arXiv:2010.04749v1 [cs.LO] 9 Oct 2020

Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification

CHRISTOPH SPRENGER, TOBIAS KLENZE, MARCO EILERS, FELIX A. WOLF, PETER MÜLLER, MARTIN CLOCHARD, and DAVID BASIN, ETH Zurich, Switzerland

Lighthouse projects such as CompCert, seL4, IronFleet, and DeepSpec have demonstrated that full verification of entire systems is feasible by establishing a refinement relation between an abstract system specification and an executable implementation. Existing approaches however impose severe restrictions on either the abstract system specifications due to their limited expressiveness or versatility, or on the executable code due to their reliance on suboptimal code extraction or inexpressive program logics.

We propose a novel methodology that combines the compositional refinement of abstract, event-based models of distributed systems with the verification of full-fledged program code using expressive separation logics, which support features of realistic programming languages like mutable heap data structures and concurrency. The main technical contribution of our work is a formal framework that soundly relates event-based system models to program specifications in separation logics, such that successful verification establishes a refinement relation between the model and the code. We formalized our framework, *Igloo*, in Isabelle/HOL.

Our framework enables the sound combination of tools for protocol development with existing program verifiers. We report on three case studies, a leader election protocol, a replication protocol, and a security protocol, for which we refine formal requirements into program specifications (in Isabelle/HOL) that we implement in Java and Python and prove correct using the VeriFast and Nagini tools.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Higher order logic**; **Separation logic**; • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Security and privacy** → *Logic and verification*; • **Computing methodologies** → *Distributed algorithms*.

1 INTRODUCTION

The full verification of entire software systems, formally relating abstract specifications to executable code, is one of the grand challenges of computer science [Hoare 2003]. Seminal projects such as seL4 [Klein et al. 2009], CompCert [Leroy 2006], IronFleet [Hawblitzel et al. 2015], and DeepSpec [Pierce 2016] have achieved this goal by formally establishing a refinement relation between a system specification and an executable implementation.

Despite this progress, substantial challenges still lay ahead. We posit that techniques for the verification of entire systems should satisfy four major requirements:

(1) *End-to-end guarantees*: Verification techniques need to provide system-wide correctness guarantees whose proofs relate global properties ultimately to verified implementations of the system components.
(2) *Versatility*: Verification techniques should be applicable to a wide range of systems. In the important domain of distributed systems, versatility requires (i) the ability to model different kinds of environments in which the system operates, capturing, for instance, different network properties, fault models, or attacker models, (ii) support for different flavors of systems,

Authors' address: Department of Computer Science, ETH Zurich, Switzerland. Email: firstname.lastname@inf.ethz.ch.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART152
https://doi.org/10.1145/3428220

comprising different types of components (such as clients and servers) and allowing an unbounded number of instances per component type, and (iii) support for heterogeneous implementations, for instance, to support the common case that clients are sequential, servers are concurrent, and each of them is implemented in a different language.

(3) *Expressiveness*: Verification techniques should support expressive languages and logics. In particular, high-level system models and proofs often benefit from the expressiveness of rich formalisms such as higher-order logic, whereas code-level verification needs to target efficiently-executable and maintainable implementations, often in multiple languages.

(4) *Tool interoperability*: While it is possible to support the previous three requirements within one generic verification tool, it is advantageous to employ specialized tools, for instance, to obtain a high degree of automation and to leverage existing tools, infrastructure, and expert knowledge. This gives rise to the additional requirement of sound interoperability of different verification tools, which is a long-standing challenge in verification. Moreover, integrating tools should ideally not require any modifications to the tools, even though they may support different logics and programming languages.

Although existing work has demonstrated that full verification is now feasible, the employed techniques do not meet all of these requirements.

Some existing approaches [Koh et al. 2019] use specifications of individual system components (such as a server), but do not explain how to formally connect them to a global model of the entire system. A global model is necessary to prove system-wide properties, especially in decentralized systems. Others [Oortwijn and Huisman 2019] do not consider the preservation of global model properties down to the implementation. Hence, these approaches do not meet our first requirement.

Most existing approaches do not match our versatility requirements. Some target particular types of systems [Klein et al. 2009; Lesani et al. 2016; Rahli et al. 2018] or make fixed environment assumptions [Koh et al. 2019; Sergey et al. 2018]. Moreover, in several works, different component types with unbounded numbers of instances are either not supported or it is unclear whether they are generically supported [Hawblitzel et al. 2015; Koh et al. 2019]. Finally, many approaches [Hawblitzel et al. 2015; Lesani et al. 2016; Rahli et al. 2018; Sergey et al. 2018; Wilcox et al. 2015] prescribe a fixed programming language and, thus, do not support heterogeneous implementations.

Most previous work does not satisfy our expressiveness requirement. Some of them [Hawblitzel et al. 2015, 2014] limit the formalism used for model development to first-order logic, to leverage SMT solvers, which complicates the formalization of common properties such as graph properties. Others restrict the executable implementation [Leroy 2006; Lesani et al. 2016; Liu et al. 2020; Rahli et al. 2018; Sergey et al. 2018; Wilcox et al. 2015; Woos et al. 2016] and extract executable code directly from formal models. This guarantees the implementation's correctness, but has several drawbacks. In particular, the extracted code is purely functional or rewriting-based, with sub-optimal performance, and any manual code optimizations invalidate the correctness argument and may compromise the intended behavior. Moreover, code extraction complicates the interaction with existing system components and libraries. Other approaches reason about manually-written implementations, but do not employ a modern verification logic [Klein et al. 2009], restricting the implementation, for instance, to sequential code, and precluding the use of existing state-of-the-art program verification tools, potentially resulting in low proof automation and non-modular proofs.

Finally, most existing approaches require the use of a single tool, typically an interactive theorem prover. This may prevent experts in both protocol and program verification from using the highly automated tools they are familiar with and from building on their existing infrastructure. An exception is Oortwijn and Huisman [2019], who combine the Viper verifier [Müller et al. 2016] with the mCRL2 model checker [Cranen et al. 2013] to reason about message passing programs.

This Work. We propose a novel approach that combines the top-down compositional refinement of abstract, event-based system models [Abadi and Lamport 1991; Abrial 2010; Lynch and Vaandrager 1995] with the bottom-up verification of full-fledged program code using separation logic [Reynolds 2002]. Our approach satisfies all four of our requirements. It offers the full expressive power of higher-order logic and the foundational guarantees of interactive theorem provers for developing formal models, as well as the expressiveness and tool support provided by modern program logics.

The core of our approach is a formal framework that soundly relates event-based system models to program specifications in separation logic, such that successful verification establishes a refinement relation between the model and the code. The program specifications link models and code and at the same time they decouple models and code, allowing us to support multiple programming languages and verification tools. This is, for instance, useful to develop multiple library implementations of a protocol. Moreover, this decoupling enables a separation of concerns where we can use specialized tools for the separate tasks of model refinement and code verification, tailored to the problem and the programming language at hand.

We focus on the development of *distributed* systems, consisting of an arbitrary number of components (of possibly heterogenous types such as clients and servers) with local states that interact by exchanging messages via an arbitrary, potentially faulty or adversarial environment. Such systems give rise to complex concurrent behaviors. In this setting, the program specification of a component's implementation prescribes the component's state changes as well as its I/O behavior and is called an *I/O specification*. For this purpose, we employ an existing encoding of I/O specifications into a separation logic to support assertions that can specify both of these aspects [Penninckx et al. 2015]. This encoding can be used with any logic that offers standard separation logic features, and can thus be used to verify components with mutable heap data structures, concurrency, and other features of realistic programming languages that enable efficient implementations.

Approach. Our methodology consists of six main steps, illustrated in Figure 1. All steps come with formal guarantees to soundly link the abstract models with the code. The first five steps are formalized in Isabelle/HOL [Nipkow et al. 2002]. **Step 1** requires formalizing an initial abstract model of the entire system and proving desired trace properties. This model and subsequent models are expressed as event systems (i.e., labeled transition systems) in a generic refinement framework that we implemented in Isabelle/HOL. **Step 2** develops a protocol model, which contains the components of the distributed system to be developed as well as assumptions about the communication network. This *environment* may, for instance, include a fault model or an attacker model, which can be used to prove properties about fault-tolerant or secure systems. So far, this is standard development by refinement, but Steps 3-5 are specific to our approach. **Step 3** prepares the model for a subsequent decomposition and refines the interfaces of the components and the environment to match the interfaces of the I/O

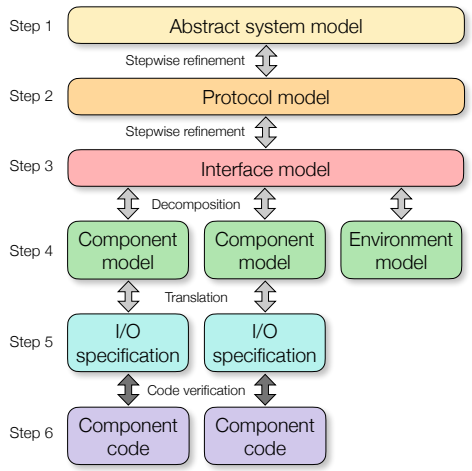

Fig. 1. The main steps of our approach. Boxes depict formal models, specifications, and programs. Light and dark gray arrows depict proofs in Isabelle/HOL and in program verification tools, respectively.

libraries to be used in the implementations. **Step 4** decomposes the, so far monolithic, model into models of the individual system components (e.g., clients and servers) and the environment. **Step 5** translates each component's event system into an I/O specification, which formalizes its valid I/O behaviors. We express this specification as an encoding into standard separation logic assertions that can describe sequences of calls to I/O libraries, e.g., for sending and receiving messages [Penninckx et al. 2015]. Each such call corresponds to one event of the component's event system. Finally, **Step 6** is standard code verification of the different system components, albeit with specifications describing I/O behavior. This verification step can be performed using an embedding of a separation logic into an interactive theorem prover (to obtain foundational guarantees) or by using separate dedicated program verifiers (to increase automation). For the latter, any existing verifier supporting standard separation logic features can be used without requiring changes to the tool, provided it satisfies our *verifier assumption*. This assumption states that proving a Hoare triple involving the I/O specification in the tool implies that the program's I/O behavior refines the one defined by the I/O specification. Crucially, our approach supports modular reasoning in that the verification of a component's code does not involve reasoning about the system's global properties, other components, or the environment. Moreover, we can employ different code verifiers to support heterogeneous implementations, where different components are written in different languages, and some are sequential, while others use local concurrency for improved performance.

Our approach ensures that the resulting distributed system's implementation does not abort due to runtime errors and satisfies, by virtue of compositional refinement, the requirements specified and proved for the formal models. These guarantees assume that the real environment, including the I/O libraries and the lower software and hardware layers, conforms to our environment model, the components are correctly instantiated, and the verification tools used are sound. As our approach "glues" together models and code through their I/O behavior, we have dubbed it "Igloo".

Contributions. Our work makes the following contributions:

(1) *Methodology:* We present a novel methodology for the sound end-to-end verification of distributed systems that combines the top-down refinement of expressive, global system specifications with bottom-up program verification based on expressive separation logics. This combination supports the verification of system-wide properties and handles heap data structures, concurrency, and other language features that are required for efficient code. Our methodology enables the sound interoperability of interactive theorem provers with existing code verification tools for different programming languages, as well as the verified interoperability of the resulting component implementations.
(2) *Theory:* We establish a novel, formal link between event system models and I/O specifications for programs expressed in separation logics by relating both of them to a process calculus. This link between these disparate formalizations is central to our methodology's soundness. It is also interesting in its own right since it shows how to formally integrate the trace semantics of event systems and processes with the permissions manipulated by separation logics.
(3) *Case studies:* We demonstrate the feasibility of our approach by developing formal models for a leader election protocol, a replication protocol, and a security protocol, deriving I/O specifications for their components, and verifying independent implementations in Java and Python, using the VeriFast [Jacobs et al. 2011] and Nagini [Eilers and Müller 2018] verifiers. Some of these components' performance is optimized using locally concurrent execution.
(4) *Formalization:* All our definitions and results are formalized and proven in Isabelle/HOL. This includes the refinement framework and its soundness, the formalization of I/O specifications, the soundness proof that formally links event systems, processes, and I/O specifications, and Steps 1–5 of our case studies. This foundational approach yields strong soundness guarantees.

Table 1. Summary of notation.

$\mathbb{1}$, $\mathbb{B}$, $\mathbb{N}$	$\{\bullet\}$, $\{\text{true}, \text{false}\}$, naturals
$A \times B$	cartesian product
$(\!\| x \in A, y \in B \|\!)$	set of records
$A \uplus B$	disjoint union (sum)
$A_\bot$	$= A \uplus \{\bot\}$
$A \to B$, $A \rightharpoonup B$	total and partial functions
$\mathbb{P}(A)$	powerset
A^*	finite sequences
$A^\#$	multisets, $= A \to \mathbb{N} \cup \{\infty\}$

$(\!\| x = a, y = b \|\!)$	concrete record
$x(r)$, $r(\!\| x := v \|\!)$	record field x, update
$f(x \mapsto v)$, f^{-1}	function update, inverse
ϵ, abc or $\langle a, b, c\rangle$	empty, concrete sequence
$x \cdot y$	concatenation
$len(x)$, $x(i)$	length, i-th value
$\{a, a, b, c\}^\#$	concrete multiset
$M +^\# M'$	multiset sum
$\emptyset^\#$, $a \in^\# M$	$\{\}^\#$, $M(a) > 0$

2 PRELIMINARIES

Although we formalize our development in Isabelle/HOL, we use standard mathematical notation where possible to enhance readability. Table 1 summarizes our notation.

2.1 Event Systems, Refinement, and Parallel Composition

2.1.1 Event Systems. An *event system* is a labeled transition system $\mathcal{E} = (S, E, \to)$, where S is a set of states, E is a set of events, and $\to \subseteq S \times E \times S$ is the transition relation. We also write $s \xrightarrow{e} s'$ for $(s, e, s') \in \to$. We extend the transition relations to finite sequences of events τ by inductively defining, for all $s, s', s'' \in S$, $s \xrightarrow{\epsilon} s$ and $s \xrightarrow{\tau \cdot \langle e \rangle} s''$, whenever $s \xrightarrow{\tau} s'$ and $s' \xrightarrow{e} s''$. Given a set of initial states $I \subseteq S$, a *trace* of an event system $\mathcal{E}$ starting in I is a finite sequence τ such that $s \xrightarrow{\tau} s'$ for some initial state $s \in I$ and reachable state s'. We denote by $traces(\mathcal{E}, I)$ the set of all traces of $\mathcal{E}$ starting in I. For singleton sets $I = \{s\}$, we also write $traces(\mathcal{E}, s)$, omitting brackets. We call a set of traces P over E a *trace property* and write $\mathcal{E}, I \models P$ if $traces(\mathcal{E}, I) \subseteq P$.

