Layered and Object-Based Game Semantics

ARTHUR OLIVEIRA VALE, Yale University, USA
PAUL-ANDRÉ MELLIÈS, CNRS and Université de Paris, France
ZHONG SHAO, Yale University, USA
JÉRÉMIE KOENIG, Yale University, USA
LÉO STEFANESCO, MPI-SWS, Germany

Large-scale software verification relies critically on the use of compositional languages, semantic models, specifications, and verification techniques. Recent work on certified abstraction layers synthesizes game semantics, the refinement calculus, and algebraic effects to enable the composition of heterogeneous components into larger certified systems. However, in existing models of certified abstraction layers, compositionality is restricted by the lack of encapsulation of state.

In this paper, we present a novel game model for certified abstraction layers where the semantics of layer interfaces and implementations are defined solely based on their observable behaviors. Our key idea is to leverage Reddy's pioneer work on modeling the semantics of imperative languages not as functions on global states but as objects with their observable behaviors. We show that a layer interface can be modeled as an object type (i.e., a layer signature) plus an object strategy. A layer implementation is then essentially a regular map, in the sense of Reddy, from an object with the underlay signature to that with the overlay signature. A layer implementation is certified when its composition with the underlay object strategy implements the overlay object strategy. We also describe an extension that allows for non-determinism in layer interfaces.

After formulating layer implementations as regular maps between object spaces, we move to concurrency and design a notion of *concurrent object space*, where sequential traces may be identified modulo permutation of independent operations. We show how to express protected shared object concurrency, and a ticket lock implementation, in a simple model based on regular maps between concurrent object spaces.

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

Additional Key Words and Phrases: object-based semantics, certified abstraction layers, game semantics, program refinement

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

1 INTRODUCTION

Certified software [Shao 2010] comes with a formal specification and a mechanized proof that the software conforms to the specification. There have been a large number of recent projects on building certified components such as compilers [Leroy 2009], program logics [Appel 2011],

Authors' addresses: Arthur Oliveira Vale, Yale University, USA, arthur.oliveiravale@yale.edu; Paul-André Melliès, Institut de Recherche en Informatique Fondamentale (IRIF), CNRS and Université de Paris, France, mellies@irif.fr; Zhong Shao, Yale University, USA, zhong.shao@yale.edu; Jérémie Koenig, Yale University, USA, jeremie.koenig@yale.edu; Léo Stefanesco, MPI-SWS, Germany, leo.stefanesco@mpi-sws.org.

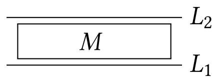
2475-1421/2022/1-ART42
https://doi.org/10.1145/3498703

OS kernels [Gu et al. 2015, 2016], file systems [Chen et al. 2015], and processor designs [Choi et al. 2017]. Unfortunately, even if these systems were developed using the same proof assistant, they use different semantic models and verification techniques. To scale up verification further (as exemplified by the DeepSpec project [dee 2021; Appel et al. 2017]), one major challenge is to identify a general-purpose model which could embed all existing components. This model should support composition and help bridge the gap between components that operate at different levels of abstraction.

1.1 Certified Abstraction Layers

Certified abstraction layers [Gu et al. 2015, 2018] are a promising technology for programming, compiling, linking, and composing certified heterogeneous components. The initial sequential CertiKOS kernel [Gu et al. 2015] was decomposed into 37 certified layers consisting of C and assembly modules such as physical and virtual memory managers, context-switch libraries, thread and process managers, virtual machine managers, and page fault and trap handlers. Later versions of CertiKOS [Chen et al. 2016; Gu et al. 2016, 2018; Liu et al. 2019] showed how to extend certified layers to support multicore and multithreaded concurrency, fine-grained locking, device drivers, and real-time scheduling; they have also been extended to verify not only the total functional correctness but also information-flow security properties [Costanzo et al. 2016; Liu et al. 2019].

As described in Gu et al. [2015], a certified abstraction layer consists of a *layer implementation* together with two *layer interfaces*: the *underlay* provides specifications for the primitives available to the layer implementation; the *overlay* provides specifications for the primitives which the layer implements. A layer M implementing the overlay interface L_2 on top of the underlay interface L_1 can be depicted on the right below.



A layer interface L has three components. First, a *signature* enumerates primitives together with their types, given as $\mathsf{op} : A \to B$ where A and B are sets. Second, the set S contains the *abstract states* of the layer interface. Finally, for each primitive $\mathsf{op} : A \to B$, its *specification* is given as a function of type $A \times S \to \mathcal{P}^1(B \times S)$ where $\mathcal{P}^1$ corresponds to the *maybe* monad: $\mathcal{P}^1(X)$ is defined as $\{x \subseteq X : |x| \leq 1\}$, and the empty set $\varnothing \in \mathcal{P}^1(X)$ denotes fault or silent divergence.

As an example (taken from Koenig and Shao [2020]), Fig. 1 presents a certified layer that implements a bounded queue with at most N elements using a ring buffer. In the underlay interface $L_1 = L_{\mathrm{rb}}$, its abstract state contains an array $f \in \mathbb{U}^N$ with N values of type $\mathbb{U}$ and two counters which take values in the interval $0 \leq c_1, c_2 < N$. The array supports the primitives get and set; the primitives fai_1 and fai_2 increment the corresponding counter and return the counter's old value.

The overlay $L_2 = L_{\mathrm{bq}}$ features two primitives enq and deq which respectively add a new element to the queue and remove the oldest element. If we add an element which overflows the queue's capacity N, or remove an element from an empty queue, the result is $\varnothing$ (i.e., the primitive aborts).

The layer implementation M_{bq} stores the queue's elements into the array, between the indices given by the counters' values. This is expressed by the simulation relation $R \subseteq S_{\mathrm{bq}} \times S_{\mathrm{rb}}$ in Fig. 1, which explains how overlay states are realized by M_{bq} in terms of underlay states. The code of M_{bq} can be interpreted using the monad $S_{\mathrm{rb}} \to \mathcal{P}^1(- \times S_{\mathrm{rb}})$, with calls to primitives of $L_1 = L_{\mathrm{rb}}$ replaced by their specifications. We write $L_{\mathrm{rb}}[M_{\mathrm{bq}}]$ to denote the result. We declare that M_{bq} defines a certified layer $L_{\mathrm{rb}} \vdash_R M_{\mathrm{bq}} : L_{\mathrm{bq}}$ when for each operation $\mathsf{op} \in \{\mathsf{enq}(v), \mathsf{deq}(*) \mid v \in \mathbb{U}\}$ of the overlay $L_2 = L_{\mathrm{bq}}$, the relation R indeed establishes a simulation of $L_{\mathrm{bq}}.\mathsf{op}$ by $L_{\mathrm{rb}}[M_{\mathrm{bq}}].\mathsf{op}$.

Certified abstraction layers bring the following benefits:

- *Compositional Specification:* A layer interface L provides not only the type signatures of its primitives but also their full functional "deep" specification [Gu et al. 2015]. The client code

$$\begin{array}{rr}
\boxed{L_{\mathsf{bq}}} & S_{\mathsf{bq}} := \mathbb{U}^* \\
\mathsf{enq} : \mathbb{U} \to \mathbf{1} & L_{\mathsf{bq}}.\mathsf{enq}(v)@\vec{q} := \{*@\vec{q}v \mid |\vec{q}| < N\} \\
\mathsf{deq} : \mathbf{1} \to \mathbb{U} & L_{\mathsf{bq}}.\mathsf{deq}(*)@\vec{q} := \{v@\vec{p} \mid \vec{q} = v\vec{p}\}
\end{array}$$

$$\begin{array}{rl}
\boxed{M_{\mathsf{bq}}} & R \subseteq S_{\mathsf{bq}} \times S_{\mathsf{rb}} \\
M_{\mathsf{bq}}.\mathsf{enq}(v) := i \leftarrow \mathsf{fai}_2;\ \mathsf{set}(i, v) & \vec{q}\ R\ (f, c_1, c_2) \Leftrightarrow (c_1 \le c_2 < N \wedge \vec{q} = f_{c_1} \cdots f_{c_2-1}) \vee \\
M_{\mathsf{bq}}.\mathsf{deq}(*) := i \leftarrow \mathsf{fai}_1;\ \mathsf{get}(i) & \qquad (c_2 \le c_1 < N \wedge \vec{q} = f_{c_1} \cdots f_{N-1} f_0 \cdots f_{c_2-1})
\end{array}$$

$$\begin{array}{rr}
\boxed{L_{\mathsf{rb}}} & S_{\mathsf{rb}} := \mathbb{U}^N \times \mathbb{N} \times \mathbb{N} \\
\mathsf{set} : \mathbb{N} \times \mathbb{U} \to \mathbf{1} & L_{\mathsf{rb}}.\mathsf{set}(i, v)@(f, c_1, c_2) := \{*@(f', c_1, c_2) \mid i < N \wedge f' = f[i := v]\} \\
\mathsf{get} : \mathbb{N} \to \mathbb{U} & L_{\mathsf{rb}}.\mathsf{get}(i)@(f, c_1, c_2) := \{f_i@(f, c_1, c_2) \mid i < N\} \\
\mathsf{fai}_1 : \mathbf{1} \to \mathbb{N} & L_{\mathsf{rb}}.\mathsf{fai}_1@(f, c_1, c_2) := \{c_1@(f, c_1', c_2) \mid c_1' = (c_1 + 1) \bmod N\} \\
\mathsf{fai}_2 : \mathbf{1} \to \mathbb{N} & L_{\mathsf{rb}}.\mathsf{fai}_2@(f, c_1, c_2) := \{c_2@(f, c_1, c_2') \mid c_2' = (c_2 + 1) \bmod N\}
\end{array}$$

Fig. 1. A certified layer $L_{\mathsf{rb}} \vdash_R M_{\mathsf{bq}} : L_{\mathsf{bq}}$ implementing a bounded queue of size N using a ring buffer. The left side of the figure shows the signatures of the overlay and underlay interfaces, and the code associated with the layer. The right side shows primitive specifications and the simulation relation used by the correctness proof. We use $*$ as a unit value of type $\mathbf{1}$, and $v@k \in A \times S$ as a pair of value $v \in A$ and state $k \in S$.

for L_{bq} can operate without seeing how L_{bq} is actually implemented. In this sense, L_{bq} fully encapsulates the implementation details of all the layers below.

- *Compositional Verification:* A certified system can be decomposed into many certified layers. Each layer implementation (e.g., M_{bq}) serves as a building block connecting one layer interface (e.g., L_{bq}) with another (e.g., L_{rb}). A layer implementation can be verified using its overlay and underlay interfaces alone.
- *Effect Encapsulation and Composition:* A layer interface behaves like an object in that its signature hides not only the implementation but also the abstract state. A layer signature is like an algebraic effect signature [Plotkin and Power 2001]. Its layer primitives are like methods or effect handlers [Plotkin and Pretnar 2009].

1.2 A Layered and Object-Based Game Model

Koenig and Shao [2020] recently presented a game-semantic model for certified abstraction layers by synthesizing ideas from game semantics [Abramsky et al. 2000; Abramsky and McCusker 1999; Blass 1992; Hyland and Ong 2000], the refinement calculus [Back and Wright 1998], and algebraic effects [Plotkin and Power 2001; Plotkin and Pretnar 2009]. They interpret each layer interface signature as a game and the interaction between the layer interface and its client as a strategy. They then model a layer implementation (e.g., M_{bq}) as an "interaction substitution" morphism from overlay strategies to underlay strategies. The resulting game semantics features specification refinement with both angelic and demonic nondeterminism.

However, in their main development, Koenig and Shao use an explicit state-passing approach where abstract states (e.g., elements of S_{bq} and S_{rb}) are communicated as part of the interaction (i.e., in game-semantic moves such as $\mathsf{deq}(*)@\vec{q}$). This is not desirable since a layer interface is supposed to encapsulate its abstract state. A client of L_{bq} should not observe the internal state $\vec{q}$ in its interaction with L_{bq}.

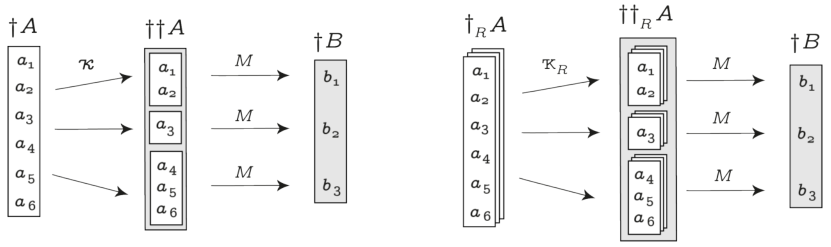


Fig. 2. The picture on the left describes how every regular map $\widehat{M} : \dagger A \longrightarrow \dagger B$ can be factored into the map $\kappa : \dagger A \longrightarrow \dagger\dagger A$ which decomposes any sequence of elements of A such as $\langle a_1, a_2, a_3, a_4, a_5, a_6\rangle \in \dagger A$ into a "sequence of sequences" such as $\langle\langle a_1, a_2\rangle, \langle a_3\rangle, \langle a_4, a_5, a_6\rangle\rangle \in \dagger\dagger A$, followed by the map $\dagger M : \dagger\dagger A \longrightarrow \dagger B$ which “replays” $M : \dagger A \multimap B$ as many times as there are elements in the output sequence of elements of B: three times in this case, in order to obtain the sequence $\langle b_1, b_2, b_3\rangle \in \dagger B$. The picture on the right explains how the construction adapts smoothly to a regular map $M : \dagger_R A \longrightarrow \dagger B$ from a concurrent object space $\dagger_R A$ equipped with an equivalence relation R on sequences of elements of A, using the decomposition map $\kappa_R : \dagger_R A \longrightarrow \dagger\dagger_R A$.

In this paper, we leverage ideas from Reddy’s pioneer work [Reddy 1996] on *object-based semantics* and develop a new model for certified abstraction layers that does not carry abstract states in the game-semantic moves and strategies. Semantics for imperative languages have mostly been described as functions on global states. Reddy’s approach, on the other hand, defines such semantics as *objects* with their observable behaviors. His key idea is precisely to restrict states as part of the internal structure of objects and make them not externally visible. He constructed a semantic model for objects based on coherence spaces [Girard 1987], and showed that an object function $\widehat{M}$ building objects of type B on top of other objects of type A can be viewed as a regular map (a linear map with extra structure):

$$\widehat{M} \quad : \quad \dagger A \longrightarrow \dagger B$$

between coherence spaces $\dagger A$ and $\dagger B$ whose elements (or tokens) describe sequences of elements of the coherence spaces A and B. Informally, the semantics of an object of type A (or B) is a set of its observable event traces, denoted as of type $\dagger A$ (or $\dagger B$), as shown in Fig.2. Each event (e.g., a_1, b_1) is just an atomic method invocation plus its return value. The fundamental property of regular maps is that they are entirely determined by their restriction $M : \dagger A \multimap B$ to the coherence space B describing a single element (instead of many) inside $\dagger B$. The regular map $\widehat{M}$ is recovered from M by the equation:

$$\dagger A \xrightarrow{\widehat{M}} \dagger B \quad = \quad \dagger A \xrightarrow{\kappa} \dagger\dagger A \xrightarrow{\dagger M} \dagger B \tag{1}$$

where κ is a canonical "decomposition" map and $\dagger M$ replays M sequentially, as explained in Fig. 2. The coherence space $\dagger A$ is called “dagger A” or more evocatively “replay A” for that reason.

Regular maps reveal the “declarative” nature of the object-based semantics. An “imperative-looking” layer implementation M_{bq} is actually quite “functional:” it transforms a stream of events for the underlay L_{rb} into one for the overlay L_{bq}. While the bounded queue interface L_{bq} encapsulates a stateful object, its effectful operations actually come from the underlay L_{rb}, orchestrated by the “stateless” object implementation M_{bq} following a *regular* pattern.

We show that there is a great synergy between object-based semantics and certified abstraction layers, and we establish a nice and useful synthesis between these two lines of work. The object-based approach can be nicely extended to support concurrency by equipping each $\dagger A$ with an equivalence relation R, yielding a set of equivalent event traces, denoted as $\dagger_R A$ in Fig. 2.

