CRIS: The Power of Imagination in Hybrid Verification

YONGHEE KIM, Seoul National University, Korea
TAEYOUNG YOON, Seoul National University, Korea
SANGHYUN YI, Seoul National University, Korea
JAEHYUNG LEE, Seoul National University, Korea
SOONWON MOON, Seoul National University, Korea
YEJI HAN, Seoul National University, Korea
SEONHO LEE, Seoul National University, Korea
TAEYOUNG RHEE, Seoul National University, Korea
YUJIN IM, Seoul National University, Korea
DONGHYUN NAM, Seoul National University, Korea
JIEUNG KIM, Yonsei University, Korea
CHUNG-KIL HUR, Seoul National University, Korea

The CCR framework unifies refinement and separation logic to provide ownership-based modular reasoning and transitive incremental reasoning in open settings that involve unverified code. However, when reasoning about function invocations, the reasoning principles available to clients remain confined to pre- and postconditions, which struggle to capture effectful behaviors such as I/O actions or interactions with arbitrary unverified code. This limitation becomes particularly acute in *hybrid verification*, where a mixture of different verification techniques is applied and some code is never formally verified but instead tested or model-checked.

To overcome this limitation, we introduce *imaginary specifications*—a novel notion that freely mixes executable code with ownership assertions—and reasoning principles for their use. Through key technical developments, we present CRIS (Contextual Refinement with Imaginary Specifications), a framework that generalizes CCR with support for imaginary specifications. We demonstrate CRIS's expressiveness and reasoning power through examples involving hybrid verification with unverified code exhibiting arbitrary side effects such as I/O or divergence, with complete mechanization in Rocq.

CCS Concepts: • **Theory of computation → Program specifications**; **Program verification**; **Separation logic**.

Additional Key Words and Phrases: hybrid verification, imaginary specification, contextual refinement, Rocq

ACM Reference Format:
Yonghee Kim, Taeyoung Yoon, Sanghyun Yi, Jaehyung Lee, Soonwon Moon, Yeji Han, Seonho Lee, Taeyoung Rhee, Yujin Im, Donghyun Nam, Jieung Kim, and Chung-Kil Hur. 2026. CRIS: The Power of Imagination in Hybrid Verification. *Proc. ACM Program. Lang.* 10, PLDI, Article 239 (June 2026), 24 pages. https://doi.org/10.1145/3808317

Authors' Contact Information: Yonghee Kim, Seoul National University, Korea, yonghee.kim@sf.snu.ac.kr; Taeyoung Yoon, Seoul National University, Korea, taeyoung.yoon@sf.snu.ac.kr; Sanghyun Yi, Seoul National University, Korea, sanghyun.yi@sf.snu.ac.kr; Jaehyung Lee, Seoul National University, Korea, jaehyung.lee@sf.snu.ac.kr; Soonwon Moon, Seoul National University, Korea, soonwon.moon@sf.snu.ac.kr; Yeji Han, Seoul National University, Korea, yeji.han@sf.snu.ac.kr; Seonho Lee, Seoul National University, Korea, seonho.lee@sf.snu.ac.kr; Taeyoung Rhee, Seoul National University, Korea, taeyoung.rhee@sf.snu.ac.kr; Yujin Im, Seoul National University, Korea, yujin.im@sf.snu.ac.kr; Donghyun Nam, Seoul National University, Korea, donghyun.nam@sf.snu.ac.kr; Jieung Kim, Yonsei University, Korea, jieungkim@yonsei.ac.kr; Chung-Kil Hur, Seoul National University, Korea, gil.hur@sf.snu.ac.kr.

ACM 2475-1421/2026/6-ART239
https://doi.org/10.1145/3808317

1 Introduction

Formal software verification has traditionally relied on two main approaches—refinement and separation logic—each with distinct advantages. CCR (Conditional Contextual Refinement) [29] recently unified these approaches by combining their strengths. In this paper, we present *CRIS (Contextual Refinement with Imaginary Specifications)*, which advances beyond CCR by creating a deeper synergy: we fuse the different notions of specification from both approaches into the novel concept of *imaginary specification* and demonstrate its power.

1.1 Refinement, Separation Logic and CCR

Before presenting the main contributions, this section reviews the three foundational approaches underlying our work: refinement, separation logic, and CCR. We particularly emphasize the expressiveness and usability of their specifications.

Refinement. Refinement-based approaches [13, 34, 35] specify concrete programs using abstract programs, which we simply call *abstractions*. We verify an implementation I against its abstraction A by constructing a simulation relation between I and A, which establishes that the observable behaviors of I refine (*i.e.,* are included in) those of A (denoted $I \sqsubseteq_{\mathsf{beh}} A$).

Refinement offers several key advantages. First, it supports transitive proofs: refinement can be incrementally established through intermediate abstractions ($I \sqsubseteq_{\mathsf{beh}} A_0 \sqsubseteq_{\mathsf{beh}} A_1 \sqsubseteq_{\mathsf{beh}} \cdots \sqsubseteq_{\mathsf{beh}} A$), enabling *incremental verification*. Second, it enables *open-setting verification* through contextual refinement (denoted $I \sqsubseteq_{\mathtt{ctx}} A$). Contextual refinement establishes that for any context C, the refinement $C[I] \sqsubseteq_{\mathsf{beh}} C[A]$ holds, allowing programs containing unverified components C to have formal correctness statements. Furthermore, since specifications are themselves executable programs, verification of $C[A]$ (including unverified parts) can leverage alternative verification techniques such as static analysis, testing, or model checking.

However, traditional refinement approaches are limited in their reasoning principles. They rely on simple simulation-based reasoning between programs, lacking powerful principles like the ownership-based reasoning found in separation logic.

Separation Logic. Separation logic-based approaches [1, 18, 26] specify concrete programs using pre- and postconditions $\langle P, Q\rangle$ that can express exclusive or shared ownership of various abilities such as memory access, network communication, and function calls. We verify an implementation I against its pre- and postconditions $\langle P, Q\rangle$ by constructing proofs in separation logic (denoted $\{P\}I\{Q\}$), which establishes that if we run I under any state satisfying P with the ownership specified by P, then I safely executes without violating ownership rules, and if it terminates, the resulting state satisfies Q, thereby providing the ownership specified by Q.

A key advantage of separation logic is its support for powerful *ownership-based verification* through the frame rule, which enables modular reasoning. It represents various abilities as *resources* and tracks ownership of these resources, ensuring that each part of a program performs only the abilities it owns. The separating conjunction $*$ then enables independent reasoning about disjoint resources, yielding the frame rule that preserves unowned resources.

However, unlike refinement-based approaches, separation logic does not support incremental and open-setting verification. Specifications are not executable programs, limiting the ability to verify systems with unverified components or to leverage alternative verification techniques like testing.

Conditional Contextual Refinement (CCR). Among those works that combine both refinement and separation logic [3, 6, 7, 9–12, 28–30, 32], only CCR [29] and CCR 2.0 [28] enable incremental, open-setting, and ownership-based verification simultaneously; CCR 2.0 further improves incremental reasoning over CCR (see §8). A CCR specification consists of a pair $A : \langle P, Q\rangle$, where A is an abstract program and $\langle P, Q\rangle$ are separation logic pre- and postconditions. Verification of an

(* $I_{\texttt{Cell}}$ *)

```
Module Cell.
private cv = 0;
def set(cb) ≡
  cv := cb();
def get() ≡
  ret cv;
End Cell.
```

$\forall x, \{\texttt{cell}(x)\}$
`set(cb)`
$\{\exists y, \texttt{cell}(y)\}$

$\forall x, \{\texttt{cell}(x)\}$
$r = \texttt{get}()$
$\{(r = x) * \texttt{cell}(x)\}$

(* $A_{\texttt{Cell}}$ *)

```
Module Cell.
def set(cb) ≡
1 x = take(ℤ);
2 Assume(cell(x));
3 i = cb();
4 Guarantee(cell(i));
End Cell.
```

```
def get() ≡
1 x = take(ℤ);
2 Assume(cell(x));
3 Guarantee(cell(x));
4 ret x;
```

Fig. 1. Implementation, separation logic spec, and imaginary spec of `Cell` module.

(* $I_{\texttt{Main}}$ *)

```
Module Main.
def cb() ≡ ...
def main() ≡
1 Cell.set(Main.cb);
2 foo();
3 i = Cell.get();
4 print(i);
End Main.
```

(* $I_{\texttt{Ctx}}$ *)

```
...
def foo() ≡
  ...
  // does not
  // use Cell
...
```

(* $T_{\texttt{Main}}$ *)

```
Module Main.
def cb() ≡ ...
def main() ≡

  i = Main.cb();
  foo();
  print(i);
End Main.
```

(* $A_{\texttt{Main}}$ *)

```
Module Main.
def cb() ≡ ...
def main() ≡
1 Assume(cell(0));
2 i = Main.cb();
3 foo();
4 print(i);
End Main.
```

Fig. 2. Implementation, top-level abstraction, and imaginary spec of `Main` module.

implementation I_1 against a specification $A_1 : \langle P_1, Q_1\rangle$, denoted $I_1 \sqsubseteq_{\texttt{ctx}} A_1 : \langle P_1, Q_1\rangle$, establishes that for any context C, the behavior of $C[I_1]$ refines that of $C[\mathcal{W}(A_1 : \langle P_1, Q_1\rangle)]$. Here the wrapper $\mathcal{W}$ encodes the CCR specification into an executable program by operationally decorating an executable program A with separation-logic conditions $\langle P, Q\rangle$ using *dual nondeterminism* [4].

CCR's encoding of correctness as standard contextual refinement provides two immediate benefits. First, it naturally supports incremental verification via the transitivity of contextual refinement. Second, it enables open-setting verification by establishing refinement under arbitrary contexts C.

Beyond these benefits, CCR enables ownership-based verification (as found in separation logic) through conditional specifications, still expressed in the form of contextual refinement. To see how this works, suppose that the implementation I_1 is the body of a function `f`, and we have another implementation I_2 that invokes `f`. In CCR, one can verify I_2 against its specification $A_2 : \langle P_2, Q_2\rangle$ while relying on the separation-logic condition $\langle P_1, Q_1\rangle$ of the function `f`, thereby enabling ownership-based verification. CCR encodes this conditional verification as contextual refinement, denoted $I_2 \sqsubseteq_{\texttt{ctx}} \mathcal{W}(\texttt{f} : \langle P_1, Q_1\rangle \vdash A_2 : \langle P_2, Q_2\rangle)$, by generalizing the wrapper $\mathcal{W}$ to also operationally encode the separation logic condition $\texttt{f} : \langle P_1, Q_1\rangle$.

1.2 Limitations of Pre/Postcondition Specifications in Hybrid Verification

While CCR successfully combines the advantages of both refinement and separation logic, it still relies on separation logic-style pre- and postconditions as the specification provided to clients of a function. As discussed earlier, when a function `f` has a CCR specification $A : \langle P, Q\rangle$, clients of `f` can only see the pre- and postcondition part $\langle P, Q\rangle$ as its specification, not the abstraction part A. In other words, the abstraction part A is only used to describe the observable behavior of the function `f`, not as a specification for clients.

However, this pre- and postcondition-style specification has limitations when it comes to *hybrid verification*, where a mixture of different verification techniques is applied. When verifying a large system, one often wants to formally verify only a critical part while applying other methodologies—such as testing or model checking—to the rest. In such cases, pre- and postconditions are insufficient: for instance, when formally verifying a library that takes a callback, we cannot assume any pre- and

postconditions about the callback if the actual function given as a callback will never be formally verified but instead remains as concrete code to be tested or model-checked.

To illustrate, consider the `Cell` module in Fig. 1 and its client in Fig. 2. The `Cell` module (Fig. 1, first column) maintains a private variable `cv`, accessible only through `set` and `get`. The function `set` updates `cv` with the return value from the callback function `cb` given as an argument, while `get` returns `cv`'s current value. Importantly, `Cell` makes no assumptions about the callback function `cb`—it may be arbitrary code that performs I/O, accesses databases, recursively calls back into `Cell`, or even crashes. The `Main` module (Fig. 2, first column) uses `Cell` as follows: it first calls `Cell.set` with `Main.cb` as the callback, which stores the return value of `Main.cb` into the cell; then calls an external function `foo`; then retrieves the stored value via `Cell.get`; and finally prints it. The context (Fig. 2, second column) defines `foo`, which performs some computation involving I/O without using `Cell`.

In hybrid verification, our goal is to formally verify the `Cell` module and abstract it away, yielding a simpler top-level program that can then be tested. Concretely, we want to prove that $I_{\texttt{Main}} \circ I_{\texttt{Cell}} \circ I_{\texttt{Ctx}}$ behaviorally refines $T_{\texttt{Main}} \circ I_{\texttt{Ctx}}$ (Fig. 2, third column), where the `Cell` module has been abstracted away and `Main` reduces to directly calling `Main.cb`, then `foo`, and printing the return value of `Main.cb`. One can then test $T_{\texttt{Main}} \circ I_{\texttt{Ctx}}$ to check whether `Main.cb` and `foo` work as expected—since the top-level code is simpler, testing at this level is more efficient than testing against the original implementation.