For concrete specifications, we often use *guarded event systems* of the form $\mathcal{G} = (S, E, G, U)$ where G and U denote the E-indexed families of *guards* $G_e : S \to \mathbb{B}$ and *update* functions $U_e : S \to S$. The associated transition relation is $\to = \{(s, e, s') \mid G_e(s) \wedge s' = U_e(s)\}$. If $S = (\!| \bar{x} \in \bar{T} |\!)$ is a record, we use the notation $e\colon\ G_e(\bar{x}) \triangleright \bar{x} := U_e(\bar{x})$ to specify events. For example, the event $dec(a)\colon\ z > a \triangleright z := z - a$ decreases z by the parameter a provided that the guard $z > a$ holds.

2.1.2 Refinement. Given two event systems, $\mathcal{E}_i = (S_i, E_i, \to_i)$ and sets of initial states $I_i \subseteq S_i$ for $i \in \{1, 2\}$, we say that $(\mathcal{E}_2, I_2)$ *refines* $(\mathcal{E}_1, I_1)$ *modulo a mediator function* $\pi : E_2 \to E_1$, written $(\mathcal{E}_2, I_2) \sqsubseteq_\pi (\mathcal{E}_1, I_1)$, if there is a simulation relation $R \subseteq S_1 \times S_2$ such that

(1) for each $s_2 \in I_2$ there exists some $s_1 \in I_1$ such that $(s_1, s_2) \in R$, and
(2) for all $s_1 \in S_1$, $s_2, s_2' \in S_2$ and $e_2 \in E_2$ such that $(s_1, s_2) \in R$ and $s_2 \xrightarrow{e_2}_2 s_2'$ there exists some $s_1' \in S_1$ such that $s_1 \xrightarrow{\pi(e_2)}_1 s_1'$ and $(s_1', s_2') \in R$.

This is standard forward simulation [Lynch and Vaandrager 1995], augmented with the mediator function π, which allows us to vary the events in a refinement. We assume that all models $\mathcal{E}$ in our developments include a special stuttering event skip $\in E$, defined by $s \xrightarrow{\text{skip}} s$; consequently, the trace properties $traces(\mathcal{E}, I)$ are closed under the addition and removal of skip to traces. Events that are added in a refinement step often refine skip.

We prove a standard soundness theorem stating that refinement implies trace inclusion. This trace inclusion in turn preserves trace properties (modulo the mediator π). Here, π is applied to each element of each trace and $\pi^{-1}(P_1)$ consists of all traces that map element-wise to a trace in P_1.

Theorem 2.1. $(\mathcal{E}_2, I_2) \sqsubseteq_\pi (\mathcal{E}_1, I_1)$ *implies* $\pi(traces(\mathcal{E}_2, I_2)) \subseteq traces(\mathcal{E}_1, I_1)$.

Lemma 2.2. *Suppose* $\mathcal{E}_1, I_1 \models P_1$ *and* $\pi(traces(\mathcal{E}_2, I_2)) \subseteq traces(\mathcal{E}_1, I_1)$. *Then* $\mathcal{E}_2, I_2 \models \pi^{-1}(P_1)$.

For complex or multi-level refinements, it may be advisable to reformulate the intended property P_1 at the concrete level as P_2 and prove that $\pi^{-1}(P_1) \subseteq P_2$, which implies $\mathcal{E}_2, I_2 \models P_2$.

2.1.3 Parallel Composition. Given two event systems, $\mathcal{E}_i = (S_i, E_i, \rightarrow_i)$ for $i \in \{1, 2\}$, a set of events E, and a partial function $\chi\colon E_1 \times E_2 \rightharpoonup E$, we define their *parallel composition* $\mathcal{E}_1 \parallel_\chi \mathcal{E}_2 = (S, E, \rightarrow)$, where $S = S_1 \times S_2$ and $(s_1, s_2) \xrightarrow{e} (s_1', s_2')$ iff there exist $e_1 \in E_1$ and $e_2 \in E_2$ such that $\chi(e_1, e_2) = e$, $s_1 \xrightarrow{e_1} s_1'$, and $s_2 \xrightarrow{e_2} s_2'$. We define the *interleaving* composition $\mathcal{E}_1 \,|||\, \mathcal{E}_2 = \mathcal{E}_1 \parallel_{\chi_I} \mathcal{E}_2$, where $E = E_1 \uplus E_2$ and $\chi_I(e_1, \mathsf{skip}) = e_1$, $\chi_I(\mathsf{skip}, e_2) = e_2$, and $\chi_I(e_1, e_2) = \bot$ if $\mathsf{skip} \notin \{e_1, e_2\}$.

We can also define a composition on sets of traces. For two trace properties T_1 and T_2 over events E_1 and E_2, a set of events E, and a partial function $\chi\colon E_1 \times E_2 \rightharpoonup E$, we define $\tau \in T_1 \parallel_\chi T_2$ iff there exist $\tau_1 \in T_1$ and $\tau_2 \in T_2$ such that $len(\tau_1) = len(\tau_2) = len(\tau)$ and, for $0 \le i < len(\tau)$, we have $\chi(\tau_1(i), \tau_2(i)) = \tau(i)$. We can then prove the following composition theorem (Theorem 2.3), which enables compositional refinement (Corollary 2.4), where we can refine individual components while preserving trace inclusion for the composed system. Similar results existed previously (see, e.g., [Silva and Butler 2010]), but we have generalized them and formalized them in Isabelle/HOL.

Theorem 2.3 (Composition theorem). $traces(\mathcal{E}_1 \parallel_\chi \mathcal{E}_2, I_1 \times I_2) = traces(\mathcal{E}_1, I_1) \parallel_\chi traces(\mathcal{E}_2, I_2)$.

Corollary 2.4 (Compositional refinement). *Suppose* $traces(\mathcal{E}_i', I_i') \subseteq traces(\mathcal{E}_i, I_i)$ *for* $i \in \{1, 2\}$. *Then* $traces(\mathcal{E}_1' \parallel_\chi \mathcal{E}_2', I_1' \times I_2') \subseteq traces(\mathcal{E}_1 \parallel_\chi \mathcal{E}_2, I_1 \times I_2)$.

2.2 I/O Specifications for Separation Logic

To satisfy the versatility and expressiveness requirements stated in the introduction, we use a verification technique that works with *any* separation logic that offers a few standard features. This approach supports a wide range of programming languages, program logics, and verification tools.

We build on the work by Penninckx et al. [2015], which enables the verification of possibly non-terminating reactive programs that interact with their environment through a given set of I/O operations, corresponding to I/O library functions, using standard separation logic. They introduce an expressive assertion language for specifying a program's allowed I/O behavior; for example, one can specify sequential, non-deterministic, concurrent, and counting I/O behavior. This language can be encoded into any existing separation logic that offers standard features such as abstract predicates [Parkinson and Bierman 2005]. Consequently, our approach inherits the virtues of these logics, for instance, local reasoning and support for language features such as mutable heap data structures and concurrency (including fine-grained and weak-memory concurrency). In particular, our approach leverages existing program verification tools for separation logic, such as VeriFast [Jacobs et al. 2011] (for Java and C), Nagini [Eilers and Müller 2018] (for Python), and GRASShopper [Piskac et al. 2013], and benefits from the automation they offer.

Syntax. We assume a given set of (basic) *I/O operations* $bio \in Bios$ and countably infinite sets of values $v, w \in Values$ and places $t, t' \in Places$. The set of *chunks* is defined by

$$Chunks ::= bio(t, v, w, t') \mid token(t),$$

where $bio \in Bios$, $t, t' \in Places$, and $v, w \in Values$. We call a chunk of the form $bio(t, v, w, t')$ an *I/O permission* to invoke the operation bio with output v, whose *source* and *target* places are t and t', respectively, and which *predicts* receiving the input value w. Note that input and output are from the perspective of the calling system component, not the environment: for example, $send(t_1, 12, 0, t_2)$ models a permission to send the value 12 (output) and a prediction that the obtained result will

be 0 (input). A chunk of the form $token(t)$ is called a *token* at place t. Intuitively, the places and I/O permissions form the nodes and edges of a multigraph. Allowed I/O behaviors are obtained by pushing tokens along these edges, which consumes the corresponding I/O permissions.

The language of assertions, intended to describe multisets of chunks representing possibly non-terminating behavior, is co-inductively defined (indicated by the subscript ν) by

$$\phi ::=_\nu b \mid c \mid \phi_1 \star \phi_2 \mid \exists v.\ \phi \mid \exists t.\ \phi,$$

where $b \in \mathbb{B}$, $c \in Chunks$, $\phi_1 \star \phi_2$ is the separating conjunction, and the two existential quantifiers are on values $v \in Values$ and places $t \in Places$, respectively. In separation logic, chunks can be modeled using abstract predicates; all other assertions are standard. In our Isabelle/HOL formalization, we use a shallow embedding of assertions. Disjunction is encoded using existential quantification. We borrow other constructs such as the conditional "if b then ϕ_1 else ϕ_2", variables, and functions operating on values from the meta-language, namely, Isabelle's higher-order logic. We also call assertions *I/O specifications* to emphasize their use as program specifications.

Example 2.5. The following I/O specification allows receiving an integer and subsequently sending the doubled value.

$$\phi = token(t) \star (\exists x, t', t''.\ recv(t, x, t') \star send(t', 2x, t'')).$$

Since the input value x is existentially quantified and unconstrained, there is no prediction about the value that will be received. Here, we use I/O permissions performing only input (*recv*) or only output (*send*) instead of both. For such permissions, we elide the irrelevant argument, implicitly setting it to a default value like the unit $\bullet$. The single token points to the source place t of *recv*.

Note that I/O specifications use places to determine the execution order of different I/O operations without requiring specific program logic support beyond normal separation logic. For example, sequential composition and choice are expressed by using separate chunks that share source or target places. Determining whether an I/O operation may be performed is therefore as simple as checking whether there is a permission that has a source place with a token. Other approaches use custom specification constructs to express this and require custom logics (e.g., Koh et al. [2019]; Oortwijn and Huisman [2019]).

Repeating behavior. The co-inductive definition of assertions allows us to define formulas co-recursively. For consistency, Isabelle/HOL requires that co-recursive calls are *productive* [Blanchette et al. 2017], namely, guarded by some constructor, which is the case for all of our co-recursive definitions. For example, for a countable set of values S, we define the iterated separating conjunction $\forall^\star v \in S.\ \phi$. We can also co-recursively define possibly non-terminating I/O behavior.

Example 2.6. The assertion $\phi = token(t) \star RS(t, 0)$ specifies the behavior of repeatedly receiving inputs and sending their sum, as long as the received values are positive.

$$RS(t, a) =_\nu \exists z, t', t''.\ recv(t, z, t') \star \text{if } z > 0 \text{ then } send(t', a + z, t'') \star RS(t'', a + z) \text{ else true}.$$

Here, the parameters t and a of RS represent the current state. Since this is a co-recursive definition, it includes the non-terminating behaviors where all received values are strictly positive.

Semantics. Assertions have both a static semantics in terms of multisets of chunks and a transition semantics for which we have given an intuition above. This intuition suffices to understand our methodology. We therefore defer the definition of the formal semantics to Section 5.1.

3 IGLOO METHODOLOGY

In this section, we present our approach for developing fully verified distributed systems, which satisfies the requirements set out in the introduction. Our approach applies to any system whose components maintain a local state and exclusively communicate over an environment such as a communication network or a shared memory system. There are neither built-in assumptions about the number or nature of the different system components nor about the environment. In particular, the environment may involve faulty or adversarial behavior. We also support different programming languages and code verifiers for the implementation and the interoperability of heterogenous components written in different languages. This versatility is enabled by separating the modeling and implementation side and using I/O specifications to link them.

After giving an overview of our methodology (Section 3.1) and the distributed leader election protocol case study (Section 3.2), we explain our methodology's steps and illustrate them by transforming an informal, high-level description of the system and its environment into real-world implementations in Java and Python with formal correctness guarantees (Sections 3.3–3.8). We summarize our approach's soundness arguments (Section 3.9): trace properties established for the models are preserved down to the implementation provided that our trust assumptions (Section 3.10) hold. We currently support the verification of safety properties, but not liveness properties.

3.1 Overview of Formal Development Steps

Before we start a formal development, we must identify the system requirements and the assumptions about the environment. The system requirements include the (informally stated) goals to be achieved by the system and structural constraints such as the types of its components. The environment assumptions describe the properties of the environment, including communication channels (e.g., asynchronous, lossy, reordering channels), the types of component faults that may occur (e.g., crash-stop or Byzantine [Cachin et al. 2011]), and possible adversarial behavior (e.g., the Dolev-Yao model of an active network adversary [Dolev and Yao 1983]).

Our methodology consists of six steps (cf. Figure 1). In Steps 1–2, we use standard refinement to develop a detailed model of the system and its environment. The number of refinements per step is not fixed. Each refinement is proven correct and may incorporate additional system requirements.

(1) *Abstract models.* We start with an abstract model that takes a global perspective on the problem. It may solve the problem in a single transition. Typically, the most central system properties are already established for this model, or the abstract models that further refine it.
(2) *Protocol models.* We then move from the global to a distributed view, where nodes execute a protocol and communicate over the environment. The result of this step is a model that incorporates all system requirements and environment assumptions.

In Steps 3–6, we produce an interface model from which we can then extract component specifications, implement the components, and verify that they satisfy their specifications.

(3) *Interface models.* We further refine the protocol model for the subsequent decomposition into system components and the environment, taking into account the I/O library interfaces to be used by the implementation.
(4) *Decomposition.* We decompose the monolithic interface model into system components and the environment. Their re-composition is trace-equivalent with the monolithic model.
(5) *Component I/O specification.* We translate the component models into trace-equivalent I/O specifications (in separation logic) of the programs that implement the components.
(6) *Component implementation and verification.* We implement the components in a suitable programming language and prove that they satisfy their I/O specification.

Steps 1–4 are supported by a generic refinement and composition framework that we have embedded in Isabelle/HOL (see Sections 2.1 and 3.6). Steps 3–5 are novel and specific to our approach. In Steps 3 and 4, we align our models' events with the implementation's I/O library functions and then separate the interface model into a set of possibly heterogeneous system components (e.g., clients and servers) and the application-specific environment (e.g., modeling a particular network semantics, faulty, or adversarial behavior). Step 5 constitutes one of the core contributions of our approach: a sound link between abstract models and I/O specifications in separation logic, also formalized in Isabelle/HOL. It will be introduced informally here and formalized in Section 5. Step 6 corresponds to standard code verification, using tools such as Nagini (for Python) and VeriFast (for Java and C). Due to our clear separation of modeling and implementation, the code verifier must check only that a component implementation follows the protocol; code verification neither needs to reason about the protocol's global properties nor about the environment, which simplifies verification and increases modularity. In Section 3.9, we will derive the overall soundness of our methodology from the individual steps' guarantees, which are summarized in Table 2.