1.3 Summary and Main Contributions

Our paper makes the following contributions:

- We present a new layered game model of interaction suitable for building certified systems (see §3). We derive an object-based game from each layer signature. Layer interfaces are modeled as *general strategies* for this game, and layer implementations are modeled as *regular strategies* from the underlay signature to the overlay signature. A layer implementation is then called *certified* when its composition with the underlay interface strategy refines the overlay interface strategy.
- We show that our new layered game model as well as its object-based interpretation can be easily extended to support a generalized form of layer interface specification that allows non-determinism (see §4).
- We connect our game semantics to Reddy's object-based model (see §2) by interpreting layer signatures as coherence spaces and layer implementations as regular maps; we then extend this interpretation to certified layer implementations (see §5).
- We design a notion of *concurrent object space*, where sequential traces may be identified modulo permutation of independent operations (see §6). We show how to express protected shared object concurrency, and a ticket lock implementation, in a simple model based on regular maps between such concurrent object spaces (see §7).

Our extended technical report [Oliveira Vale et al. 2021] contains a set of appendices which give a formal presentation of §3 and the detailed proofs for various results discussed in this paper.

1.4 Connecting Semantics to Code: A Broader Perspective

More broadly, this paper continues the work by Koenig and Shao [2020] and aims to *develop a compositional model for certified abstraction layers so that the model can be used to build certified heterogeneous systems such as CertiKOS*. The abstract reformulation of certified abstraction layers enables us to benefit from a wealth of semantics research toward our goal. While our model allows us to have a bird's-eye view of a system's behavior, when dealing with a concrete system we must establish a connection between this large-scale view and the fine-grained operational semantics of the code implementing it. To do so we plan to leverage and enhance the following technologies developed recently by the Yale FLINT group:

`CompCertO` [Koenig and Shao 2021]. The `CompCertO` compiler provides an open semantics based on Koenig and Shao [2020] to the CompCert certified C compiler. The semantics is inspired by the game-semantics approaches and compatible with the model of certified abstraction layers in this paper. This tier of abstraction, unfortunately, suffers from the problem of lack of encapsulation of memory state, which we resolve here in the case of certified layers.

`DeepSEA` [Sjöberg et al. 2019]. The process of manually connecting the C and assembly semantics from CompCert with the Certified Abstraction Layers framework in the original development of CertiKOS led to the development of the `DeepSEA` programming language. The `DeepSEA` code is compiled into C (and then compiled to assembly using `CompCertO`) plus a deep specification of its behavior in Coq and an automatically generated proof of refinement between them. The `DeepSEA` platform has been revamped so that it now follows the semantics presented in this paper, and the C code generated now comes with a proof of refinement with a specification in our new semantic model. `DeepSEA` bridges the gap between the more abstract model (presented in our current paper) and CompCertO's model, crucially enforcing the encapsulation of C memory state by different layers.

To verify a concrete system, we can first use `DeepSEA` and `CompCertO` to move from the small-step operational semantics of CompCert to our model; we can then verify the system using the more abstract model as described in this paper. Ultimately we believe this connection between systems verification and the semantic models of linear logic and state brings to the forefront of systems verification well-established semantic techniques for characterizing programming language and systems behavior, with a focus on expressiveness and compositionality.

2 OBJECT-BASED SEMANTICS

Reddy [1996] introduces a semantic domain for imperative languages based on the model of state introduced in Reddy [1993], itself based on the coherence spaces semantics for linear logic [Girard 1987] with the before operator [Retoré 1997]. He calls this semantic domain object-based semantics. This section gives a quick introduction to coherence spaces and object-based semantics.

2.1 The Basic Principles

Reddy [1996] advocates the idea that a program with an internal state can be entirely characterized in terms of its interactive behavior, as what he calls an *object*. This object-based semantics is based on four basic principles:

(1) An object can in general be used only sequentially,
(2) The behavior of an object is in general affected by its past history of operations,
(3) Object functions must be linear maps,
(4) Object functions are regular maps.

Principle 1 means that objects can be described by the linear trace they produce, and Principle 2 that the prefix of a trace influences what comes next. Principles 3 and 4 are then elegantly formalized using the notion of coherence spaces originally introduced by Girard [1987].

2.2 Coherence Spaces and Linear Maps

Definition 2.1. A *coherence space* $A = (|A|, \coh_A)$ is a set of tokens $|A|$ together with a reflexive and symmetric coherence relation $\coh_A \subset |A| \times |A|$.

Example 2.2. The coherence space Var encodes the operations over a variable (or memory cell) storing an integer value $n \in \mathbb{N}$. The web |Var| of Var is defined as:

$$|\mathsf{Var}| := \{\mathsf{get}.n \mid n \in \mathbb{N}\} \uplus \{\mathsf{set}(n).\mathsf{ok} \mid n \in \mathbb{N}\}$$

Each token of Var encodes both a call and return event: the token $\mathsf{set}(n).\mathsf{ok}$ encodes a call to set with argument n returning ok; the token $\mathsf{get}.n$ encodes a call to get with no arguments and returning n. The coherence relation of Var is defined as:

$$\mathsf{op}.v \coh_{\mathsf{Var}} \mathsf{op}'.v' \iff \big(\mathsf{op} = \mathsf{op}' \Rightarrow v = v'\big)$$

The definition of $\coh_{\mathsf{Var}}$ conveys the intuition that the operations in Var are deterministic, in the sense that two tokens $\mathsf{op}.v$ and $\mathsf{op}.v'$ with the same underlying operation op are coherent precisely when the operation returns the same value $v = v'$.

We proceed in the same way to define the coherence space Counter which encodes the operations of a counter:

$$|\mathsf{Counter}| := \{\mathsf{get}.n \mid n \in \mathbb{N}\} \cup \{\mathsf{inc.ok}\} \qquad \mathsf{op}.v \coh_{\mathsf{Counter}} \mathsf{op}'.v' \iff \big(\mathsf{op} = \mathsf{op}' \Rightarrow v = v'\big)$$

Morphisms between coherence spaces are defined as *linear maps*, in the following way, where we use the notation $a \mapsto b$ to denote a pair $(a, b) \in A \times B$:

Definition 2.3. A relation $f \subseteq |A| \times |B|$ is a *linear map* $f : A \multimap B$ when for all $a_1 \mapsto b_1, a_2 \mapsto b_2 \in f$ the following holds:

$$(1) \quad a_1 \coh_A a_2 \Rightarrow b_1 \coh_B b_2 \qquad (2) \quad a_1 \coh_A a_2 \wedge b_1 = b_2 \Rightarrow a_1 = a_2$$

A *clique* in a coherence space A is defined as a subset $f \subseteq |A|$ of tokens of A which are pairwise coherent. Note that a clique f of a coherence space A is the same thing as a linear map $f : \mathbf{1} \multimap A$ where $\mathbf{1}$ is defined as the coherence space with web $|\mathbf{1}| := \{*\}$ containing a single token such that $* \frown_{\mathbf{1}} *$.

2.3 The Replay Modality

The main thesis of Reddy [1996] is that the informal notion of object is appropriately captured by the notion of clique of a coherence space of the form $\dagger A$, whose tokens are finite sequences of tokens of A.

Definition 2.4. Given a coherence space A, the *(free) object space* associated to A is the coherence space $\dagger A$ with tokens in $|\dagger A| := |A|^*$ and the coherence relation $\frown_{\dagger A}$ which relates $\langle a_1, \dots, a_n\rangle \frown_{\dagger A} \langle b_1, \dots, b_m\rangle$ if and only if:

$$\forall i \leq \min(n, m). \langle a_1, \dots, a_{i-1}\rangle = \langle b_1, \dots, b_{i-1}\rangle \Rightarrow a_i \frown_A b_i.$$

An *object* is a clique of $\dagger A$.

The coherence relation over $\dagger A$ ensures that at the first point where the two sequences differ, they differ coherently. Note that if $s \in |\dagger A|$ is a prefix of $t \in |\dagger A|$ then $s \frown_{\dagger A} t$.

Example 2.5. The object space associated to Var is the space of all sequences of call and return events that can be performed on a variable. Note that $\dagger A$ does not enforce a particular semantics for the returns across this sequence. For instance,

$$\langle \mathsf{get}.3, \mathsf{set}(7).\mathsf{ok}, \mathsf{get}.5\rangle \in |\dagger\mathsf{Var}| \qquad \langle \mathsf{get}.0, \mathsf{set}(1).\mathsf{ok}, \mathsf{get}.1\rangle \in |\dagger\mathsf{Var}|$$

On the other hand, an object in $\dagger$Var specifies a particular semantics by restricting those sequences. For instance, the following set of sequences forms a clique in $\dagger$Var:

$$V_{\mathsf{Var}} := \{s \in |\dagger\mathsf{Var}| \mid (s = \mathsf{get}.i \cdot s' \Rightarrow i = 0) \wedge (s = p \cdot \mathsf{get}.i \cdot \mathsf{get}.i' \cdot t \Rightarrow i = i') \\ \wedge (s = p \cdot \mathsf{set}(i).\mathsf{ok} \cdot \mathsf{get}.i' \cdot t \Rightarrow i = i')\}$$

The properties over the sequences in V_{Var} enforce that: (1) The variable initially responds to a get with a 0; (2) Consecutive calls to get return the same value; (3) A call to get following a set(i) must return i. This defines a prefix-closed clique encoding all possible behaviors starting from a certain state. Similarly, we can define:

$$V_{\mathsf{Counter}} := \{s \in |\dagger\mathsf{Counter}| \mid (s = p \cdot \mathsf{get}.i \cdot t \Rightarrow i = \#\mathsf{inc}(p))\}$$

where #inc(p) counts occurrences of inc.ok in the sequence p.

2.4 Regular Maps between Object Spaces

Maps between object spaces are required to satisfy a regularity requirement. In Reddy [1996], regular functions are first defined using a structural property of the map, then an equivalence is proven with the co-Kleisli category of $\dagger$ on coherence spaces.

Definition 2.6. For object spaces $\dagger A$ and $\dagger B$, a *regular map* $f : \dagger A \rightarrow_{\mathsf{Reg}} \dagger B$ is a linear map satisfying:

(1) $(s_1 \mapsto t_1), \dots, (s_n \mapsto t_n) \in f \Rightarrow (s_1 \cdots s_n \mapsto t_1 \cdots t_n) \in f$
(2) If $(s \mapsto t_1 \cdots t_n) \in f$, then there exists $s_1, \dots, s_n \in |\dagger A|$ such that $s = s_1 \cdot \ldots \cdot s_n$ and $s_1 \mapsto t_1 \in f, \dots, s_n \mapsto t_n \in f$

The equivalence is then given by the following theorem.

Theorem 2.7 (Reddy). *There is an isomorphism*

$$\dagger A \multimap B \quad \cong \quad \dagger A \to_{\mathsf{Reg}} \dagger B$$

between the linear maps from $\dagger A$ *to* B *and the regular maps from* $\dagger A$ *to* $\dagger B$.

The isomorphism is based on the observation by Reddy that every linear function $f : \dagger A \multimap B$ extends uniquely to a regular map $\widehat{f} : \dagger A \to \dagger B$ defined in the following way:

$$\widehat{f} \;:=\; \{s_1 \cdots s_n \mapsto \langle b_1, \ldots, b_n\rangle \mid n \geq 0 \;\wedge\; \forall i.\; s_i \mapsto b_i \in f\} \quad : \quad \dagger A \to_{\mathsf{Reg}} \dagger B$$

As explained in the introduction, the regular map $\widehat{f}$ can be equivalently defined as the composite (1) where the canonical "decomposition" map $\kappa : \dagger A \to \dagger\dagger A$ is defined as:

$$\kappa \;:=\; \{s_1 \cdot \ldots \cdot s_n \mapsto \langle s_1, \ldots, s_n\rangle \mid s_1, \ldots, s_n \in |\dagger A|\}$$

Example 2.8. We can define a regular map $M : \dagger\mathsf{Var} \multimap \mathsf{Counter}$ by first defining two maps M^{get} and M^{inc} which define the sequences that map to the $\mathsf{get}.i$ token and the $\mathsf{inc.ok}$ token in Counter, respectively,

$$M^{\mathsf{inc}} = \{\langle \mathsf{get}.i, \mathsf{set}.(i+1)\rangle \mapsto \mathsf{inc.ok} \mid i \in \mathbb{N}\} \qquad M^{\mathsf{get}} = \{\langle \mathsf{get}.i\rangle \mapsto \mathsf{get}.i \mid i \in \mathbb{N}\}$$

This intuitively corresponds to the following pieces of code one might write to implement a counter using a variable:

```
inc() {                    get() {
  i ← get();                 i ← get();
  set(i+1);                  return i;
  return ok;               }
}
```

And then, the map M defined as $M := M^{\mathsf{get}} \uplus M^{\mathsf{inc}}$ is indeed a linear map $M : \dagger\mathsf{Var} \multimap \mathsf{Counter}$ which can be extended to a regular map $\widehat{M} : \dagger\mathsf{Var} \to_{\mathsf{Reg}} \dagger\mathsf{Counter}$, which can be regarded as a function from objects in $\dagger$Var to objects in $\dagger$Counter. One important equation which we will formalize in §3 is that

$$\widehat{M} \circ V_{\mathsf{Var}} \quad = \quad V_{\mathsf{Counter}}. \tag{2}$$

The equation expresses that the object V_{Counter} is correctly implemented by the object V_{Var} by using the implementation specified by the regular map M. This can be seen as follows. Since M^{get} and M^{inc} both start with get, it must be that whenever $s \mapsto t \in \widehat{M} \circ V_{\mathsf{Var}}$ we have that s starts with get.0. But, as we will see in §3, traces in V_{Var} are deterministic so that there is only one possible return for a get. This way, there is a single sequence in V_{Var} that is mapped to any particular sequence in V_{Counter} by $\widehat{M}$.

Composition of linear functions is defined by usual relational composition:

Definition 2.9. Given linear maps $f : A \multimap B$ and $g : B \multimap C$ we define the composition $g \circ f : A \multimap C$ as

$$g \circ f \;:=\; \{a \mapsto c \mid \exists b \in |B|.\; a \mapsto b \in f \;\wedge\; b \mapsto c \in g\}$$

The identity $\mathbf{id}_A : A \multimap A$ is defined as $\mathbf{id}_A := \{a \mapsto a \mid a \in |A|\}$.

We define the composition of linear maps generating regular functions $f : \dagger A \multimap B$ and $g : \dagger B \multimap C$ as $g \circ \widehat{f}$.

In summary, object spaces are modeled as coherence spaces $\dagger A$ and morphisms between these object spaces are regular maps $\dagger A \to_{\mathsf{Reg}} \dagger B$. Regular maps can be instead described by linear maps $\dagger A \multimap B$, and such linear maps can be composed in accordance to regular map composition. A single object of type A is a clique in the graph of the relation $\frown_{\dagger A}$.

3 AN INTERACTIVE MODEL OF CERTIFIED LAYERS

In this section, we describe a layered model of interaction based on game semantics, suitable for defining certified systems. We choose to present it informally here for the sake of simplicity, see Oliveira Vale et al. [2021] for a formal presentation. A *layer interface* (E, V_E) consists of an effect signature E and of a deterministic specification V_E, which we call an *object strategy*, of the interactive behavior of the interface. A *certified implementation* $M : (E, V_E) \to (F, V_F)$ between two such layer interfaces is essentially a collection of strategies $M : E \to F$ which implement the overlay interface (F, V_F) using the effects and capabilities provided by the underlay interface (E, V_E).

3.1 Effect Signatures as Layer Signatures

An important observation of Koenig and Shao [2020] is that effect signatures can be used to specify layer interface signatures. We recall their notion of *effect signature* here.

Definition 3.1. An *effect signature* is a set E of operations together with a mapping $\mathsf{ar}(-)$, which assigns to each $e \in E$ a set $\mathsf{ar}(e)$ called the *arity* of e. We will use the notation

$$E = \{e_1 : \mathsf{ar}(e_1), e_2 : \mathsf{ar}(e_2), \ldots\}$$

to describe effect signatures.