To achieve this, we need to give a specification of the `Cell` module, which provides reasoning principles for `set` and `get` that clients can rely on without examining the module's implementation. With pre- and postconditions, these functions can be specified using an *ownership predicate*[1] `cell` as shown in the second column of Fig. 1. This ownership predicate provides two key properties. First, it ensures exclusive access: only the owner of $\texttt{cell}(x)$ can invoke `set` and `get`, since both require it as a precondition and ownership cannot be duplicated. Second, $\texttt{cell}(x)$ encodes the invariant that the `cv` variable of the `Cell` module contains the value x. This property is witnessed by the postcondition of `get`, which guarantees that the return value r equals x when $\texttt{cell}(x)$ is provided as a precondition. Note that both `set` and `get` return ownership of `cell` in their postconditions to allow continued access.

However, this pre- and postcondition specification is insufficient for reasoning about callbacks, and we recall a traditional remedy. The postcondition $\exists y,\ \texttt{cell}(y)$ of `set` indicates that `cv` changes to some new value y, but does not express that this value comes from calling `cb`. The traditional approach to address this is to assume that `cb` satisfies a client-provided pre- and postcondition, and then use this assumption to give a more precise specification of `set`.

Unfortunately, this traditional remedy does not apply in this hybrid verification setting. The assumption that `Main.cb` satisfies a pre- and postcondition cannot be discharged, since `Main.cb` is not formally verified but only tested. This limitation calls for a richer specification mechanism that can provide precise ownership-based reasoning principles for `set` without requiring the callback itself to be formally verified.

1.3 CRIS

The limitations of pre- and postcondition-style reasoning principles motivate our framework, *CRIS*. The key innovation of CRIS is a novel notion of specification, called *imaginary specifications*, providing greater expressive power and reasoning principles than pre- and postconditions.

CRIS's imaginary specifications generalize CCR's specifications in two significant ways. First, unlike CCR's specifications $A : \langle P, Q \rangle$ where the abstraction A and the pre- and postconditions $\langle P, Q \rangle$

[1] In the literature, these are called separation logic assertions or Iris propositions [18].

are completely separated (*i.e.,* they cannot depend on each other), CRIS's imaginary specifications freely mix executable code and ownership assertions, allowing them to depend on each other. Second, and more importantly, while CCR uses the abstraction A to describe observable behaviors and the pre- and postcondition $\langle P, Q\rangle$ to provide reasoning principles to clients, CRIS's imaginary specifications serve both roles simultaneously: they both describe observable behaviors and provide reasoning principles.

To achieve this integration, CRIS uses four special operators that serve as primitive building blocks for expressing ownership assertions within programs: **`take`**, **`choose`**, **`Assume`**, and **`Guarantee`**. The operators **`take`** and **`choose`**, inherited from CCR [29], represent dual forms of nondeterminism: **`take`** imaginarily receives an arbitrary value (serving as universal quantification ∀), while **`choose`** makes a concrete selection (serving as existential quantification ∃). The operators **`Assume`** and **`Guarantee`** handle ownership transfer: **`Assume`** acquires ownership of resources (analogous to requiring a precondition), while **`Guarantee`** releases ownership (analogous to establishing a postcondition). Note that while CCR 2.0 [28] also employs similar constructs, CRIS interprets **`Assume`** and **`Guarantee`** differently in a way that enables the meaningful integration of code and ownership assertions.

Equipped with these operators, we can now express the `Cell` module's behavior using imaginary specifications, as shown in the third column of Fig. 1. The `get` specification demonstrates how standard pre- and postconditions translate into this framework. It imaginarily takes an arbitrary integer x (line 1) and assumes ownership of `cell(`x`)` (line 2), then returns the value x (line 4) while guaranteeing ownership of `cell(`x`)` (line 3). More significantly, the `set` specification demonstrates the distinctive power of imaginary specifications. While maintaining the pre- and postcondition structure, it embeds the call to `cb` between the assertions, explicitly capturing that the updated value comes from this call. It imaginarily takes a value x (line 1) and assumes ownership of `cell(`x`)` (line 2), then invokes `cb()` to obtain a return value i (line 3) and guarantees ownership of `cell(`i`)` (line 4). Note that the postcondition `cell(i)` depends on the program code `i = cb()`, and this dependency is precisely what pre- and postconditions alone cannot express.

Since imaginary specifications represent a new form of specification, we also need a new reasoning principle for clients. The reasoning principle follows naturally from the nature of imaginary specifications as abstract programs: specifications are inlined at call sites. For example, when verifying a client calling `set`, we replace the call to `set` with its imaginary specification, treating the inlined specification as if it were the client's own code (see §5 for more details).

However, this inlining approach creates a fundamental tension with supporting module-local reasoning. Both CCR and CRIS support reasoning about a module's private state via module-local invariants. Yet, inlining across modules seemingly breaks this modularity because the inlining principle treats code inlined from a different module as if it were part of the caller's own module.

We resolve this tension through carefully designed module semantics and simulation relations. We make the inlining principle sound by allowing **`Assume`** and **`Guarantee`** in inlined specifications to access the caller module's own resources, while simultaneously preserving support for module-local invariants (see §6 for details).

To demonstrate the power of imaginary specifications, we verify the motivating example given in Figs. 1 and 2, where we use the imaginary specification for `Main` given in the fourth column of Fig. 2 as an intermediate step in the verification (§3). We also provide a hybrid specification of a memory module that allows clients to freely choose between pre- and postcondition specifications or operational specifications (§7).

In summary, this paper introduces *imaginary specifications*, a novel notion of specification that enables *hybrid verification* where a mixture of different verification techniques is applied. Imaginary specifications extend pre- and postcondition-style specifications to express interactions with arbitrary unverified code involving any side effects such as I/O or nontermination. We present

the CRIS framework with formal semantics and proof rules, demonstrate its expressive power through examples involving unverified code, and provide a complete mechanization in Rocq [19].

2 Background

This section introduces the essential background on dual nondeterminism, resource algebras, and interaction trees that underpin our approach.

2.1 Dual Nondeterminism

Dual nondeterminism introduces two complementary forms of nondeterministic choice: demonic and angelic [4]. In the following, we explain these two notions by comparing how they affect the observable behaviors of programs. Before we proceed, we suppose that the semantics of a program p is given by the set of behaviors $\mathrm{Beh}(p)$, consisting of finite or infinite sequences of I/O events that the program exhibits.

Demonic Nondeterminism. Demonic nondeterminism, triggered by the **choose** operator, is the standard form of nondeterminism found in programming languages: when a program reaches a branching point, it may evolve into any one of several possible next states. Formally, the semantics of **choose** is defined as the union of the behaviors arising from each branch:

$$\mathrm{Beh}(\mathtt{x} = \textbf{choose}(X);\ K[\mathtt{x}]) = \bigcup_{x \in X} \mathrm{Beh}(K[x])$$

The following example demonstrates demonic nondeterminism:

$$\mathrm{Beh}(\mathtt{c} = \textbf{choose}(\{1, 2\});\ \mathtt{print(c)}) = \{\, \mathtt{print}(1), \mathtt{print}(2) \,\}$$

A value c is nondeterministically chosen from a given set and then printed. The overall behavior set is obtained by taking the union of the behaviors from each possible choice.

Angelic Nondeterminism. Angelic nondeterminism, triggered by the **take** operator, is the dual of demonic nondeterminism. Formally, its semantics is defined by taking the intersection of the behaviors of each branch, retaining only the behaviors common to all possible executions.

$$\mathrm{Beh}(\mathtt{x} = \textbf{take}(X);\ K[\mathtt{x}]) = \bigcap_{x \in X} \mathrm{Beh}(K[x])$$

The following example demonstrates angelic nondeterminism:

$$\mathrm{Beh}\left(\begin{array}{l}\mathtt{b} = \textbf{take}(\{\mathtt{true}, \mathtt{false}\});\ \mathtt{if\ (b)}\ \{\ \mathtt{c} = \textbf{choose}(\{1, 2\});\ \mathtt{print(c);}\ \}\\ \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \ \{\ \mathtt{c} = \textbf{choose}(\{2, 3\});\ \mathtt{print(c);}\ \}\end{array}\right) = \{\, \mathrm{print}(2) \,\}$$

Here, **take** constrains the behaviors to those common across both branches, leaving only $\mathtt{print}(2)$.

Implementability of Dual Nondeterminism. The two forms of nondeterminism differ in real-world implementability. Union-based (demonic) nondeterminism is directly implementable: any single branch yields a semantically valid behavior. Intersection-based (angelic) nondeterminism, while mathematically well-defined, cannot be realized in general: determining which behaviors survive requires comparing all branches simultaneously. Accordingly, CRIS treats **choose** as implementable, while restricting **take** to *imaginary specifications*, where executability in the mathematical sense suffices.

2.2 Resources and Ownership Predicates

We briefly review resources and ownership predicates in Iris [18]. Resources form a resource algebra R equipped with a commutative addition operator $\cdot : R \times R \to R$ and a validity predicate $\mathcal{V} : R \to \mathbf{Prop}$. For the example in Fig. 1, we define valid resources $\mathtt{cell}(x)$, $\mathtt{cellA}(x)$, $\mathtt{cellF}(x)$ for all integers x, and an invalid resource $\bot$, with addition defined by:

- $\mathtt{cell}(x) \cdot \mathtt{cellA}(x) = \mathtt{cellA}(x) \cdot \mathtt{cell}(x) = \mathtt{cellF}(x)$ for every x;

- $\sigma_1 \cdot \sigma_2 = \bot$ for all other cases.

A global validity invariant requires that the addition of all resources owned by all components must be valid, ensuring that component interactions never violate system correctness. Each component may only update its own resources in ways that preserve validity, preventing any component from invalidating others' assumptions.

In the example, `Cell` always owns[2] $\texttt{cellA}(x)$ when `cv` contains x, while the client owns $\texttt{cell}(x)$. Global validity forces both to agree on x, preventing the client from independently changing $\texttt{cell}(x)$ to $\texttt{cell}(x')$ since $\texttt{cell}(x') \cdot \texttt{cellA}(x) = \bot$. On a call to `set`, $\texttt{cell}(x)$ is transferred to `Cell`, giving it $\texttt{cellF}(x)$, which it can update to $\texttt{cellF}(y)$ for any y (choosing y to be the return value of `cb`), before splitting back into $\texttt{cell}(y)$ and $\texttt{cellA}(y)$ and returning $\texttt{cell}(y)$ to the client.

A collection of resource algebras is combined into a global resource algebra Σ. An *ownership predicate* $P \in \mathbf{iProp}\,\Sigma$ is a predicate over Σ satisfying a monotonicity condition. Pre- and postconditions are expressed as ownership predicates: a function with precondition P and postcondition Q receives a resource satisfying P and returns one satisfying Q. We overload $\texttt{cell}(x)$ to also denote the minimal ownership predicate containing the resource $\texttt{cell}(x)$.

3 Verification of Motivating Example

We now illustrate CRIS's reasoning principles—particularly the inlining principle for imaginary specifications—through the motivating example in Figs. 1 and 2.

Overall Proof Structure. Recall that our goal is to prove the following refinement for any $I_{\texttt{Ctx}}$:

$$\mathrm{Beh}(I_{\texttt{Main}} \circ I_{\texttt{Cell}} \circ I_{\texttt{Ctx}}) \subseteq \mathrm{Beh}(T_{\texttt{Main}} \circ I_{\texttt{Ctx}})$$

where $I_{\texttt{Cell}}$, $I_{\texttt{Main}}$, $I_{\texttt{Ctx}}$ are the implementations of `Cell`, `Main` and the context; $T_{\texttt{Main}}$ is the top-level specification of `Main`; and $\circ$ is the module linking operator, formally defined in §4.2.

Intuitively, this refinement holds as follows. When the function `foo` invokes any function of `Cell`, the refinement holds vacuously because the absence of `Cell` from the top-level spec triggers *undefined behavior*, which subsumes all possible behaviors. If `foo` does not use `Cell` (and in particular does not invoke `Cell.set()`), then `Cell.set()` is invoked only once and therefore the return value from `Cell.get()` should be equal to the return value from the first call to `cb()`, thereby printing the same value in both the implementation and top-level spec.

In CRIS, we decompose the above end-to-end refinement proof into the following sub-refinements via the intermediate imaginary specifications $A_{\texttt{Cell}}$ for `Cell` (third column of Fig. 1) and $A_{\texttt{Main}}$ for `Main` (fourth column of Fig. 2):

$$\begin{aligned} I_{\texttt{Cell}} &\sqsubseteq_{\texttt{ctx}} A_{\texttt{Cell}} && (\mathrm{Ref}_{\texttt{Cell}}) \\ I_{\texttt{Main}} \circ A_{\texttt{Cell}} &\sqsubseteq_{\texttt{ctx}} A_{\texttt{Main}} \circ A_{\texttt{Cell}} && (\mathrm{Ref}_{\texttt{Main}}) \\ A_{\texttt{Main}} \circ A_{\texttt{Cell}} \circ I_{\texttt{Ctx}} &\sqsubseteq_{\texttt{beh}} T_{\texttt{Main}} \circ I_{\texttt{Ctx}} && (\mathrm{Ref}_{\mathrm{Cancel}}) \end{aligned}$$

Note that $A_{\texttt{Main}}$ is the same as $T_{\texttt{Main}}$ except that it starts with imaginarily acquiring $\texttt{cell}(0)$. The refinement $\mathrm{Ref}_{\texttt{Cell}}$ allows us to modularly reason about `Cell` alone using our simulation relation (formally defined in §5), which yields contextual refinement. $\mathrm{Ref}_{\texttt{Main}}$ allows us to modularly reason about `Main` using the simulation relation, relying on the imaginary specification $A_{\texttt{Cell}}$ of `Cell`. Finally, the global refinement $\mathrm{Ref}_{\mathrm{Cancel}}$ eliminates all imaginary operations by eliminating the `Cell` module and the **`Assume`** command in $A_{\texttt{Main}}$.