Our three case studies demonstrate the versatility and expressiveness of our approach. We cover different types of systems, including fault-tolerant and secure ones, different component types with unbounded numbers of instances, and TCP and UDP communication. We have written and verified implementations in Python and Java, including concurrent ones. This section illustrates our approach using the leader election case study; the other case studies are presented in Section 4.

3.2 Case Study: Leader Election

The main requirement of a distributed leader election protocol is to elect at most one leader in a network of uniquely identified but otherwise identical nodes, whose total number is a priori unknown. Since we do not consider liveness properties in this work, we do not prove that the protocol will terminate with an elected leader.

We model an algorithm by Chang and Roberts [1979], which assumes a ring network and a strict total order on the set of node identifiers. The algorithm elects the node with the maximum identifier as follows. Each node initially sends out its identifier to the next node in the ring and subsequently forwards all received identifiers greater than its own. When a node receives its own identifier, this is guaranteed to be the maximum identifier in the ring, and the node declares itself the leader. For the environment, we assume that each node asynchronously sends messages to the next node in the ring over an unreliable, duplicating, and reordering channel. We do not consider other faults or adversarial behavior in this example, but see Section 4 for case studies that do.

3.3 Step 1: Abstract Models

A common approach to develop systems by refinement is to start from a very abstract model whose correctness is either obvious or can be proved by a set of simple invariants or other trace properties. This model takes a global "bird's eye" view of an entire run of the protocol in that it does not explicitly model the network communication or represent the individual protocol steps.

Example 3.1. The abstract model of leader election elects a leader in a single "one-shot" transition. We assume a given set ID of node identifiers. The model's state space is defined as an ID-indexed family of local states containing a single boolean state variable identifying the leader, i.e., $S_0 = ID \rightarrow (\!| \mathit{leader} \in \mathbb{B} |\!)$. Initially, $\mathit{leader}(s_0(i)) = \mathsf{false}$, for all $i \in ID$. There is a single event *elect* that elects the leader. The guard ensures that this event can be performed only by a single, initially arbitrary node that updates its local variable *leader* to true.

$$\mathit{elect}(i) : \quad (\forall j.\ \mathit{leader}_j \Rightarrow i = j) \quad \triangleright \quad \mathit{leader}_i := \mathsf{true}.$$

We use indexing to refer to different instances of variables, e.g., $leader_j$ refers to node j's local state. Note that the guard refers to other nodes' local states; hence, this model takes a global point of view. We have proved that this model satisfies the main requirement for leader election, namely, the uniqueness of the leader. This is formalized as the trace property

$$U_0 = \{\tau \mid \forall i, j.\ elect(i) \in \tau \wedge elect(j) \in \tau \Rightarrow i = j\},$$

where $e \in \tau$ means that the event e occurs in the trace τ. This model is sufficiently abstract to specify *any* leader election algorithm, and will be refined to the protocol described above next.

3.4 Step 2: Protocol Models

In Step 2, we move from a global to a distributed perspective, and distinguish system components (e.g., nodes or clients and servers) that communicate over an environment (e.g., a wide-area network). The way that we model the environment accounts for any assumptions made about network communication. For example, we can represent a reliable, non-duplicating, reordering channel as a multiset of messages. This step may also introduce a failure model for fault-tolerant systems or an adversary model for secure systems. The result of this step is a complete model of our system and environment that satisfies all system requirements.

Example 3.2. We refine our abstract model into a protocol model. We model the environment by assuming a finite, totally ordered set of identifiers ID and that the nodes are arranged in a ring defined by a function $next\colon ID \to ID$, where $next(i)$ yields node i's successor in the ring. We extend the state with communication channels, which we model as sets, from which messages are never removed; this represents our assumption that the network may reorder and duplicate messages. Since we do not consider liveness properties, message loss is implicitly represented by never receiving a message. Since messages contain node identifiers, our state space becomes $S_1 = ID \to (\!|\ leader \in \mathbb{B}, chan \in \mathbb{P}(ID)\ |\!)$.

Three events model the protocol steps: a *setup* event where nodes send their own identifier to the next node in the ring, an *accept* event where they forward received identifiers greater than their own, and an *elect* event where a node receiving its own identifier declares itself the leader.

$$\begin{array}{lcll} setup(i) & : & \mathsf{true} & \triangleright\ chan_{next(i)} := chan_{next(i)} \cup \{i\} \\ accept(i,j) & : & j \in chan_i \wedge j > i & \triangleright\ chan_{next(i)} := chan_{next(i)} \cup \{j\} \\ elect(i) & : & i \in chan_i & \triangleright\ leader_i := \mathsf{true} \end{array}$$

We have proved that this protocol model refines the abstract model defined in Example 3.1. For this we use the simulation relation that removes the field *chan* from the local state and the mediator function that maps *elect* to itself and the new events to skip. The proof involves showing that the guard of this model's *elect* event implies the guard of the abstract model's *elect* event. We prove two invariants that together imply this. The first one is inductive and states that if a node ID i is in the channel of node j then $k < i$ for all node IDs k in the channels in the ring interval from i to j. From this it follows that if $i \in chan_i$ then i is the maximal node ID. The second invariant expresses that only the node with the maximal node ID can become a leader.

3.5 Step 3: Interface Models

This is the first step towards an implementation. Its purpose is twofold: first, we prepare the model for the subsequent decomposition step (Step 4) and, second, we align the I/O events with the API functions of the I/O libraries to be used in the implementation. The resulting interface model must satisfy the following structural *interface requirements*:

$$
\begin{array}{lcll}
setup_i() & : & \mathsf{true} & \triangleright\ obuf_i := obuf_i \cup \{i\} \\
receive_i(j) & : & j \in chan_{addr(i)} & \triangleright\ ibuf_i := ibuf_i \cup \{j\} \\
accept_i(j) & : & j \in ibuf_i \wedge j > i & \triangleright\ obuf_i := obuf_i \cup \{j\} \\
send_i(j, a) & : & j \in obuf_i \wedge a = addr(next(i)) & \triangleright\ chan_a := chan_a \cup \{j\} \\
elect_i() & : & i \in ibuf_i & \triangleright\ leader_i := \mathsf{true}.
\end{array}
$$

Fig. 2. Event system resulting from interface refinement step.

(1) The state space is a product of the components' local state spaces and the environment's state space. The events are partitioned into *I/O events*, which model the communication with the environment, and *internal events*, which model local computations.
(2) Each I/O event can be associated with a single I/O library function (e.g., receiving or sending a message on a socket, but not both). It must have the same parameters as that library function, each of which can be identified as an output parameter (e.g., the message to send) or an input parameter (e.g., an error code returned as a result).
(3) Each I/O event's guard must be the conjunction of
    - a *component guard*, which refers only to the component's local state, the event's output parameters, and the component identifier, and
    - an *environment guard*, referring only to the environment's state, the input parameters, and the component identifier.

Our approach leaves the choice of the abstraction level of the interface model's I/O events to the user. For example, the APIs of network socket libraries typically represent payloads as bitstrings, which the application must parse into and marshal from its internal representation. We may choose to either (i) define I/O events (and thus I/O operations) that operate on bitstrings, which requires modeling and verifying parsing and marshalling explicitly, or (ii) keep their interface on the level of parsed data objects, and trust that these functions are implemented correctly.

Example 3.3. We refine the protocol model into a model satisfying the interface requirements. The protocol model's *accept* event receives, processes, and sends a message. To satisfy Conditions 1–2 above, we introduce two local buffers, *ibuf* and *obuf*, for each node and split *accept* into three events: *receive* transfers a message from the previous node to the input buffer *ibuf*, *accept* processes a message from *ibuf* and places the result in the output buffer *obuf*, and *send* sends a message from *obuf* to the next node.

We also align the I/O events *send* and *receive* with the I/O operations $UDP\_send\_int(msg, addr)$ and $UDP\_receive\_int(msg)$, which are offered by standard socket libraries. Here, we represent messages as integers, but as stated above, we could alternatively represent them as bitstrings, and model parsing and marshalling explicitly (including bounds and endianness), resulting in stronger correctness guarantees. Since each I/O event must match the corresponding I/O operation's parameters (Condition 2), we add the send operation's destination address as an event parameter. Hence, we introduce an injective function $addr : ID \to Addr$, where $Addr$ is the set of addresses. UDP communication is unreliable and messages sent may be reordered, duplicated, or lost; our environment model faithfully represents this behavior by modeling channels as sets (Section 3.4).

We define the state space as the product $S_2 = S_2^{\mathrm{s}} \times S_2^{\mathrm{e}}$ (Condition 1) of a system state space $S_2^{\mathrm{s}} = ID \to (\!|\ leader \in \mathbb{B}, ibuf \in \mathbb{P}(ID), obuf \in \mathbb{P}(ID)\ |\!)$ and an environment state space $S_2^{\mathrm{e}} = Addr \to (\!|\ chan \in \mathbb{P}(ID)\ |\!)$. The events are specified in Figure 2. We henceforth consider the component identifier i as a component parameter and therefore write it as a subscript of the event. Only *receive* and *send* are I/O events; all others are internal (Condition 1). These I/O events have the required

form and parameters (Condition 2) and their guards have the required separable form (Condition 3). The parameter j of *receive* is the only input parameter and all others are outputs. The simulation relation with the protocol model projects away the internal buffers. The mediator function maps *elect* to itself, $send_i(j, a)$ to $setup(i)$ if $i = j$ and to $accept(i, j)$ otherwise, and all other events to skip. The refinement proof requires an invariant relating internal buffers to channels, e.g., stating that $j \in ibuf_i$ implies $j \in chan_{addr(i)}$.

3.6 Step 4: Decomposition

To support distributed systems with different component types (such as nodes or clients and servers), we decompose the monolithic interface model from Step 3 into a parallel composition of an environment model and (a family of) component models for each component type.

We first decompose the interface model into a parallel composition $\mathcal{E} = \mathcal{E}^{\mathrm{s}} \parallel_{\chi} \mathcal{E}^{\mathrm{e}}$ of a system model $\mathcal{E}^{\mathrm{s}}$ and an environment model $\mathcal{E}^{\mathrm{e}}$. We have already distinguished their respective state spaces S^{s} and S^{e} in the interface model. The I/O events e of $\mathcal{E}$ are split into a system part e^{s}, consisting of e's component guard and system state updates, and an environment part e^{e}, consisting of e's environment guard and environment state updates. We define χ such that it synchronizes the split I/O events and interleaves the internal events. The system model is further subdivided into models of different component types, which are composed using interleaving composition $\mathcal{E}^{\mathrm{s}} = \mathcal{E}^{\mathrm{s}}_1 \,|||\, \cdots \,|||\, \mathcal{E}^{\mathrm{s}}_n$. This reflects our assumption that the components exclusively communicate via the environment. If there are multiple *instances* of a component type, parametrized by a countable index set I of identifiers, the respective model, say $\mathcal{E}^{\mathrm{s}}_k$, becomes an interleaving composition over I, that is, $|||_{i \in I}\, \mathcal{E}^{\mathrm{s}}_k(\vec{\gamma}_k(i))$. Each component model $\mathcal{E}^{\mathrm{s}}_k(\vec{p})$ may have some parameters $\vec{p}$. We instantiate these using a *configuration map* $\vec{\gamma}_k$, which represents assumptions on the *correct system configuration*. Note that component models may be further refined before translating them to I/O specifications.

In preparation for the subsequent translation to I/O specifications, we model (instances of) system components in a subclass of guarded event systems. An *I/O-guarded event system* $\mathcal{G} = (S, E, G, U)$ is a guarded event system, where E consists of events of the form $bio(v, w)$ (formally introduced as *I/O actions* in Section 5.1) and all guards $G_{bio(v,w)}$ are component guards as in Condition (3), i.e., they must not depend on the I/O action's input w. This models that an input becomes available only as the result of an I/O operation and cannot be selected before the I/O operation is invoked. Furthermore, we model a component's internal events as *ghost I/O actions*; these actions change the state of the abstract model, but do not correspond to real I/O operations. The implementation may have to perform a corresponding state change to stay aligned with the abstract model.

We prove the correctness of the decomposition by showing that the parallel (re)composition of all parts is trace-equivalent to the original system.

Example 3.4. All nodes instantiate the same component type. We thus decompose the model from the previous step into an environment event system $\mathcal{E}^{\mathrm{e}}$ and an I/O-guarded event system $\mathcal{E}^{\mathrm{s}}(i, a)$, parametrized by a node identifier $i \in ID$ and an address $a \in Addr$. These will also be the parameters of the future program $c(i, a)$ implementing $\mathcal{E}^{\mathrm{s}}(i, a)$. For the system's (re)composition, we use the configuration map $\vec{\gamma}(i) = (i, addr(next(i)))$, which instantiates the destination address a for i's outbound messages with the address of i's successor in the ring. The environment operates on the state $S^{\mathrm{e}}_3 = Addr \to (\!|\, chan \in \mathbb{P}(ID) \,|\!)$ and the state space of each node model $\mathcal{E}^{\mathrm{s}}(i, a)$ is $S^{\mathrm{s}}_3 = (\!|\, leader \in \mathbb{B}, ibuf \in \mathbb{P}(ID), obuf \in \mathbb{P}(ID) \,|\!)$. The environment has the following events, where '$-$' represents the identity update function:

$$
\begin{array}{lcll}
receive^{\mathrm{e}}(i, m) & : & m \in chan_{addr(i)} & \triangleright\ - \\
send^{\mathrm{e}}(i, m, a) & : & \mathsf{true} & \triangleright\ chan_a := chan_a \cup \{m\}.
\end{array}
$$

$$
\begin{aligned}
P(t,(i,a),s) =_\nu\ & (\exists t'.\ setup(t,t') \star P(t',(i,a),s(\!|\ obuf := obuf(s) \cup \{i\}\ |\!))) \star \\
& (\exists m, t'.\ UDP\_receive\_int(t,m,t') \star P(t',(i,a),s(\!|\ ibuf := ibuf(s) \cup \{m\}\ |\!))) \star \\
& (\forall^\star m.\ \mathsf{if}\ m \in ibuf(s) \wedge i < m\ \mathsf{then}\ \exists t'.\ accept(t,m,t') \star \\
& \qquad P(t',(i,a),s(\!|\ obuf := obuf(s) \cup \{m\}\ |\!))\ \mathsf{else\ true}) \star \\
& (\forall^\star m, a'.\ \mathsf{if}\ m \in obuf(s) \wedge a' = a\ \mathsf{then}\ \exists t'.\ UDP\_send\_int(t,(m,a'),t') \star \\
& \qquad P(t',(i,a),s)\ \mathsf{else\ true}) \star \\
& (\mathsf{if}\ i \in ibuf(s)\ \mathsf{then}\ \exists t'.\ elect(t,t') \star P(t',(i,a),s(\!|\ leader := \mathsf{true}\ |\!))\ \mathsf{else\ true}).
\end{aligned}
$$

Fig. 3. I/O specification of leader election nodes.