Example 3.2. We can define signatures Var, for a layer describing a variable interface, and Counter, describing a counter, as follows:

$$\mathsf{Var} := \{\mathsf{get} : \mathbb{N}, \mathsf{set} : \mathbb{N} \to \mathbf{1}\} \qquad \mathsf{Counter} := \{\mathsf{get} : \mathbb{N}, \mathsf{inc} : \mathbf{1}\}$$

Note that a primitive of type $A \to B$ is described in the signature as an A-indexed family of operations of arity B. For example, $\mathsf{set} : \mathbb{N} \to \mathbf{1}$ corresponds to one operation $\mathsf{set}(i) : \mathbf{1}$ for each possible index $i \in \mathbb{N}$.

Example 3.3. The operations of the layer interfaces presented in Fig. 1 can be described by the following effect signatures:

$$E_{\mathsf{bq}} := \{\mathsf{enq}(v) : \mathbf{1},\ \mathsf{deq} : \mathbb{U} \mid v \in \mathbb{U}\} \qquad E_{\mathsf{rb}} := \{\mathsf{set}(i, v) : \mathbf{1},\ \mathsf{get}(i) : \mathbb{U},\ \mathsf{fai}_1 : \mathbb{N},\ \mathsf{fai}_2 : \mathbb{N} \mid i \in \mathbb{N}, v \in \mathbb{U}\}$$

An effect signature already defines a certain structure of interaction in the sense that a caller issues an effect $e \in E$ and potentially receives a response $v \in \mathsf{ar}(e)$ from its environment. In this way, an effect signature E defines a very small game where the possible moves are effects of E or responses $\cup_{e \in E} \mathsf{ar}(e)$ to effects of E. The only valid plays in this game are:

$$\epsilon \qquad e \qquad e \cdot v$$

which are simply the empty play, a call to $e \in E$, and a call to $e \in E$ followed by a return value $v \in \mathsf{ar}(e)$ to e.

3.2 Layer Implementations

Our primary goal in this section is to express how an overlay with effect signature F is implemented using an underlay with effect signature E. To that purpose, we introduce the notion of implementation $M : E \to F$ of the signature F in terms of the signature E. This notion of implementation is formulated as a family $M = (M^f)_{f \in F}$ of game-semantics strategies M^f, which we call implementation strategies, over the signature E associated to each effect $f \in F$.

When implementing an overlay with signature F using an underlay with signature E a single operation f of the overlay F may require several operations over E to respond with a value $v \in \mathsf{ar}(f)$. This suggests a different pattern of interaction than what we discussed in §3.1, as the game associated with E may be replayed several times in sequence. This leads to considering a game, call it Replay E, which allows for the same moves as E but lets the game defined by E be replayed as many times as necessary, so that the set of valid plays is given by sequences of shape:

$$\epsilon \qquad e_1 \qquad e_1 \cdot v_1 \qquad e_1 \cdot v_1 \cdot e_2 \qquad \dots \qquad e_1 \cdot v_1 \cdot \dots \cdot e_n \cdot v_n$$

that is, a sequence of completed plays of E followed by a potentially partial play of E.

Then, we consider a play of $E \longrightarrow F$, a complete interaction leading to the implementation of an effect f, as a play of Replay E bracketed by a play of F, like so:

$$f \cdot e_1 \cdot v_1 \cdot \dots \cdot e_n \cdot v_n \cdot v$$

where $f \in F$, $v \in \mathsf{ar}(f)$ and for all i, $e_i \in E$ and $v_i \in \mathsf{ar}(e_i)$. Any partial interactions matching this shape are also possible plays, for instance

$$\epsilon \qquad f \qquad f \cdot e_1 \cdot v_1 \cdot \dots \cdot e_k \qquad f \cdot e_1 \cdot v_1 \cdot \dots \cdot e_k \cdot v_k$$

These remarks allow us to define a notion of implementation as follows:

Definition 3.4. Let E and F be effect signatures. An implementation $M : E \longrightarrow F$ is a non-empty set of plays of $E \longrightarrow F$ such that

(1) M is closed under the prefix order $\sqsubseteq$: If $s \in M$ and $p \sqsubseteq s$ then $p \in M$.
(2) M is receptive: $f \in M$ for every $f \in F$, and for every $e \in E$ if $s \cdot e \in M$ and $v \in \mathsf{ar}(e)$ then $s \cdot e \cdot v \in M$.
(3) M is deterministic: If $s \cdot m \cdot n$, $s \cdot m \cdot n' \in M$ are even-length plays then $n = n'$.

Receptivity means that M accepts any operation $f \in F$ played by its client, as well as any return value $v \in \mathsf{ar}(e)$ played by its underlay in response to a call made by M to the operation e. Note also that determinism means that

$$s \cdot e,\ s \cdot e' \in M \Rightarrow e = e'$$

so that the implementation calls the same effect of the underlay next if their past histories are the same. It also implies that

$$s \cdot v,\ s \cdot v' \in M \Rightarrow v = v'$$

so that if the same code in the underlay was executed with the same returns, then the same return is given to the overlay effect being implemented. Furthermore, because of determinism, in no condition may the plays $f \cdot s \cdot v$ and $f \cdot s \cdot e$, where $v \in \mathsf{ar}(f)$ and $e \in E$, belong to the same implementation M.

An implementation $M : E \longrightarrow F$ may be decomposed into sets

$$M^f := \{s \in M \mid f \sqsubseteq s\}$$

that is, M^f is the set of plays that implement the effect f. This is verified by the equation

$$M = \{\epsilon\} \uplus \biguplus_{f \in F} M^f$$

In fact, given a collection $(M^f)_{f \in F}$ such that for each f the set of plays M^f is an implementation $M^f : E \longrightarrow F$ that only has plays starting with f, the set M defined as

$$M = \{\epsilon\} \uplus \bigcup_{f \in F} M^f$$

is an implementation $M : E \longrightarrow F$. This way, implementations are in one-to-one correspondence to collections $(M^f)_{f \in F}$ of implementations of each effect $f \in F$.

Example 3.5. The code presented in Example 2.8 can easily be encoded as the sets of plays

$$M^{\mathsf{inc}} := \downarrow\{\mathsf{inc} \cdot \mathsf{get}_{\mathsf{Var}} \cdot n \cdot \mathsf{set}(n+1) \cdot \mathsf{ok} \cdot \mathsf{ok} \mid n \in \mathbb{N}\} \quad M^{\mathsf{get}} := \downarrow\{\mathsf{get}_{\mathsf{Counter}} \cdot \mathsf{get}_{\mathsf{Var}} \cdot n \cdot n \mid n \in \mathbb{N}\}$$

where $\downarrow S = \{s \mid \exists t \in S. s \sqsubseteq t\}$ is the prefix ordering down-closure of S. The correspondence between the code and the sets of plays should be apparent. The full implementation $M : \mathsf{Var} \to \mathsf{Counter}$ is then simply $M^{\mathsf{inc}} \cup M^{\mathsf{get}}$.

Example 3.6. The strategy associated with the implementation $M_{\mathsf{bq}} : E_{\mathsf{rb}} \to E_{\mathsf{bq}}$ outlined in Fig. 1 can be described as:

$$M_{\mathsf{bq}}^{\mathsf{enq}(v)} := \downarrow\{\mathsf{enq}(v) \cdot \mathsf{fai}_2 \cdot n \cdot \mathsf{set}(n, v) \cdot \mathsf{ok} \cdot \mathsf{ok} \mid n \in \mathbb{N}\}$$
$$M_{\mathsf{bq}}^{\mathsf{deq}} := \downarrow\{\mathsf{deq} \cdot \mathsf{fai}_1 \cdot n \cdot \mathsf{get}(n) \cdot v \cdot v \mid n \in \mathbb{N} \wedge v \in \mathbb{U}\}$$

In order to model the vertical composition operation of Gu et al. [2015, 2018] it will be necessary to compose implementations. So consider an implementation $M : E \to F$ and an implementation $N : F \to G$. We would like to produce an implementation $N \circ M : E \to G$. In order to do so the implementation M will need to be used several times, as N might make several calls to effects in F in order to implement a single call/return event from G. To this end, given an implementation $M : E \to F$ we define the set $\widehat{M}$ of plays, called its regular extension, as the set

$$\widehat{M} := \{s_1 \cdot \ldots \cdot s_n \mid s_1, \ldots, s_n \in M \text{ and } s_1 \cdot \ldots \cdot s_n \text{ is a Replay } E \multimap \text{Replay } F \text{ play}\}$$

so that $\widehat{M}$ describes the plays resulting from using M several times to implement a play of Replay F.

Example 3.7. Consider the $M : \mathsf{Var} \to \mathsf{Counter}$ defined in Example 3.5. Its regular extension includes plays such as

$$\underbrace{\mathsf{get}_{\mathsf{Counter}} \cdot \mathsf{get}_{\mathsf{Var}} \cdot a \cdot a}_{M} \cdot \underbrace{\mathsf{inc} \cdot \mathsf{get}_{\mathsf{Var}} \cdot b \cdot \mathsf{set}(b+1) \cdot \mathsf{ok} \cdot \mathsf{ok}}_{M} \cdot \underbrace{\mathsf{get}_{\mathsf{Counter}} \cdot \mathsf{get}_{\mathsf{Var}} \cdot c \cdot c}_{M}$$

In the following definition, if s is a sequence involving events in signatures E, F, G we use the notation $s\restriction_{E,F}$ to denote the subsequence of s including all but only the events in E or F. Later we use the unary variation $s\restriction_{E}$ for the subsequence including all but only the events in E. Then, implementation composition is defined as

Definition 3.8. Let $M : E \to F$ and $N : F \to G$ be implementations. Then, the implementation $N \circ M : E \to G$ is defined as

$$N \circ M := \{s\restriction_{E,G} \mid s\restriction_{E,F} \in \widehat{M} \text{ and } s\restriction_{F,G} \in N\}$$

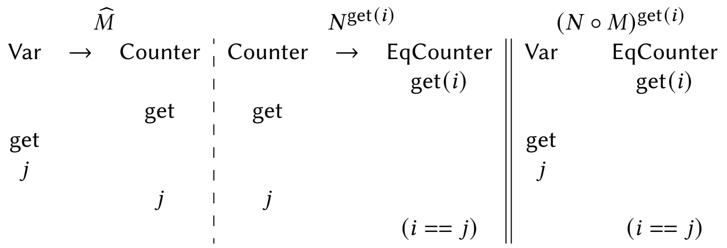
Example 3.9. Suppose we want to use a counter, with signature Counter as in example 3.2, to implement an interface with signature

$$\mathsf{EqCounter} := \{\mathsf{get} : \mathbb{N} \to \mathbb{B}, \mathsf{inc} : \mathbf{1}\}$$

where $\mathbb{B} = \{\mathsf{True}, \mathsf{False}\}$. The difference between Counter and EqCounter is that in EqCounter the get operation takes an integer as argument, compares it against the current value of the counter, and returns whether or not the value of the counter is equal to the argument to get. This can be implemented by $N : \mathsf{Counter} \to \mathsf{EqCounter}$ defined as

$$N^{\mathsf{inc}} := \downarrow\{\mathsf{inc}_{\mathsf{EqCounter}} \cdot \mathsf{inc}_{\mathsf{Counter}} \cdot \mathsf{ok} \cdot \mathsf{ok}\} \quad N^{\mathsf{get}(i)} := \downarrow\{\mathsf{get}(i) \cdot \mathsf{get} \cdot j \cdot (i == j)\}$$

where we use $- == -$ to denote the boolean function checking for equality of two integers. Now, given the implementation $M : \mathsf{Var} \to \mathsf{Counter}$ from Example 3.5 we can construct $N \circ M : \mathsf{Var} \to \mathsf{EqCounter}$ using Definition 3.8. Then, the general shape for a play in $(N \circ M)^{\mathsf{get}(i)}$ is depicted by the following graphical descriptions:



For the particular play we depicted the interaction s in the definition of composition is

$$s = \mathsf{get}(i) \cdot \mathsf{get}_{\mathsf{Counter}} \cdot \mathsf{get}_{\mathsf{Var}} \cdot j \cdot j \cdot (i == j)$$

so that we verify that

$$s\upharpoonright_{\mathsf{Var},\mathsf{Counter}} = \mathsf{get}_{\mathsf{Counter}} \cdot \mathsf{get}_{\mathsf{Var}} \cdot j \cdot j \in \widehat{M} \quad s\upharpoonright_{\mathsf{Counter},\mathsf{EqCounter}} = \mathsf{get}(i) \cdot \mathsf{get}_{\mathsf{Counter}} \cdot j \cdot (i == j) \in N$$

and therefore

$$s\upharpoonright_{\mathsf{Var},\mathsf{EqCounter}} = \mathsf{get}(i) \cdot \mathsf{get}_{\mathsf{Var}} \cdot j \cdot (i == j) \in N \circ M$$

Definition 3.10. At this point we are ready to define a category **Layer** whose objects are effect signatures E, F and whose morphisms from E to F are the implementations $M : E \longrightarrow F$. Composition is as in Definition 3.8 and the identity implementation for an effect signature E is given by

$$I_E := \downarrow\{e \cdot e \cdot v \cdot v \mid e \in E \wedge v \in \mathsf{ar}(E)\}$$

3.3 Layer Interfaces

We introduce in this section the notion of *layer interface* defined as a pair (E, V_E) consisting of an effect signature E specifying the interface for the objects, together with an object strategy V_E which specifies the interactive behavior of the layer interface. We first define the notion of object strategy and then give an illustration with our running example of ring buffers and bounded queues.

Definition 3.11. A (deterministic) *object strategy* over an effect signature E is a non-empty set of plays V_E of Replay E which satisfies

(1) the strategy V_E is prefix-closed.
(2) the strategy V_E is receptive:

If $s \in V_E$ is an even-length play and $e \in E$ then $s \cdot e \in V_E$.

(3) the strategy V_E is deterministic:

If $s \cdot e \cdot v, s \cdot e \cdot v' \in V_E$ are even-length plays then $v = v'$.

We denote by $\mathbf{S}_E$ the set of object strategies over E.

Definition 3.12. A *layer interface* is a pair $L = (E, V_E)$ of an effect signature E and of an object strategy V_E over E.

Example 3.13. In general, given a state-based description of a layer interface L of the kind used in Fig. 1, we can obtain the set of plays $L\sharp q$ induced by a state q with the recursive condition:

$$\epsilon \in L\sharp q\,; \quad m \cdot n \cdot s \in L\sharp q \Leftrightarrow \exists q'\,.\,(n, q') \in L.m@q \wedge s \in L\sharp q'$$

The empty queue ϵ is a natural initial state, so we define (where L^S_{bq} is the state-based specification in Fig. 1):

$$L_{\mathsf{bq}} := (E_{\mathsf{bq}}, V_{\mathsf{bq}}) \qquad V_{\mathsf{bq}} := L^S_{\mathsf{bq}}\sharp\epsilon$$

For example, for all $u, v \in \mathbb{U}$, L_{bq} allows the following play, as witnessed by the sequence of states $\epsilon, u, uv, v, \epsilon$.

$$\mathsf{enq}(u) \cdot \mathsf{ok} \cdot \mathsf{enq}(v) \cdot \mathsf{ok} \cdot \mathsf{deq} \cdot u \cdot \mathsf{deq} \cdot v \in V_{\mathsf{bq}}\,,$$

For ring buffers, we prefer not to make any assumptions on initial contents, so that the get operation on a location which has not yet been set is undefined. The corresponding layer interface is (again, we denote by L^S_{rb} the state-based specification in Fig. 1):

$$L_{\mathrm{rb}} := (E_{\mathrm{rb}}, V_{\mathrm{rb}}) \qquad V_{\mathrm{rb}} := \bigcap_{f \in \mathbb{U}^N} L^S_{\mathrm{rb}} \sharp (f, 0, 0)$$

In this case, $\mathrm{set}(i, v) \cdot \mathrm{ok} \cdot \mathrm{get}(i) \cdot v$ is a play in V_{rb} for all $i < N$, $v \in \mathbb{U}$, but $\mathrm{get}(i) \cdot v$ on its own is never accepted when $|\mathbb{U}| > 1$.