This cancellation refinement is fundamentally different from the other two: it is discharged automatically by the *cancellation theorem* (§5.3) and applied only as the last step to the whole closed program without any context (*i.e.,* $\sqsubseteq_{\texttt{beh}}$ instead of $\sqsubseteq_{\texttt{ctx}}$). Although the cancellation step holds for arbitrary $I_{\texttt{Ctx}}$ in this example, this is not the case in general: the cancellation theorem is applied to

[2]The notion of a module owning a resource is needed for local reasoning about the module's private state [29].

the whole program without any completely unknown context, and one may need to verify that context functions do not invoke functions with non-trivial preconditions. We discuss $\mathrm{Ref}_{\mathtt{Cancel}}$ in more detail in §5.3.

Composing these refinements by transitivity (with appropriate contexts at each step) establishes the end-to-end refinement:

$$I_{\mathtt{Main}} \circ I_{\mathtt{Cell}} \circ I_{\mathtt{Ctx}} \sqsubseteq_{\mathtt{ctx}} I_{\mathtt{Main}} \circ A_{\mathtt{Cell}} \circ I_{\mathtt{Ctx}} \sqsubseteq_{\mathtt{ctx}} A_{\mathtt{Main}} \circ A_{\mathtt{Cell}} \circ I_{\mathtt{Ctx}} \sqsubseteq_{\mathtt{beh}} T_{\mathtt{Main}} \circ I_{\mathtt{Ctx}}$$

yielding $\mathrm{Beh}(I_{\mathtt{Main}} \circ I_{\mathtt{Cell}} \circ I_{\mathtt{Ctx}}) \subseteq \mathrm{Beh}(T_{\mathtt{Main}} \circ I_{\mathtt{Ctx}})$.

This refinement proof highlights several key aspects of CRIS reasoning. First, it requires *partially open reasoning*. For example, the refinement $\mathrm{Ref}_{\mathtt{Main}}$ relies on a complete specification of the `Cell` module but leaves the implementations of the functions `Main.cb` and `foo` completely open. Second, it requires *ownership-based reasoning*. The simulation proof of refinement $\mathrm{Ref}_{\mathtt{Main}}$ logically relies on the assumption that `foo` does not invoke `Cell.set()` because otherwise the top-level spec and implementation could print different values. As in separation logic, we encode such logical assumptions via the `cell`-related resources. Third, it requires *inlining reasoning*. The simulation proof of $\mathrm{Ref}_{\mathtt{Main}}$ employs the imaginary specification $A_{\mathtt{Cell}}$ of `Cell` by inlining the specification.

Proof of $\mathrm{Ref}_{\mathtt{Cell}}$. We examine the sub-refinement $\mathrm{Ref}_{\mathtt{Cell}}$, which illustrates local reasoning about a module's private state via a module-local invariant. CRIS allows each module to define its own private state—such as `cv`—that is inaccessible to other modules, and to expose abstract predicates—such as `cell`—that give other modules a logical abstraction over that state. Crucially, such abstraction is possible only when the private state is physically protected from context modules: in a hybrid setting where context code is unverified, any unprotected state can be arbitrarily modified by the context, invalidating the abstraction.

The correctness of this abstraction—between the private state `cv` and the abstract predicate `cell`—is established by the refinement proof $\mathrm{Ref}_{\mathtt{Cell}}$. For this, our simulation relation takes a *module-local invariant* $\mathbb{I} : \mathrm{State} \to \mathrm{State} \to \mathbf{iProp}\,\Sigma$ as a parameter, specifying what resources the module must own given its source and target states. The relation then relates runtime configurations (consisting of a module-local state and program code) on the source and target sides, enforcing $\mathbb{I}$ at each interaction point (*i.e.,* the entry and return points and each call site).

To prove $\mathrm{Ref}_{\mathtt{Cell}}$, we instantiate $\mathbb{I}$ to assert that whenever the target's `cv` variable holds x, the resource $\mathtt{cellA}(x)$ must be owned. Using the intuition behind the `cell`-related resources from §2.2, one can then establish the simulation between $I_{\mathtt{Cell}}$ and $A_{\mathtt{Cell}}$ via the simulation relation formalized in §5.1. We refer to CCR [29] for further details on this style of module-local abstraction.

Proof of $\mathrm{Ref}_{\mathtt{Main}}$. We show how to prove the refinement $\mathrm{Ref}_{\mathtt{Main}}$ in detail since it uses the inlining principle. For this refinement, we use the trivial module-local invariant $\lambda \_\ \_ .\ \ulcorner \mathtt{True} \urcorner$ since `Main` has no module-local state. The proof repeatedly uses the **`take`**, **`Assume`**, and **`Guarantee`** cases of the simulation relation formalized later in §5.1, so we present simplified forms here.

$$\textsc{take-src}\ \frac{\forall x \in X.\ t \lesssim K\ x}{t \lesssim (\mathtt{x} = \mathtt{take}(X); K\ \mathtt{x})} \qquad \textsc{choose-src}\ \frac{\exists x \in X.\ t \lesssim K\ x}{t \lesssim (\mathtt{x} = \mathtt{choose}(X); K\ \mathtt{x})} \qquad \textsc{assume-src}\ \frac{P \mathrel{-\!\!*} (t \lesssim s)}{t \lesssim (\mathtt{Assume}(P); s)} \qquad \textsc{guarantee-src}\ \frac{P\ *\ (t \lesssim s)}{t \lesssim (\mathtt{Guarantee}(P); s)}$$

$$\textsc{take-tgt}\ \frac{\exists x \in X.\ K\ x \lesssim s}{(\mathtt{x} = \mathtt{take}(X); K\ \mathtt{x}) \lesssim s} \qquad \textsc{choose-tgt}\ \frac{\forall x \in X.\ K\ x \lesssim s}{(\mathtt{x} = \mathtt{choose}(X); K\ \mathtt{x}) \lesssim s} \qquad \textsc{assume-tgt}\ \frac{P\ *\ (t \lesssim s)}{(\mathtt{Assume}(P); t) \lesssim s} \qquad \textsc{guarantee-tgt}\ \frac{P \mathrel{-\!\!*} (t \lesssim s)}{(\mathtt{Guarantee}(P); t) \lesssim s}$$

The first row applies to source-side steps, and the second row to target-side steps. For **`take`** and **`choose`**, a value is selected universally or existentially; the quantifier is swapped between the source and target sides, and between the two operations. For **`Assume`** and **`Guarantee`**, ownership

Target		Resource	Source	
		{ emp }	`Assume (cell(0));`	// add cell(0)
`Cell.set(Main.cb);`	// inline Cell.set	{ cell(0) }		
`x = take(ℤ);`	// select 0	{ cell(0) }		
`Assume (cell(x));`	// remove cell(0)	{ cell(0) }		
`i = Main.cb();`	// call Main.cb & get i	{ emp }	`i = Main.cb();`	// call Main.cb & get i
`Guarantee (cell(i));`	// add cell(i)	{ emp }		
`foo();`	// call foo	{ cell(i) }	`foo();`	// call foo
`i = Cell.get();`	// inline Cell.get	{ cell(i) }		
`x = take(ℤ);`	// select i	{ cell(i) }		
`Assume (cell(x));`	// remove cell(i)	{ cell(i) }		
`Guarantee (cell(x));`	// add cell(i)	{ emp }		
`return x;`	// return i	{ cell(i) }		
`print(i);`	// print i	{ cell(i) }	`print(i);`	// print i

Fig. 3. Simulation proof outline for $\text{Ref}_{\texttt{Main}}$.

predicate P is acquired or consumed using the magic wand $-\!\!*$ and the separating conjunction $*$, respectively; again, the roles are swapped between the two sides and between the two operations.

Fig. 3 presents the outline of the simulation proof. The **Resource** column shows the resource owned by the simulation relation at each step. The grey vertical bars mark the inlined `Cell` code on the target side, and the grey comments describe each step in the simulation proof.

The proof proceeds as follows. We start with both sides at their initial states owning the empty resource. The source begins with `Assume`(`cell(0)`), which adds `cell(0)` to the owned resource by ASSUME-SRC. The target inlines `Cell.set(Main.cb)` using the imaginary specification in $A_{\texttt{Cell}}$, and executes two steps: it selects `x` = 0 by TAKE-TGT and removes `cell(0)` by ASSUME-TGT. Both sides then take a matching step: calling `Main.cb()` and receiving the same return value i. The target gains ownership of `cell`(i) by GUARANTEE-TGT, completing the execution of the inlined code. It is this inlining-based reasoning that breaks in CCR [29] and CCR 2.0 [28] (see §6 for details).

Both sides then take another matching step: calling `foo()`, which returns no value. Since we maintain `cell`(i) as our resource throughout, the resource remains unchanged after the call, meaning that `Cell` still contains i. It may seem surprising that this is sound, since `foo` is arbitrary code that may call `Cell.set`, thereby changing the state of `Cell`. The reason for soundness is that if `foo` calls `Cell.set` or `Cell.get` on the source side, it executes their imaginary specifications in $A_{\texttt{Cell}}$, which attempt to acquire `cell(_)`. This conflicts with the existing `cell`(i), triggering undefined behavior. Since undefined behavior exhibits all possible behaviors, the simulation holds vacuously.

The target then inlines `Cell.get()` using the imaginary specification in $A_{\texttt{Cell}}$, and executes four steps: it selects `x` = i, removes `cell`(i), re-acquires `cell`(i), and returns i, completing the execution of the inlined code. Finally, both sides take a matching step: printing the same value i, which completes the simulation proof.

The key insight is that imaginary ownership transfer via `Assume` and `Guarantee`, combined with inlining, ensures that the state of `Cell` is fully tracked by the resource flow throughout the simulation. From the client's perspective, the resource `cell`(i) serves a dual role: it acts as a logical witness for the value stored in `Cell`, and as permission to call `Cell.set` and `Cell.get`.

4 CRIS Framework

This section establishes the formal foundation of CRIS by defining its modules and semantics, with particular attention to the operational encodings of `Assume` and `Guarantee`.

$$\textbf{Mod} \ni \mathrm{M} \triangleq \{(\mathsf{scopes}, \mathsf{funs}, \mathsf{init}) \mid \mathsf{scopes} \in \mathtt{list}\ \mathtt{scope} \wedge \mathsf{funs} \in \mathtt{list}\ \mathtt{func} \wedge \mathsf{init} \in \mathtt{list}\ \mathtt{state} \wedge \mathtt{wellscoped}\}$$

$$\mathtt{func} \triangleq \{(\mathsf{name}, \mathsf{scopes}, \mathsf{body}) \mid \mathsf{name} \in \mathtt{string} \wedge \mathsf{scopes} \in \mathtt{list}\ \mathtt{scope} \wedge \mathsf{body} \in \mathtt{Any} \to \mathtt{itree}\ \mathsf{E}_{\mathrm{mod}}\ \mathtt{Any}\}$$

$$\mathtt{scope} \triangleq \mathtt{string} \qquad \mathtt{state} \triangleq \{(\mathsf{key}, \mathsf{value}) \mid \mathsf{key} \in \mathtt{scope} \times \mathtt{string} \wedge \mathsf{value} \in \mathtt{Any}\}$$

$$\mathrm{M}.\mathtt{wellscoped} \triangleq (\forall \mathrm{f} \in \mathrm{M}.\mathsf{funs},\ \mathrm{f}.\mathsf{scopes} \subseteq \mathrm{M}.\mathsf{scopes}) \wedge (\forall \mathrm{st} \in \mathrm{M}.\mathsf{init},\ \mathrm{st}.\mathsf{key}.1 \in \mathrm{M}.\mathsf{scopes}) \wedge (\mathrm{NoDup}(\mathrm{M}.\mathsf{scopes}) \to \mathrm{NoDup}([s.\mathsf{key} \mid s \in \mathrm{M}.\mathsf{init}]))$$

$$\mathrm{WF}(\mathrm{M}) \triangleq \mathrm{NoDup}(\mathrm{M}.\mathsf{scopes}) \ \wedge \ \mathrm{NoDup}([f.\mathsf{name} \mid f \in \mathrm{M}.\mathsf{funs}])$$

Fig. 4. Definitions of modules.