These events execute synchronously with their matching system parts:

$$
\begin{array}{lcll}
receive^{\mathrm{s}}_{i,a}(m) & : & \mathsf{true} & \triangleright\ ibuf := ibuf \cup \{m\} \\
send^{\mathrm{s}}_{i,a}(m,a') & : & m \in obuf \wedge a' = a & \triangleright\ -\,.
\end{array}
$$

Note that the $receive^{\mathrm{s}}$ event's guard does not depend on its input parameter m and the $send^{\mathrm{s}}_{i,a}$ event's single output parameter is a pair of a message and an address. The equality $a' = a$ in the guard of $send^{\mathrm{s}}_{i,a}$ enforces that messages are sent only to the node at the address a, which is a component parameter. This is a constraint on the future program's use of the I/O library function. The internal events $setup_{i,a}()$, $accept_{i,a}(m)$, and $elect_{i,a}()$ of $\mathcal{E}^{\mathrm{s}}(i,a)$ are ghost I/O actions, which are identical to their counterparts in the previous model modulo their slightly different parametrization. We have proved that the composition of all parts is trace-equivalent to the original monolithic system.

3.7 Step 5: I/O Specifications

We can now perform the central step of our approach: we extract, for each component, an I/O specification that defines the implementation's I/O behavior. Our translation maps an I/O-guarded parametrized event system $\mathcal{E}^{\mathrm{s}}(\vec{p})$ to an I/O specification of the form

$$\phi(\vec{p}) = \exists t.\ token(t) \star P(t,\vec{p},s_0),$$

where P is a co-recursively defined predicate encoding the events and s_0 is the event system's initial state.[1] The predicate P takes a place t, the event system's (and future program's) parameters $\vec{p}$, and the event system's abstract state s as arguments. The predicate P contains, for each event and all values of its output parameters satisfying the guard, a permission to execute the I/O operation represented by the event, and another instance of itself with the argument representing the new state resulting from applying the event's update function. This translation is formally defined and proved correct in Section 5. Here, we explain the intuition behind it using our example.

Example 3.5. Figure 3 defines the predicate $P(t,(i,a),s)$ for our example, where i and a denote the local node identifier i and the address a of the next node in the ring. The fourth top-level conjunct of P corresponds to the $send^{\mathrm{s}}_{i,a}(m,a')$ event from the previous step. It states that for all possible values of the output parameter (m,a') that fulfill the event's guard $m \in obuf(s) \wedge a' = a$, there is a permission to perform the I/O operation *UDP_send_int* (which is mapped to the $send^{\mathrm{s}}_{i,a}$ event) and another instance of P at the operation's target place with the same state, since $send^{\mathrm{s}}_{i,a}$ does not change the local state. The second (simplified) conjunct corresponds to the $receive^{\mathrm{s}}_{i,a}$ event and

[1] The formal development of our theory (Section 5) is based on event systems with single initial states. This is without loss of generality since multiple initial states can easily be introduced by a non-deterministic initialization event.

```
def main(my_id: int, out_host: str):
    #@ PRE: exists t . token(t) and P(t, (my_id, out_host), INIT_STATE)
    #@ POST: true
    to_send = my_id     # variable stores only the largest identifier seen so far
    setup()             # ghost I/O operation
    while True:
        #@ INVARIANT: exists t, s . token(t) and P(t, (my_id, out_host), s)
        #@ INVARIANT: to_send in s.obuf and to_send >= my_id
        send_int(out_host, to_send)
        msg = try_receive_int()      # returns None on timeout
        if msg is not None:
            if msg is my_id:
                elect()     # ghost I/O operation
                break
            elif msg > to_send:
                accept(msg)     # ghost I/O operation
                to_send = msg
```

Listing 1. Pseudocode of the leader election algorithm with proof annotations (simplified). The method *try_receive_int* tries a receive operation and either returns an identifier or times out and returns *None*.

existentially quantifies over the event's input parameter and contains another predicate instance with an updated state $s(\!| \mathit{ibuf} := \mathit{ibuf}(s) \cup \{m\} |\!)$ as defined by $\mathit{receive}^s_{i,a}$. The remaining conjuncts correspond to the internal events *setup*, *accept*, and *elect*.

3.8 Step 6: Component Implementation and Verification

In the final step, we prove for every component that its implementation c fulfills the I/O specification ϕ that was extracted in the previous step. This requirement is expressed as

$$\mathit{traces}(c) \subseteq \mathit{traces}(\phi), \tag{1}$$

i.e., the I/O traces of the component implementation c, as defined by its operational semantics, are included in those specified by the I/O specification ϕ. Here, we elide possible parameters $\vec{p}$ of the program c and the I/O specification ϕ for the sake of a lighter notation. Since I/O specifications are language-agnostic, the implementation may use any programming language. Verifying (1) typically requires defining a suitable I/O-aware semantics of the chosen language that defines the I/O traces produced by its programs. We assume that the verification technique used defines an interpretation of Hoare triples of the form $\models \{\phi\}\ c\ \{\psi\}$, and a sound program logic to prove them. We only rely on the *verifier assumption* stating that the correctness of a command with respect to a precondition implies the trace inclusion between the command and the precondition assertion. Since the I/O permissions in the precondition restrict which I/O operations may be performed, these triples do not trivially hold even though the postcondition is true:

$$\models \{\phi\}\ c\ \{\mathsf{true}\} \quad \text{implies} \quad \mathit{traces}(c) \subseteq \mathit{traces}(\phi). \tag{VA}$$

Our approach leaves open the mechanism for proving such Hoare triples. In principle, proofs can be constructed using an interactive theorem prover, an SMT-based automated verifier, or even as pen-and-paper proof. I/O specifications consist only of constructs that can be expressed using standard separation logic with abstract predicates. This allows us to leverage existing tool support, in particular, proof automation. For instance, encoding such specifications in VeriFast required less than 25 LoC to declare types for places and abstract predicates for chunks, and no tool modifications.

The I/O specification is (currently manually) converted to the syntax of the respective tool, and the program verifier is then used to prove the correctness of the program with respect to its I/O specification. Assuming (VA) holds for the tool, this guarantees the required trace inclusion (1).

```
def send_int(address: str, msg: int):
    #@ PRE: token(?t) * UDP_send_int(t, (msg, address), ?tp)
    #@ PRE: connected(this, address)
    #@ PRE: 0 <= msg <= MAX_MSG_VAL
    #@ POST: token(tp)
    #@ EXCEPTIONAL POST: token(t) * UDP_send_int(t, (msg, address), tp)
```

Listing 2. The simplified pseudocode contract for a library method for sending packets via UDP. The names starting with question marks are implicitly existentially quantified. *connected* is a separation logic predicate that contains the heap memory of a UDP socket object connected to the shown address.

Besides the verification, we must manually justify that the actual program's I/O operations satisfy the assumptions encoded in the environment model. For example, we may implement an order-preserving network channel model using TCP sockets, but not UDP sockets. Conversely, it is sound to implement an unordered channel model using either TCP or UDP communication.

Example 3.6. We have implemented three versions of the leader election algorithm, a sequential and a concurrent one in Java and a sequential version in Python, and verified in VeriFast and Nagini that these implementations conform to their I/O specification: $\models \{\phi(i, a)\}\ main(i, a)\ \{\mathsf{true}\}$. All three implementations are interoperable and successfully elect a leader in actual networks.

Listing 1 shows a slightly simplified pseudocode version of the sequential implementation; the Java and Python versions share the same structure but contain additional annotations as required by the respective verifier. The concurrent version uses two separate threads for receiving and sending identifiers. We use the standard UDP socket libraries of the respective languages; since the API in both cases is structured differently, we defined the I/O operations used in the specification to be compatible with both. We annotated the relevant I/O library operations with contracts, whose correctness is assumed and must be validated manually against the environment model.

Listing 2 shows a simplified pseudocode specification of a message sending function. Its precondition consists of three typical parts that (i) specifiy the I/O behavior of this function in terms of tokens and I/O permissions, (ii) constrain the program state, in this case requiring that our socket is already connected to the receiver address, and (iii) impose additional restrictions on messages that do not exist on more abstract levels, in this case, that the sent message falls within a valid range.

Since the I/O specification describes the allowed I/O behavior in terms of the model's state, the verification process requires relating the program to the model state. The latter is represented in the program as a *ghost state*, which is present only for verification purposes, but not during program execution. If the verifier can prove for a given program point that a token for a place t and the predicate $P(t, (i, a), s)$ are held for some model state s, this means that the current program state corresponds to the model state s. The invocations of the internal operations *setup*, *accept*, and *elect* in the code update the ghost state to stay aligned with the program state.

As an optimization, the implementations store and forward only the largest identifier seen so far, since smaller ones can never belong to the leader. The verifier proves the loop invariant that this largest integer is always in the output buffer and may therefore be sent out.

Note that, although we do not prove liveness, our implementation repeatedly resends UDP packets since packets may be lost. This will continue even after a leader has been elected since our simple protocol does not include a leader announcement phase.

3.9 Overall Soundness Guarantees

Our methodology provides a general way of proving properties of a distributed system. Table 2 summarizes the soundness guarantees of each step (see also Figure 1). We now show how to

Table 2. Method overview with guarantees of each step (initial states elided).

steps	activity	guarantee	justification
1–3	model refinements and interface refinement	$\hat{\pi}_i(traces(\mathcal{M}_m)) \subseteq traces(\mathcal{M}_i)$ where $\hat{\pi}_i = \pi_m \circ \cdots \circ \pi_{i+1}$	ref. $\mathcal{M}_{i+1} \sqsubseteq_{\pi_{i+1}} \mathcal{M}_i$ Theorem 2.1
4	decompose $\mathcal{M}_m$ into $\mathcal{E}^{\mathrm{s}}_1, \ldots, \mathcal{E}^{\mathrm{s}}_n$, and $\mathcal{E}^{\mathrm{e}}$	$traces(\mathcal{M}_m) = traces((\mathcal{E}^{\mathrm{s}}_1 \mid\mid\mid \ldots \mid\mid\mid \mathcal{E}^{\mathrm{s}}_n) \parallel_{\chi_e} \mathcal{E}^{\mathrm{e}})$	mutual refinement Theorem 2.1
5	translate $\mathcal{E}^{\mathrm{s}}_j$ into ϕ_j	$traces(\phi_j) = traces(\mathcal{E}^{\mathrm{s}}_j)$	Theorems 5.5 and 5.7
6	verify $\models \{\phi_j\}\ c_j\ \{\mathrm{true}\}$	$traces(c_j) \subseteq traces(\phi_j)$	Assumption (VA)

combine them to obtain the overall soundness guarantee that the models' properties are preserved down to the implementation. We will first discuss the simpler case with a single instance of each component and later extend this reasoning to multiple instances.

Let our implemented system be defined by $\mathcal{S} = (C_1 \mid\mid\mid \ldots \mid\mid\mid C_n) \parallel_{\chi_e} \mathcal{E}^{\mathrm{re}}$, where each C_j is the event system defining the operational semantics of c_j, i.e., $traces(C_j) = traces(c_j)$ and suppose the event system $\mathcal{E}^{\mathrm{re}}$ corresponds to the real environment. From Steps 5–6's guarantees listed in Table 2, we derive $traces(C_j) \subseteq traces(\mathcal{E}^{\mathrm{s}}_j)$. Furthermore, we assume that the environment model faithfully represents the real environment, i.e.,

$$traces(\mathcal{E}^{\mathrm{re}}) \subseteq traces(\mathcal{E}^{\mathrm{e}}). \tag{EA}$$

Using Corollary 2.4 and Step 4's guarantee, we derive that the implemented system's traces are included in the interface model's traces: $traces(\mathcal{S}) \subseteq traces(\mathcal{M}_m)$. Using Lemma 2.2 and the guarantees of Steps 1–3, we derive our overall soundness guarantee stating that any trace property P_i of model $\mathcal{M}_i$ is preserved all the way down to the implementation:

$$\mathcal{M}_i \models P_i \implies \mathcal{S} \models \hat{\pi}_i^{-1}(P_i). \tag{SOUND}$$

With multiple component instances, the event systems $\mathcal{E}^{\mathrm{s}}_j$ and C_j have the form of compositions $|||_{i \in I}\ \mathcal{E}^{\mathrm{s}}_j(\vec{\gamma}_j(i))$ and $|||_{i \in I}\ C_j(\vec{\gamma}_j(i))$ for some configuration map $\vec{\gamma}_j$ and we have $traces(C_j(\vec{p})) = traces(c_j(\vec{p}))$ for all parameters $\vec{p}$. The guarantees of Step 4 in Table 2 hold for these compositions and those of Steps 5–6 for the individual parametrized components. We then easily derive the guarantee (SOUND) using a theorem for indexed interleaving composition similar to Corollary 2.4.

3.10 Trust Assumptions

The guarantees described in the last subsection hold under the following trust assumptions:

(1) *Environment assumptions:* The modeled environment includes all possible behaviors of the real system's environment, as formulated in Assumption (EA). This means it faithfully represents the behavior of all real components below the interface of the I/O library used in the implementation. These may include the I/O library, the operating system, the hardware, the communication network, as well as potential attackers and link or node failures. Recent work by Mansky et al. [2020] demonstrates how to connect the verification of I/O behavior to a verified operating system to remove the I/O library and operating system from the trust assumptions. Their approach could be adapted to our setting. Other environment assumptions, such as the attacker model, remain and cannot be eliminated through formal proofs.

(2) *Correct program configuration:* The programs are called with parameters conforming to the configuration map $\vec{\gamma}$. For instance, our case study assumes that each node program is initialized with parameters $\vec{\gamma}(i) = (i, a)$ where i is a node identifier and $a = addr(next(i))$ is

the network address of i's successor in the ring. Verifying the configuration, which typically happens using scripts or manual procedures, is orthogonal to program correctness.

(3) *Manual translation of I/O specification:* The I/O specification is translated correctly from the Isabelle/HOL to the code verifier tool's syntax. This translation is manual, but well-defined and can thus be automated in the future by a translator implemented and verified in Isabelle.

(4) *Toolchain soundness.* The verification logics and tools are sound and all proofs are thus correct. They agree on the semantics of I/O specifications and the code verifier satisfies the verifier assumption (VA). In our case, this concerns Isabelle/HOL and either VeriFast (which uses Z3 [De Moura and Bjørner 2008]) or Nagini (which depends on Viper [Müller et al. 2016] and Z3). The trusted codebase could be reduced further by using a foundational verifier such as VST [Appel 2012], at the cost of a higher verification effort.