3.4 Certified Layer Implementations

We have just seen in §3.2 how to define a notion of implementation $M : E \to F$ of an effect signature F in terms of an effect signature E. We now adapt and refine this definition to obtain a notion of certified implementation

$$M : (E, V_E) \to (F, V_F)$$

between layer interfaces, as defined in Definition 3.12.

For an implementation $M : E \to F$ we will use the notation $s \overset{M}{\hookrightarrow} t$ to mean that M can implement the play t of Replay F using the underlay play s of Replay E. Formally:

$$s \overset{M}{\hookrightarrow} t \iff \exists p \in \widehat{M}. p \upharpoonright_E = s \text{ and } p \upharpoonright_F = t$$

Definition 3.14. Let $L_E = (E, V_E)$ and $L_F = (F, V_F)$. A *certified implementation* $M : L_E \to L_F$ is an implementation $M : E \to F$ such that

$$\forall t \in V_F . \exists s \in V_E . s \overset{M}{\hookrightarrow} t$$

We also find convenient to use the notation:

$$V_E \overset{M}{\hookrightarrow} V_F \equiv \forall t \in V_F . \exists s \in V_E . s \overset{M}{\hookrightarrow} t.$$

Example 3.15. Building on Example 3.13, the correctness of M_{bq} can be established using the simulation relation R given in Fig. 1. We can show by induction on plays that:

$$\vec{q}\ R\ (f, c_1, c_2) \Rightarrow L_{\mathrm{rb}} \sharp (f, c_1, c_2) \overset{M_{\mathrm{bq}}}{\hookrightarrow} L_{\mathrm{bq}} \sharp \vec{q}$$

For $\epsilon \in L_{\mathrm{bq}} \sharp \vec{q}$ we have $\epsilon \overset{M_{\mathrm{bq}}}{\hookrightarrow} \epsilon$ and $\epsilon \in L_{\mathrm{rb}} \sharp (f, c_1, c_2)$. For $(n, \vec{q}\,') \in L_{\mathrm{bq}}.m$ and $s \in L_{\mathrm{bq}} \sharp \vec{q}\,'$, we only need to witness a related state $\vec{q}\,'\ R\ (f', c_1', c_2')$ of L_{rb} reached by the corresponding sequence of operations in M_{bq}. Since the initial states $\epsilon\ R\ (f, 0, 0)$ are related for all $f \in \mathbb{U}^N$, we can conclude $V_{\mathrm{rb}} \overset{M_{\mathrm{bq}}}{\hookrightarrow} V_{\mathrm{bq}}$.

Definition 3.16. The category **CertiLayer** has layer interfaces as objects and certified implementations $M : (E, V_E) \to (F, V_F)$ as morphisms, with composition and identities defined as in the category **Layer**.

Layer interfaces support a simple notion of refinement defined by

$$(E, V_E) \sqsubseteq (E, V_E') \iff V_E \subseteq V_E'$$

The refinement order $\sqsubseteq$ defines a refinement system satisfying the usual refinement law.

Proposition 3.17. *$(E, V_E) \sqsubseteq (E, V_E')$ if and only if the identity implementation on E is a certified implementation from (E, V_E) to (E, V_E').*

Which immediately implies that:

Corollary 3.18. *Suppose that $L_1 \sqsubseteq L_1'$ and $L_2' \sqsubseteq L_2$. Then, if $M : L_1 \to L_2$ then $M : L_1' \to L_2'$.*

Remark 1. *Another way to formulate Definition 3.8 would be to proceed along the lines of game semantics, and to see the implementation $M : E \to F$ as a strategy from* Replay *E to F, and similarly for N. In that prospect, the set $\widehat{M}$ of plays defines a strategy from* Replay *E to* Replay *F which may be then composed with the strategy N from* Replay *F to G in order to obtain the strategy $N \circ M$ from* Replay *E to G. Definition 3.14 of certified implementation may be reformulated in this spirit by observing that the property $V_E \overset{M}{\hookrightarrow} V_F$ is equivalent to the fact that V_F seen as a strategy of* Replay *F is refined by the composite of V_E seen as a strategy of* Replay *E with the strategy $\widehat{M}$ from* Replay *E to* Replay *F, see Oliveira Vale et al. [2021] for details.*

4 NON-DETERMINISTIC LAYER INTERFACES

In this section, we generalize the notions of layer interface formulated in §3 in order to accommodate specific forms of nondeterminism in the specification of layers. We start by introducing the notion of nondeterministic layer interface.

Definition 4.1. A *non-deterministic layer interface* $\mathcal{L} = (E, \mathcal{V}_E)$ is a pair consisting of an effect signature E and set $\mathcal{V}_E \subseteq \mathbf{S}_E$ of object strategies. We further require $\mathcal{V}_E$ to be upward closed under the refinement order:

$$\forall V_E \in \mathcal{V}_E.\ \forall V_E' \in \mathbf{S}_E.\ \ V_E \sqsubseteq V_E' \ \Rightarrow\ V_E' \in \mathcal{V}_E$$

Given an arbitrary set $\mathcal{V}_E$ of object strategies, we will write its upward closure $\uparrow\mathcal{V} = \{V' \in \mathbf{S}_E \mid \exists V \in \mathcal{V}. V \sqsubseteq V'\}$ This means in particular that a layer interface (E, V) can be promoted to its nondeterministic counterpart as $(E, \uparrow\{V\})$.

Definition 4.2. A *certified implementation* $M : \mathcal{L}_E \to \mathcal{L}_F$ between nondeterministic layer interfaces is an implementation $M : E \to F$ such that

$$\forall V_E \in \mathcal{V}_E.\ \exists V_F \in \mathcal{V}_F.\ (E, V_E) \overset{M}{\hookrightarrow} (F, V_F).$$

The intuition behind these definitions is that the behavior of a *nondeterministic* layer interface $(E, \mathcal{V}_E)$ is described by the set $\mathcal{V}_E$ of *deterministic* object strategies potentially chosen to implement the layer interface. An implementation $M : E \to F$ defines a certified implementation $M : \mathcal{L}_E \to \mathcal{L}_F$ when for all object strategies V_E of the underlay, there is an object strategy V_F of the overlay included in the composite of M and V_E.

The category **CertiLayerND** has non-deterministic layer interfaces $\mathcal{L}_E, \mathcal{L}_F$ as objects and certified implementations $M : \mathcal{L}_E \to \mathcal{L}_F$ as morphisms. Composition and identities are as in **Layer**.

Example 4.3. Recall that in Example 3.13, we defined

$$L_{\text{rb}} := (E_{\text{rb}}, V_{\text{rb}}) \qquad\qquad V_{\text{rb}} := \bigcap_{f \in \mathbb{U}^N} L^S_{\text{rb}} \sharp (f, 0, 0)$$

allowing arbitrary initial contents in L_{rb}. In fact, the correctness of M_{bq} is also insensitive to the counters' initial value, since $\epsilon\ R\ (f, c, c)$ for all $c < N$. However, we cannot define V_{rb} as

$$V_{\text{rb}}^{\text{wrong}} := \bigcap_{f \in \mathbb{U}^N} \bigcap_{c < N} L^S_{\text{rb}} \sharp (f, c, c)$$

which would make the behavior of fai_1 and fai_2 completely undefined, similarly to the initial behavior of get.

By contrast, the model introduced by Definition 4.1 gives a more fine-grained way to weaken constraints on L_{rb}:

$$\mathcal{L}_{\mathrm{rb}} := (E_{\mathrm{rb}}, \mathcal{V}_{\mathrm{rb}}) \qquad \mathcal{V}_{\mathrm{rb}} := \uparrow\{L^S_{\mathrm{rb}} \sharp (f, c, c) \mid f \in \mathbb{U}^N, c < N\}$$
$$\mathcal{L}_{\mathrm{bq}} := (E_{\mathrm{bq}}, \mathcal{V}_{\mathrm{bq}}) \qquad \mathcal{V}_{\mathrm{bq}} := \uparrow\{V_{\mathrm{bq}}\}$$

Then $M_{\mathrm{bq}} : \mathcal{L}_{\mathrm{rb}} \to \mathcal{L}_{\mathrm{bq}}$ remains a certified implementation in the sense of Definition 4.2.

We can easily adapt the notion of refinement $\sqsubseteq$ between deterministic layer interfaces, in the following way:

$$(E, \mathcal{V}') \sqsubseteq (E, \mathcal{V}) \iff \mathcal{V}' \supseteq \mathcal{V}$$

Just as in §3.4, we have that $(E, \mathcal{V}') \sqsubseteq (E, \mathcal{V})$ if and only if the identity implementation $\mathbf{I}_E : E \to E$ on the effect signature E is a certified implementation. From this follows an immediate adaptation of Proposition 3.18:

Proposition 4.4. *Given $\mathcal{L}_1 \sqsubseteq \mathcal{L}'_1$ and $\mathcal{L}'_2 \sqsubseteq \mathcal{L}_2$, if $M : \mathcal{L}_1 \to \mathcal{L}_2$, then $M : \mathcal{L}'_1 \to \mathcal{L}'_2$.*

5 CORRESPONDENCE WITH OBJECT-BASED SEMANTICS IN COHERENCE SPACES

In §2 we reviewed Reddy's object-based semantics in coherence spaces, and in §3 we introduced an interactive game model of certified layers with many similarities to Reddy's object-based semantics. In this section we discuss a way of connecting the two semantics.

5.1 The Category Reg of Regular Maps

We start by observing that in §2 we have delineated all of the structure for the category of coherence spaces, defined simply as

Definition 5.1. The category **Coh** has coherence spaces A, B as objects and linear maps $f : A \multimap B$ as morphisms. Composition and identity are relational composition $- \circ -$, and the diagonal relation $\mathbf{id}_{-}$ respectively.

We also take the opportunity to define the category **Reg** of object spaces:

Definition 5.2. The category **Reg** has coherence spaces A, B as objects and regular maps $f : \dagger A \multimap B$ as morphisms. The composite of two regular maps $f : \dagger A \multimap B$ and $g : \dagger B \multimap C$ is defined as the regular map $g \circ \widehat{f} : \dagger A \multimap C$ as explained in §2. The identity morphism of A in **Reg** is the regular map $\epsilon_A : \dagger A \multimap A$ defined as $\epsilon_A := \{\langle a \rangle \mapsto a \mid a \in |A|\}$. Note that the category **Reg** is the co-Kleisli category associated to the comonad $\dagger : \mathbf{Coh} \to \mathbf{Coh}$ on the category **Coh** of coherence spaces.

We then introduce the category **CertiReg** which refines the category **Reg** of regular maps in the same way as the category **CertiLayer** refines **Layer** in §3.

Definition 5.3. The objects of **CertiReg** are the pairs (A, W_A) consisting of a coherence space A and of a clique W_A of the coherence space $\dagger A$. A morphism $M : (A, W_A) \to (B, W_B)$ of the category **CertiReg** is defined as a regular map $M : \dagger A \multimap B$ such that $\widehat{M} \circ W_A \supseteq W_B$ where $\subseteq$ is the (set-theoretic) inclusion of linear maps.

5.2 Effect Signatures to Coherence Spaces

We can associate to every effect signature E a coherence space $[\![E]\!]$ defined as

$$|[\![E]\!]| = \{e.v \mid e \in E \text{ and } v \in \mathsf{ar}(e)\} \qquad e.v \frown_{[\![E]\!]} e'.v' \iff (e = e' \Rightarrow v = v')$$

Every token $e.v \in [\![E]\!]$ is a pair consisting of an effect and a return value (or arity) associated to this effect. Coherence encodes a form of determinism, which ensures that there exists *at most* one possible return value $v \in \mathrm{ar}(e)$ for a given effect $e \in E$ in a clique of $[\![E]\!]$. This means in particular that given an effect e and two possible return values $v, v' \in \mathrm{ar}(m)$, two tokens of the coherence space $\dagger[\![E]\!]$ of the form $s \cdot (e.v)$ and $s \cdot (e.v')$ are coherent precisely when $v = v'$.

The translation from effect signatures E to the underlying coherence space $[\![E]\!]$ suggests that we can see plays $e \cdot v$ in E as tokens $e.v \in [\![E]\!]$. We can then interpret any even-length play of Replay E as sequence in $\dagger[\![E]\!]$ as follows:

$$[\![e_1 \cdot v_1 \cdot \ldots e_n \cdot v_n]\!] = \langle e_1.v_1, \ldots, e_n.v_n \rangle$$

which in fact defines an order-preserving bijection (with respect to prefix ordering on sequences) between the plays encoded by Replay E and tokens of $\dagger[\![E]\!]$. From now on we allow ourselves to apply this bijection tacitly whenever we need it.

5.3 From Implementations to Certified Regular Maps

We start by noting that every implementation $M : E \longrightarrow F$ between effect structures E, F can be translated to a regular map $[\![M]\!] : \dagger[\![E]\!] \multimap [\![F]\!]$ between coherence spaces in the following way:

$$[\![M]\!] := \{s \mapsto f.v \mid f \cdot s \cdot v \in M\}$$

We can establish that the translation is functorial in the sense that

Proposition 5.4. *The translation $[\![-]\!]$ defines a* full *(but not faithful) functor* $[\![-]\!]$: **Layer** $\longrightarrow$ **Reg**

The functor $[\![-]\!]$ is full because every regular map $N : \dagger[\![E]\!] \multimap [\![F]\!]$ can be turned into an object strategy $M : E \longrightarrow F$ such that $N = [\![M]\!]$ defined as the receptive closure of

$$M = \downarrow\{f \cdot s \cdot v \mid s \mapsto f.v \in N\}$$

On the other hand the functor $[\![-]\!]$ is not faithful because two object strategies $M, M' : E \longrightarrow F$ which differ only on partial behaviors are translated to the same regular map $[\![M]\!] = [\![M']\!]$. The reason is that the functor $[\![-]\!]$ captures exactly the complete behaviors of object strategies.

We then extend the functor $[\![-]\!]$: **Layer** $\longrightarrow$ **Reg** defined in §5.3 to a functor

$$[\![-]\!] : \mathbf{CertiLayer} \longrightarrow \mathbf{CertiReg} \tag{3}$$

To that purpose we observe that for every effect signature E,

Proposition 5.5. *If V_E is an object strategy over the effect signature E then its set of even-length plays is a non-empty, prefix-closed clique of the associated coherence space $\dagger[\![E]\!]$.*

Thanks to this observation we can associate to every layer interface (E, V_E) in **CertiLayer** the corresponding pair in **CertiReg**,

$$(E, V_E) \mapsto ([\![E]\!], [\![V_E]\!])$$

where $[\![V_E]\!]$ is simply the clique in $\dagger[\![E]\!]$ corresponding to the even-length plays of the object strategy V_E. This then allows us to extend the functor $[\![-]\!]$: **CertiLayer** $\longrightarrow$ **CertiReg** defined in §5.3 to a functor

$$[\![-]\!] : \mathbf{CertiLayerND} \longrightarrow \mathbf{CertiRegND}$$

by applying to sets $\mathcal{V}_E$ of object strategies the action of the original functor $[\![-]\!]$ in (3) to object strategies $V_E \in \mathcal{V}_E$, in the following wayL

$$[\![\mathcal{V}_E]\!] := \{[\![V_E]\!] \mid V_E \in \mathcal{V}_E\} \qquad [\![(E, \mathcal{V}_E)]\!] := ([\![E]\!], [\![\mathcal{V}_E]\!])$$

The image of a non-deterministic layer interface $(E, \mathcal{V}_E)$ is defined as seen above, while implementations $M : (E, \mathcal{V}_E) \longrightarrow (F, \mathcal{V}_F)$ are mapped to the same regular map $[\![M]\!]$ as in §5. We obtain in this way a commutative diagram:

$$\begin{tikzcd}
\mathbf{CertiLayer} \arrow[r, "{[\![-]\!]}"] \arrow[d, "\text{inclusion}"'] & \mathbf{CertiReg} \arrow[r, "\text{inclusion}"] \arrow[d, "\text{inclusion}"] & \mathbf{CertiRegConc} \arrow[d, "\text{inclusion}"] \\
\mathbf{CertiLayerND} \arrow[r, "{[\![-]\!]}"] \arrow[d, "\text{forget}"'] & \mathbf{CertiRegND} \arrow[r, "\text{inclusion}"] \arrow[d, "\text{forget}"] & \mathbf{CertiRegConcND} \arrow[d, "\text{forget}"] \\
\mathbf{Layer} \arrow[r, "{[\![-]\!]}"] & \mathbf{Reg} \arrow[r, "\text{inclusion}"] & \mathbf{RegConc}
\end{tikzcd}$$

which may be seen as a map of functorial refinement systems [Melliès and Zeilberger 2015]. This expresses very concisely in what sense the categories **CertiReg** and **CertiRegND** refine the category **Reg** in the same way as the categories **CertiLayer** and **CertiLayerND** refine the category **Layer**. The grayed-out categories will be discussed in §6, except for **CertiRegConcND** which is the result of applying the abstract construction from §4 to **CertiRegConc**.