4.1 Interaction Tree

We begin by reviewing interaction trees (ITrees) [31], which CRIS uses to represent effectful computations—those involving, *e.g.,* nondeterminism and I/O. An interaction tree is a coinductive structure that models computation as a potentially infinite tree of interactions with the environment. Specifically, the set of ITrees `itree` $E\ R$ is parameterized by an event type $E : \mathsf{Type} \to \mathsf{Type}$ and a return type R. Each node represents one of three kinds of computation: (i) an internal, unobservable deterministic step τ; (ii) termination with a return value of type R; or (iii) an interaction with the environment via an external event $e : E\ X$ for some response type X, together with a continuation for each possible response $x \in X$ (*i.e.,* a map from X to `itree` $E\ R$). With this structure, interaction trees form a monad, supporting standard monadic operations such as **`ret`** v to return a value and $i >>= k$ (or $x = i; k\ x$) to sequence computations via bind. Moreover, ITrees can be extracted to executable code (*e.g.,* OCaml), making them amenable to testing.

4.2 Module Definition

Modules. Fig. 4 shows the definition of a CRIS module M, which consists of three components scopes, funs, and init satisfying well-scopedness:

- scopes: the list of scopes that are permitted to be accessed in this module.
- funs: a list of triples (name, scopes, body) consisting of a function name, the list of scopes the function is permitted to access, and its body of type `Any` $\to$ `itree` $\mathsf{E}_{\mathrm{mod}}$ `Any`, where `Any` is the set of all mathematical values (*i.e.,* $\Sigma_{T \in \mathsf{Type}}\ T$) and $\mathsf{E}_{\mathrm{mod}}$ is the event type defined in §4.3.
- init: the module's initial state, given as a (partially defined) key-value map whose keys are pairs (scope, variable name) and whose values have type `Any`.

The predicate `wellscoped`(M) requires: (i) the scopes of all functions in M must be included in M.scopes; (ii) the scope of each key in M.init must occur in M.scopes; and (iii) if M.scopes is duplication-free, then the keys of M.init must also be duplication-free, permitting identical variable names in distinct scopes.

The core idea of our module system is to allow multiple modules to be treated as a single unified module without sacrificing isolation. Module composition—well-formed only when the modules' scopes and function names are disjoint—performs element-wise concatenation of all module components. This composition is purely structural: once two modules are linked, the resulting module no longer records which component originated from which submodule.

Isolation is instead preserved through a scope-based state model. Each module-local state cell is stored under a scoped key, and every function declares the scopes it is allowed to access. A function may only read or update state associated with scopes appearing in its annotation; any out-of-scope access is blocked by the sandboxing mechanism described in §4.3. Thus, even after composition, modules remain logically separated: no function can interfere with another module's state.

Note that CRIS does not fix any specific programming language; instead, since interaction trees provide a general model of computation, different languages can be encoded at the user level.

$$\mathsf{E}_{\text{core}}(X) \triangleq \{\texttt{Choose}\ X\} \uplus \{\texttt{Take}\ X\} \uplus \{\texttt{IO}\ O\ I\ \mathit{fn}\ \mathit{arg} \mid O, I \in \texttt{Type} \land \mathit{fn} \in \texttt{string} \land \mathit{arg} \in O \land X = I\}$$
$$\mathsf{E}_{\text{state}}(X) \triangleq \{\texttt{Put}\ k\ v \mid k \in \texttt{scope}\times\texttt{string} \land v \in \texttt{Any} \land X = \texttt{unit}\} \uplus \{\texttt{Get}\ k \mid k \in \texttt{scope}\times\texttt{string} \land X = \texttt{Any}\}$$
$$\mathsf{E}_{\text{ctrl}}(X) \triangleq \{\texttt{Call}\ \mathit{fn}\ \mathit{arg} \mid \mathit{fn} \in \texttt{string},\ \mathit{arg} \in \texttt{Any} \land X = \texttt{Any}\}$$
$$\mathsf{E}_{\text{logic}}(X) \triangleq \{\texttt{Assume}\ P \mid P \in \mathbf{iProp}\,\Sigma \land X = \texttt{unit}\} \uplus \{\texttt{Guarantee}\ P \mid P \in \mathbf{iProp}\,\Sigma \land X = \texttt{unit}\}$$
$$\mathsf{E}_{\text{mod}}(X) \triangleq \mathsf{E}_{\text{core}}(X) \uplus \mathsf{E}_{\text{state}}(X) \uplus \mathsf{E}_{\text{ctrl}}(X) \uplus \mathsf{E}_{\text{logic}}(X)$$

Fig. 5. Event types in CRIS.

As a concrete example, we provide a simple imperative language, IMP, inherited from CCR [29], which includes a notion of values with function pointers and a memory module storing IMP values. Higher-order functions can also be handled (see Landin's knot example in the Rocq artifact [19]).

Well-formedness and Linking. A module is *well-formed* when (i) scopes contains no duplicates and (ii) all function names are pairwise distinct. Linking of modules, denoted $\circ$, is defined as list concatenation applied to each component (scopes, funs, and init) and therefore always succeeds, even when the modules share scopes or function names. Well-formedness is not enforced at linking time; rather, CRIS's key theorems require all modules involved to be well-formed. It is therefore the user's responsibility to verify that linked modules satisfy well-formedness in order to apply these theorems.

4.3 Semantics and Contextual Refinement

We model the semantics of CRIS using interaction trees parameterized by the event types defined in Fig. 5. At a high level, the semantics of a linked CRIS module is defined in two steps:

(1) The linked module is *translated* into a single global interaction tree containing only I/O, `Choose`, and `Take` events. During this translation, all auxiliary events—including `Assume`, `Guarantee`, and `Call` events—are interpreted and eliminated. Sandboxing is also realized at this stage (see **Sandboxing** below).
(2) The resulting global interaction tree is *interpreted* as a set of observable I/O traces using a coinductive relation Beh. At this stage, `Choose` and `Take` are interpreted as the union and intersection of their continuation branches, respectively.

The following paragraphs define the semantic ingredients of each step.

Events. E_{mod} is the event type that expresses the full range of operational and logical events relevant to CRIS, including state manipulation, control flow, and logical assertions. It serves as the basis for defining the operational semantics of modules in CRIS. Four broad categories of events are shown in Fig. 5:

(i) E_{core}: `Choose` and `Take` represent dual forms of nondeterministic choice, allowing the program to either select or be provided with a value from an arbitrary set X. `IO` represents observable I/O operations.
(ii) $\mathsf{E}_{\text{state}}$: `Put` and `Get` update and retrieve values from a module's local state, respectively.
(iii) E_{ctrl}: `Call` represents function invocation.
(iv) $\mathsf{E}_{\text{logic}}$: `Assume` and `Guarantee` embed separation logic assertions[3] into programs.

We also define shorthand notations for triggering these events in interaction trees:

$$\textbf{choose}(X) \triangleq \texttt{trigger(Choose}\ X\texttt{)} \quad \textbf{take}(X) \triangleq \texttt{trigger(Take}\ X\texttt{)} \quad f(v) \triangleq \texttt{trigger(Call}\ f\ v\texttt{)}$$

[3]Following Iris for separation logic, we use **iProp** but employ a non-step-indexed [12, 29] resource algebra [17] with the strong update modality—a restricted form of the standard update modality proposed by CCR 2.0 [28]. This restriction is essentially necessary for supporting transitive composition of conditional refinements. Although the standard update modality's additional flexibility is exploited in a few resources—such as those used for ghost name allocation in Iris—simple workarounds for all such cases have been found.

$\mathbf{io}(O, I, fn, arg) \triangleq \texttt{trigger}(\texttt{IO}\ O\ I\ fn\ arg) \qquad k := v \triangleq \texttt{trigger}(\texttt{Put}\ k\ v) \qquad k \triangleq \texttt{trigger}(\texttt{Get}\ k)$
$\texttt{Assume}(P) \triangleq \texttt{trigger}(\texttt{Assume}\ P) \qquad \texttt{Guarantee}(P) \triangleq \texttt{trigger}(\texttt{Guarantee}\ P)$

Translation. The translation carries a global state consisting of two components. The first component is a key-value map indexed by pairs of (scope, variable name), used to model module-local state and initialized with the linked module's init component. The second component is a *global resource repository* that stores a logical resource of type Σ. The repository is accessed exclusively via two internal events, **putres** and **getres**, which are emitted only by **Assume** and **Guarantee** during translation (see §4.4); consequently, only **Assume** and **Guarantee** can access the repository. The initial content of the repository is given as a parameter to the translation and is later quantified over in the definition of contextual refinement.

Once all modules are linked together into a module M and an initial resource σ for the global resource repository is given, the translation, denoted $\mathrm{trans}(M, \sigma)$, starts from the distinguished entry function and turns the interaction trees of type $\texttt{itree}\ \mathsf{E}_{\mathrm{mod}}\ \texttt{Any}$ in M into a single global interaction tree of type $\texttt{itree}\ \mathsf{E}_{\mathrm{core}}\ \texttt{Any}$. This translation corecursively eliminates auxiliary events at each step as follows:

(i) $\mathsf{E}_{\mathrm{logic}}$ events are compiled away using the operational encoding shown in Fig. 7.
(ii) $\mathsf{E}_{\mathrm{ctrl}}$ events are removed by inlining function bodies at each call site.
(iii) $\mathsf{E}_{\mathrm{state}}$ events are elaborated using the monadic computation power of ITrees in the style of a state monad.

Note that the `iter` combinator of interaction trees [31] plays a crucial role in representing nonterminating computation. Given a function $f : A \to \texttt{itree}\ E\ (A + B)$ and $a : A$, `iter` repeatedly applies f starting from a until it returns a value of type B, or diverges otherwise, yielding an interaction tree of type $\texttt{itree}\ E\ B$. $\mathrm{trans}(M, \sigma)$ is defined using `iter` since the resulting interaction tree may be nonterminating—due to, *e.g.,* step (ii) above for mutually recursive calls. Users can also use `iter` inside function bodies to express (possibly nonterminating) loops.

Sandboxing. Sandboxing is also realized during the translation. At every $\texttt{Put}\ k\ v$ or $\texttt{Get}\ k$ event in a function f, the translation inserts scope-checking code, $\texttt{assume}(k.\texttt{scope} \in S)$, where S is the scope annotation of f—inherited from the module to which f belongs—ensuring that the scope of the accessed variable k—inherited from the module to which k belongs—is included in S. This prevents a function of one module from reading or modifying variables belonging to another module. Crucially, since the sandboxing code is hard-coded into each function body during translation, inlining a function into another module's context does not change its sandboxing behavior: the function retains its original scope annotation regardless of where it is inlined.

Traces and Behavior. We formally define traces as finite or infinite sequences of I/O events as shown in Fig. 6. Infinite traces correspond to executions that produce I/O events indefinitely. Finite traces conclude in one of four ways: (i) normal termination with an `Any` value; (ii) divergence, where the program performs an infinite internal computation without producing further observable I/O; (iii) an I/O hang, where the program blocks indefinitely waiting for a response that never arrives; or (iv) abort due to an external signal (*e.g.,* a user interrupt). Note that finite τ steps are invisible in the observable I/O sequence, which is why divergence appears as an explicit concluding event rather than being represented by an infinite sequence of τ steps. Also, I/O hang (case (iii)) captures divergence on the environment side, distinguishing it from program-side divergence (case (ii)).

