zhang et al. 2024b].

6.5 Certified Abstraction Layers

The bounded queue example in §2 was ad-hoc and relied on our framework as a versatile glue. However, in many contexts additional structure is preferable. The methodology of Gu et al. [2015] divides the code of a large system into standardized *certified abstraction layers*. The functionality exposed to client code at each layer is specified in a *layer interface*. Within the terms of our formalism, a layer interface is a set D of *abstract states* together with a specification $L : 0 \twoheadrightarrow C @ \mathsf{mem} @ D$. The semantics of client code then takes this *underlay* interface as a parameter:

$$L : 0 \twoheadrightarrow C @ \mathsf{mem} @ D \quad \vdash \quad \mathsf{Clight}_L[M] : 0 \twoheadrightarrow C @ \mathsf{mem} @ D\,. \tag{9}$$

A certified abstraction layer involves an *underlay* interface L_1, an *overlay* interface L_2, a program module M and an abstraction relation $R \subseteq D_2 \times (D_1 \times \mathsf{mem})$. They must satisfy the property:

$$L_1 \vdash_R M : L_2 \quad :\Leftrightarrow \quad L_2 \leq_{\varnothing \twoheadrightarrow C @ \hat{R}} \mathsf{Clight}_{L_1}[M]\,,$$

where $\hat{R} \subseteq (\mathsf{mem} \times D_2) \times (\mathsf{mem} \times D_1)$ extends R to a relationship between the entire states of the source and target programs. The main challenge is then to prove the vertical composition property

$$\frac{\psi_{12} \;:\; L_1 \vdash_R M : L_2 \qquad \psi_{23} \;:\; L_2 \vdash_S N : L_3}{\psi_{13} \;:\; L_1 \vdash_{R \cdot S} M + N : L_3} \tag{10}$$

Implementing Layers. This methodology is implemented in CompCertX, a modified version of CompCert where every language semantics and correctness proof has been updated to take into account the abstract state and underlay interface. A complex *memory injection* is used in $\hat{R}$ to express the embedding of the source memory into the target, alongside the concretized abstract state of the overlay. Finally, the proof of vertical compositionality is complex and largely monolithic, involving aspects of our frame property, CompCertO's linking theorem, and more.

By contrast, the toolbox provided by our framework makes it straightforward to formulate a comparable theory of certified abstraction layers. A layer-aware semantics can be defined as:

$$\mathsf{Clight}_L[M] := (\mathsf{Clight}(M) @ D) \odot L$$

and does not require any compiler change. Our memory join relation can be leveraged to define:

$$\hat{R} := (\mathsf{mem} @ R) \odot (\Upsilon @ D_1) \qquad R \cdot S := S \fatsemi (\mathsf{mem} @ R) \fatsemi (\Upsilon @ D_1)$$

such that the composition property $\alpha : (\hat{S} \fatsemi \hat{R}) \cong \widehat{R \cdot S}$ holds by associativity of the join operation $\bullet$. Finally, the vertical composition property (10) can be established with the single-line proof term:

$$\psi_{13} \;:=\; \alpha \odot \big(\psi_{23} \fatsemi \big((\mathsf{Clight}(N) @ R) \fatsemi (\mathsf{FP}(N) @ D_1)\big) \odot \psi_{12}\big)\,.$$

We provide additional details in Appendix D [Zhang et al. 2024b].

6.6 Clight with Module-Local State

Beyond verification-oriented applications, incorporating state encapsulation into CompCert semantics opens the door to new language features. As an example, we have defined a language called ClightP which supports encapsulated module-local state and can be soundly compiled to Clight.

Semantics. In ClightP, 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 semantics of a ClightP program M are defined using an underlying transition system of type:

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

We can then extract from the program M the initial private environment $p_0 = \mathsf{init\_penv}(M)$ and obtain the encapsulated semantics $\mathsf{ClightP}\langle M \rangle : \mathcal{C} \mathbin{@} \mathsf{mem} \rightarrow \mathcal{C} \mathbin{@} \mathsf{mem}$ as:

$$\mathsf{ClightP}\langle M \rangle := (\mathcal{C} \mathbin{@} \mathsf{mem} \mathbin{@} [p_0\rangle) \odot \mathsf{ClightP}(M)\,.$$

Note that the resulting type means that ClightP semantics in this form can be composed directly.

Compiling to Clight. We have defined a simple transformation $M' := \mathsf{ClightUnP}(M)$ which turns a ClightP program M into a regular Clight program M' by erasing the private annotations from all variables. We can then show the associated correctness property:

$$\mathsf{ClightP}\langle M \rangle \leq_{\mathcal{C}@\mathsf{mem}\circledast\langle\mathsf{mem}]^* \twoheadrightarrow \mathcal{C}@\mathsf{mem}\circledast[\mathsf{m}_0\rangle_*} \mathsf{Clight}(M')\,,$$

where m_0 is a memory share computed from M containing the initial values of its private variables. The incoming simulation convention $\mathsf{mem} \circledast [m_0\rangle_*$ requires m_0 to be added to the target global memory state. The outgoing convention $\mathsf{mem} \circledast \langle\mathsf{mem}]^*$ allows the target program to include this additional memory region into its outgoing calls, with a guarantee that it will not be changed.