4 CASE STUDIES: FAULT-TOLERANCE AND SECURITY

We evaluate our methodology with two additional case studies that demonstrate the generality and versatility of Igloo. Concretely, we study a fault-tolerant, primary-backup replication system and an authentication protocol. These case studies showcase different features of our approach: (1) proofs of global, protocol-level properties, (2) environments with different types of networks as well as faulty and adversarial environments, (3) different component types with unbounded numbers of instances, and (4) sequential and concurrent implementations in different programming languages.

4.1 Primary-backup Replication

We apply our methodology to a primary-backup replication protocol presented by Charron-Bost et al. [2010, Sec. 2.3.1]. This case study exhibits the following features supported by our approach: (i) an environment that includes a fault model for components, (ii) reliable, in-order communication implemented by TCP, and (iii) sequential as well as concurrent implementations.

4.1.1 Description. The primary-backup replication protocol maintains an append-only distributed data structure, called *log*, which is a sequence of arbitrary data (e.g., database commands). One server, the *primary*, receives requests from clients to append elements to the log. The primary server first synchronizes a given append request with all other servers, the *backups*, before extending its own log and responding to the client. The protocol's goal is to maintain *backup consistency*, i.e., the log stored on the primary when it replies to the client is a prefix of the logs stored at all backups. We assume a fail-stop fault model, where servers can fail but not recover, and perfect failure detection, where all clients and servers eventually detect server failures (see, e.g., [Cachin et al. 2011]). The servers are totally ordered, and initially, the first server is the primary. A backup server becomes the new primary once it detects that all previous servers in the order have failed.

4.1.2 Protocol Model. In this case study, we have chosen not to model an abstract version of the protocol (Step 1), but rather the concrete protocol (Step 2) directly. While the normal (fault-free) operation of the protocol is straightforward, the non-deterministic failures and their detection add significant complexity. When a new primary server takes over, its log may diverge from those of the backups. By synchronizing its log with those of the backup servers, it reestablishes a consistent state before responding to a client. Once backup b has replied to a sync request from primary a, all logs contained in their states and sent between them are totally ordered in a prefix relation:

$$\mathit{ordered\text{-}wrt\text{-}prefix}(\langle \mathit{log}(a)\rangle \cdot \mathit{transit}(b,a) \cdot \langle \mathit{log}(b)\rangle \cdot \mathit{transit}(a,b) \cdot \langle \mathit{pend}(a)\rangle).$$

Here, $\mathit{pend}(a)$ is the primary's log including all pending additions, and $\mathit{transit}(a,b)$ is the sequence of logs in transit from a to b (in sync requests or responses). Additional inductive invariants and history variables are needed to derive this invariant. Careful modeling is also required for the

failure detection. The environment state contains a set $live_{env}$, consisting of all live servers. Since clients and servers may detect failures only after some delay, each of them has its own set *live* containing all servers except for those whose failure was noticed. As we show in an invariant, *live* sets are supersets of $live_{env}$. In total, our proof of backup consistency relies on nine invariants.

4.1.3 Towards an Implementation. Our protocol model already includes the input and output buffers that are typically only added in Step 3. This allows us to directly decompose the model into a trace-equivalent composition of client and server components and an environment (Step 4).

Besides *send* and *receive*, the clients and servers have a third I/O event, *detect-failure*, to query the failure detector. Its system part removes a server from the component's *live* set whereas its environment part has of a guard ensuring that the removed server is not in the $live_{env}$-set.

We extract I/O specifications for both the server and the client component types (Step 5). This extraction, as well as the equivalence proof between the component's event systems and their I/O specifications, follows the same standard pattern as in our security case study below. Thus, this step could likely be automated in the future.

4.1.4 Code Level. We implement a sequential client and a concurrent server in Java. To handle requests in parallel, we split the server into multiple threads, communicating over shared buffers, guarded by a lock. For TCP, we use Java's standard socket library. For failure detection, clients and servers get a failure detector object as an argument. This object provides methods to query whether a server has failed. If Java's socket API determines that a connection attempt times out, the failure detector is queried. According to the protocol, failed servers are removed from the set of *live* servers, otherwise the last action is repeated. For this case study, we provide a dummy failure detector instance, which stores the set of failed server ids. When we kill a server in our execution script, the server process is terminated and its id is added to that set. The setup of contracts for trusted libraries and the verification of our client and server implementations in VeriFast against their respective I/O specifications closely follows our approach described in Section 3.8.

In the server, all shared data is protected by a lock. For this lock, we define a *monitor invariant* [Leino and Müller 2009], declaring that the lock owns all shared data structures and the I/O permissions. The ownership of these resources is transferred to a thread when it acquires the lock and is transferred back when the lock is released. The I/O permissions define which I/O operations may be performed depending on the component model's state. Since the implementation's behavior depends on the actual program state, in particular the state guarded by the lock, we also need to link the model state to the actual state in the monitor invariant. We do this using an abstraction function. Thus, when executing an I/O operation, the associated model state update must be matched by a corresponding program state update before the lock can be released.

4.2 Two-party Entity Authentication

We also use our methodology to derive and implement a two-party authentication protocol standardized as ISO/IEC 9798-3. This case study demonstrates the following features of our approach: (i) an adversarial environment, (ii) the use of cryptography, (iii) unreliable, unordered channels implemented by UDP, and (iv) the use of data abstraction linking abstract message terms and their concrete cryptographic bitstring representations.

4.2.1 Description. In standard (informal) Alice&Bob notation, the protocol reads as follows.

$$\begin{aligned} &\text{M1.} \quad A \to B \quad : \quad A, B, N_A \\ &\text{M2.} \quad B \to A \quad : \quad [N_B, N_A, A]_{\mathrm{pri}(B)} \end{aligned}$$

Here N_A and N_B are the nonces generated by the initiator A and the responder B respectively, and $[M]_{\mathsf{pri}(B)}$ denotes the digital signature of the message M with B's private key. First, the initiator generates a fresh nonce and sends it to the responder. Afterwards, the responder generates his own nonce, signs it together with the initiator's nonce and name, and sends the signed message to the initiator. The nonces provide replay protection. The protocol's authentication goal is that the initiator agrees with the responder on their names and the two nonces.

4.2.2 Abstract and Protocol Models. We follow the four-level refinement strategy for security protocols proposed by Sprenger and Basin [2018]. Its levels corresponds to the first two steps of our methodology. We start from an abstract model of the desired security property, in our case, injective agreement [Lowe 1997]. We then refine this into a protocol model that introduces the two protocol roles (i.e., Igloo component types) and their runs. Each protocol run instantiates a role and stores the participants' names and received messages in its local state. We model an unbounded number of runs as a finite map from run identifiers to the runs' local states. The runs communicate over channels with intrinsic security properties. In our case, A sends her nonce on an insecure channel to B, who returns it with his own nonce on an authentic channel back to A. The attacker can read, but not modify, messages on authentic channels.

In a second refinement, we represent messages as terms, use digital signatures to implement authentic channels, and refine the attacker into a standard Dolev-Yao attacker [Dolev and Yao 1983] who completely controls the network. The attacker's capabilities are defined by a closure operator $DY(M)$, denoting the set of messages he can derive from the set of observed messages M using both message composition (such as encryption) and decomposition (such as decryption). All refined models correspond to Igloo protocol models with different levels of detail. The refinement proofs imply that they all satisfy injective agreement.

4.2.3 Towards an Implementation. We further refine the final protocol model into an interface model, where the messages are represented by bitstrings of an abstract type bs, which we later instantiate to actual bitstrings. We assume a surjective abstraction function $\alpha : bs \to msg$ from bitstrings to messages. A special term Junk represents all unparsable bitstrings. We define a concrete attacker operating on (sets of) bitstrings by $DY_{bs} = \alpha^{-1} \circ DY \circ \alpha$, where α^{-1} is the inverse of α lifted to message sets. The simulation relation includes the equation $\alpha(CIK) = IK$, where CIK and IK respectively represent the concrete attacker's knowledge and the Dolev-Yao attacker's knowledge. This allows us to transfer the Dolev-Yao model's security guarantees to the implementation.

We also augment each role's state with buffers for receiving and sending bitstring messages. The send and receive I/O events each take a network address and a bitstring message. The two roles' events still operate on message terms, but exchange messages with I/O buffers. For example, we refine a guard $m \in IK$ modeling a message reception in the protocol model by $bs \in ibuf \wedge \alpha(bs) = m$ in the interface model. The roles' events have several parameters, including the run id, the participants' names, and the long-term key bitstrings, which later become program parameters.

Finally, we decompose the interface model into an environment and (an unbounded number of) initiator and responder components. From these, we derive the initiator's and the responder's I/O specifications. Our framework provides lemmas to support the corresponding proofs.

4.2.4 Code Level. We implement each component type (protocol role) in both Java and Python. Each role's implementation sends and receives one message and checks that received messages have the correct form. Our implementation provides a (trusted) parsing and marshalling function for each type of abstract message (e.g., signatures, identifiers, pairs), each specified by a contract relating the message to its bitstring representation using α. This yields a partial definition of α, which we otherwise leave abstract to avoid modeling bit-level cryptographic operations. Since α relates

```
def verify(signed_msg: bytes, key: bytes) -> bytes:
    #@ PRE: alpha(key) == Key(publicKey(?a)))
    #@ POST: alpha(signed_msg) == Crypt(Key(privateKey(a))), alpha(result))
    #@        && len(signed_msg) == len(result) + 1 + SIGNATURE_SIZE
    #@ EXCEPTIONAL POST: forall msg. alpha(signed_msg) != Crypt(Key(privateKey(a)), alpha(msg))
    if len(signed_msg) == 0 or signed_msg[0] != SIGNATURE_TAG:
        raise InvalidDataException("Invalid tag on signature")
    return nacl.verify(signed_msg[1:])    # raises exception if signature is not valid
```

Listing 3. Simplified pseudocode implementation of a function for verifying signatures and its (trusted) specification. The pre- and postconditions relate the bitstrings to their abstract message representations that are used in the I/O specification. Variable *a* is implicitly existentially quantified (denoted by a question mark).

each bitstring to a unique message term (or Junk), we ensure that every bitstring representing a non-junk message can be parsed unambiguously by tagging bitstring messages with their type. We employ widely-used cryptographic libraries: PyNaCl for Python and Java's standard library.

Listing 3 shows the contract and implementation of the signature verification function. It checks the signature's tag and then calls the cryptographic library's signature verification function, which either returns the corresponding plaintext message or raises an exception for invalid signatures. The contract requires that the key bitstring represents some agent's public key and guarantees that the function terminates normally iff the input bitstring was signed with that agent's private key.

We use VeriFast and Nagini to prove the implementations correct. As an overall result, we obtain a proof that the entire system satisfies the intended authentication property.

5 FROM EVENT SYSTEMS TO I/O SPECIFICATIONS IN SEPARATION LOGIC

A central aspect of our methodology is to soundly link protocol models and code. We use I/O specifications to bridge the substantial semantic gap between the component models that result from the interface model decomposition step and the code. The component models are given as event systems, whereas I/O specifications are separation logic formulas built over I/O permissions, possibly employing co-recursive predicates to specify non-terminating behavior. What event systems and I/O specifications share is a trace semantics. We therefore define a translation of the components' event systems into I/O specifications and prove that they are trace-equivalent by establishing a simulation in each direction. It is this trace equivalence that, together with the verifier assumption (VA) and the compositional refinement theorem (Corollary 2.4), enables the seamless switching from models to code verification in our methodology (cf. Section 3.9).

Instead of translating component models directly into I/O specifications, we will pass through an intermediate translation to a sequential process calculus. This intermediate step has several benefits. First, it shifts the focus from data (guards and state updates) to I/O interactions. Second, it introduces a minimal syntax for these interactions, providing a useful structure for the correctness proofs. Third, it builds a bridge between two popular specification formalisms: process calculi on the modeling level and permission-based I/O specifications in separation logic on the code level.

The main challenge in proving our result stems from the disparate semantics of event systems and processes on one hand and I/O specifications on the other hand. Concretely, we must show that a process P can simulate the traces induced by its corresponding assertion ϕ_P. As we shall see, an assertion's behavior is the intersection of all its models' behaviors. This is challenging as some models of ϕ_P exhibit spurious behavior not present in P and also due to the absence of a single model representing exactly the behavior of ϕ_P.

$$\frac{w \in Ty(bio, v)}{\{token(t), bio(t, v, w, t')\}^{\#} +^{\#} h \xrightarrow{bio(v,w)}_{\mathcal{H}} \{token(t')\}^{\#} +^{\#} h}\ \text{Bio}$$

$$\frac{w \neq w' \quad w, w' \in Ty(bio, v)}{\{token(t), bio(t, v, w, t')\}^{\#} +^{\#} h \xrightarrow{bio(v,w')}_{\mathcal{H}} \bot}\ \text{Contradict} \qquad \frac{w \in Ty(bio, v)}{\bot \xrightarrow{bio(v,w)}_{\mathcal{H}} \bot}\ \text{Chaos}$$

Fig. 4. Heap transition rules.

5.1 Background: Semantics of I/O Specifications for Separation Logic

We slightly extend the semantics of the I/O specifications of Penninckx et al. [2015] by enforcing a typing discipline on inputs by using a *typing* function $Ty : Bios \times Values \to (\mathbb{P}(Values) \setminus \{\emptyset\})$, which assigns to each I/O operation and output value a type, given as a non-empty set of accepted inputs. An I/O permission $bio(t, v, w, t')$ and its input value w are *well-typed* if $w \in Ty(bio, v)$, and a chunk is well-typed if it is a well-typed I/O permission or a token. The typing function specifies a relational contract for each I/O operation. The set $Ty(bio, v)$ typically captures the possible results of an I/O operation, which is useful to match I/O operations to I/O library functions.

Assertion Semantics. The formal semantics of our assertions is co-inductively defined over *heaps* $h \in Heap$, where $Heap = Chunks^{\#}$ is the set of multisets of chunks (see Section 2.2), as follows.

$$\begin{array}{lcl}
h \models b & \iff & b = \text{true} \\
h \models c & \iff & c \in^{\#} h \ \wedge\ c \text{ is well-typed} \\
h \models \phi_1 \star \phi_2 & \iff & \exists h_1, h_2 \in Heap.\, h = h_1 +^{\#} h_2 \ \wedge\ h_1 \models \phi_1 \ \wedge\ h_2 \models \phi_2 \\
h \models \exists v.\, \phi & \iff & \exists v' \in Values.\, h \models \phi[v'/v] \\
h \models \exists t.\, \phi & \iff & \exists t' \in Places.\, h \models \phi[t'/t]
\end{array}$$

Note that a *heap* here is different from a program's heap memory; its chunks represent permissions to perform I/O operations or tokens, not memory locations and their values. Here, we elide the ordinary program heap for simplicity and since it is orthogonal to modeling I/O behavior.