We write $\mathrm{Beh}(-)$ for the behavior of a program, defined as the set of traces obtained from an interaction tree over $\mathsf{E}_{\mathrm{core}}$, as shown in Fig. 6. It is defined via a mixed induction-coinduction (following [14]), where inductive cases are marked with ⬚ and coinductive cases with □. Abort can occur at any point during execution, modeling a user interrupt. The set of interaction trees

$$\text{ObsIO} \triangleq \{(\texttt{Interact}\ O\ I\ \mathit{fn}\ \mathit{arg}\ \mathit{ret}) \mid O, I \in \texttt{Type}, \mathit{fn} \in \texttt{string}, \mathit{arg} \in O, \mathit{ret} \in I\}$$

$$\text{ObsHang} \triangleq \{\texttt{Hang}\ O\ \mathit{fn}\ \mathit{arg} \mid O \in \texttt{Type}, \mathit{fn} \in \texttt{string}, \mathit{arg} \in O\}$$

$$\text{Trace} \overset{\text{coind}}{=} \{e :: \mathit{tr} \mid e \in \text{ObsIO}, \mathit{tr} \in \boxed{\text{Trace}}\} \uplus \{\text{Term}\ v \mid v \in \texttt{Any}\} \uplus \{\text{Diverge}\} \uplus \{o \mid o \in \text{ObsHang}\} \uplus \{\text{Abort}\}$$

$$\text{Beh}(\mathit{itr} \in \texttt{itree}\ \texttt{E}_{\text{core}}\ \texttt{Any}) \in \mathbb{P}(\text{Trace}) \overset{\text{ind-coind}}{=} \{\text{Abort}\} \cup \{\text{Diverge} \mid \mathit{itr} \in \text{div}\} \cup$$

$$\begin{array}{l}
\quad \textbf{match}\ \mathit{itr}\ \textbf{with} \\
\quad \mid \textbf{ret}\ v \Rrightarrow \{\text{Term}\ v\} \qquad\qquad \mid \textbf{tau} \gg\!= K \Rrightarrow \boxed{\text{Beh}(K())} \\
\quad \mid \textbf{choose}(X) \gg\!= K \Rrightarrow \bigcup_{x \in X} \boxed{\text{Beh}(K(x))} \qquad \mid \textbf{take}(X) \gg\!= K \Rrightarrow \bigcap_{x \in X} \boxed{\text{Beh}(K(x))} \\
\quad \mid \textbf{io}(O, I, \mathit{fn}, \mathit{arg}) \gg\!= K \Rrightarrow \{\texttt{Hang}\ O\ \mathit{fn}\ \mathit{arg}\} \cup \bigcup_{\mathit{ret} \in I} (\texttt{Interact}\ O\ I\ \mathit{fn}\ \mathit{arg}\ \mathit{ret}) :: \boxed{\text{Beh}(K(\mathit{ret}))} \\
\quad \textbf{end}
\end{array}$$

$$\text{div} \in \mathbb{P}(\texttt{itree}\ \texttt{E}_{\text{core}}\ \texttt{Any}) \overset{\text{coind}}{=} \{\ \textbf{tau} \gg\!= K \mid K\,() \in \boxed{\text{div}}\ \} \cup$$
$$\{\ \textbf{choose}(X) \gg\!= K \mid \exists x \in X.\, K(x) \in \boxed{\text{div}}\ \} \cup \{\ \textbf{take}(X) \gg\!= K \mid \forall x \in X.\, K(x) \in \boxed{\text{div}}\ \}$$

$$(M, P) \sqsubseteq_{\text{beh}} (M', P') \triangleq (\text{WF}(M) \to \text{WF}(M')) \wedge$$
$$(\forall \sigma'.\, (\texttt{Own}\ \sigma' \vdash P') \to \exists \sigma.\, (\texttt{Own}\ \sigma \vdash P) \wedge \text{Beh}(\text{trans}(M, \sigma)) \subseteq \text{Beh}(\text{trans}(M', \sigma')))$$

$$(M, P) \sqsubseteq_{\text{ctx}} (M', P') \triangleq \forall\, M_c \in \textbf{Mod},\ P_c \in \textbf{iProp}\,\Sigma,\ (M \circ M_c, P * P_c) \sqsubseteq_{\text{beh}} (M' \circ M_c, P' * P_c)$$

Fig. 6. Definitions of trace, behavior, and contextual refinement.

exhibiting silent divergence, denoted div, is separately defined coinductively. We interpret **choose** and **take** as the union and intersection of the behaviors of their continuation branches, respectively. In particular, executing **choose**($\emptyset$) yields the empty set of traces, which we call *No Behavior* (**NB**), whereas executing **take**($\emptyset$) yields the set of all traces, which we call *Undefined Behavior* (**UB**).

Contextual Refinement. CRIS defines behavioral and contextual refinement as shown in Fig. 6. Since a module's behaviors (defined via trans) depend on an initial resource for the global repository, we define a pair (M, P) consisting of a module M and an initial condition $P : \textbf{iProp}\,\Sigma$ as the unit of refinement. Intuitively, the initial condition P specifies the set of initial resources for which the execution of M is intended. Behavioral refinement $(M, P) \sqsubseteq_{\text{beh}} (M', P')$ states that (i) if M is well-formed then so is M', and (ii) if P' is consistent (*i.e.,* satisfied by some σ'), then P is also consistent (*i.e.,* satisfied by some σ), and the behaviors of $\text{trans}(M, \sigma)$ are included in those of $\text{trans}(M', \sigma')$. Contextual refinement $(M, P) \sqsubseteq_{\text{ctx}} (M', P')$ holds when, for all context modules M_c and context initial conditions P_c, we have $(M \circ M_c,\ P * P_c) \sqsubseteq_{\text{beh}} (M' \circ M_c,\ P' * P_c)$.

4.4 Assume and Guarantee

We now give details on the translation of **Assume** and **Guarantee**, which enable imaginary ownership transfer within the semantics. The key idea, inspired by CCR's wrapper encoding, is to virtualize ownership transfer via dual nondeterminism, adapted here for the setting of imaginary specifications.

In a nutshell, for an ownership predicate $P : \textbf{iProp}\,\Sigma$, the operation **Assume**(P) imaginarily receives a resource satisfying P from the context, and **Guarantee**(P) imaginarily sends a resource satisfying P to the context. To support this, an imaginary specification maintains a single global resource repository together with two internal operations: **getres**, which retrieves the current resource from the repository, and **putres**, which updates it with a new resource.

The precise definitions of **Assume** and **Guarantee** are given in Fig. 7. First, **assume**(P) for a normal proposition $P : \textbf{Prop}$ does nothing if P holds; otherwise it exhibits undefined behavior (**UB**). Dually, **guarantee**(P) does nothing if P holds; otherwise it exhibits no behavior (**NB**). For an ownership predicate $P \in \textbf{iProp}\,\Sigma$, the operation **Assume**(P) enlarges the resource in the global repository by imaginarily receiving an additional resource satisfying P, while **Guarantee**(P) shrinks it by imaginarily sending some resource satisfying P.

```
assume(P: Prop) ≜ if not P then take(∅)
Assume(P: iProp Σ) ≜
  σ_cur = getres();
  σ_new = take(Σ);
  assume(V(σ_new));
  assume(Own(σ_new) ⊢ |⇛ (P ∗ Own(σ_cur)));
  putres(σ_new)
```

```
guarantee(P: Prop) ≜ if not P then choose(∅)
Guarantee(P: iProp Σ) ≜
  σ_cur = getres();
  σ_new = choose(Σ);
  guarantee(V(σ_new));
  guarantee(Own(σ_cur) ⊢ |⇛ (P ∗ Own(σ_new)));
  putres(σ_new)
```

Fig. 7. Encoding of Assume and Guarantee.

More specifically, **Assume**(P) proceeds as follows:

1. **Get current resource:** It retrieves the current resource σ_{cur} from the global repository.
2. **Take a new resource:** It imaginarily takes a new resource σ_{new} via **take**(Σ).
3. **Assume validity:** It assumes that σ_{new} is valid. If not, this branch can be ignored in the intersection because it exhibits all possible behaviors by triggering **take**$(\emptyset)$.
4. **Assume ownership transfer constraint:** It assumes $\mathsf{Own}(\sigma_{\text{new}}) \vdash \dot{\Rightarrow}(P{*}\mathsf{Own}(\sigma_{\text{cur}}))$, meaning σ_{new} can be updated to the composition of a resource satisfying P and the original resource σ_{cur}. This formalizes the idea that we have received a resource satisfying P from the context.
5. **Update repository:** It updates the global repository with σ_{new}.

The operation **Guarantee**(P) works dually:

1. **Get current resource:** It retrieves the current resource σ_{cur} from the global repository.
2. **Choose a new resource:** It chooses a resource σ_{new} to retain via **choose**(Σ).
3. **Guarantee validity:** It guarantees that σ_{new} is valid. If not, this branch can be ignored in the union because it exhibits no behaviors by triggering **choose**$(\emptyset)$.
4. **Guarantee ownership transfer constraint:** It guarantees $\mathsf{Own}(\sigma_{\text{cur}}) \vdash \dot{\Rightarrow}(P * \mathsf{Own}(\sigma_{\text{new}}))$, meaning σ_{cur} can be split into a part satisfying P and the remaining part σ_{new}. This formalizes the idea that we have sent a resource satisfying P to the context.
5. **Update repository:** It updates the global repository with σ_{new}.

The key insight is that **Guarantee**(P) and **Assume**(P) are logically paired operations. **Guarantee**(P) removes a part satisfying P from the global repository, making room for the context to safely add it back via **Assume**(P). This process of shrinking and enlarging the global resource constitutes *imaginary ownership transfer* without any physical communication: components exchange resources logically, through **Guarantee** and **Assume**, rather than through any direct interaction.

Remark. The update modality in **Assume** is not essential: including or omitting it yields an equivalent definition. More specifically, the encoding of **Assume**(P) intersects each branch with a resource satisfying $\dot{\Rightarrow}(P * \mathsf{Own}(\sigma_{\text{cur}}))$ in the repository, and this is equivalent to intersecting with a resource satisfying $P * \mathsf{Own}(\sigma_{\text{cur}})$. This holds because repository resources can only be consumed by **Guarantee** assertions, which always contain the update modality.

The key observation is that holding a resource σ that is updatable to σ' is *more powerful* than holding σ' itself: whenever the repository resource is used by a **Guarantee**(Q) assertion, if σ' satisfies $\dot{\Rightarrow}(Q{*}\mathsf{Own}(\sigma_{\text{new}}))$, then so does σ, since σ can first be updated to σ'. Hence σ admits at least as many behaviors as σ', because σ' is more likely to violate **Guarantee**(Q) and exhibit no behavior. From this, it follows that the two intersections coincide: every σ satisfying $\dot{\Rightarrow}(P * \mathsf{Own}(\sigma_{\text{cur}}))$ can be updated to some σ' satisfying $P * \mathsf{Own}(\sigma_{\text{cur}})$, and since σ admits no fewer behaviors than σ', including σ in the intersection does not further restrict it beyond what σ' already imposes.

The update modality in **Guarantee**, on the other hand, is important: it enables a convenient lemma that allows the update modality to be freely eliminated from any hypothesis during simulation proofs (see §5 for details).

$$\textsc{tau-src}\ \frac{tgt \lesssim (st, s)}{tgt \lesssim (st, \mathbf{tau}; s)} \qquad \textsc{put-src}\ \frac{tgt \lesssim (st[k \mapsto v], s)}{tgt \lesssim (st, k := v; s)} \qquad \textsc{get-src}\ \frac{tgt \lesssim (st, K\ st[k])}{tgt \lesssim (st, k \gg\!= K)}$$

$$\textsc{inline-src}\ \frac{(f \mapsto \mathbf{F}) \in \Lambda_{\mathtt{src}} \quad tgt \lesssim (st, \mathbf{F}(v) \gg\!= K)}{tgt \lesssim (st, f(v) \gg\!= K)}$$

$$\textsc{tau-tgt}\ \frac{(st, t) \lesssim src}{(st, \mathbf{tau}; t) \lesssim src} \qquad \textsc{put-tgt}\ \frac{(st[k \mapsto v], t) \lesssim src}{(st, k := v; t) \lesssim src} \qquad \textsc{get-tgt}\ \frac{(st, K\ st[k]) \lesssim src}{(st, k \gg\!= K) \lesssim src}$$

$$\textsc{inline-tgt}\ \frac{(f \mapsto \mathbf{F}) \in \Lambda_{\mathtt{tgt}} \quad (st, \mathbf{F}(v) \gg\!= K) \lesssim src}{(st, f(v) \gg\!= K) \lesssim src}$$

$$\textsc{take-src}\ \frac{\forall x \in X.\ tgt \lesssim (st, K\ x)}{tgt \lesssim (st, \mathtt{take}(X) \gg\!= K)} \qquad \textsc{choose-src}\ \frac{\exists x \in X.\ tgt \lesssim (st, K\ x)}{tgt \lesssim (st, \mathtt{choose}(X) \gg\!= K)}$$

$$\textsc{assume-src}\ \frac{P \mathrel{-\!\!*} (tgt \lesssim (st, s))}{tgt \lesssim (st, \mathtt{Assume}(P); s)} \qquad \textsc{guarantee-src}\ \frac{P * (tgt \lesssim (st, s))}{tgt \lesssim (st, \mathtt{Guarantee}(P); s)}$$

$$\textsc{take-tgt}\ \frac{\exists x \in X.\ (st, K\ x) \lesssim src}{(st, \mathtt{take}(X) \gg\!= K) \lesssim src} \qquad \textsc{choose-tgt}\ \frac{\forall x \in X.\ (st, K\ x) \lesssim src}{(st, \mathtt{choose}(X) \gg\!= K) \lesssim src}$$

$$\textsc{assume-tgt}\ \frac{P * ((st, t) \lesssim src)}{(st, \mathtt{Assume}(P); t) \lesssim src} \qquad \textsc{guarantee-tgt}\ \frac{P \mathrel{-\!\!*} ((st, t) \lesssim src)}{(st, \mathtt{Guarantee}(P); t) \lesssim src}$$

$$\textsc{io}\ \frac{\forall r \in I, (st_{\mathtt{t}}, K_{\mathtt{t}}\ r) \lesssim (st_{\mathtt{s}}, K_{\mathtt{s}}\ r)}{(st_{\mathtt{t}}, \mathbf{io}\ O\ I\ f\ arg \gg\!= K_{\mathtt{t}}) \lesssim (st_{\mathtt{s}}, \mathbf{io}\ O\ I\ f\ arg \gg\!= K_{\mathtt{s}})} \qquad \textsc{call}\ \frac{\mathbb{I}\ st_{\mathtt{t}}\ st_{\mathtt{s}} * (\forall r, st'_{\mathtt{t}}, st'_{\mathtt{s}}.\ \mathbb{I}\ st'_{\mathtt{t}}\ st'_{\mathtt{s}} \mathrel{-\!\!*} (st'_{\mathtt{t}}, K_{\mathtt{t}}\ r) \lesssim (st'_{\mathtt{s}}, K_{\mathtt{s}}\ r))}{(st_{\mathtt{t}}, f(v) \gg\!= K_{\mathtt{t}}) \lesssim (st_{\mathtt{s}}, f(v) \gg\!= K_{\mathtt{s}})}$$

$$\textsc{ret}\ \frac{\Phi\ (st_{\mathtt{t}}, r_{\mathtt{t}})\ (st_{\mathtt{s}}, r_{\mathtt{s}})}{(st_{\mathtt{t}}, \mathbf{ret}\ r_{\mathtt{t}}) \lesssim (st_{\mathtt{s}}, \mathbf{ret}\ r_{\mathtt{s}})}$$

Fig. 8. Inference rules of the `isim` relation. Omitted parameters: function environments $\Lambda_{\mathtt{src}}$ and $\Lambda_{\mathtt{tgt}}$, module-local invariant $\mathbb{I}$, return type R, return relation Φ, progress flags $p_{\mathtt{t}}$ and $p_{\mathtt{s}}$, and coinductive hypothesis r.