Composition. One challenge is that the correctness property depicted above is not directly compositional, because the incoming and outgoing simulation conventions are different. Fortunately, the frame property for Clight ensures that the correctness properties for multiple ClightP translation units can be combined in a meaningful way. See Appendix E [Zhang et al. 2024b] for more details.

7 Related Work

Finally, we briefly discuss past and future research relevant to the work and goals we have described.

Interaction Trees. As a "semantics toolbox" of sorts, interaction trees share some goals and techniques with our model. In fact, an interaction tree $t : \mathsf{ITree}_E(X)$ can be interpreted into our framework as a strategy $\langle t \rangle : E \twoheadrightarrow \{* : X\}$. However, strategies generalize ITrees in several ways:

- Strategies are two-sided and encode incoming as well as outgoing interactions, forming the basis for layered composition.
- By design, ITrees must be executable programs, whereas strategies can be described logically using arbitrary Coq specifications.
- Strategies that exhibit the same external behavior are formally equal. By contrast, ITrees are compared using bisimulation equivalences. Equational reasoning requires Coq's setoid support, which can be slower and more fragile than rewriting with eq.
- Our strategies come with built-in notions of partial definition, refinement and data abstraction, whereas similar notions for ITrees have to be defined and tailored to a particular application.

Game Semantics. The horizontal fragment of our framework is a particularly simple form of game semantics. The framework's novelty resides in the vertical and spatial fragments, for which, to our knowledge, there exists no precedent in the game semantics literature. In particular, refinement conventions involve alternations of angelic and demonic choices; we were surprised to find they can be modeled using a fairly standard approach, although a rather unconventional ordering of plays must be used. An interesting question for further research would be to investigate how far this can be pushed and whether games more complex than effect signatures could admit their own forms of refinement conventions.

Refinement Calculus. The refinement calculus [Back and Wright 1998] was a source of inspiration for our framework. One defining feature of the refinement calculus is *dual nondeterminism*, which provides very powerful abstraction mechanisms. At the same time, models like predicate transformers do not deal with external interactions or state encapsulation.

CompCertO. The semantic model of CompCertO [Koenig and Shao 2021; Zhang et al. 2024c] introduced *simulation conventions* and the associated idea of a full-blown, two-dimensional refinement framework, so it is worth pointing out the ways in which our framework generalizes the CompCertO model, especially when it comes to refinement conventions:

- CompCertO transition systems and simulation conventions use explicit states and Kripe worlds in their definitions, whereas strategies and refinement conventions provide canonical representations for the components' observable behaviors.
- Effect signatures are more general than the language interfaces used in CompCertO, which force all questions to use the same set of answers.
- CompCertO transition systems do not retain any history between successive incoming questions; as such, they cannot support the kind of state encapsulation which our framework enables. Likewise, simulation conventions only specify 4-way relationships between isolated pairs of questions and answers, but unlike refinement conventions they cannot be sensitive to the history of the computation.

Other CompCert-based Verification Frameworks. CompCertM [Song et al. 2019] is another project which builds on CompCert to provide a compositional verification framework. Like CompCertO, it introduces a better model of the interaction between C and assembly programs and more flexibility in simulation conventions. However, while it permits some form of localized state, CompCertM still does not support full-blown data abstraction and state encapsulation of the kind we have presented. See Koenig and Shao [2021]; Zhang et al. [2024c] for a detailed comparison between Compositional CompCert, CompCertM and CompCertO.

We have also touched on certified abstraction layers and CompCertX in §6.5. Subsequent work has extended CAL to support concurrency [Gu et al. 2018]. There are more recent treatments of CAL which, like our work, attempt to streamline the underlying theory [Koenig and Shao 2020; Oliveira Vale et al. 2022], but this work has not been mechanized or interfaced with CompCert.

Separation Logic. For the most part, the frameworks discussed above do not provide program-level verification facilities, but rather focus on a more coarse-grained, module-level "glue". Likewise, we have assumed that elementary module correctness properties such as ϕ_1, ϕ_2 and $\phi_{\mathsf{bq}}^{\mathsf{min}}$ were provided by the user[2] and focused on the problem of connecting such proofs. Nevertheless, program logics in general and separation logic in particular are relevant to our work in the following ways.

First, it would be beneficial to incorporate such program logics into our framework. For example, Gu et al. [2015] provides a rudimentary Clight program logic which can be used to help prove

[2]Our example is simple enough that, in our implementation, manual simulation proofs were sufficient.

abstraction layers correct. It may be useful to investigate whether the Clight separation logic provided by the Verified Software Toolchain [Appel 2011] could be interfaced with our model.