6 CONCURRENT OBJECT SPACES

So far we have only discussed models of sequential systems. The models we defined are expressive enough to capture stateful sequential computation with an elegant decomposition of statefulness into a state-less implementation and a stateful sequential specification. Challenges arise when attempting to faithfully model concurrent computation. In order to discuss this situation, we take full advantage of the correspondence shown in §5.

So let us consider a very simple concurrent system, where two variable objects are used concurrently to implement two independent counters. This can be modeled by an underlay signature

$$\mathsf{Var} + \mathsf{Var} = \{\mathbf{1}{:}\mathsf{get} : \mathbb{N}, \mathbf{1}{:}\mathsf{set} : \mathbb{N} \to \mathbf{1}\} \cup \{\mathbf{2}{:}\mathsf{get} : \mathbb{N}, \mathbf{2}{:}\mathsf{set} : \mathbb{N} \to \mathbf{1}\}$$

In coherence spaces this signature nicely corresponds to the product

$$[\![\mathsf{Var} + \mathsf{Var}]\!] = [\![\mathsf{Var}]\!] \,\&\, [\![\mathsf{Var}]\!]$$

where the *with* of coherence spaces A and B, $A \,\&\, B$, is defined by

$$|A \,\&\, B| := |A| + |B| \qquad \boldsymbol{X}{:}x \frown_{A\&B} \boldsymbol{Y}{:}y \iff X = Y \Rightarrow x \frown_X y$$

The unit for & is the empty coherence space $\top$. For conciseness we will omit applications of the functor $[\![-]\!]$ to effect signatures to no harm, so we may write Var & Var, for example.

Now, an implementation in our models corresponds to a regular map $\dagger(\mathsf{Var} \,\&\, \mathsf{Var}) \multimap (\mathsf{Counter} \,\&\, \mathsf{Counter})$. If we assume each agent can only call the operations labelled with their own name, such a map corresponds to two maps $\dagger\mathsf{Var} \multimap \mathsf{Counter}$ each representing the local implementation that each agent is running. We will use the usual implementation of Counter in terms of Var, as in Example 2.8. Then, the implementation for all the agents is given by

$$M \,\&\, M \quad : \quad \dagger(\mathsf{Var} \,\&\, \mathsf{Var}) \multimap (\mathsf{Counter} \,\&\, \mathsf{Counter})$$

where $M \,\&\, M$ is a labelled disjoint union of the implementation M with itself regarded as a set (including labeling the events within each copy of M). But note that this map only expresses a very limited form of concurrency. Namely, the implementation of a trace $t = \langle \mathbf{1}{:}\mathsf{inc.ok}, \mathbf{2}{:}\mathsf{inc.ok}\rangle$ by $\widehat{M}$ is always given by a sequence of shape

$$s_1 = \langle \mathbf{1}{:}\mathsf{get}.i, \mathbf{1}{:}\mathsf{set}(i+1).\mathsf{ok}, \mathbf{2}{:}\mathsf{get}.j, \mathbf{2}{:}\mathsf{set}(j+1).\mathsf{ok}\rangle \in \dagger(\mathsf{Var} \,\&\, \mathsf{Var})$$

which is completely atomic. The issue lies much deeper. Consider another interleaving on the underlay corresponding to calls to inc on the overlay. For instance,

$$s_2 = \langle \mathbf{1}{:}\mathsf{get}.i, \mathbf{2}{:}\mathsf{get}.j, \mathbf{1}{:}\mathsf{set}(i+1).\mathsf{ok}, \mathbf{2}{:}\mathsf{set}(j+1).\mathsf{ok}\rangle \in \dagger(\mathsf{Var} \,\&\, \mathsf{Var})$$

A purposed linear map $f : \dagger(\mathsf{Var} \,\&\, \mathsf{Var}) \multimap \dagger(\mathsf{Counter} \,\&\, \mathsf{Counter})$ that models the usual counter implementation on each counter should map both s_1, s_2 to the same sequence t: $s_1 \mapsto t \in f$ and $s_2 \mapsto t \in f$. But note that according to the definition of a linear map (Def. 2.3) this implies that $s_1 = s_2$, as $s_1 \frown_{\dagger(\mathsf{Var}\&\mathsf{Var})} s_2$. Therefore, there is no such linear map. Despite that, coherence spaces do support a notion of parallelism in the tensor product $\otimes$:

$$|A \otimes B| := |A| \times |B| \qquad (a, b) \frown_{A\otimes B} (a', b') \iff a \frown_A a' \wedge b \frown_B b'$$

So that a map $f : \dagger\mathsf{Var} \otimes \dagger\mathsf{Var} \multimap \dagger\mathsf{Counter} \otimes \dagger\mathsf{Counter}$ is possible. On the other hand, the category **Reg** does not have tensor products, a fact noted by Reddy [1996].

In this section we explore the issue by dissecting the category **Reg** as the category of free †-coalgebras, that is, coalgebras of the form $\dagger A$. We then consider a larger category of all †-coalgebras where we pinpoint a particularly elegant class of †-coalgebras, equipped with a tensor product, and with the expressive power to model a variety of concurrent systems.

6.1 The Replay Modality's Co-monadic Structure

We start by noting that $\dagger-$ is a comonad in **Coh**, with the structural maps

$$\begin{aligned} &\delta_A : \dagger A \multimap \dagger\dagger A && \epsilon_A : \dagger A \multimap A \\ &\delta_A := \{s_1 \cdot \ldots \cdot s_n \mapsto \langle s_1, \ldots, s_n\rangle \mid s_1, \ldots, s_n \in |\dagger A|\} && \epsilon_A := \{\langle a\rangle \mapsto a \mid a \in |A|\} \end{aligned}$$

which justifies the construction of **Reg** as the co-Kleisli category of $\dagger-$. As a functor, the action of $\dagger-$ on a linear map $f : A \multimap B$ is

$$\dagger f := \{\langle a_1, \ldots, a_n\rangle \mapsto \langle b_1, \ldots, b_n\rangle \mid \forall i \leq n. a_i \mapsto b_i \in f\}$$

which applies f element-wise through the input sequences. Note that in **Reg** the map ϵ_A plays the role of the identity morphism. In particular, the identity implementation I_E is mapped by $[\![-]\!]$ to ϵ_E, in other words: $[\![I_E]\!] = \epsilon_E$.

As explained in the introduction, see Fig. 2, the "decomposition" map $\kappa = \delta_A$ plays an essential role in lifting a map $f : \dagger A \multimap B$ to the regular map $\widehat{f} : \dagger A \rightarrow \dagger B$ defined as the composite

$$\dagger A \xrightarrow{\widehat{f}} \dagger B \quad = \quad \dagger A \xrightarrow{\delta_A} \dagger\dagger A \xrightarrow{\dagger f} \dagger B$$

Example 6.1. With the usual counter implementation $M : \dagger\mathsf{Var} \rightarrow \mathsf{Counter}$ we observe the composition

$$\langle \mathsf{get}.a, \mathsf{get}.b, \mathsf{set}(b+1).\mathsf{ok}, \mathsf{get}.c\rangle \overset{\delta_{\mathsf{Var}}}{\longmapsto} \langle\langle \mathsf{get}.a\rangle, \langle \mathsf{get}.b, \mathsf{set}(b+1).\mathsf{ok}\rangle, \langle \mathsf{get}.c\rangle\rangle \overset{\dagger M}{\longmapsto} \langle \mathsf{get}.a, \mathsf{inc}.\mathsf{ok}, \mathsf{get}.c\rangle$$

where δ_{Var} plays the role of decomposing the input trace before M can be replicated to map the input trace of $\dagger\mathsf{Var}$ to a trace of $\dagger\mathsf{Counter}$. Note also that there are many decompositions of the input trace that do not get mapped though $\dagger M$ and therefore do not appear in $\widehat{M}$.

Although the role of $\kappa = \delta_A$ in this setting is rather simple, it is fundamental for the structure of regular maps. We will see that this decomposition step plays a much subtler role for general †-coalgebras, which is fundamental to the simplicity of our model.

6.2 Identifying Interleavings

In §6 we noted that it is rather challenging to model the independent composition of objects because different interleavings of the underlay are all coherent, and we can't represent a $\otimes$ in **Reg**. On the other hand, as the objects are independent, we could identify all those interleavings as representing the same computation. We define a relation $R \subseteq |\dagger(\mathsf{Var} \mathbin{\&} \mathsf{Var})| \times |\dagger(\mathsf{Var} \mathbin{\&} \mathsf{Var})|$ as the smallest equivalence relation relating, for any $s, t \in |\dagger(\mathsf{Var} \mathbin{\&} \mathsf{Var})|$, $e.v, e'.v' \in \mathsf{Var}$:

$$i \neq j \Rightarrow s \cdot \langle \boldsymbol{i}{:}e.v\rangle \cdot \langle \boldsymbol{j}{:}e'.v'\rangle \cdot t \;\; R \;\; s \cdot \langle \boldsymbol{j}{:}e'.v'\rangle \cdot \langle \boldsymbol{i}{:}e.v\rangle \cdot t$$

Then, we can define a coherence space $\dagger_R(\mathsf{Var} \mathbin{\&} \mathsf{Var})$ by

$$|\dagger_R(\mathsf{Var} \mathbin{\&} \mathsf{Var})| := |\dagger(\mathsf{Var} \mathbin{\&} \mathsf{Var})|/R \qquad x \frown_{\dagger_R(\mathsf{Var\&Var})} y \iff \forall s \in x. \forall t \in y. s \frown_{\dagger(\mathsf{Var\&Var})} t$$

where $|\dagger(\mathsf{Var} \mathbin{\&} \mathsf{Var})|/R$ is the set of equivalence classes of R over $|\dagger(\mathsf{Var} \mathbin{\&} \mathsf{Var})|$. Similarly, we can define an analogous relation $S \subseteq |\dagger(\mathsf{Counter} \mathbin{\&} \mathsf{Counter})| \times |\dagger(\mathsf{Counter} \mathbin{\&} \mathsf{Counter})|$ and a space $\dagger_S(\mathsf{Counter} \mathbin{\&} \mathsf{Counter})$.

The usual Counter implementation in this setting can be formulated instead by defining two maps $M[i] : \iota_i(\mathsf{Var}) \multimap \iota_i(\mathsf{Counter})$, one for each $i \in \{1, 2\}$:

$$M[i] := \{\langle \boldsymbol{i}{:}\mathsf{get}.n\rangle \mapsto \boldsymbol{i}{:}\mathsf{get}.n \mid n \in \mathbb{N}\} \cup \{\langle \boldsymbol{i}{:}\mathsf{get}.n, \boldsymbol{i}{:}\mathsf{set}(n+1).\mathsf{ok}\rangle \mapsto \boldsymbol{i}{:}\mathsf{inc.ok} \mid n \in \mathbb{N}\}$$

Then, we can define a map $\widehat{M[1]} \otimes \widehat{M[2]} : \dagger_R(\mathsf{Var} \mathbin{\&} \mathsf{Var}) \multimap \dagger_S(\mathsf{Counter} \mathbin{\&} \mathsf{Counter})$ as

$$\widehat{M[1]} \otimes \widehat{M[2]} := \{[s_1 \cdot \ldots \cdot s_n]_R \mapsto [t_1 \cdot \ldots \cdot t_n]_S \mid \forall i \leq n. s_i \mapsto t_i \in \widehat{M[1]} \vee s_i \mapsto t_i \in \widehat{M[2]}\}$$

where $[s]_R$ denotes the equivalence class of R in which s belongs, and similarly for $[-]_S$ (we will often omit the subscript when it causes no confusion). Then, in our usual graphical presentation we observe that

$$\begin{array}{c} [s_2] = [\langle \mathbf{1}{:}\mathsf{get}.i, \mathbf{2}{:}\mathsf{get}.j, \mathbf{1}{:}\mathsf{set}(i+1).\mathsf{ok}, \mathbf{2}{:}\mathsf{set}(j+1).\mathsf{ok}\rangle] \\ = \\ [s_1] = [\langle \mathbf{1}{:}\mathsf{get}.i, \mathbf{1}{:}\mathsf{set}(i+1).\mathsf{ok}, \mathbf{2}{:}\mathsf{get}.j, \mathbf{2}{:}\mathsf{set}(j+1).\mathsf{ok}\rangle] \end{array} \xmapsto{\widehat{M[1]} \otimes \widehat{M[2]}} [\langle \mathbf{1}{:}\mathsf{inc.ok}, \mathbf{2}{:}\mathsf{inc.ok}\rangle] = t$$

6.3 Concurrent Object Spaces

We say that an equivalence relation $R \subseteq |\dagger A| \times |\dagger A|$ is *coherent* when

$$\forall s, t \in |\dagger A|. \quad s \mathrel{R} t \Rightarrow s \frown_{\dagger A} t$$

which we write more concisely $R \subseteq \frown_{\dagger A}$. Furthermore, we say that R is a *congruence* when

$$\forall p, s, s', t \in \dagger A. \quad s \mathrel{R} s' \Rightarrow p \cdot s \cdot t \mathrel{R} p \cdot s' \cdot t$$

Definition 6.2. The coherence space $\dagger_R A$ associated to a coherent congruence R is defined as

$$\dagger_R A := |\dagger A|/R \qquad x \frown_{\dagger_R A} y \iff \forall s \in x. \forall t \in y. s \frown_{\dagger A} t$$

A coherence space $\dagger_R A$ is called a *concurrent object space.*

The fact that R is a coherent congruence ensures that there is a linear map $\kappa_R : \dagger_R A \multimap \dagger\dagger_R A$ playing a similar role for $\dagger_R A$ as the "decomposition" map $\kappa = \delta_A : \dagger A \multimap \dagger\dagger A$ plays for the free $\dagger$-coalgebra $\dagger A$. The map κ_R is defined as

$$\kappa_R \quad := \quad [s_1 \cdot \ldots \cdot s_n] \mapsto \langle [s_1], \ldots, [s_n]\rangle \quad : \quad \dagger_R A \multimap \dagger\dagger_R A.$$

An important observation of the paper is that this map equips the concurrent object space $\dagger_R A$ with the structure of a $\dagger$-coalgebra. Recall that a $\dagger$-coalgebra is a pair $(C, \kappa : C \multimap \dagger C)$ of a coherence space C and linear map $\kappa : C \multimap \dagger C$ making the diagrams below commute:

$$\begin{tikzcd} C \arrow[r, "\kappa"] \arrow[d, "\kappa"'] & \dagger C \arrow[d, "\dagger\kappa"] \\ \dagger C \arrow[r, "\delta_C"] & \dagger\dagger C \end{tikzcd} \qquad \begin{tikzcd} C \arrow[r, "\kappa"] \arrow[dr, "\mathbf{id}_C"'] & \dagger C \arrow[d, "\epsilon_C"] \\ & C \end{tikzcd} \tag{4}$$

The †-coalgebras define a category †-**Coalg**, known as the Eilenberg-Moore category associated to the comonad †. Its objects are the †-coalgebras just described, and its morphisms $f : (C, \kappa) \to (C', \kappa')$ are the maps $f : C \to C'$ of **Coh** making the diagram below commute:

$$\begin{tikzcd} C \arrow[r, "f"] \arrow[d, "\kappa"'] & C' \arrow[d, "\kappa'"] \\ \dagger C \arrow[r, "\dagger f"] & \dagger C' \end{tikzcd} \tag{5}$$

We can summarize the observations made so far into the following proposition:

Proposition 6.3. *For every coherence space A and coherent congruence $R \subseteq |\dagger A| \times |\dagger A|$, the pair*

$$(\dagger_R A \, , \; \kappa_R : \dagger_R A \multimap \dagger\dagger_R A \,)$$

defines a †-coalgebra.