5 Reasoning Principles of CRIS

This section presents the reasoning principles of CRIS via the simulation relation `isim` over **iProp** Σ and related theorems, enabling users to verify programs directly inside the Iris Proof Mode [22, 23].

5.1 Simulation Relation

The simulation relation `isim` is defined by a mixed induction-coinduction using Paco (parameterized coinduction) [14, 33], in the style of FreeSim by Cho et al. [5], with the rules given in Fig. 8. We write $tgt \lesssim src$ for `isim`, omitting the following parameters for brevity: function environments $\Lambda_{\mathtt{src}}$ and $\Lambda_{\mathtt{tgt}}$ mapping function names to their bodies; module-local invariant $\mathbb{I} : \mathtt{state} \to \mathtt{state} \to$ **iProp** Σ; return type $R : \mathtt{Type}$ and return relation $\Phi : \mathtt{state} \times R \to \mathtt{state} \times R \to$ **iProp** Σ; boolean progress flags $p_{\mathtt{t}}$ and $p_{\mathtt{s}}$; and coinductive hypothesis $r : (R : \mathtt{Type}) \to (\mathtt{state} \times R \to \mathtt{state} \times R \to$ **iProp** $\Sigma) \to \mathtt{bool} \to \mathtt{bool} \to$ **iProp** Σ. These parameters are explained when needed below.

The module-local invariant $\mathbb{I}$ is set by the user to express a module-local state invariant for a particular module. The return relation Φ is used when decomposing an `isim` proof via the bind lemma introduced below, and is set to the default return condition $\lambda\ (st_{\mathtt{t}}, r_{\mathtt{t}})\ (st_{\mathtt{s}}, r_{\mathtt{s}}).\ \mathbb{I}\ st_{\mathtt{t}}\ st_{\mathtt{s}} * \ulcorner r_{\mathtt{t}} = r_{\mathtt{s}} \urcorner$ when relating the whole body of each function.

Intuitively, `isim` is coinductive in structure, but uses induction to rule out unsound cases (according to the adequacy theorem given below), such as taking silent steps indefinitely on the source side while stuttering on the target side, and vice versa. For this, we use three parameters: r from Paco for accumulating coinductive hypotheses, and $p_{\mathtt{t}}$ and $p_{\mathtt{s}}$ from FreeSim for recording progress on the source and target sides, respectively. Specifically, r is used to express the coinduction principle for `isim` introduced below. The flags $p_{\mathtt{t}}$ and $p_{\mathtt{s}}$ are set to `true` when any $*$-SRC or $*$-TGT rule is applied (setting $p_{\mathtt{t}}$ and $p_{\mathtt{s}}$, respectively), or both are simultaneously set to `true` when IO or CALL is applied; once both are `true`, `isim` may take a coinductive step, at which point the coinductive hypotheses in r become available, after which $p_{\mathtt{t}}$ and $p_{\mathtt{s}}$ are reset to `false`.

One might wonder why sandboxing does not appear in Fig. 8. The reason is that the translation function in §4 first applies a sandboxing pass that transforms interaction trees of type `itree` $\mathtt{E}_{\mathtt{mod}}\ R$ into sandboxed interaction trees of the same type, inserting scope-checking code

$$\textsc{isim-upd}\ \frac{\dot{\Rrightarrow} tgt \lesssim src}{tgt \lesssim src} \qquad \textsc{isim-bind}\ \frac{(st_t, t) \lesssim_\Psi (st_s, s) \ *\ (\forall st_t, st_s, r_t, r_s.\ \Psi\ (st_t, r_t)\ (st_s, r_s) \mathrel{-\!\!*} (st_t, K_t\ r_t) \lesssim_\Phi (st_s, K_s\ r_s))}{(st_t, t \gg\!= K_t) \lesssim_\Phi (st_s, s \gg\!= K_s)}$$

$$\textsc{isim-coind}\ \frac{\forall r'.\ (\forall tgt, src.\ r\ tgt\ src \vdash r'\ tgt\ src) \to (\forall a.\ P\ a \vdash r'\ (st_t\ a, t\ a)\ (st_s\ a, s\ a)) \to \forall a.\ P\ a \vdash (st_t\ a, t\ a) \lesssim^{r'} (st_s\ a, s\ a)}{\forall a.\ P\ a \vdash (st_t\ a, t\ a) \lesssim^{r} (st_s\ a, s\ a)}$$

Fig. 9. Additional inference rules for `isim`.

(*i.e.,* **assume**$(k.\mathsf{scope} \in S)$) explicitly into each function body. The simulation relation `isim` is then defined over these sandboxed trees, so sandboxing is already baked into the trees being related. Contextual refinement, on the other hand, quantifies over arbitrary context code *before* the sandboxing pass, ensuring that sandboxing is enforced on all context code as well.

Inference Rules. The rules in Fig. 8 define `isim` and simultaneously serve as reasoning principles for constructing simulation proofs.

- TAU-SRC/TAU-TGT takes a silent step on the source or target side.
- PUT-SRC/PUT-TGT and GET-SRC/GET-TGT update or read a state cell on the source or target side.
- INLINE-SRC/INLINE-TGT looks up the called function in the function environment $\Lambda_{\mathsf{src}}/\Lambda_{\mathsf{tgt}}$ and inlines its body.
- TAKE-SRC/TAKE-TGT and CHOOSE-SRC/CHOOSE-TGT select a value universally or existentially; the quantifier is swapped between the source and target sides, and between **take** and **choose**.
- ASSUME-SRC/ASSUME-TGT and GUARANTEE-SRC/GUARANTEE-TGT acquire or consume ownership; again, the roles are swapped between the two sides, and between the two operations.
- IO requires identical I/O operations on both sides and continues with their continuations.
- CALL requires identical function calls on both sides, but additionally requires the module-local invariant $\mathbb{I}$ to hold before the call and guarantees it for the updated states after the call.
- RET requires the return condition Φ to hold and completes the simulation.

Inlining-based Reasoning. The core reasoning principle of CRIS is the inlining of imaginary specifications, carried out by the rules INLINE-SRC and INLINE-TGT. This mechanism allows arbitrary function bodies to be inlined as if they were part of the local continuation, without imposing any restrictions on their access to logical resources; physical state, meanwhile, remains isolated by the sandboxing mechanism (§4.3). As a result, once a function is inlined, it can be reasoned about exactly as the surrounding continuation.

Additional Inference Rules. Fig. 9 presents additional rules for `isim`, proved as lemmas. ISIM-UPD states that the update modality can be freely added to the simulation relation. Consequently, given an ownership hypothesis $\dot{\Rrightarrow} H$ in the context, one can drop the update modality by first applying ISIM-UPD to relax the goal to $\dot{\Rrightarrow} tgt \lesssim src$, and then reducing $\dot{\Rrightarrow} H$ to H and the goal back to $tgt \lesssim src$ via the standard IPM update tactics. ISIM-BIND allows users to split a simulation proof at a sequential composition point, with a chosen intermediate return relation Ψ. ISIM-COIND allows users to apply coinductive reasoning by accumulating the current goal into the coinductive hypothesis r; the accumulated hypothesis becomes available once progress has been made on both the source and target sides.

CRIS Proof Tactics. CRIS provides Rocq notations to pretty-print proof goals by hiding less important information, and Rocq tactics to minimize boilerplate and automate simulation reasoning—using these tactics, users rarely need to apply the inference rules directly. The key tactics are:

- `cStepsS`/`cStepsT` automatically takes all consecutive steps on the source/target side that require no user input.

$$\textsc{consequence}\ \frac{P \vdash P'}{(M, P') \sqsubseteq_{\mathtt{ctx}} (M, P)} \qquad \textsc{frame-cond}\ \frac{(M, P) \sqsubseteq_{\mathtt{ctx}} (M', P')}{(M, P * Q) \sqsubseteq_{\mathtt{ctx}} (M', P' * Q)} \qquad \textsc{frame-mod}\ \frac{(M, P) \sqsubseteq_{\mathtt{ctx}} (M', P')}{(M \circ M_c, P) \sqsubseteq_{\mathtt{ctx}} (M' \circ M_c, P')}$$

$$\textsc{compose-vertical}\ \frac{(M_a, P_a) \sqsubseteq_{\mathtt{ctx}} (M_b, P_b) \quad (M_b, P_b) \sqsubseteq_{\mathtt{ctx}} (M_c, P_c)}{(M_a, P_a) \sqsubseteq_{\mathtt{ctx}} (M_c, P_c)} \qquad \textsc{compose-horizontal}\ \frac{(M_a, P_a) \sqsubseteq_{\mathtt{ctx}} (M'_a, P'_a) \quad (M_b, P_b) \sqsubseteq_{\mathtt{ctx}} (M'_b, P'_b)}{(M_a \circ M_b, P_a * P_b) \sqsubseteq_{\mathtt{ctx}} (M'_a \circ M'_b, P'_a * P'_b)}$$

$$\textsc{compose-mix}\ \frac{(M_a \circ M_c, P_a) \sqsubseteq_{\mathtt{ctx}} (M'_a \circ M_c, P'_a) \qquad (M_b \circ M_c, P_b) \sqsubseteq_{\mathtt{ctx}} (M'_b \circ M_c, P'_b)}{(M_a \circ M_b \circ M_c, P_a * P_b) \sqsubseteq_{\mathtt{ctx}} (M'_a \circ M'_b \circ M_c, P'_a * P'_b)}$$

Fig. 10. Useful properties and composition lemmas for contextual refinement.

- `cForceS` x/`cForceT` x takes a single step on the source/target side by using x as user input.
- `cStep` applies either IO or RET.
- `cCall` Hs applies CALL, generating a subgoal to prove $\mathbb{I}$ using the hypotheses Hs.
- `cInlineS`/`cInlineT` inlines a function body at a call site on the source/target side.
- `cCoind` accumulates the current goal into the coinductive hypothesis via ISIM-COIND, and `cByCoind` closes the goal by applying the coinductive hypothesis.

Adequacy of Simulation. Contextual refinement follows from a simulation proof as follows.

THEOREM 5.1 (ADEQUACY). *Let M_s and M_t be two well-formed modules, and let $\mathbb{I}$ be a module-local invariant. Suppose that for every function f_s in M_s, there exists a function f_t of the same name in M_t such that $\forall st_t, st_s, v.\ \mathbb{I}\ st_t\ st_s \vdash (st_t, f_t\ v) \lesssim^{\mathbb{I}} (st_s, f_s\ v)$. Then $(M_t, emp) \sqsubseteq_{\mathtt{ctx}} (M_s, \mathbb{I}\ M_t.\mathtt{init}\ M_s.\mathtt{init})$.*

Compositions of Contextual Refinement. Proof manipulation and composition are performed at the level of contextual refinement using the lemmas in Fig. 10. The CONSEQUENCE rule strengthens the initial condition. The FRAME-COND and FRAME-MOD rules support framing over initial conditions and modules, respectively. The COMPOSE-VERTICAL rule gives transitivity: two refinements for the same module can be chained together. The COMPOSE-HORIZONTAL rule composes refinements for different modules in parallel. The COMPOSE-MIX rule enables mixed composition, allowing refinements that share a common context module M_c to be composed without duplicating M_c—supporting the reuse of shared library modules across different verification proofs.

5.2 Pre/Postcondition Wrappers and Derived Reasoning

As a user-level addition to CRIS, we adopt the function wrapper mechanism $\mathcal{W}$ from CCR [29] to support (i) reasoning about external functions using only their pre/postconditions, without inlining, in the style of CCR (see **Pre/Postcondition-Style Reasoning** below), and (ii) eliminating imaginary operations via the cancellation theorem (see §5.3).