Secondly, spatial composition is in fact the defining feature of separation logic. Our treatment of memory separation draws extensively from separation algebra [Calcagno et al. 2007], an approach to building models of separation logic. More recently, Conditional Contextual Refinement (CCR) [Song et al. 2023] combined (vertical) refinement and (spatial) separation logic into a unified, mechanized framework. CCR however does not support state encapsulation or certified compilation.

Multi-language Semantics. We have demonstrated that our framework is able to reason across languages through non-trivial examples such as the one in Fig 1. In Compositional CompCert and CompCertM, assembly programs are given C-level semantics, making it possible to directly reason about composite programs (but only for Asm code, which behaves according to the C calling convention). CAL uses the opposite approach and can translate C-level layer specification into assembly behaviors. Recent work on the DimSum framework [Sammler et al. 2023] attempts to give a more general account of multi-language semantics by introducing wrappers to translate between different languages.

These various approaches all attempt to represent *horizontally* what the simulation conventions of CompCertO represent vertically. In our framework, the notions of companion and conjoint could provide a natural way to formalize approaches of this kind, so that, for example, the CompCertO calling convention $\mathbb{C} : \mathcal{C} \leftrightarrow \mathcal{A}$ would be in companion/conjoint relationships with adapter components $\mathbb{C}_* : \mathcal{A} \twoheadrightarrow \mathcal{C}$ and $\mathbb{C}^* : \mathcal{C} \twoheadrightarrow \mathcal{A}$. The complexity of CompCertO's convention as presently stated makes this challenging, but we do not believe it to be a fundamental issue.

Event-based Semantics. The DimSum framework [Sammler et al. 2023] employs a language-agnostic, event-based semantics as a generic framework for multi-language semantics. Both the DimSum framework and our strategy model feature rich compositional structures, and support private states across function invocations. However, there are several key differences set DimSum apart from our approach. First, DimSum introduces explicit angelic and demonic nondeterminism alongside events. These nondeterministic structures facilitate the transformation and ordering of event sequences at different abstraction levels. However, this also adds complexity due to the intricate commuting properties between events and nondeterministic choices. In contrast, our strategy model adheres to a transitional approach where plays solely consist of events. Here, dual nondeterminism is concealed within the construction of refinement conventions and simulations, activating only when necessary. Second, events in DimSum are not well-bracketed, allowing for modeling complex interactions such as coroutines, which are challenging to implement within our current strategy model. Generalization to asynchronous games semantics would be required to accommodate such behaviors. Third, the DimSum framework does not support spacial composition. Instead, data abstraction must go through the semantics wrapper, which is a rather heavy mechanism. Lastly, the DimSum framework features a four-pass compiler that translates idealized source- and target-level programs. By contrast, our strategy model integrates a realistic optimizing compiler that compiles C program into assembly.

8 Conclusion

Combining compositional semantics, abstraction, encapsulation and certified compilation is an important step towards the construction of large-scale systems certified end-to-end. Moreover, we believe that the underlying algebraic structures that we have uncovered in this process constitute an elegant conceptual framework with applications beyond the present work, and may become an important facet of future certified systems engineering work.

Acknowledgments

We wish to thank the anonymous POPL 2025 reviewers, whose feedback significantly improved the quality of this paper. This material is based upon work supported in part by NSF grants 2313433 and 2019285 and by the National Natural Science Foundation of China (NSFC) under Grant No. 62372290 and 62002217. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

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

Ralph-Johan Back and Joakim von Wright. 1998. *Refinement Calculus: A Systematic Introduction*. Springer, New York. https://doi.org/10.1007/978-1-4612-1674-2

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 407–419. https://doi.org/10.1145/1328438.1328487

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 366–378. https://doi.org/10.1109/LICS.2007.30

Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational Integration Verification of a Cryptographic Server. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1704–1729.

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

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

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A program logic for verified interoperability between OCaml and C. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 716–744.

Jérémie Koenig. 2016–2024. *Coqrel: a binary logical relations library for the Coq proof assistant.* https://github.com/CertiKOS/coqrel

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

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.

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.

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

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. *ACM SIGPLAN Notices* 42, 1 (2007), 3–10.

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

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (Jan. 2023), 31 pages. https://doi.org/10.1145/3571220

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

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (Jan. 2023), 31 pages. https://doi.org/10.1145/3571232

Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2024c. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules. *Proc. ACM Program. Lang.* 8, POPL, Article 72 (Jan. 2024), 31 pages. https://doi.org/10.1145/3632914

Yu Zhang, Jérémie Koenig, Yuting Wang, and Zhong Shao. 2024a. *Unifying compositional verification and certified compilation with a three-dimensional refinement algebra (artifact).* https://doi.org/10.5281/zenodo.14202535

Yu Zhang, Jérémie Koenig, Yuting Wang, and Zhong Shao. 2024b. *Unifying compositional verification and certified compilation with a three-dimensional refinement algebra (extended version).* Technical Report YALEU/DCS/TR1572. Yale University.

Received 2024-07-11; accepted 2024-11-07