The semantics of assertions satisfies the following monotonicity property.

Lemma 5.1 (Monotonicity). *If $h \models \phi$ then $g +^{\#} h \models \phi$.*

Example 5.2. Consider the I/O specification $\phi = token(t) \star (\exists x, t', t''.\, recv(t, x, t') \star send(t', 2x, t''))$ from Example 2.5. Examples of heaps that satisfy ϕ are $h_1 = \{token(t), recv(t, 12, t_1), send(t_1, 24, t_2)\}^{\#}$, $h_2 = \{token(t), recv(t, 12, t), send(t, 24, t)\}^{\#}$, and $h_3 = h_1 +^{\#} \{send(t_1, 35, t_2)\}^{\#}$. More generally, all heaps satisfying ϕ have the form $H_\phi(x, t', t'', h) = \{token(t), recv(t, x, t'), send(t', 2x, t'')\}^{\#} +^{\#} h$ for some value x, places t' and t'', and heap h. We will return to this example below.

Heap Transitions. Heaps have a transition semantics, where I/O permissions are consumed by pushing a token through them. This semantics is given by the event system $\mathcal{H} = (Heap_\bot, Act, \to_{\mathcal{H}})$ with the set of states $Heap_\bot$ and the set of events $Act = \{bio(v, w) \mid bio \in Bios \wedge v, w \in Values\}$, called *I/O actions*. Note that *bio* is overloaded, with the 2-argument version yielding trace events and the 4-argument one defining a chunk. An I/O action $bio(v, w)$ is *well-typed* if $w \in Ty(bio, v)$ and a trace $\tau \in Act^*$ is well-typed if all its events are.

The transition relation $\to_{\mathcal{H}}$ of $\mathcal{H}$ is defined by the rules in Figure 4 and mostly matches the place-I/O-permission multigraph intuition given in Section 2.2. The rule Bio corresponds to a normal heap transition executing an I/O operation. The input read is well-typed. The token moves

to the I/O permission's target place and the permission is consumed and removed from the heap. The rule Contradict describes the situation where a transition $bio(v, w)$ would be possible, but the environment provides an input $w' \neq w$ that is different from the one predicted by the chunk. In this case, the heap can perform a transition $bio(v, w')$ to the special state $\bot$. In this state, arbitrary (well-typed) behavior is possible by the rule Chaos. Hence, all traces of $\mathcal{H}$ are well-typed. For a set of heaps H, we define the set of traces of H to contain the traces executable in *all* heaps of H, i.e.,

$$traces^{\mathcal{H}}(H) = \{\tau \mid \forall h \in H.\, \tau \in traces(\mathcal{H}, h)\}.$$

The set of traces of an assertion ϕ is then defined to be the set of traces of its heap models, i.e.,

$$traces^{\mathcal{H}}(\phi) = traces^{\mathcal{H}}(\{h \mid h \models \phi\}).$$

This universal quantification over all heap models of an assertion constitutes the main challenge in our soundness proof (Theorem 5.7). Let us now look at an example illustrating these definitions.

Example 5.3 (Heap and assertion traces). Consider the heap models h_1, h_2, and h_3 of the I/O specification ϕ from Example 5.2. First focusing on *regular behaviors*, i.e., ignoring the rules Contradict and Chaos, their traces are given by the following sets, where $\downarrow$ denotes prefix closure:

- $traces(\mathcal{H}, h_1) = \{recv(12) \cdot send(24)\}\downarrow$,
- $traces(\mathcal{H}, h_2) = traces(\mathcal{H}, h_1) \cup \{send(24) \cdot recv(12)\}\downarrow$, and
- $traces(\mathcal{H}, h_3) = traces(\mathcal{H}, h_1) \cup \{recv(12) \cdot send(35)\}\downarrow$.

The first heap, h_1, exhibits an instance of the expected behavior: receive a value and send the doubled value. The heaps h_2 and h_3, however, also allow unintended behaviors. Heap h_2 has a trace where receive and send are inverted. This comes from the semantics of existential quantification, which does not ensure that the places are distinct. Heap h_3 can send a value different from the doubled input value, which is possible due to the monotonicity property in Lemma 5.1. Due to these additional behaviors, which we call *spurious*, the set $traces^{\mathcal{H}}(\psi)$ of traces of an I/O specification ψ is defined to contain those traces that are possible in *all* heap models of ψ. The three heaps above only share the traces of h_1, which corresponds to the intended behavior.

Note that these spurious behaviors are not an artifact of the particular formalism we use, but a standard part of the permission-based specification style of separation logics in general. For example, all program heaps satisfying a standard points-to assertion $x \mapsto e$ allow the program to dereference the pointer x, but some heaps may also allow dereferencing the pointer z because z and x happen to alias in a particular interpretation (analogous to "aliasing" places in h_2), or, for logics with monotonicity, may contain (and therefore allow access to) extra memory pointed to by y. However, like in our case, the program logic must not allow dereferencing y or z because it is not possible in *all* program heaps satisfying the assertion.

The rules Contradict and Chaos add, for any regular trace of the form $\tau_1 \cdot recv(w) \cdot \tau_2$, a spurious traces of the form $\tau_1 \cdot recv(w') \cdot \tau$ for each well-typed $w' \neq w$ and well-typed trace τ. These rules formalize that a heap reading some (well-typed) input different from the one predicted by the I/O permission may behave arbitrarily. For example, both $h_1' = \{token(t), recv(t, 19, t_1), send(t_1, 38, t_2)\}^{\#}$ and h_1 are models of ϕ and ϵ is their only shared regular trace. However, the regular traces of h_1' are also spurious traces of h_1 and vice versa. Hence, $traces^{\mathcal{H}}(\{h_1, h_1'\})$ consists of the regular traces of h_1 and h_1'. This ensures that $traces^{\mathcal{H}}(\phi) = \{recv(x) \cdot send(2x) \mid x \in Values\}\downarrow$ is the trace property intended by the assertion ϕ. Without these two rules, we would have $traces^{\mathcal{H}}(\phi) = \{\epsilon\}$.

5.2 Embedding I/O-guarded Event Systems into Processes

We co-inductively define a simple language of (sequential) processes:

$$P ::=_{\nu} \mathsf{Null} \mid bio(v, z).P \mid P_1 \oplus P_2.$$

Here, Null is the inactive process, $bio(v, z).P$ is prefixing with an I/O operation, which binds the input variable z in P, and $P_1 \oplus P_2$ is a binary choice operator. Let *Proc* be the set of all processes.

We can then co-recursively define processes. For example, we define a countable choice operator $\bigoplus_{v \in S} P(v)$ over a set of values S with Null as the neutral element, analogous to the definition of the iterated separating conjunction. We can also co-recursively define non-terminating processes.

Example 5.4. A process corresponding to the I/O specification from Example 2.6 is specified by $RSP(0)$, where $RSP(a) =_\nu recv(z).$ if $z > 0$ then $send(a + z).RSP(a + z)$ else Null.

The operational semantics of processes is given by the event system $\mathcal{P} = (Proc, Act, \rightarrow_{\mathcal{P}})$, where the transition relation $\rightarrow_{\mathcal{P}}$ is inductively defined by the following rules:

$$\frac{w \in Ty(bio, v)}{bio(v, z).P \xrightarrow{bio(v,w)}_{\mathcal{P}} P[w/z]}\ \text{Pref} \qquad \frac{P_1 \xrightarrow{a}_{\mathcal{P}} P_1'}{P_1 \oplus P_2 \xrightarrow{a}_{\mathcal{P}} P_1'}\ \text{Choice}_1 \qquad \frac{P_2 \xrightarrow{a}_{\mathcal{P}} P_2'}{P_1 \oplus P_2 \xrightarrow{a}_{\mathcal{P}} P_2'}\ \text{Choice}_2$$

We write $traces(P)$ as a shorthand for $traces(\mathcal{P}, P)$.

Translation. We define a translation from I/O-guarded event systems $\mathcal{G} = (S, Act, G, U)$ to processes. The process $proc(\mathcal{G}, s)$ represents $\mathcal{G}$ in state s and is co-recursively defined by

$$proc(\mathcal{G}, s) =_\nu \bigoplus_{bio \in Bios} \bigoplus_{v \in Values} \text{if } G_{(bio,v)}(s) \text{ then } bio(v, z).\, proc(\mathcal{G}, U_{bio(v,z)}(s)) \text{ else Null.}$$

Recall that here we borrow the conditional from our meta-language HOL. The following correctness result is established by a simulation in each direction.

Theorem 5.5 (Correctness of event system translation). *For any I/O-guarded event system $\mathcal{G} = (S, Act, G, U)$ and state $s \in S$, we have $traces(\mathcal{G}, s) = traces(proc(\mathcal{G}, s))$.*

5.3 Embedding Processes into I/O Specifications

We now co-recursively define the embedding *emb* from processes and places into I/O specifications:

$$\begin{aligned} emb(\text{Null}, t) &=_\nu \text{true} \\ emb(bio(v, z).P, t) &=_\nu \exists t', z'.\, bio(t, v, z', t') \star emb(P[z'/z], t') \\ emb(P_1 \oplus P_2, t) &=_\nu emb(P_1, t) \star emb(P_2, t). \end{aligned}$$

We define the *process assertion* of P by $emb(P) = \exists t.\, token(t) \star emb(P, t)$. We then prove by co-induction that countable choice translates to iterated separating conjunction.

Lemma 5.6. $emb(\bigoplus_{v \in S} P(v), t) = \forall^\star v \in S.\, emb(P(v), t)$.

We now turn to our main result, namely, the trace equivalence of process P and its I/O specification $emb(P)$. We focus on the intuition here and defer the formal details to Appendix A.

Theorem 5.7 (Correctness of process translation). $traces(P) = traces^{\mathcal{H}}(emb(P))$.

The proof follows from Propositions 5.8, 5.12, and 5.13 to which the remainder of Section 5 is devoted. Together with Theorem 5.5, this result allows us to translate any I/O-guarded event system $\mathcal{E}$ modeling some component of our system into an I/O specification $\phi_{\mathcal{E}} = emb(proc(\mathcal{E}))$ with identical behavior. We can then use $\phi_{\mathcal{E}}$ as a specification for the code implementing $\mathcal{E}$'s behavior.

The left-to-right trace inclusion of this theorem is captured by the following proposition, which we prove by a simulation between process P and heap models of $emb(P)$ (see Appendix A.1).

Proposition 5.8. $traces(P) \subseteq traces^{\mathcal{H}}(emb(P))$.

The main difficulty lies in the proof of the reverse set inclusion and stems from the meaning of $traces^{\mathcal{H}}(emb(P))$, which contains exactly those traces τ that are a trace of *all* models of $emb(P)$. From Example 5.3, we know that many models of $emb(P)$ (or of any assertion ϕ for that matter) exhibit spurious behaviors that are not in $traces^{\mathcal{H}}(emb(P))$ (or in $traces^{\mathcal{H}}(\phi)$, respectively). Therefore, picking an arbitrary heap model of $emb(P)$ and trying to simulate its transitions by the process P's transitions will fail. Instead, we restrict our attention to *canonical* models that do not exhibit spurious behaviors. We denote by $can(P)$ the set of all canonical models of P (introduced in Section 5.4). We then decompose the proof into the following chain of set inclusions:

$$traces^{\mathcal{H}}(emb(P)) \subseteq traces^{\mathcal{H}}(can(P)) \subseteq traces(P). \tag{2}$$

The first inclusion expresses that the canonical models cover all behaviors of $emb(P)$. We will establish the second inclusion by simulating the behavior of canonical models by process transitions.

5.4 Canonical Heap Models for Processes

A natural canonical model candidate for a process P would be the heap h_P that is isomorphic to P's computation tree, where a process $bio(v, w).Q$ would result in one I/O permission $bio(t, v, w, t_w)$ for each input w on the heap. Although this proposal avoids spurious behaviors due to additional permissions and place identifications (cf. Example 5.3), it fails as the following example shows.

Example 5.9 (Failed attempt). Let $P = in(x).out(x).\mathsf{Null}$, $Values = \mathbb{B}$, and $Places = \{\mathsf{L}, \mathsf{R}\}^*$ (for tree positions). Then h_P contains both I/O permissions $in(\epsilon, \mathsf{false}, \mathsf{L})$ and $in(\epsilon, \mathsf{true}, \mathsf{R})$. This would lead to $traces^{\mathcal{H}}(can(P)) = traces(h_P) = \{\epsilon\} \cup \{in(v) \cdot \tau \mid v \in \mathbb{B}, \tau \in Act^*\}$ according to the rules Contradict and Chaos and hence to $traces^{\mathcal{H}}(can(P)) \supset traces(P)$.

We will therefore construct the canonical heap models of a process P with respect to an input schedule, which is essentially a prophecy variable that uniquely determines the inputs read by the process. An *input schedule* is a function $\rho : Act^* \times Bios \times Values \to Values$ mapping an I/O trace τ, an I/O operation bio, and an output value v to an input value $\rho(\tau, bio, v)$. Hence, there will be a canonical model $cmod(P, \rho)$ for each input schedule ρ, which intuitively corresponds to the projection of P's computation tree to the inputs prescribed by ρ. The set $can(P)$ contains such a model for each input schedule ρ. Our construction uses the set of places $Places = \{\mathsf{L}, \mathsf{R}\}^*$, i.e., the places are positions of a binary tree. The inputs being fixed, the only branching stems from the choice operator. The following example illustrates our construction. We defer its formal definition and the proofs of the corresponding results to Appendix A.2.

Example 5.10 (Canonical model). Consider the process P defined by

$$P = in(x).Q(x) \oplus fail.\mathsf{Null} \qquad Q(x) = out(x).\mathsf{Null} \oplus (in(y).out(x + y).\mathsf{Null} \oplus drop.\mathsf{Null}).$$

For simplicity, the I/O operations *drop* and *fail* have no arguments. Let ρ be the input schedule defined by $\rho(\tau, bio, v) = len(\tau) + 1$. Figure 5 (left) shows the projection of P's syntax tree to the input schedule ρ. Edges arising from action prefixes are labeled with the corresponding action. Each node is annotated with its current position $cpos = ppos \cdot x$, which is composed of $ppos$, the target position of the previous action-labeled edge in the tree (or ϵ if there is none), and a rest x. Each edge labeled by some action $bio(v, w)$ and connecting position $cpos = ppos \cdot x$ to $cpos \cdot \mathsf{L}$ translates into an I/O permission $bio(ppos, v, w, cpos \cdot \mathsf{L})$ in the resulting canonical heap $cmod(P, \rho)$, which is shown in Figure 5 (right).

The following result states that the canonical model for a process P and a schedule ρ is indeed a model of the assertion corresponding to the process P. The first inclusion in (2) then easily follows.