A detailed proof of proposition 6.3 can be found in Oliveira Vale et al. [2021].

Example 6.4. Note that the equivalence relations R and S introduced in §6.2 are both coherent congruences, so that $\dagger_R(\mathrm{Var} \mathbin{\&} \mathrm{Var})$ and $\dagger_S(\mathrm{Counter} \mathbin{\&} \mathrm{Counter})$ assemble into concurrent object spaces which are †-coalgebras by Proposition 6.3.

It should be noted that coalgebra morphisms of †-coalgebras described in (4) generalize in a very natural and pleasant way the notion of regular map $\dagger A \to \dagger B$ defined in §2.4 between object spaces. Indeed, an important (and well-known) fact is that the map $\delta_A : \dagger A \multimap \dagger\dagger A$ defines the free †-coalgebra generated by A:

$$(\dagger A, \delta_A : \dagger A \multimap \dagger\dagger A)$$

It then turns out that a regular map $\dagger A \to \dagger B$ in the sense of Def. 2.6 in Section §2.4 is same as a coalgebra morphism $\dagger A \to \dagger B$ between free †-coalgebras in the sense of (5). In particular, every regular map $\widehat{f} : \dagger A \to \dagger B$ associated to the linear map $f : \dagger A \multimap B$ makes the diagram below commute:

$$\begin{tikzcd} \dagger A \arrow[r, "\widehat{f}"] \arrow[d, "\delta_A"'] & \dagger B \arrow[d, "\delta_B"] \\ \dagger\dagger A \arrow[r, "\dagger\widehat{f}"] & \dagger\dagger B \end{tikzcd} \tag{6}$$

Looking backwards, this means that we have been working all along in the previous sections with †-coalgebras, even if only the free ones, of the form $\dagger A$. The challenges which arise with concurrency lead us to consider more general †-coalgebras such as concurrent object spaces, of the more general form $\dagger_R A$ for $R \subseteq |\dagger A| \times |\dagger A|$ for a coherent congruence. It should come as no surprise that we can define a full subcategory **RegConc** of the category of †-coalgebras given by restricting the objects of †-**Coalg** to such concurrent †-spaces. At the same time, we can recover $\dagger A$ as the concurrent object space $\dagger_= A$ associated to the specific identity relation R defined by equality =. The situation is nicely summarized by the chain of inclusion functors

$$\mathbf{Reg} \hookrightarrow \mathbf{RegConc} \hookrightarrow \dagger\text{-}\mathbf{Coalg}$$

Note that one main difference is that **RegConc** and †-**Coalg** are equipped with a parallel tensor product (discussed in §6.4), while this is not the case for the original category **Reg** of regular maps.

We have just seen in (6) how coalgebra morphisms are similar (and in fact extend) the usual notion of regular map $\dagger A \to \dagger B$ in the category **Reg**. A special case is of particular relevance to us: imagine that one is given a linear map $f : \dagger_R A \multimap B$ which, by analogy with regular maps, one would like to lift to a map $\dagger_R A \multimap \dagger_S B$. While this is in general *not* possible, in the case where S is the identity relation we may exploit the $\dagger$-coalgebra structure of $\dagger_R A$ to construct a morphism $\widehat{f} : \dagger_R A \multimap \dagger B$ in the following way:

$$\dagger_R A \xrightarrow{\widehat{f}} \dagger B \quad = \quad \dagger_R A \xrightarrow{\kappa_R} \dagger\dagger_R A \xrightarrow{\dagger f} \dagger B$$

The structural morphism κ_R plays here a very similar role as the "decomposition" map $\kappa = \delta_A$ discussed in the introduction for the sequential setting. One fundamental difference however is that κ_R may take advantage of the equational theory encoded in R prior to decomposing a trace.

6.4 A Parallel Tensor Product on Concurrent Object Spaces

Every pair of $\dagger$-coalgebras (C_1, κ_1) and (C_2, κ_2) defines a $\dagger$-coalgebra $C_1 \otimes C_2$ with structural map κ_{12} defined as the composition

$$C_1 \otimes C_2 \xrightarrow{\kappa_{12}} \dagger(C_1 \otimes C_2) \quad = \quad C_1 \otimes C_2 \xrightarrow{\kappa_1 \otimes \kappa_2} \dagger C_1 \otimes \dagger C_2 \longrightarrow \dagger(C_1 \otimes C_2)$$

where the second map is an instance of the structural map

$$\dagger\, A \otimes \dagger B \multimap \dagger(A \otimes B) \qquad (\langle a_1, \dots, a_n\rangle, \langle b_1, \dots, b_n\rangle) \mapsto \langle (a_1, b_1), \dots, (a_n, b_n)\rangle \tag{7}$$

This construction turns $\dagger$-**Coalg** into a symmetric monoidal category. The coherence space $\mathbf{1}$ (the usual unit for $\otimes$) is equipped with a $\dagger$-coalgebra structure provided by the structural map

$$\mathbf{1} \multimap \dagger\mathbf{1} \qquad * \mapsto \underbrace{\langle *, \dots, * \rangle}_{n \text{ times}} \tag{8}$$

The monoidal structure of $\dagger$-**Coalg** comes from the fact that (7) and (8) equip the comonad $\dagger$ with the structure of a symmetric monoidal comonad over **Coh** [Kock 1972][Melliès 2009].

We saw in §6.3 that every concurrent object space $\dagger_R A$ defines a $\dagger$-coalgebra. We establish now that our class of concurrent object spaces is closed under tensor product in the sense that

Proposition 6.5. *Given two concurrent object spaces $\dagger_R A$ and $\dagger_S B$ the tensor product of $\dagger_R A$ and $\dagger_S B$ is a concurrent object space $\dagger_{R \otimes S}(A \,\&\, B)$.*

Indeed, given relations $R \subseteq |\dagger A| \times |\dagger A|$ and $S \subseteq |\dagger B| \times |\dagger B|$ we define the relation

$$R \otimes S \subseteq |\dagger(A \,\&\, B)| \times |\dagger(A \,\&\, B)| \qquad s\ (R \otimes S)\ t \iff s{\upharpoonright_A}\ R\ t{\upharpoonright_A} \wedge s{\upharpoonright_B}\ S\ t{\upharpoonright_B}$$

which in addition to any equations from R and S also adds equations allowing for tokens of A and B to be swapped. This congruence has the remarkable property that it induces an isomorphism of $\dagger$-coalgebras

$$\dagger_R\, A \otimes \dagger_S B \quad \cong \quad \dagger_{R \otimes S}(A \,\&\, B) \tag{9}$$

which elegantly captures an equivalence between a true concurrency and an interleaving concurrency presentation of the same concurrent object, and should be seen as an analogue of the Seely isomorphism satisfied by the exponential modality $A \mapsto\, !A$ of linear logic (see Melliès [2009] for details). Thanks to this isomorphism (9) proved in Oliveira Vale et al. [2021], we establish the important property that our category **RegConc** of concurrent object spaces is equipped with a notion of parallel tensor product:

Proposition 6.6. ***RegConc** is a symmetric monoidal category.*

Example 6.7. As an illustration, coming back to the motivating equivalence relations R and S formulated in §6.2, we observe that they satisfy the isomorphisms

$$\dagger\mathsf{Var} \otimes \dagger\mathsf{Var} \;\cong\; \dagger_R(\mathsf{Var} \,\&\, \mathsf{Var}) \qquad \dagger\,\mathsf{Counter} \otimes \dagger\mathsf{Counter} \;\cong\; \dagger_S(\mathsf{Counter} \,\&\, \mathsf{Counter})$$

mentioned in (9) because the equivalence relations R and S in §6.2 are equal to $= \otimes =$.

6.5 Certified Concurrent Object Spaces

We are now ready to define our category of certified concurrent systems **CertiRegConc**, which refines **RegConc** in the same way **CertiReg** refines **Reg**. Its objects are triples (A, R, V_A) of a coherence space A, a coherent congruence $R \subseteq |\dagger A| \times |\dagger A|$ and $V_A : \mathbf{1} \multimap \dagger A$ a clique of $\dagger A$. Morphisms $M : (A, R, V_A) \longrightarrow (B, S, V_B)$ are coalgebra morphisms $M : \dagger_R A \multimap \dagger_S B$ satisfying the additional requirement that

$$\forall t \in V_B . \exists s \in V_A . \qquad [s]_R \mapsto [t]_S \in M$$

Identity and composition are as in $\dagger$-**Coalg**.

We have seen in §6.4 that **RegConc** comes equipped with a tensor product. We now extend the parallel tensor to **CertiRegConc**. Given (A, R, V_A) and (B, S, V_B) we would like that the underlying coherence space of their product

$$(A, R, V_A) \otimes (B, S, V_B)$$

be given by $A \,\&\, B$ so to match the relation $R \otimes S$. But taking the tensor product of the cliques V_A and V_B we obtain

$$V_A \otimes V_B : \mathbf{1} \multimap \dagger A \otimes \dagger B$$

which is *not* a clique of $\dagger(A \,\&\, B)$. In order to obtain such a clique, we make use of the interleaving morphism

$$\mathsf{inter}_{A,B} : \dagger A \otimes \dagger B \multimap \dagger(A \,\&\, B) \quad \mathsf{inter}_{A,B} := \{(s_A, s_B) \mapsto s \mid s{\restriction_A} = s_A \wedge s{\restriction_B} = s_B\}$$

which produces all the possible interleavings of the pair of input traces. Then, we define the product $V_A \bullet V_B$ of the cliques V_A and V_B as the composition

$$\mathbf{1} \xrightarrow{V_A \bullet V_B} \dagger(A \,\&\, B) \quad = \quad \mathbf{1} \xrightarrow{iso} \mathbf{1} \otimes \mathbf{1} \xrightarrow{V_A \otimes V_B} \dagger A \otimes \dagger B \xrightarrow{\mathsf{inter}} \dagger(A \,\&\, B)$$

which is simply the set of all possible interleavings of traces in V_A with traces in V_B. This endows the category **CertiRegConc** with a monoidal structure encoding independent parallel composition:

$$(A, R, V_A) \otimes (B, S, V_B) := (A \,\&\, B, R \otimes S, V_A \bullet V_B)$$

7 CONCURRENT OBJECT SPACES: TWO CASE STUDIES

In §6 we defined a notion of concurrent object spaces supporting independent parallel composition. In this section we present two case studies showcasing that concurrent object spaces can express more complex forms of concurrency. In §7.1 we discuss a simple model of protected shared object concurrency which uses a lock primitive to synchronize several computational agents. We show atomic concurrent overlays can be certified by proving a local sequential refinement condition. Then, §7.2 discusses how the lock interface, which §7.1 uses as underlay, can be encoded in concurrent object spaces in a simple fashion by means of a carefully constructed equational theory.

7.1 Protected Shared Object Concurrency

A common form of concurrency in systems is protected access to a shared object. By this we mean that different agents (say threads, or processors) have their accesses to a shared object protected by a synchronization primitive such as a lock. This allows an object that in principle is shared concurrently to implement atomic interfaces.

To ground this discussion we will assume a set of computational agents Υ. Given an effect signature E we can construct an effect signature $E[\Upsilon]$ which labels the operations described by E with the name of who is executing the operation, formally defined as the labelled disjoint union

$$E[\Upsilon] := \sum_{\tau \in \Upsilon} E \text{ corresponding to the coherence space } \mathop{\&}_{\tau \in \Upsilon} [\![E]\!]$$

We define a signature Lock for a lock interface as

$$\mathsf{Lock} := \{\mathsf{acq} : \mathbf{1}, \mathsf{rel} : \mathbf{1}\}$$

which is shared by a set of agents Υ in the signature $\mathsf{Lock}[\Upsilon]$. We give a simple sequential specification to the Lock interface with the (prefix-closed) clique V_{Lock} defined as

$$\begin{aligned} V_{\mathsf{Lock}} := \{s \in |\dagger\mathsf{Lock}| \mid \forall p, t \in |\dagger\mathsf{Lock}|. \forall \tau, \tau' \in \Upsilon. \forall m \in \mathsf{Lock}.(&s = \boldsymbol{\tau}{:}m \cdot t \Rightarrow m = \mathsf{acq.ok}) \\ \wedge\ (&s = p \cdot \boldsymbol{\tau}{:}\mathsf{acq.ok} \cdot \boldsymbol{\tau'}{:}m \cdot t \Rightarrow m = \mathsf{rel.ok} \wedge \tau' = \tau) \\ \wedge\ (&s = p \cdot \boldsymbol{\tau}{:}\mathsf{rel.ok} \cdot \boldsymbol{\tau'}{:}m \cdot t \Rightarrow m = \mathsf{acq.ok})\}, \end{aligned}$$

where each of the conditions say, respectively, that: (1) Every trace starts with an acq move; (2) If acq is called by agent τ then the next event is a call to rel by agent τ; (3) Any rel call may only be followed by an acq call.

Now, given an object encoded by the signature E and a clique $V_E : \mathbf{1} \multimap \dagger E$ we construct the interface for its sharing among the agents in Υ as the signature $E[\Upsilon]$ and the clique

$$V_E[\Upsilon] := \{\langle \boldsymbol{\tau_1}{:}e_1.v_1, \ldots, \boldsymbol{\tau_n}{:}e_n.v_n\rangle \in \dagger E[\Upsilon] \mid \langle e_1.v_1, \ldots, e_n.v_n\rangle \in V_E\}$$

that is, all sequences such that if we "forget" which agent is calling each operation the trace obeys the specification V_E.

Given an object specification (E, V_E) we can always construct the object specification

$$(\dagger_{=\otimes=}(\mathsf{Lock} \,\&\, E)[\Upsilon], V_{\mathsf{Lock}} \bullet V_E[\Upsilon])$$

where we make use of the isomorphism

$$\mathsf{Lock}[\Upsilon] \,\&\, E[\Upsilon] \cong (\mathsf{Lock} \,\&\, E)[\Upsilon].$$

The equivalence classes of $= \otimes =$, the tensor of the equality relation over $\dagger\mathsf{Lock}[\Upsilon]$ with the equality relation over $\dagger E[\Upsilon]$ as defined in §6.5, allow for Lock and E events to be commuted liberally.

Then, given an implementation $M : \dagger E \multimap F$ we denote by $M[\tau] : \dagger E[\tau] \multimap F[\tau]$ the implementation obtained by labelling every event that appears in M with agent τ. We construct the protected implementation $\langle M\rangle[\tau] : \dagger_{=\otimes=}(\mathsf{Lock} \,\&\, E)[\Upsilon] \multimap F[\Upsilon]$:

$$\langle M\rangle[\tau] := \{[\langle \boldsymbol{\tau}{:}\mathsf{acq.ok}\rangle \cdot s \cdot \langle \boldsymbol{\tau}{:}\mathsf{rel.ok}\rangle] \mapsto \boldsymbol{\tau}{:}f.v \mid s \mapsto \boldsymbol{\tau}{:}f.v \in M[\tau]\}$$

which surrounds the body of the implementation by acquiring a lock and then releasing it when done. The implementation on behalf of all the agents is given by

$$\langle M\rangle[\Upsilon] := \biguplus_{\tau\in\Upsilon} \langle M\rangle[\tau].$$

It is easy to check that if $\widehat{M} \circ V_E \supseteq V_F$ that is, the refinement condition holds locally, then

$$\widehat{\langle M\rangle[\Upsilon]} : ((\mathsf{Lock} \,\&\, E)[\Upsilon], = \otimes =, V_{\mathsf{Lock}} \bullet V_E[\Upsilon]) \to (F, =, V_F[\Upsilon]),$$

that is, $\widehat{\langle M\rangle[\Upsilon]}$ is a certified implementation of **CertiRegConc**.