Pre/Postcondition Wrappers. For each function f, the user applies the wrapper $\mathcal{W}_{\mathtt{fun}}$ shown in Fig. 11 by choosing a pre/postcondition triple s consisting of $W \in \mathtt{Type}$, $P \in W \times \mathtt{Any} \times \mathtt{Any}$, and $Q \in W \times \mathtt{Any} \times \mathtt{Any}$, and a pre/postcondition map S from function names to pre/postcondition triples. The wrapper $\mathcal{W}_{\mathtt{fun}}$ wraps f by (i) inserting the pre/postcondition from s at the beginning and end of f, and (ii) decorating every call to a function fn within f with its triple $S(fn)$, via the `interp` combinator for interaction trees [31] with the event handler $\mathcal{W}_{\mathtt{call}}$ shown in Fig. 11.

This encoding shows how values and ownership can be imaginarily transferred using imaginary operations. Specifically, at each call site to fn, $\mathcal{W}_{\mathtt{call}}$ imaginarily sends out metadata w, a virtual argument x_v, and the ownership specified by the precondition P of fn via **`choose`** and **`Guarantee`**, while physically passing a physical argument x related to x_v by P. Correspondingly, $\mathcal{W}_{\mathtt{fun}}$ causes the function fn to physically receive x and imaginarily receive w, x_v, and the ownership specified

```
W_fun(S)(s, f ∈ Any → itree E_mod Any) ≜
  λ x ∈ Any. let (W, P, Q) = s in
    (w, x_v) = take(W × Any);
    Assume(P(w, x_v, x));
    r_v = interp (W_call(S)) (f x_v);
    r = choose(Any);
    Guarantee(Q(w, r_v, r));
    ret r
```

```
W_call(S)(e ∈ E_mod) ≜ match e with Call fn x_v ⇒
    let (W, P, Q) = S(fn) in
    (w, x) = choose(W × Any);
    Guarantee(P(w, x_v, x));
    r = trigger(Call fn x);
    r_v = take(Any);
    Assume(Q(w, r_v, r));
    ret r_v          | _ ⇒ trigger(e) end
```

Fig. 11. Pre/postcondition wrappers for function definitions and call sites.

by its precondition P via **take** and **Assume**. On the return side, *fn* imaginarily sends out a virtual return value r_v and the ownership specified by its postcondition Q via **choose** and **Guarantee**, while physically returning a value r related to r_v by Q. Correspondingly, the caller physically receives r and imaginarily receives r_v and the ownership specified by the postcondition Q of *fn*.

We note a few important points, following CCR [29]. First, dual nondeterminism via **choose** and **take** is the key mechanism that enables logically transferring values and ownership between the caller and the callee. Second, the physical values x and r are those appearing in an implementation, while the related virtual values x_v and r_v are those appearing in a top-level specification. For example, in a program using linked lists, functions pass 64-bit pointer values in the implementation, while the top-level specification uses mathematical lists. To establish refinement between them, we use intermediate imaginary specifications where the concrete 64-bit pointers and abstract mathematical lists are related via physical values x, r, and virtual values x_v, r_v, as shown in Fig. 11 (see the Celliostk example in the Rocq artifact [19], which abstracts pointer values into mathematical lists). Finally, concrete implementations can be represented using the trivial pre/postcondition triple, which imposes only equality between physical and virtual values with no ownership transfer (*i.e.*, $W = \texttt{unit}$, $P((), x_v, x) = \ulcorner x_v = x \urcorner$, and $Q((), r_v, r) = \ulcorner r_v = r \urcorner$).

In the motivating example, all functions use the trivial wrapper except `Main.main` in $A_{\texttt{Main}}$, with precondition $\ulcorner x_v = x \urcorner$ $*$ **Assume**(`cell(0)`) and postcondition $\ulcorner r_v = r \urcorner$. Note that Figs. 1 and 2 present equivalent but simplified versions of the wrapped functions. In particular, the first line of `Main.main` in $A_{\texttt{Main}}$ (Fig. 2) is generated by the wrapper, while the imaginary operations in the `Cell` module are not generated by the wrapper but manually written, intended for inlining reasoning.

Pre/Postcondition-Style Reasoning. With the pre/postcondition wrapper, the user can also reason about external functions using only their pre/postconditions, without inlining. Specifically, at a call site, for the pre/postcondition triple (W, P, Q) of f (given from S) we have:

$$(st_t, r = f(x);\ K\ r) \lesssim (st_s, (w, x) = \textbf{choose}(W \times \text{Any});\ \textbf{Guarantee}(P(w, x_v, x));\ r = f(x);\ r_v = \textbf{take}(\text{Any});\ \textbf{Assume}(Q(w, r_v, r));\ K\ r)$$

Then we can proceed as follows: (*i*) prove $P(w, x_v, x)$ for chosen w and x; (*ii*) apply the CALL rule by proving $\mathbb{I}\ st_t\ st_s$; (*iii*) assume $\mathbb{I}\ st'_t\ st'_s$ for any updated states st'_t and st'_s, and $Q(w, r_v, r)$ for any virtual and physical return values r_v and r; and (*iv*) continue with $(st'_t, K\ r) \lesssim (st'_s, K\ r)$. Note that since CRIS provides both INLINE-SRC/INLINE-TGT and CALL, users may choose between inlining-based and pre/postcondition-style reasoning.

5.3 Elimination of Imaginary Operations

A top-level specification is typically a program free of imaginary operations. CRIS provides two ways to eliminate imaginary operations such as **take**, **Assume**, and **Guarantee** from specifications.

Elimination by Inlining. Imaginary operations may be reduced by simulation reasoning following inlining, together with elimination of the inlined function or module. As illustrated by the

proof of $\mathrm{Ref}_{\texttt{Main}}$ in §3, the imaginary operations in $A_{\texttt{Cell}}$ are eliminated by simulation reasoning after inlining, leaving only one imaginary operation in $A_{\texttt{Main}}$—the one generated by the pre/postcondition wrapper (§5.2)—and rendering $A_{\texttt{Cell}}$ dead code. One can then eliminate $A_{\texttt{Cell}}$ entirely: removing a function or module is always sound, since invoking a nonexistent function triggers **UB**.

Cancellation Theorem. Imaginary operations introduced by the pre/postcondition wrapper can also be eliminated automatically via the cancellation theorem, where $\mathcal{W}(S)(M)$ denotes the module derived from M by wrapping every function with the pre/postcondition map S.

Theorem 5.2 (Cancellation). *Let M be a well-formed module, S a pre/postcondition map, and $(W, P, Q) = S(f_{\texttt{init}})$ for the initial function $f_{\texttt{init}}$. For any $w \in W$ such that $\forall r_v, r.\ Q(w, r_v, r) \vdash \ulcorner r_v = r \urcorner$, we have $(\mathcal{W}(S)(M), I) \sqsubseteq_{\texttt{beh}} (M, I * P(w, (), ()))$ for any $I \in \textbf{\textit{iProp}}\,\Sigma$.*

This theorem is proved by eliminating the precondition of the initial function using the given initial resource, and then cancelling out all matching **choose**/**Guarantee** and **take**/**Assume** pairs at each call and return site. For example, $\mathrm{Ref}_{\mathrm{Cancel}}$ is proved by this theorem since **Assume**(`cell(0)`) in $A_{\texttt{Main}}$ is generated by the wrapper as the precondition of the initial function.

The cancellation theorem yields behavioral refinement rather than contextual refinement, since all functions must be wrapped with the same pre/postcondition map S, leaving no room for an additional context module. Nevertheless, unknown functions may still be accommodated as parts of the cancellation refinement by assigning them the trivial pre/postcondition triple (§5.2) in S, provided they invoke no function with a non-trivial pre/postcondition in S. This is illustrated by $\mathrm{Ref}_{\mathrm{Cancel}}$, which holds for arbitrary $I_{\texttt{Ctx}}$.

Note, however, that the reason why $I_{\texttt{Ctx}}$ invokes no function with a non-trivial pre/postcondition is special. Since `Main.main` is the only function with a non-trivial pre/postcondition in S, one would normally need to prove that `foo` does not invoke `Main.main`. In CRIS, however, the initial function is used only to start execution and cannot be called during execution, so this obligation is discharged automatically. If `Main.main` were not set as the initial function and a new initial function were defined to invoke `Main.main` instead, one would need to prove that `foo` does not invoke `Main.main`. This can be systematically achieved by wrapping `foo` with a wrapper that triggers **UB** on any call to `Main.main` (see the Cellio example in the Rocq artifact [19]).

6 Inlining vs. Module-Local Invariants

This section examines the tension between support for inlining and support for module-local invariants (§6.1), and explains how CRIS resolves it (§6.2).

6.1 Tension between Inlining and Module-Local Invariants

Inlining and module-local invariants impose seemingly conflicting requirements. To support inlining an imaginary specification f from a module M into a module M', we must allow the imaginary operations in f to access resources owned by M' across module boundaries. On the other hand, module-local invariants $\mathbb{I}$ rely on the absence of interference from other modules. In particular, the CALL rule guarantees $\mathbb{I}$ even after a call to an unknown function in a context module, which is sound only if arbitrary code in the unknown function cannot interfere with the resources owned by the caller module.

A natural approach to supporting module-local invariants is to maintain a separate resource repository for each module, rather than a single global repository shared among all modules. Separating repositories physically prevents functions of one module from accessing another module's resources. This is indeed the approach taken by CCR [29] and CCR 2.0 [28].

However, separating repositories conflicts with the inlining principle, as shown below. This is precisely why CCR and CCR 2.0 cannot support inlining of imaginary operations.

Counterexample. The motivating example in Figs. 1 and 2 serves as a counterexample. Recall that in §3 we established the refinement between $I_{\texttt{Main}} \circ I_{\texttt{Cell}} \circ I_{\texttt{Ctx}}$ (denoted `Imp`) and $A_{\texttt{Main}} \circ A_{\texttt{Cell}} \circ I_{\texttt{Ctx}}$ (denoted `Abs`) using the inlining principle. We now show that this refinement fails under module-local repositories, demonstrating that inlining is unsound in that setting.

To see this, suppose that `foo()` simply calls `Cell.set()` once. In `Imp`, this call overwrites the cell value set by the initial `Cell.set()` in $I_{\texttt{Main}}$, so the overwritten value is eventually printed. In `Abs`, by contrast, the printed value is determined by the first `Main.cb()` call, regardless of what `foo()` does. For the refinement to hold, `Abs` must therefore trigger undefined behavior (**`UB`**) before the mismatch becomes observable.

Under CRIS's global repository semantics, this is exactly what happens. During the execution of `Abs`, the exclusive resource `cell(−)` is assumed twice: once by the **`Assume`** at the beginning of $A_{\texttt{Main}}$, and once by $A_{\texttt{Cell}}$ itself. The second **`Assume`** attempts to duplicate the `cell(−)` resource, thereby triggering **`UB`**, as intended.

Under module-local repositories, however, this argument breaks down. The two assumptions are checked against separate module-local repositories, so both succeed independently. The duplication of the exclusive resource goes undetected, and `Abs` can print a value different from `Imp`. Hence, the inlining principle is unsound under module-local repositories.

6.2 Reconciling Inlining and Module-Local Invariants

How does CRIS support module-local invariants with a global repository? Our key observation is that supporting both requires (1) merging the resource repository across modules, while (2) maintaining separation of module-local states across modules. We achieve (1) by removing module boundaries at the semantic level (*i.e.,* there is no semantic difference between two linked modules and the same functions written in a single module from the start). We achieve (2) through the sandboxing mechanism introduced in §4: functions with disjoint scopes can access only disjoint keys, thereby preventing interference with other functions' states.

One may still wonder how module-local invariants are supported when resource repositories are merged into a single global one. The key insight is that our simulation relation is a unary **iProp** (*i.e.,* a predicate over resources, not a relation over pairs of resources), specifying the resource difference between the global repositories of the source and target. More specifically, the source repository must contain all resources of the target repository, and the additional resources in the source repository must satisfy the simulation relation.

Whenever an external function attempts to interfere with a module's resources, one of two outcomes follows. Either (i) the attempt fails on the target side because the target repository contains insufficient resources, in which case the simulation holds vacuously since the target cannot take the interfering step; or (ii) the attempt succeeds in modifying the target repository, in which case the corresponding attempt on the source side also succeeds, since the source repository contains all resources of the target. Moreover, the source-target difference remains unchanged, because the interference modifies the shared resources identically on both sides. This explains why module-local invariants are preserved even in the presence of interference from external functions.

It is important to note that the operational semantics places no restriction on resources—meaning the target side can hold more resources than the source side—while it is the simulation relation that prevents this. In other words, if no source execution can be found that always holds a resource at least as large, the simulation cannot be established. For example, when both sides hold the same resource, the simulation owns the difference (*i.e.,* the empty resource). If the target then attempts to acquire an additional resource, the simulation reasoning gets stuck: acquiring a resource on the target side requires taking it from the resource currently owned by the simulation, which is empty.