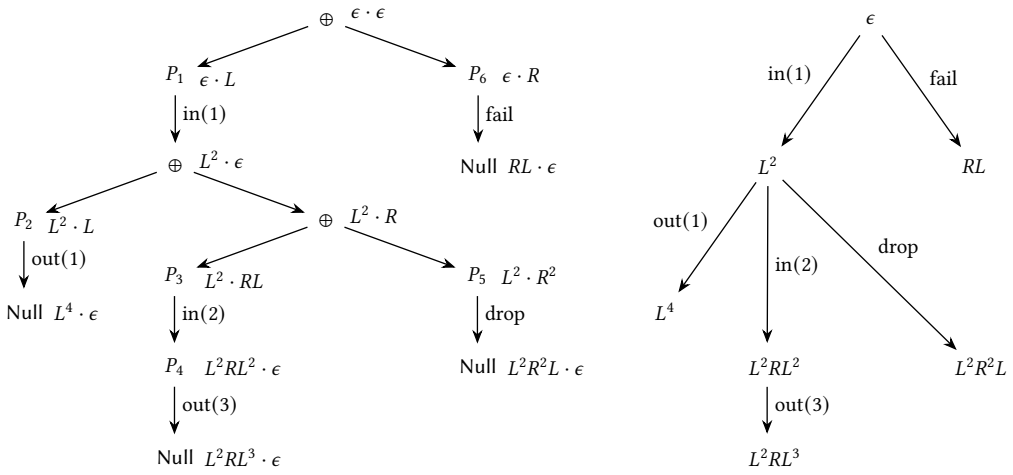

Fig. 5. Process P and the schedule ρ of Example 5.10 (left) and resulting canonical model (right). Each process (node in the left graph) is annotated with its position $cpos = ppos \cdot x$.

Proposition 5.11 (Canonical model property). *$cmod(P, \rho) \models emb(P, \epsilon)$ for all processes P and well-typed schedules ρ.*

Proposition 5.12. *$traces^{\mathcal{H}}(emb(P)) \subseteq traces^{\mathcal{H}}(can(P))$.*

5.5 Processes Simulate Canonical Models

We now turn to the second trace inclusion in (2): each trace of the canonical model set $can(P)$ is also a trace of P. Writing $cmod^{\mathrm{t}}(P, \rho)$ for the canonical model $cmod(P, \rho)$ with a token added at its root place, we would like transitions of the heap $cmod^{\mathrm{t}}(P, \rho)$ to lead to a heap $cmod^{\mathrm{t}}(P', \rho)$ for some process P', so we can simulate it with the corresponding process transition from P to P'.

There are two obstacles to this plan: (1) dead heap parts, which correspond to untaken choices in processes $P \oplus Q$ and cannot perform any transitions, and (2) chaotic transitions where, given a trace of the set of canonical models $can(P)$, some of the models $cmod^{\mathrm{t}}(P, \rho)$ in $can(P)$ transit to the "chaotic" state $\bot$ at some point along the trace. The problem here is that a given process cannot in general simulate the (arbitrary) I/O actions that are possible in the state $\bot$.

Our proofs must take such dead heap parts into account to address problem (1) and carefully pick a particular schedule to avoid problem (2). Here, we focus on problem (2) from an intuitive perspective (see Appendix A.3 for a more precise and detailed account). Its solution is based on the observation that executing some I/O action $bio(v, w_\rho)$ with *scheduled input* $w_\rho = \rho(\tau, bio, v)$ from $cmod^{\mathrm{t}}(P, \rho)$ indeed leads to a heap $cmod^{\mathrm{t}}(P', \rho)$ for some process P' (and, in particular, not to $\bot$). Hence, to simulate a given trace τ of the heap $cmod^{\mathrm{t}}(P, \rho)$ by transitions of the process P, we must ensure that the schedule ρ is consistent with the trace τ. We therefore define a witness schedule $\rho_{wit}(\tau)$, which returns the inputs appearing on the trace τ and has the property:

$$cmod^{\mathrm{t}}(P, \rho_{wit}(\tau)) \xrightarrow{\tau} h = cmod^{\mathrm{t}}(P', \rho_{wit}(\tau)) \tag{3}$$

for some process P', i.e., in particular, $h \neq \bot$. The final trace inclusion in Equation (2) then follows immediately, since any trace $\tau \in traces^{\mathcal{H}}(can(P))$ is also a trace of $cmod^{\mathrm{t}}(P, \rho_{wit}(\tau))$.

Proposition 5.13. *$traces^{\mathcal{H}}(can(P)) \subseteq traces(P)$.*

6 RELATED WORK

Numerous formalisms have been developed for modeling and verifying systems. In the following, we focus on those approaches that combine models and code, and target distributed systems.

Model Verification with Code Extraction. Various approaches verify models of distributed systems in formalisms that support the extraction of executable code. The following four approaches are all embedded in Coq and support the extraction of OCaml programs.

In Verdi [Wilcox et al. 2015; Woos et al. 2016], a system is specified by defining types and handlers for external I/O and for network messages. The developer can focus on the application and its correctness proof by essentially assuming a failure-free environment. These assumptions can be relaxed by applying Verdi's verified system transformers to make the application robust with respect to communication failures or node crash failures. DISEL [Sergey et al. 2018] offers a domain-specific language for defining protocols in terms of their invariants and atomic I/O primitives. It enables the modular verification of programs that participate in different protocols, using separation logic to represent protocol state separation. Component programs are verified in the context of one or more protocol models using a Hoare logic embedded in a dependent type theory. The program verification can be understood as a single refinement step. Velisarios [Rahli et al. 2018] is a framework for verifying Byzantine fault-tolerant state-machine replication protocols in Coq based on a logic of events. It models systems as deterministic state machines and provides an infrastructure for modeling and reasoning about distributed knowledge and quorum systems. Chapar [Lesani et al. 2016] is a formal framework in Coq for the verification of causal consistency for replicated key-value stores. The technique uses an abstract operational semantics that defines all the causally-consistent executions of a client of the store. The implementation of the store is verified by proving that its concrete operational semantics refines this abstract semantics. Liu et al. [2020] model distributed systems in Maude's rewriting logic [Clavel et al. 2007]. These are compiled into distributed implementations using mediator objects for the TCP communication. They prove that the generated implementation is stuttering equivalent to the original model, hence preserving next-free CTL* properties. The implementation runs in distributed Maude sessions.

All of these approaches enable the development of distributed systems that are correct by construction. However, code extraction has three major drawbacks. First, the produced code is either purely functional or based on rewriting logic, which precludes common optimizations (e.g., using mutable heap data structures). Second, it is difficult for extracted code to interface existing software modules such as libraries; incorporating existing (possibly unverified) modules is often necessary in practice. Third, the approaches prescribe a fixed implementation language; however, it is often useful in practice to be able to combine components, such as clients and servers, written in different languages. Our approach avoids all three problems by supporting the bottom-up development and verification of efficient, flexible implementations.

PSync [Dragoi et al. 2016] is a domain-specific language for implementing round-based distributed, fault-tolerant systems. PSync programs are executed via an embedding into Scala. A dedicated verifier allows one to prove safety and liveness properties of PSync programs, and a refinement result shows that these carry over to the executable system. The focus of PSync is mostly on developing specific verified distributed *algorithms* rather than entire software systems.

Combinations of Model and Code Verification. The works most closely related to ours are those of Koh et al. [2019] and of Oortwijn and Huisman [2019]. The former work is part of DeepSpec [Pierce 2016], which is a research program with the goal of developing fully-verified software and hardware. The DeepSpec developments are based on the Verified Software Toolchain (VST) [Cao et al. 2018], a framework for verifying C programs via a separation logic embedded in Coq. Koh et al. [2019]

use *interaction trees* [Xia et al. 2020], which are similar to our processes, to specify a program's I/O behavior and directly embed these into VST's separation logic using a special predicate. In contrast, our embedding of processes into separation logic using the encoding of Penninckx et al. [2015] allows us to apply standard separation logic and existing program verifiers. In both their and our work, a successful program verification guarantees an inclusion of the program's I/O traces in those of the I/O specification or interaction tree. Koh et al. [2019] verify a simple networked server in a TCP networking environment, for which they use two interaction trees at different abstraction levels and relate them by a form of contextual refinement that establishes linearizability. Their paper leaves open the question whether their approach can be used to verify system-wide global properties of distributed systems with different types of components and operating in different environments (e.g., exhibiting faulty and adversarial behavior). For example, it is unclear whether they could verify our case study protocols. Oortwijn and Huisman [2019] use a process calculus for modeling, which they embed into a concurrent separation logic (CSL). Their approach relies on automated tools and combines the mCRL2 model checker with an encoding of CSL into Viper. The modeling-level expressiveness is limited by mCRL2 being a finite-state model checker. Moreover, while the soundness of CSL implies the preservation of state assertions from modeling to implementation level, it is unclear whether arbitrary trace properties are preserved.

IronFleet [Hawblitzel et al. 2015] combines TLA-style refinement with code verification. Abstract models as well as the implementation are expressed in Dafny [Leino 2010]. Dafny is a powerful verification framework that supports, among other features, mutable heap data structures, inductive and coinductive data types, and proof authoring. Reasoning is supported by an SMT solver, which is restricted to first-order logic. Dafny enables different kinds of higher-order reasoning by encoding it into first-order logic internally, but nevertheless has some restrictions both in expressivity and practicality for larger proofs when compared to native higher-order theorem provers. By using Isabelle/HOL as modeling language, our approach provides the full expressiveness of higher-order logic, which also allows us to formalize our meta-theory. By using a single framework, Ironfleet avoids the problems we had to solve when linking abstract models to separation logic specifications. However, it lacks the flexibility to support different logics or modeling languages. Dafny currently compiles to sequential C#, Go, and JavaScript, while existing separation logic based verifiers support concurrent implementations and allow developers to write the code directly in familiar programming languages rather than in Dafny. IronFleet supports both safety and liveness properties, whereas our approach focuses on safety properties and leaves liveness as future work.

Project Everest [Bhargavan et al. 2017] uses an approach similar to IronFleet to develop a verified implementation of TLS. An abstract implementation is developed and verified in Low* [Protzenko et al. 2017], a subset of F* [Swamy et al. 2016] geared toward imperative C-like code that is compiled to C. A main focus of this project is on verifying cryptographic algorithms. Like IronFleet, Low* verification uses an SMT solver and the extracted C code is sequential.

7 CONCLUSIONS AND FUTURE WORK

We proposed a novel approach for the formal development of distributed systems. Our approach combines the top-down development of system models via compositional refinement with bottom-up program verification. This supports a clean separation of concerns and simplifies the individual verification tasks, which is crucial for managing the additional complexity arising in systems operating in faulty or adversarial environments. For program verification, we support state-of-the-art separation logics, which support mutable heap data structures, concurrency, and other features needed to develop efficient, maintainable code. We demonstrated that our approach bridges the gap between abstract models and concrete code, both through the theoretical foundations underpinning

its soundness and with three complete case studies. The theory and case studies are mechanized in Isabelle/HOL and the Nagini and VeriFast program verifiers.

As future work, we plan to reduce the need for boilerplate Isabelle code by automating the translation of interface models into the components' I/O specifications that are input to the code verifiers. We also plan to support liveness properties, which will require a more complex refinement framework in the style of TLA [Lamport 1994], including support for fairness notions. Finally, we are currently applying our approach to verify substantial parts of the SCION secure Internet architecture [Perrig et al. 2017]. We show protocol-level global security properties in the Dolev-Yao symbolic attacker model and verify the I/O behavior (as well as memory safety, secure information flow, and other properties) of the currently deployed implementation of SCION routers.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press.

Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings.* 2. https://doi.org/10.1007/978-3-642-28891-3_2

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 1:1–1:12.

Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. 2017. Friends with Benefits - Implementing Corecursion in Foundational Proof Assistants. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 111–140. https://doi.org/10.1007/978-3-662-54434-1_5

Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer. I–XIX, 1–367 pages.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reasoning* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Ernest J. H. Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (1979), 281–283. https://doi.org/10.1145/359104.359108

Bernadette Charron-Bost, Fernando Pedone, and André Schiper. 2010. Replication. *LNCS* 5959 (2010), 19–40.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic.* Lecture Notes in Computer Science, Vol. 4350. Springer. https://doi.org/10.1007/978-3-540-71999-1

Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. An Overview of the mCRL2 Toolset and Its Recent Advances. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 199–213.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Information Theory* 29, 2 (1983), 198–207. https://doi.org/10.1109/TIT.1983.1056650

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 400–415. https://doi.org/10.1145/2837614.2837650

Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 596–603. https://doi.org/10.1007/978-3-319-96145-3_33

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Operating Systems Design and Implementation (OSDI)*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 165–181.

Tony Hoare. 2003. The Verifying Compiler: A Grand Challange for Computing Research. In *Modular Programming Languages, Joint Modular Languages Conference, (JMLC) (Lecture Notes in Computer Science, Vol. 2789)*, László Böszörményi and Peter Schojer (Eds.). Springer, 25–35.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP)*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923. https://doi.org/10.1145/177492.177726

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370.

K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-Threaded Programs. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 5502)*, G. Castagna (Ed.). Springer, 378–393.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 42–54.

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 357–370. https://doi.org/10.1145/2837614.2837622

Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. 2020. Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs. In *NFM 2020 (LNCS)*. Springer.

Gavin Lowe. 1997. A Hierarchy of Authentication Specification. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society, 31–44. https://doi.org/10.1109/CSFW.1997.596782

Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

William Mansky, Wolf Honoré, and Andrew W. Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 428–455. https://doi.org/10.1007/978-3-030-44914-8_16

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

Wytse Oortwijn and Marieke Huisman. 2019. Practical Abstractions for Automated Verification of Message Passing Concurrency. In *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11918)*, Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa (Eds.). Springer, 399–417. https://doi.org/10.1007/978-3-030-34968-4_22

Matthew Parkinson and Gavin Bierman. 2005. Separation Logic and Abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. ACM, New York, NY, USA, 247–258.