Example 7.1. We take our usual example of implementing a counter using a variable. Now, we consider implementing an atomic concurrent counter interface making use of a lock and a concurrent shared variable. From our discussion, the underlay can be modeled by the signature and specification

$$(\mathsf{Lock} \,\&\, \mathsf{Var})[\Upsilon] \qquad\qquad V_{\mathsf{Lock}} \bullet V_{\mathsf{Var}}[\Upsilon]$$

where V_{Var} is the usual variable specification, as seen in Example 2.5. The usual implementation $M : \dagger\mathsf{Var} \multimap \mathsf{Counter}$ is lifted to

$$\langle M\rangle[\Upsilon] : \dagger_{=\otimes=}(\mathsf{Lock} \,\&\, \mathsf{Var})[\Upsilon] \multimap \dagger\mathsf{Counter}[\Upsilon].$$

Note that the underlay's equational theory relates the traces s_1 and s_2 below:

$$s_1 = \langle \mathbf{1}{:}\mathsf{acq.ok}, \mathbf{1}{:}\mathsf{get}.0, \mathbf{1}{:}\mathsf{set}(1).\mathsf{ok}, \mathbf{1}{:}\mathsf{rel.ok}, \mathbf{2}{:}\mathsf{acq.ok}, \mathbf{2}{:}\mathsf{get}.1, \mathbf{2}{:}\mathsf{set}(2).\mathsf{ok}, \mathbf{2}{:}\mathsf{rel.ok}\rangle$$

$$s_2 = \langle \mathbf{1}{:}\mathsf{acq.ok}, \mathbf{1}{:}\mathsf{rel.ok}, \mathbf{2}{:}\mathsf{acq.ok}, \mathbf{2}{:}\mathsf{rel.ok}, \mathbf{1}{:}\mathsf{get}.0, \mathbf{1}{:}\mathsf{set}(1).\mathsf{ok}, \mathbf{2}{:}\mathsf{get}.1, \mathbf{2}{:}\mathsf{set}(2).\mathsf{ok}\rangle$$

and therefore both map under $\widehat{\langle M\rangle[\Upsilon]}$ to $t = \langle \mathbf{1}{:}\mathsf{inc.ok}, \mathbf{2}{:}\mathsf{inc.ok}\rangle$ despite the fact that s_2 does not match the shape of the implementation M. The presence of the synchronization primitives with semantics given by the clique $V_{\mathsf{Lock}} \bullet V_{\mathsf{Var}}[\Upsilon]$ together with the equational theory $= \otimes =$ means that s_2 carries the information that the two increments were indeed performed atomically. Here, the structural map $\kappa_{=\otimes=}$ plays a very important role as it makes use of the equational theory to decompose s_2 in the following way:

$$[s_2] \overset{\kappa_{=\otimes=}}{\longmapsto} \langle [\mathbf{1}{:}\mathsf{acq.ok}, \mathbf{1}{:}\mathsf{get}.0, \mathbf{1}{:}\mathsf{set}(1).\mathsf{ok}, \mathbf{1}{:}\mathsf{rel.ok}], [\mathbf{2}{:}\mathsf{acq.ok}, \mathbf{2}{:}\mathsf{get}.1, \mathbf{2}{:}\mathsf{set}(2).\mathsf{ok}, \mathbf{2}{:}\mathsf{rel.ok}]\rangle$$

which $\dagger\langle M\rangle[\Upsilon]$ is then able to map to t. There is no other choice: the synchronization primitives do not commute with each other and neither do the variable primitives. We note that a more specialized map that does not protect calls to get could have been used instead.

This example showcases that the coalgebra structural map κ plays a much more subtle role than δ did in the completely sequential models. It does not only split a trace, but it also may transform the trace according to the equational theory it has access to. We will see that it is a key feature of our handling of even more complex concurrent objects.

7.2 Ticket Lock

We have just discussed in §7.1 a simple framework for handling protected shared object concurrency. In that setting we assume a sequentially specified lock interface $(\mathsf{Lock}[\Upsilon], V_{\mathsf{Lock}})$ is available as underlay. For instance, a particular system architecture may implement an array of ticket locks to be used throughout the system. Often such a lock interface is implemented using some other synchronization primitives. In the context of a certified system, the ticket lock implementation itself might be certified to be correctly implemented using its underlay.

We will take as example a ticket lock algorithm. The ticket lock is implemented using a fetch-and-increment primitive and a shared counter. We model this underlay with the signature FAI & Counter where Counter is the usual Counter interface and FAI is given by the signature and specification

$$\mathsf{FAI} := \{\mathsf{fai} : \mathbb{N}\} \qquad V_{\mathsf{FAI}} := \{s \in |\dagger\mathsf{FAI}| \mid s = p \cdot \mathsf{fai}.n \cdot t \Rightarrow n = \#\mathsf{fai}(p)\}$$

where $\#\mathsf{fai}(p)$ is the number of fai operations in the sequence p. We construct the interfaces $(\mathsf{FAI}[\Upsilon], V_{\mathsf{FAI}}[\Upsilon])$ and $(\mathsf{Counter}[\Upsilon], V_{\mathsf{Counter}}[\Upsilon])$ as in §7.1. We will define the underlay specification $V_{\mathsf{FAI\&Counter}}$ to be all possible interleavings of traces in $V_{\mathsf{FAI}}[\Upsilon]$ and $V_{\mathsf{Counter}}[\Upsilon]$. That is,

$$V_{\mathsf{FAI\&Counter}} := \{s \in |\dagger(\mathsf{FAI}\ \&\ \mathsf{Counter})[\Upsilon]| \mid s\restriction_{\mathsf{FAI}[\Upsilon]} \in V_{\mathsf{FAI}}[\Upsilon] \wedge s\restriction_{\mathsf{Counter}[\Upsilon]} \in V_{\mathsf{Counter}}[\Upsilon]\}$$

In order to justify the equational theory we will be using, it is vital to understand exactly how the ticket lock is implemented. We wish to encode the code:

```
acq() {                              rel() {
  my_t := fai();                       inc();
  while (get() != my_t) {};            return ok
  return ok                          }
}
```

The intuition for the code is that each contestant for the lock acquires a ticket number from the FAI object. Then, each contestant keeps checking for the currently serving ticket number obtained from the shared counter. As soon as a contestant checks for the currently serving ticket number

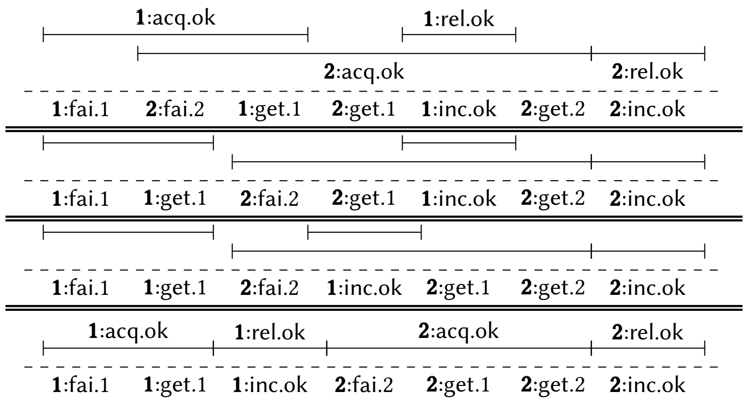

1:acq.ok
1:rel.ok
2:acq.ok
2:rel.ok
1:fai.1 **2**:fai.2 **1**:get.1 **2**:get.1 **1**:inc.ok **2**:get.2 **2**:inc.ok
1:fai.1 **1**:get.1 **2**:fai.2 **2**:get.1 **1**:inc.ok **2**:get.2 **2**:inc.ok
1:fai.1 **1**:get.1 **2**:fai.2 **1**:inc.ok **2**:get.1 **2**:get.2 **2**:inc.ok
1:acq.ok
1:rel.ok
2:acq.ok
2:rel.ok
1:fai.1 **1**:get.1 **1**:inc.ok **2**:fai.2 **2**:get.1 **2**:get.2 **2**:inc.ok

Fig. 3. We depict several L related traces. Vertically adjacent traces require a single swap to be related. Above the traces are the intervals in which call/return events of the Lock overlay are active. The swaps, from top to bottom, preserve the happens before ordering of the lock overlay. The bottom-most trace introduces a new happens before relation, which is allowed by linearizability. Although the bottom-most trace does not satisfy the sequential specification of Counter, it preserves local program order and behavior. Therefore, from the perspective of each agent there is no difference between the traces displayed.

and verifies that it is the same as the ticket number it holds it acquires the lock. In order to release the lock the current lock holder simply increments the currently serving ticket number.

Note that while the underlay interface is atomic, at least with respect to each of the independent objects available, the implementation of the overlay events themselves may interleave non-atomically creating friction with the completely atomic overlay lock specification. A common correctness criterion for atomicity of concurrent objects is linearizability. In defining our equational theory for the ticket lock we take inspiration from the fact that the ticket lock implementation yields a linearizable lock interface. Our equational theory is carefully constructed so to preserve "happens before" order as defined in Herlihy and Wing [1990].

We are now ready to define the relation

$$L \subseteq |\dagger(\mathsf{FAI}\ \&\ \mathsf{Counter})[\Upsilon]| \times |\dagger(\mathsf{FAI}\ \&\ \mathsf{Counter})[\Upsilon]|$$

encoding the equational theory for the Lock implementation. We define L as the smallest congruence satisfying the rules:

(1) $\tau \neq \tau' \wedge (e$ and e' are events of different shared objects$) \Rightarrow \langle \boldsymbol{\tau}{:}e.v, \boldsymbol{\tau'}{:}e'.v' \rangle\ L\ \langle \boldsymbol{\tau'}{:}e'.v', \boldsymbol{\tau}{:}e.v \rangle$
(2) $\tau \neq \tau' \Rightarrow \langle \boldsymbol{\tau}{:}\mathsf{get}.i, \boldsymbol{\tau'}{:}\mathsf{inc.ok} \rangle\ L\ \langle \boldsymbol{\tau'}{:}\mathsf{inc.ok}, \boldsymbol{\tau}{:}\mathsf{get}.i \rangle$
(3) $\tau \neq \tau' \Rightarrow \langle \boldsymbol{\tau}{:}\mathsf{get}.i, \boldsymbol{\tau'}{:}\mathsf{get}.j \rangle\ L\ \langle \boldsymbol{\tau'}{:}\mathsf{get}.j, \boldsymbol{\tau}{:}\mathsf{get}.i \rangle$

Rule (1) says that if two events come from different agents and different shared objects they may be swapped. Rule (2) might seem counter-intuitive, as it allows an inc event to swap with a get event. Despite that, it still preserves the program order of each agent involved, as the swap can only be performed between events of different threads. Therefore, from the local perspective of each agent they still see the same history. Furthermore, the swap does not change the real-time ordering of operations from the perspective of the overlay events, it at most refines it. Rule (3) enforces the passivity of get. In Figure 3 we consider a few traces related by L.

Now that we have carefully described the intuition for the L equational theory, we are ready to discuss the encoding of the implementation of the ticket lock. Locally the implementation is simply given by the map

$$M[\tau] : \dagger_L(\mathsf{FAI}\ \&\ \mathsf{Counter})[\tau] \multimap \mathsf{Lock}[\tau]$$

defined as

$$M[\tau]^{\mathsf{acq}} := \{[\langle \boldsymbol{\tau}{:}\mathsf{fai}.i, \boldsymbol{\tau}{:}\mathsf{get}.i_1, \ldots, \boldsymbol{\tau}{:}\mathsf{get}.i_n, \boldsymbol{\tau}{:}\mathsf{get}.i\rangle] \mapsto \boldsymbol{\tau}{:}\mathsf{acq.ok} \mid \forall j \leq n. i_j \neq i\}$$

$$M[\tau]^{\mathsf{rel}} := \{[\langle \boldsymbol{\tau}{:}\mathsf{inc.ok}\rangle] \mapsto \boldsymbol{\tau}{:}\mathsf{rel.ok}\}$$

note that all the equivalence classes involved in $M[\tau]$ are singleton equivalence classes. It is notorious that the definition of M is essentially just the code for the implementation.

M can be lifted to the map

$$\widehat{M[\Upsilon]} : \dagger_L(\mathsf{FAI}\ \&\ \mathsf{Counter})[\Upsilon] \multimap \dagger\mathsf{Lock}[\Upsilon]$$

and shown to be correct by verifying the refinement condition:

$$\forall t \in V_{\mathsf{Lock}}. \exists s \in V_{\mathsf{FAI\&Counter}}. [s]_L \mapsto t \in \widehat{M[\Upsilon]}$$

For this, the structural map κ_L plays a fundamental role. Consider for instance the following graphical depiction of an input/output pair in $\widehat{M[\Upsilon]}$:

$$\begin{aligned}
& [\langle \mathbf{1}{:}\mathsf{fai}.0, \mathbf{2}{:}\mathsf{fai}.1, \mathbf{1}{:}\mathsf{get}.0, \mathbf{2}{:}\mathsf{get}.0, \mathbf{1}{:}\mathsf{inc.ok}, \mathbf{2}{:}\mathsf{get}.1\rangle] \\
\xrightarrow{\kappa_L}\ & \langle [\langle \mathbf{1}{:}\mathsf{fai}.0, \mathbf{1}{:}\mathsf{get}.0\rangle], [\langle \mathbf{1}{:}\mathsf{inc.ok}\rangle], [\langle \mathbf{2}{:}\mathsf{fai}.1, \mathbf{2}{:}\mathsf{get}.0, \mathbf{2}{:}\mathsf{get}.1\rangle]\rangle \\
\xrightarrow{\widehat{M[\Upsilon]}}\ & \langle \mathbf{1}{:}\mathsf{acq.ok}, \mathbf{1}{:}\mathsf{rel.ok}, \mathbf{2}{:}\mathsf{acq.ok}\rangle
\end{aligned}$$

The $V_{\mathsf{FAI\&Counter}}$ trace is transformed using the equational theory encoded in L. This transformation is performed by the structural map κ_L which then decomposes the trace into the components which $M[\Upsilon]$ is able to map. This is a much more subtle operation than the sequential decomposition performed by δ. This nice coalgebraic structure greatly simplifies reasoning and makes for particularly simple implementation definitions. After the decomposition the map M, which simply encodes the body of the code implementing each method, is applied directly.

8 RELATED WORK

Object-Based Semantics. While we have already discussed the relationship between our work and Reddy's work on object-based semantics [Reddy 1996], we have not mentioned Reddy's work with †-coalgebras in the Appendix of Reddy [1996]. Reddy faces similar problems with the tensor product as we do and presents two solutions. One of them [Reddy 1996] defines a class of †-coalgebras characterized by partial monoids, which he calls finitary object spaces. Our work in §6 may be seen as a subcategory of Reddy's finitary object spaces characterized instead as presentations of partial monoids. This equational formulation is more convenient for our purposes, as it is leads to a smooth treatment of concurrency. While Reddy's finitary object spaces are monoidal closed, concurrent objects are not. Despite that, concurrent object spaces are still rich enough to encode Reddy's model of interference-controlled Algol.

The second approach pioneered by Reddy, called dependence spaces [Reddy 1994], was one of many inspirations for our work. They differ substantially in that our work remains in the category of coherence spaces, while dependence spaces endow coherence spaces with extra structure. We believe there is an instructive embedding of dependence spaces into a generalization of our category of concurrent object spaces using partial equivalence relations instead of equivalence relations and mediated by a reformulation of dependence spaces as Mazurkiewicz traces [Mazurkiewicz 1995] which we leave for future work.