$[I_{\mathsf{Mem}}]$`def alloc(sz)` ≡ *["Det" alloc model]*	`def store(loc,v)` ≡ *[store model]*	`load,free,...`
$[T_{\mathsf{Mem}}]$`def alloc(sz)` ≡ *["Nondet" alloc model]*	`def store(loc,v)` ≡ *[store model]*	`load,free,...`
$[A_{\mathsf{Mem}}]$`def alloc(sz)` ≡ **Assume**($\ulcorner 0 < \mathsf{sz} \urcorner$); `loc = choose`($\mathbb{N}$); **Guarantee**($\circledast_{\mathsf{k}\in[0,\mathsf{sz})}$ loc+k $\mapsto$ 0); **ret** loc;	`def store(loc,v)` ≡ **Assume**($\exists \mathsf{v}_0, \mathsf{loc} \mapsto \mathsf{v}_0$); **Guarantee**($\mathsf{loc} \mapsto \mathsf{v}$); `def load(loc)` ≡ ... `def free(loc)` ≡ ...	
$[H_{\mathsf{Mem}}]$`def alloc(sz)` ≡ `b = take(bool);` `if b then` **Assume**($\ulcorner 0 < \mathsf{sz} \urcorner$); `loc = choose`($\mathbb{N}$); **Guarantee**($\circledast_{\mathsf{k}\in[0,\mathsf{sz})}$ (loc+k) $\mapsto$ 0); **ret** loc; `else` *["Nondet" alloc model]*	`def store(loc,v)` ≡ `b = take(bool);` `if b then` **Assume**($\exists \mathsf{v}_0, \mathsf{loc} \mapsto \mathsf{v}_0$); **Guarantee**($\mathsf{loc} \mapsto \mathsf{v}$); `else` *[store model]* `def load(loc)` ≡ ... `def free(loc)` ≡ ...	

Fig. 12. A concrete memory model, its ownership-based specification and a hybrid specification.

7 Advanced Examples

We present additional examples demonstrating the expressive power of imaginary specifications within CRIS, all formally verified in our artifact [19].

Hybrid Specifications for Memory. We present a hybrid specification for a memory module that allows clients to freely choose between pre/postcondition-style and operational-style reasoning.

Consider an integer-address-based memory model I_{Mem} with deterministic allocation, a stack module I_{Stack} using this memory, and an application I_{App} that uses both modules directly. To enable more efficient but less formal verification of I_{App} via techniques such as static analysis or testing, we aim to replace the concrete memory-based stack implementation with an abstract mathematical-list-based stack model A_{Stack} by establishing $I_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} \circ I_{\mathsf{App}} \sqsubseteq_{\mathsf{beh}} T_{\mathsf{Mem}} \circ A_{\mathsf{Stack}} \circ I_{\mathsf{App}}$, where T_{Mem} is the concrete memory model that I_{App} uses in the top-level spec.

The natural choice for T_{Mem} is I_{Mem} itself, but this does not work: its deterministic allocation causes the refinement to fail, since I_{Stack} consumes memory addresses while A_{Stack} does not. Instead, we define T_{Mem} as I_{Mem} with nondeterministic allocation, which allows the top-level spec to match the addresses allocated by the implementation.

While verifying $I_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} \sqsubseteq_{\mathsf{ctx}} T_{\mathsf{Mem}} \circ A_{\mathsf{Stack}}$ would directly establish this refinement, doing so forces us to reason with the concrete models I_{Mem} and T_{Mem} (first and second rows of Fig. 12) rather than an ownership-based specification (third row of Fig. 12). The challenge is that we must retain a concrete memory model at the top level because I_{App} uses it directly, while we want to abstract the stack's memory usage into an abstract specification to simplify the stack verification.

We resolve this conflict using an imaginary specification H_{Mem} (fourth row of Fig. 12) that allows clients to choose between the ownership-based specification and the concrete operational model. The specification H_{Mem} uses **take** to branch between A_{Mem} and T_{Mem}. By the TAKE-TGT rule in Fig. 8, clients can select either branch during verification.

The verification proceeds as follows: we verify $I_{\mathsf{Mem}} \sqsubseteq_{\mathsf{ctx}} H_{\mathsf{Mem}}$ for memory and $H_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} \sqsubseteq_{\mathsf{ctx}} H_{\mathsf{Mem}} \circ A_{\mathsf{Stack}}$ for the stack; $H_{\mathsf{Mem}} \sqsubseteq_{\mathsf{ctx}} T_{\mathsf{Mem}}$ then follows almost immediately from the TAKE-TGT rule, since **take** takes the intersection of its branches and thus refines any individual branch. Composing these yields: $I_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} \circ I_{\mathsf{App}} \sqsubseteq_{\mathsf{ctx}} H_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} \circ I_{\mathsf{App}} \sqsubseteq_{\mathsf{ctx}} H_{\mathsf{Mem}} \circ A_{\mathsf{Stack}} \circ I_{\mathsf{App}} \sqsubseteq_{\mathsf{ctx}} T_{\mathsf{Mem}} \circ A_{\mathsf{Stack}} \circ I_{\mathsf{App}}$.

This example illustrates a subtle unsoundness: naively using I_{Mem} in the top-level spec breaks the refinement. This subtlety highlights the need for formal verification via imaginary specifications.

More Examples. CRIS can express a range of advanced features, including mutual recursion and higher-order functions. Beyond the motivating example `CellioCB`, the Rocq artifact [19] includes the following examples, among others: (*i*) `Cellio`: similar to `CellioCB` but with `Main.main` not set as the initial function; (*ii*) `MutSum`: a mutually recursive computation of a sum; (*iii*) `Repeat`:

a higher-order function invoking a function pointer a specified number of times; (*iv*) Landin's knot: higher-order references and general recursion; and (*v*) `Ring`: a linking of n `Cell` modules for arbitrary n, demonstrating verification with a variable number of modules.

8 Related Work

The main distinguishing feature of CRIS relative to all existing work is formal support for hybrid verification via imaginary specifications.

Contextual Refinement and Relational Separation Logics. Contextual refinement requires a strong assumption: it must hold across *all possible contexts*, making it difficult to establish directly. A series of works [3, 6, 7, 9–12, 30, 32] has developed logical methods for proving contextual refinement and related relational properties. ReLoC and ReLoC Reloaded [9–11] developed mechanized Iris-based proof rules for contextual refinement of higher-order state and fine-grained concurrency. Blaze [6] extended relational separation logic to effect handlers with mutable state and concurrency. SimulIris [12] established fair termination-preserving contextual refinement.

However, none of these systems simultaneously support *termination-preservation*, *conditionality*, and *transitivity*. ReLoC, ReLoC Reloaded, and Blaze do not establish termination-preserving refinement, while SimulIris, although termination-preserving, separates the latter two properties across different judgments: its simulation judgment is conditional but not transitive, while its induced contextual refinement is transitive but unconditional.

Conditional Contextual Refinement. As reviewed in §1, CCR [29] introduced a refinement framework whose contextual refinement is both *conditional* and *transitive*, with a wrapper-based approach using dual nondeterminism to transfer ownership between caller and callee. However, CCR's proof principles are asymmetric: they work well for $I \sqsubseteq_{\mathtt{ctx}} S \vdash A : \langle P, Q\rangle$ but do not provide high-level reasoning principles for chains where the intermediate term is itself a wrapped specification (*e.g.*, $S \vdash A : \langle P, Q\rangle \sqsubseteq_{\mathtt{ctx}} S' \vdash A' : \langle P', Q'\rangle$). CCR 2.0 [28] addresses this by introducing **`*assume`** and **`*assert`** as user-level operations and providing reasoning principles for them at both source and target sides. CRIS subsumes all reasoning principles of CCR 2.0, while neither CCR nor CCR 2.0 supports imaginary specifications or hybrid verification (see §6.1).

Recent work [13, 34, 35] also attempts to marry conditional and transitive refinements, but relies on simple rely-guarantee conditions rather than ownership reasoning.

Dual Nondeterminism. Dual nondeterminism (angelic and demonic) is a classical concept extensively studied in game semantics. Refinement Calculus [2] pioneered the use of assume/assert statements and dual nondeterminism for writing specifications as programs to enable incremental verification, but was designed for simple languages with global state and does not support ownership-based reasoning. Refinement-Based Game Semantics (RBGS) [20, 21] extends this to settings with layers and local state, unifying refinement, game semantics, and algebraic effects, but also without ownership-based reasoning. Inspired by the use of dual nondeterminism in CCR [29], Dimsum [27] leverages dual nondeterminism to logically convert between integer-based addresses and block-based addresses, enabling composition of modules written in different languages such as assembly and C. VeriFast [15, 16] introduced angelic nondeterminism into symbolic execution to check assumptions about memory operations via automatically inserted assertions, rather than supporting separation logic assertions at arbitrary locations.

Viper. Viper [8, 24, 25] is an intermediate verification language that supports permission-based automated reasoning through constructs such as `inhale` and `exhale`, enabling the encoding of assume-guarantee reasoning within code. These constructs are conceptually similar to CRIS's **`Assume`** and **`Guarantee`**, though Viper's primary focus is on automated verification whereas CRIS focuses on proving refinement. Additionally, Viper's permission model is tightly coupled to the heap, whereas CRIS follows Iris in supporting a more general non-step-indexed resource algebra.

Acknowledgments

We thank the anonymous reviewers for their detailed and constructive feedback, which significantly helped us improve the presentation of this paper. This work was supported in part by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03, and by a National Research Foundation of Korea (NRF) grant funded by the Korean Ministry of Science and ICT (MSIT) (Grant No. RS-2024-00355459). Chung-Kil Hur is the corresponding author.

Data Availability Statement

The artifact for this paper is publicly available on Zenodo: https://doi.org/10.5281/zenodo.19491861.

References

[1] Andrew W. Appel. 2011. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

[2] Ralph-Johan Back and Joakim Wright. 2012. *Refinement calculus: a systematic introduction.* Springer Science & Business Media.

[3] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 14–25. doi:10.1145/964001.964003

[4] Manfred Broy and Martin Wirsing. 1981. On the Algebraic Specification of Nondeterministic Programming Languages. In *Proceedings of the 6th Colloquium on Trees in Algebra and Programming (CAAP '81)*. Springer-Verlag, Berlin, Heidelberg, 162–179.

[5] Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. Stuttering for Free. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 281 (Oct. 2023), 28 pages. doi:10.1145/3622857

[6] Paulo Emílio de Vilhena, Simcha van Collem, Ines Wright, and Robbert Krebbers. 2026. A Relational Separation Logic for Effect Handlers. *Proc. ACM Program. Lang.* 10, POPL, Article 34 (Jan. 2026), 29 pages. doi:10.1145/3776676

[7] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A Relational Modal Logic for Higher-Order Stateful ADTs. *SIGPLAN Not.* 45, 1 (jan 2010), 185–198. doi:10.1145/1707801.1706323

[8] M. Eilers, M. Schwerhoff, A. J. Summers, and P. Müller. 2025. Fifteen Years of Viper. In *Computer Aided Verification (CAV)*. To appear.

[9] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 442–451.

[10] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* Volume 17, Issue 3 (Jul 2021). doi:10.46298/lmcs-17(3:9)2021

[11] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). doi:10.46298/lmcs-17(3:9)2021

[12] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498689

[13] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. *ACM SIGPLAN Notices* 53, 4 (2018), 646–661.

[14] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The power of parameterization in coinductive proof. *SIGPLAN Not.* 48, 1 (Jan. 2013), 193–206. doi:10.1145/2480359.2429093

[15] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: a powerful, sound, predictable, fast verifier for C and java. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 41–55.

[16] Bart Jacobs, Frédéric Vogels, and Frank Piessens. 2015. Featherweight verifast. *Logical Methods in Computer Science* 11 (2015).

[17] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

[18] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *SIGPLAN Not.* 50, 1 (Jan. 2015), 637–650. doi:10.1145/2775051.2676980

[19] Yonghee Kim, Taeyoung Yoon, Sanghyun Yi, Jaehyung Lee, Soonwon Moon, Yeji Han, Seonho Lee, Taeyoung Rhee, Yujin Im, Donghyun Nam, Jieung Kim, and Chung-Kil Hur. 2026. Artifact for CRIS: The Power of Imagination in Hybrid Verification *(PLDI 2026 Artifact)*. doi:10.5281/zenodo.19491861

[20] Jérémie Koenig. 2020. Refinement-Based Game Semantics for Certified Components. https://flint.cs.yale.edu/flint/publications/koenig-phd.pdf

[21] 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* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 633–647. doi:10.1145/3373718.3394799

[22] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

[23] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 205–217. doi:10.1145/3009837.3009855

[24] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution. In *Computer Aided Verification (CAV) (LNCS, Vol. 9779)*, S. Chaudhuri and A. Farzan (Eds.). Springer-Verlag, 405–425. http://link.springer.com/chapter/10.1007/978-3-319-41528-4_22

[25] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS, Vol. 9583)*, B. Jobstmann and K. R. M. Leino (Eds.). Springer-Verlag, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[26] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic*, Laurent Fribourg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–19.

[27] 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. doi:10.1145/3571220

[28] Youngju Song and Minki Cho. 2025. CCR 2.0: High-level Reasoning for Conditional Refinements. arXiv:2507.04298 [cs.PL] https://arxiv.org/abs/2507.04298

[29] 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. doi:10.1145/3571232

[30] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 377–390.

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

[32] Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 308–334. doi:10.1016/j.tcs.2006.12.036

[33] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 71–84. doi:10.1145/3372885.3373813

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

[35] 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 (Jan. 2025), 31 pages. doi:10.1145/3704900