Reddy [1993] has also investigated in detail the categorical structures surrounding object-based semantics, what he calls a LLMS for Linear Logic Model of State. The appendix in our extended technical report [Oliveira Vale et al. 2021] places our model from §3 in a standard game-semantics model by defining a † modality. Although we don't discuss it there, the † is constructed so to endow the category of games defined with a LLMS. A careful discussion from this point-of-view would be lengthy, so we curb our remarks on the matter. Reddy's subsequent work [O'Hearn and Reddy 1999; Reddy 2002, 2013; Reddy and Dunphy 2012] focused on combining the event-based and state-based approaches to define the full semantics of Algol-like languages.

Game Semantics. Game semantics has been around for more than 30 years. It has been extremely successful in describing the fine-grained semantics of a large class of programming languages including PCF [Abramsky et al. 2000; Hyland and Ong 2000], imperative languages [Abramsky and McCusker 1997, 1999; Ghica and Murawski 2008], and object-oriented languages [Murawski and Tzevelekos 2014]. Despite its importance and promising support to compositional reasoning, it has not been used in large formal verification projects based on proof assistants. Instead, the formal verification community has preferred to use simple small-step or mixed-step operational semantics to verify programs because game semantics is often seen as too complex to be smoothly mechanized in any proof assistant. Our work as well as Koenig and Shao [2020] can be seen as significant steps toward applying game semantics to the mechanized verification of large systems. We have had a pleasant experience in mechanizing coherence spaces due to their simplicity. Furthermore, while we give a traditional game semantics presentation to our model in §3, we believe Koenig [2021] provides an equivalent model amenable to convenient mechanization.

The game semantics literature focused on giving the semantics for a specific programming language and then using it to prove the soundness and full abstraction properties. They are complex because they use game semantics to model command- or expression-level interaction in the core programs. These languages and their game semantics are not primarily designed for program verification; and there are no equivalent notions of layer interfaces or certified layers. By focusing on certified layers, we take the best idea from game semantics to support certified composition. To make things simple, our key idea is that these certified layers must fully encapsulate their states, otherwise, their interfaces would be too complex and then make composition difficult.

This is why Reddy's approach to handling global state is particularly attractive. Starting from the seminal work by Abramsky and McCusker on Idealized Algol [Abramsky and McCusker 1997], the game semantics community took inspiration from Reddy's idea to give fully abstract models to imperative languages—for programming in the small. Here, we show that the very idea could have a big impact for certified layer programming in the large. This happens to match the best practice on how abstraction layers are used by the real-world engineers. Also, despite its early influence on game semantics, the † modality has been largely forgotten in the game semantics community in the benefit of linear logic's ! modality. Our work seeks to bring back into focus the relevance of the † modality by showcasing its simplicity and expressiveness.

Calderon and McCusker [2010] presented a full, faithful strong monoidal embedding of a category of games into a category of coherence posets and hinted about a possible deep connection between games semantics and Reddy's object-based semantics. The correspondence which we established in §5 can be viewed as a first attempt toward addressing this problem in the context of certified abstraction layers. Our functor differs from that of Calderon and McCusker [2010] in that it maps less plays. This way, while their functor is lax with respect to † our functor distributes strictly over †. This is fundamental for our development as we need a precise connection with † in coherence spaces, and is what prevents us from using the functor in Calderon and McCusker [2010].

Finally, there are similarities but also intrinsic differences between our model of certified layers based on concurrent objects and the model of Idealized Concurrent Algol (ICA) developed by Ghica and Murawski [2008]. A first key difference is that their model is based on arena games, which means that they have to take care of the intricacies associated to the justification of pointers. Our model based on coherence spaces and regular functions is for that reason simpler to manipulate and to certify in a proof assistant, and this is a main point of our work. In particular, regular functions describe alternating strategies where each token of the coherence space describes a pair consisting of an Opponent move followed by a Player move. In contrast, the model based on arena games enables interactions where Opponent moves and Player moves do not necessarily alternate — which complicates the construction of the model, even on first-order functions. The information provided by the coherence relation as well as the dagger structure are moreover missing from the game model of ICA. For these foundational and practical reasons, our model does not coincide with the game model of ICA restricted to first-order functions.

Certified Abstraction Layers. Koenig and Shao [2020] model certified abstraction layers using categories whose objects are effect signatures and whose morphisms are *strategy specifications*, enriched with a complete refinement lattice structure. Layer interfaces and implementations are both modeled as strategy specifications. Layer correctness can be stated as $L_F \sqsubseteq M \circ L_E$, where $L_E : 1 \longrightarrow E$ is the underlay interface, $M : E \longrightarrow F$ is the layer implementation, and $L_F : 1 \longrightarrow F$ is the overlay interface. However, this elegant picture is complicated by their treatment of state. The set of states used by a layer interface must be encoded as part of its signature, and interactions must follow a "state-passing" discipline. Likewise, the simulation relation used to establish a layer's correctness must be internalized as a morphism, then composed with the implementation to translate between underlay and overlay states. While Koenig and Shao also explore a model featuring stateful and reentrant strategies, which could in principle realize the encapsulation of state, this comes at the cost of the simplicity and elegance of their main development, and they do not extend their treatment of certified layers to this setting. They also do not consider layers with concurrency.

By contrast, our approach avoids complex combinations of features by maintaining a strong distinction between layer interfaces and implementations. Layer implementations are two-sided (they both *use* underlay operations and *provide* overlay operations) but they can remain stateless and deterministic. Layer interfaces are stateful, but because they are one-sided the structure of their plays can remain simple. In turn, the statefulness of layer interfaces and our direct approach to formalizing layer correctness mean we do not need an explicit internalization of simulation relations. This allows us to limit our treatment of nondeterminism to *demonic* nondeterminism, which is sufficient to express implementation freedom.

Concurrency. Gu et al. [2018] developed *Certified Concurrent Abstraction Layers (CCAL)* and applied them to build a certified concurrent OS kernel [Gu et al. 2019, 2016]. They used game-semantic strategies to model the interaction behavior of each thread (or CPU core) against its environment context, and developed a program logic for reasoning about both the safety and progress of concurrent objects. The marriage of Reddy's work [Reddy 1994, 1996] with our new layered game semantics offers a promising direction for developing compositional models for CCAL-style shared-memory concurrency. An appealing challenge for future work in that direction will be to articulate the results of this paper with the asynchronous and interactive accounts based on action and footstep trace semantics [Brookes 2006, 2007] and template game semantics [Melliès and Stefanesco 2018, 2020] of Concurrent Separation Logic [O'Hearn 2004].

Similarly, there is a significant body of work on correctness conditions for concurrent programs [Cerone et al. 2014; Filipović et al. 2009; Herlihy and Wing 1990; Murawski and Tzevelekos 2019]. Most notably Cerone et al. [2014] and Murawski and Tzevelekos [2019] provide generalizations of linearizability to layers encompassing both an underlay and an overlay, including potentially higher-order computation. As far as we are aware this is the only work in this line that discusses a notion of layer with underlay and overlay, and we believe that there is an opportunity to connect the ideas from there with our model. Indeed, while we present a framework for certifying concurrent programs, we provide no correctness criterion for our coherent congruences. It is a key part of the proof of some of the claims in §6 and §7 that any congruence which is a subrelation of the equivalence up-to sequential consistency relation is a coherent congruence. This includes preservation of happens-before order as in Herlihy and Wing [1990]. As the Lock example in §7 shows, we often need even more precise coherent congruences. There is a complex interplay between the coherent congruence and the implementation M. Despite that, as §7.1 shows, once a synchronization primitive such as Lock has been verified, general compositional rules for shared state concurrency become available. This is a promising avenue for future work and will likely involve a connection with concurrent models such as Ghica and Murawski [2008], Cerone et al. [2014], and Murawski and Tzevelekos [2019].

9 CONCLUSION

The idea of certified abstraction layers [Gu et al. 2015] was inspired by the systems community's best practice in using abstraction layers to build large-scale software and hardware systems [Saltzer and Kaashoek 2009]. Certified abstraction layers rely on using a pair of underlay and overlay interfaces to encapsulate the implementation effects and eliminate undesirable dependencies from other components. Gu et al. [2016] has shown the effectiveness of using certified abstraction layers to build large-scale certified concurrent OS kernels. However, the main semantic ingredients that make certified abstraction layers so effective have been unclear for many years.

In this paper, we have demonstrated that there is a close connection between certified abstraction layers and Reddy's object-based semantics of states based on coherent spaces. The major new conceptual contribution of this paper is our model of certified layer implementation (e.g., Definitions 3.16 and 4.2). Modeling a layer interface L as a pair of an effect signature E and an object strategy V_E (or a set of object strategies $\mathcal{V}_E$) is by no means obvious. Here, the signature E imposes a *syntactic* well-formedness of the system-environment interface, and V_E (or $\mathcal{V}_E$) imposes more refined semantic constraints to the layer's behaviors.

This is very different from how existing module languages model a module implementation and its import and export interfaces, and how Koenig and Shao [2020] model certified layer implementation which is still based on a simulation relation between the underlay and overlay states. This reformulation comes with great benefit. Looking to the past, it helps clarify some of what made certified abstraction layers so effective. Looking to the future, it provides an abstract model of certified abstraction layers that can be studied and extended in its own right, as the preliminary advances into the territory of concurrent systems in §6 and §7 showcase.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 2019285, 1763399, 1521523, and 2118851, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. The third author is a co-founder of and has an equity interest in CertiK Global Ltd. CertiK has licensed Yale University's intellectual property, which is related to the NSF grants 1521523 and 1763399. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

2015-2021. DeepSpec: The Science of Deep Specifications. https://deepspec.org/.

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Inf. Comput.* 163, 2 (2000), 409–470. https://doi.org/10.1006/inco.2000.2930

Samson Abramsky and Guy McCusker. 1997. *Linearity, Sharing and State: A Fully Abstract Game Semantics for Idealized Algol with Active Expressions.* Birkhäuser Boston, Boston, MA, 297–329. https://doi.org/10.1007/978-1-4757-3851-3_10

Samson Abramsky and Guy McCusker. 1999. Game Semantics. In *Computational Logic*, Ulrich Berger and Helmut Schwichtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–55.

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

Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position Paper: The Science of Deep Specification. *Phil. Trans. R. Soc. A* 375, 2104 (2017), 20160331. https://doi.org/10.1098/rsta.2016.0331

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

Andreas Blass. 1992. A Game Semantics for Linear Logic. *Ann. Pure Appl. Log.* 56, 1–3 (1992), 183–220. https://doi.org/10.1016/0168-0072(92)90073-9

Stephen Brookes. 2006. A Grainless Semantics for Parallel Programs with Shared Mutable Data. *Electronic Notes in Theoretical Computer Science* 155 (2006), 277 – 307. https://doi.org/10.1016/j.entcs.2005.11.060 Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

Stephen Brookes. 2007. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science* 375, 1 (2007), 227 – 270. https://doi.org/10.1016/j.tcs.2006.12.034 Festschrift for John C. Reynolds's 70th birthday.

Ana C. Calderon and Guy McCusker. 2010. Understanding Game Semantics Through Coherence Spaces. *Electron. Notes Theor. Comput. Sci.* 265 (Sept. 2010), 231–244. https://doi.org/10.1016/j.entcs.2010.08.014

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2014. Parameterised Linearisability. In *Automata, Languages, and Programming*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–109.

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16).* Association for Computing Machinery, New York, NY, USA, 431–447. https://doi.org/10.1145/2908080.2908101

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15).* Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

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

David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-End Verification of Information-Flow Security for C and Assembly Programs. *SIGPLAN Not.* 51, 6 (June 2016), 648–664. https://doi.org/10.1145/2980983.2908100

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for Concurrent Objects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 252–266.

Dan R. Ghica and Andrzej S. Murawski. 2008. Angelic Semantics of Fine-Grained Concurrency. *Annals of Pure and Applied Logic* 151, 2 (2008), 89 – 114. https://doi.org/10.1016/j.apal.2007.10.005

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

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

Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building Certified Concurrent OS Kernels. *Commun. ACM* 62, 10 (Sept. 2019), 89–99. https://doi.org/10.1145/3356903

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16).* USENIX Association, USA, 653–669.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

J. M. E. Hyland and C.-H. L. Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. https://doi.org/10.1006/inco.2000.2917

A. Kock. 1972. Strong functors and monoidal monads. *Archiv der Mathematik* 23 (1972), 113–120.

Jérémie Koenig. 2021. Grounding Game Semantics in Categorical Algebra. In *Proceedings of the Fourth International Conference on Applied Category Theory (ACT 2021)*, Kohei Kishida (Ed.). To appear.

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. https://doi.org/10.1145/3373718.3394799

Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1095–1109. https://doi.org/10.1145/3453483.3454097

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

Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proc. ACM Program. Lang.* 4, POPL, Article 20 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371088

Antoni W. Mazurkiewicz. 1995. Introduction to Trace Theory. In *The Book of Traces*, Volker Diekert and Grzegorz Rozenberg (Eds.). World Scientific, 3–41. https://doi.org/10.1142/9789814261456_0001

Paul-André Melliès and Léo Stefanesco. 2018. An Asynchronous Soundness Theorem for Concurrent Separation Logic. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 699–708. https://doi.org/10.1145/3209108.3209116

Paul-André Melliès and Léo Stefanesco. 2020. Concurrent Separation Logic Meets Template Games. 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, 742–755. https://doi.org/10.1145/3373718.3394762

Paul-André Melliès and Noam Zeilberger. 2015. Functors Are Type Refinement Systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/2676726.2676970

Paul-André Melliès. 2009. Categorical Semantics of Linear Logic. In *Interactive Models of Computation and Program Behaviour, Panoramas et Synthèses 27*. Société Mathématique de France, Paris, France, 1–196.

Andrzej S. Murawski and Nikos Tzevelekos. 2014. Game Semantics for Interface Middleweight Java. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 517–528. https://doi.org/10.1145/2535838.2535880

Andrzej S. Murawski and Nikos Tzevelekos. 2019. Higher-order Linearisability. *Journal of Logical and Algebraic Methods in Programming* 104 (2019), 86–116. https://doi.org/10.1016/j.jlamp.2019.01.002

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67.

Peter W. O'Hearn and Uday S. Reddy. 1999. Objects, interference, and the Yoneda embedding. *Theoretical Computer Science* 228, 1 (1999), 253–282. https://doi.org/10.1016/S0304-3975(98)00360-0

Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. 2021. *Layered and Object-Based Game Semantics*. Technical Report YALEU/DCS/TR-1559. Yale Univ. https://flint.cs.yale.edu/publications/layered.html

Gordon Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2001)*. Springer, Berlin, Heidelberg, 1–24. https://doi.org/10.1007/3-540-45315-6_1

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*. Springer, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

U.S. Reddy. 1994. Passivity and independence. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. 342–352. https://doi.org/10.1109/LICS.1994.316055

Uday S. Reddy. 1993. *A Linear Logic Model of State*. Technical Report. Dept. of Computer Science, UIUC, Urbana, IL.

Uday S. Reddy. 1996. Global State Considered Unnecessary: An Introduction to Object-Based Semantics. *LISP Symb. Comput.* 9, 1 (1996), 7–76.

Uday S. Reddy. 2002. Objects and Classes in Algol-like Languages. *Inf. Comput.* 172, 1 (Feb. 2002), 63–97. https://doi.org/10.1006/inco.2001.2927

Uday S. Reddy. 2013. Automata-Theoretic Semantics of Idealized Algol with Passive Expressions. *Electronic Notes in Theoretical Computer Science* 298 (2013), 325–348. https://doi.org/10.1016/j.entcs.2013.09.020 Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIX.

Uday S. Reddy and Brian P. Dunphy. 2012. An Automata-Theoretic Model of Idealized Algol. In *Automata, Languages, and Programming*, Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–350.

Christian Retoré. 1997. Pomset logic: A non-commutative extension of classical linear logic. In *Typed Lambda Calculi and Applications*, Philippe de Groote and J. Roger Hindley (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 300–318.

Jerome H. Saltzer and M. Frans Kaashoek. 2009. *Principles of Computer System Design.* Morgan Kaufmann.

Zhong Shao. 2010. Certified Software. *Commun. ACM* 53, 12 (December 2010), 56–66.

Vilhelm Sjöberg, Yuyang Sang, Shu-chun Weng, and Zhong Shao. 2019. DeepSEA: A Language for Certified System Software. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 136 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360562