Larry Paulson. 1998. The inductive approach to verifying cryptographic protocols. *J. Computer Security* 6 (1998), 85–128. http://www.cl.cam.ac.uk/users/lcp/papers/Auth/jcs.pdf

Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 158–182. https://doi.org/10.1007/978-3-662-46669-8_7

Adrian Perrig, Pawel Szalachowski, Raphael M. Reischuk, and Laurent Chuat. 2017. *SCION: A Secure Internet Architecture*. Springer. https://doi.org/10.1007/978-3-319-67080-5

Benjamin C. Pierce. 2016. The science of deep specification (keynote). In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, Eelco Visser (Ed.). ACM, 1. https://doi.org/10.1145/2984043.2998388

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *PACMPL* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 619–650. https://doi.org/10.1007/978-3-319-89884-1_22

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. IEEE Computer Society Press.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *PACMPL* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

Renato Silva and Michael J. Butler. 2010. Shared Event Composition/Decomposition in Event-B. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers (Lecture Notes in Computer Science, Vol. 6957)*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). Springer, 122–141. https://doi.org/10.1007/978-3-642-25271-6_7

Christoph Sprenger and David A. Basin. 2018. Refining security protocols. *Journal of Computer Security* 26, 1 (2018), 71–120. https://doi.org/10.3233/JCS-16814

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages (POPL)*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 154–165. https://doi.org/10.1145/2854065.2854081

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

A THEORY DETAILS

This section provides details on the main part of our soundness theorem, the equivalence of processes and their I/O specifications (Theorem 5.7).

$$traces(P) = traces^{\mathcal{H}}(emb(P)).$$

Recall that we prove this theorem using a series of trace inclusions (Propositions 5.8, 5.12, and 5.13).

$$traces(P) \subseteq traces^{\mathcal{H}}(emb(P)) \subseteq traces^{\mathcal{H}}(can(P)) \subseteq traces(P).$$

A.1 Formal Definitions and Proofs for Section 5.3

A.1.1 Process Traces Are Process Assertion Traces.

Proposition A.1. $traces(P) \subseteq traces^{\mathcal{H}}(emb(P))$.

Proof. It suffices to prove that $traces(P) \subseteq traces(ho)$ for any P and ho in the simulation relation

$$R(P, ho) = (\exists t, h.\, ho = \{token(t)\}^{\#} +^{\#} h \wedge h \models emb(P, t)) \vee ho = \bot.$$

The proof proceeds by establishing a simulation between P and ho using this relation. If P is related to a heap h (first disjunct in R) then a given transition of P can either be simulated by a transition to another heap h' (using rule Bio) or to $\bot$ (using rule Contradict). In each case, the resulting states are again related by R. We prove this by induction on the operational semantics of processes. Otherwise, if P is related to $\bot$, then any of P's transitions can be simulated using rule Chaos. □

A.2 Formal Definitions and Proofs for Section 5.4

A.2.1 Formal Definition of Canonical Model $cmod(P, \rho)$. Recall from Example 5.3 that there are two sources of spurious behaviors: (a) unintended control flow stemming from the identification of places and (b) extra permissions not explicitly described by the assertion. Also recall that non-unique inputs in a heap allow arbitrary subsequent behavior through Contradict and Chaos rules. We will therefore construct our canonical models of a process P with respect to an input schedule ρ, which uniquely determines the inputs read by the process. These observations lead us to the construction of a canonical (heap) model $cmod(P, \rho)$ for each input schedule ρ. The set $can(P)$ contains such a model for each input schedule ρ. The construction of $cmod(P, \rho)$ satisfies the following properties:

- $cmod(P, \rho) \models emb(P, t_\epsilon)$, i.e., canonical models are indeed models of $emb(P, t_\epsilon)$, where t_ϵ is the distinguished starting place of $cmod(P, \rho)$.
- A token never returns to the same place: the I/O permissions of $cmod(P, \rho)$ induce a tree on places where each $bio(t, v, w, t')$ gives rise to an edge from t to t'; this solves problem (a).
- $cmod(P, \rho)$ does not contain any extra permissions, i.e., every proper sub-multiset of $cmod(P, \rho)$ fails to satisfy $emb(P, t_\epsilon)$. This addresses problem (b). We do not explicitly prove this property, but some of our trace inclusion proofs implicitly rely on it.

Intuitively, we construct a canonical heap model, $cmod(P, \rho)$, given a process P and an input schedule ρ by transforming the (syntactic) tree of P for the input schedule ρ to a corresponding heap model.

An *input schedule* is a function $\rho : Act^* \times Bios \times Values \to Values$ mapping an I/O trace τ, an I/O operation bio, and an output value v to an input value $\rho(\tau, bio, v)$. An input schedule ρ is well-typed, written $welltyped(\rho)$, if $\rho(\tau, bio, v) \in Ty(bio, v)$ for all τ, bio, and v. We use the set of positions in a binary tree as our set of places $Places = \{\mathsf{L}, \mathsf{R}\}^*$.

$$
\begin{aligned}
pm(bio(v,z).P, \rho, \tau, ppos, cpos, \epsilon) &= \{bio(ppos, v, \rho(\tau, bio, v), cpos \cdot \langle \mathsf{L} \rangle)\}^{\#} \\
pm(bio(v,z).P, \rho, \tau, ppos, cpos, \mathsf{L} \# pos) &= pm(P[w_\rho/z]), \rho, \tau \cdot \langle bio(v, w_\rho) \rangle, cpos \cdot \langle \mathsf{L} \rangle, cpos \cdot \langle \mathsf{L} \rangle, pos) \\
pm(P_1 \oplus \_, \rho, \tau, ppos, cpos, \mathsf{L} \# pos) &= pm(P_1, \rho, \tau, ppos, cpos \cdot \langle \mathsf{L} \rangle, pos) \\
pm(\_ \oplus P_2, \rho, \tau, ppos, cpos, \mathsf{R} \# pos) &= pm(P_2, \rho, \tau, ppos, cpos \cdot \langle \mathsf{R} \rangle, pos) \\
pm(\_, \_, \_, \_, \_, \_) &= \emptyset^{\#}, \text{otherwise.}
\end{aligned}
$$

Fig. 6. Function pm maps a process and a position to a singleton multiset containing an I/O permission or $\emptyset^{\#}$. We abbreviate $\rho(\tau, bio, v)$ as w_ρ and write $x \# xs$ for prefixing an element to a list (cons).

We then construct the canonical model $cmod(P, \rho)$ in two steps (see also Example 5.10):

(1) We define a recursive function pm, where $pm(P, \rho, \tau, ppos, cpos, pos)$ returns a singleton multiset containing an I/O permission $bio(t, v, w, t')$ corresponding to the I/O operation at position pos of process P under the input schedule ρ (if any, otherwise $\emptyset^{\#}$). Its starting place is given by $t = ppos$, where $ppos$ is the position $ppos$ of the last process appearing directly under an I/O operation prefix (initially ϵ). Its target place is $t' = cpos \cdot \langle L \rangle$, where $cpos$ is the current position in the original process (i.e., the path already traversed). The trace τ records the traversed I/O actions and is used to determine the scheduled input $w = \rho(\tau, bio, v)$. More precisely, for a prefix process $P = bio(v, z).P'$, pm behaves as follows. If $pos = \epsilon$, it returns the corresponding I/O permission $bio(t, v, w_\rho, t')$, where $w_\rho = \rho(\tau, bio, v)$ is the scheduled input, $t = ppos$ is the starting place and $t' = cpos \cdot \langle \mathsf{L} \rangle$ is the target place. If $pos = \mathsf{L} \# pos'$ then the prefix is "traversed", calling pm recursively with process $P[w_\rho/z]$ the updated trace $\tau \cdot \langle bio(v, w_\rho) \rangle$ and updated previous position $cpos \cdot \langle \mathsf{L} \rangle$. Otherwise, it returns $\emptyset^{\#}$. Choices are traversed recursively. Figure 6 shows the formal definition of the function pm, which we discuss below.

(2) We define the canonical heap model for a process P and input schedule ρ by $cmod(P, \rho) = gmod(P, \rho, \epsilon, \epsilon, \epsilon)$, i.e., as an instance of an auxiliary function $gmod(P, \rho, \tau, ppos, cpos)$, which is defined by collecting all I/O permissions generated by the function pm using the multiset sum over all positions pos:

$$gmod(P, \rho, \tau, ppos, cpos) = \sum_{pos}^{\#} pm(P, \rho, \tau, ppos, cpos, pos)$$

We also define some derived heaps, adding a token to a canonical model, indicated by a superscript, i.e., $gmod^{\mathsf{t}}(P, \rho, \tau, ppos, cpos) = \{token(ppos)\}^{\#} +^{\#} gmod(P, \rho, \tau, ppos, cpos)$ and $cmod^{\mathsf{t}}(P, \rho) = gmod^{\mathsf{t}}(P, \rho, \epsilon, \epsilon, \epsilon)$. Finally, we define the set $can(P)$ of canonical heap models of P by

$$can(P) = \{h \mid \exists \rho.\ welltyped(\rho) \wedge h = cmod^{\mathsf{t}}(P, \rho)\}.$$

Figure 6 shows the formal definition of the function pm. The first two equations defining pm cover the case of a prefixed process $bio(v, w).P$. In the first equation, the desired position is reached ($pos = \epsilon$) and a singleton multiset containing the corresponding I/O permission with source place $ppos$, target place $cpos \cdot \langle L \rangle$, and scheduled input $\rho(\tau, bio, v)$ is returned. In the second equation, the position has head L and the search continues in the process $P[w_\rho/z]$, where the scheduled input $w_\rho = \rho(\tau, bio, v)$ replaces the bound variable z, for the trace τ extended with the traversed I/O event $bio(v, w_\rho)$ and with the arguments $ppos$ and $cpos$ both set to $cpos \cdot \langle L \rangle$. The third and fourth equations recursively navigate into a choice process in the direction given by the position, updating $cpos$ but not $ppos$ in the recursive call. The final equation catches all cases not covered by

the previous equations and returns the empty multiset. Note that the concatenation $cpos \cdot pos$ is invariant throughout the recursive calls.

A.2.2 Canonical Model Property. The following lemma provides fixed-point equations for the canonical models, with one case per process form:

Lemma A.2 (Canonical model as fixed-point).

(1) $gmod(\mathsf{Null}, \rho, \tau, ppos, cpos) = \emptyset^{\#}$,
(2) $gmod(bio(v, z).P, \rho, \tau, ppos, cpos) =$
$\{bio(ppos, v, w_\rho, cpos')\}^{\#} +^{\#} gmod(P[w_\rho/z], \rho, \tau \cdot \langle bio(v, w_\rho)\rangle, cpos', cpos')$
where $w_\rho = \rho(\tau, bio, v)$ *and* $cpos' = cpos \cdot \langle \mathsf{L}\rangle$, *and*
(3) $gmod(P_1 \oplus P_2, \rho, \tau, ppos, cpos) =$
$gmod(P_1, \rho, \tau, ppos, cpos \cdot \langle \mathsf{L}\rangle) +^{\#} gmod(P_2, \rho, \tau, ppos, cpos \cdot \langle \mathsf{R}\rangle)$.

Proposition A.3 (Canonical model property). $cmod(P, \rho) \models emb(P, \epsilon)$ *for all processes* P *and well-typed schedules* ρ.

Proof. The lemma's statement follows from $gmod(P, \rho, \tau, ppos, cpos) \models emb(P, ppos)$, which we prove by coinduction using the relation X on heaps and formulas defined by

$$X(h, \phi) = \exists P, \tau, ppos, cpos.\, h = gmod(P, \rho, \tau, ppos, cpos) \wedge \phi = emb(P, ppos),$$

and a case analysis on the structure of P. The different cases are proved using the fixed point property of *gmod* stated in Lemma A.2. □

A.3 Formal Definitions and Proofs for Section 5.5

We now turn to the second trace inclusions of Equation (2), given on page 24. It states that each trace of the canonical model set $can(P)$ is also a trace of P. We would like heap transitions of the canonical model $cmod^{\mathrm{t}}(P, \rho)$ to lead to a heap $cmod^{\mathrm{t}}(P', \rho)$ for some process P', so that we can simulate it with the corresponding process transition from P to P'. Recall that there are two problems:

(1) Dead heap parts: Consider the process $P = Q \oplus bio(v, z).R$. The canonical model $cmod^{\mathrm{t}}(P, \rho)$ has a transition labeled $bio(v, w)$ to $gmod^{\mathrm{t}}(R[w/z], \rho, \langle bio(v, w)\rangle, cpos', cpos') +^{\#} g$ where $w = \rho(\epsilon, bio, v)$, $g = cmod^{\mathrm{t}}(Q, \rho)$, and $cpos' = \langle \mathsf{R}, \mathsf{L}\rangle$ with the resulting token at place $cpos'$ (see Lemma A.2). Since this token can subsequently only visit places in $\{pos \mid cpos' \leq pos\}$, this means that the g portion of the heap will never be able to make a transition. Our proof must take such *dead* heap parts into account.
(2) Chaotic transitions: Let τ be a trace of all canonical models $cmod^{\mathrm{t}}(P, \rho)$ (i.e., for all input schedules ρ). Some of these models transit to the "chaotic" state $\bot$ at some point along the trace. However, a given process cannot in general simulate the (arbitrary) I/O actions possible in that state. We will have to carefully pick a particular schedule to avoid this problem.

We address problem (1) by considering heaps with dead parts in our transition lemmas. Let $srcs(h)$ be the set of source places occurring in I/O permissions in the heap h. A heap h is called *dead* with respect to a position pos if h contains no tokens and $srcs(h) \cap \{pos' \mid pos \leq pos'\} = \emptyset$, meaning a token at position pos in a canonical model will never activate a transition in h.

The solution to problem (2) is based on the observation that executing some I/O action $bio(v, w_\rho)$ with *scheduled input* $w_\rho = \rho(\tau, bio, v)$ from $cmod^{\mathrm{t}}(P, \rho)$ indeed leads to a heap $cmod^{\mathrm{t}}(P', \rho)$ for some process P' (and, in particular, not to $\bot$). Hence, to simulate a given trace τ of the heap $cmod^{\mathrm{t}}(P, \rho)$ by transitions of the process P, we must ensure that the schedule ρ is consistent with

the trace τ. We therefore define a "witness" schedule $\rho_{wit}(\tau)$ such that, roughly speaking,

$$cmod^{\mathrm{t}}(P, \rho_{wit}(\tau)) \xrightarrow{\tau} h = cmod^{\mathrm{t}}(P', \rho_{wit}(\tau))$$

for some process P', i.e., in particular, $h \neq \bot$. We define the schedule $\rho_{wit}(\tau)$ to return the inputs appearing on the trace τ:

$$\begin{aligned}
\rho_{wit}(bio'(v', w) \# \tau, (\epsilon, bio, v)) &= \text{if } bio' = bio \wedge v' = v \text{ then } w \text{ else } pick(Ty(bio, v)) \\
\rho_{wit}(a \# \tau', (b \# \tau, bio, v)) &= \text{if } a = b \text{ then } \rho_{wit}(\tau', (\tau, bio, v)) \text{ else } pick(Ty(bio, v)) \\
\rho_{wit}(_, (_, bio, v)) &= pick(Ty(bio, v)).
\end{aligned}$$

That is, for proper prefixes τ' of the trace τ, I/O operation bio, and output v, the schedule $\rho_{wit}(\tau)$ returns the input w, if $bio(v, w)$ is the next step in τ after the prefix τ'. For other traces, it returns an arbitrary well-typed input (i.e., $pick(S)$ selects an arbitrary element from a non-empty set S).

The following three lemmas make the intuition given above more precise. We first prove a lemma about the individual transitions of canonical models.

Lemma A.4 (Canonical heap transitions). *Suppose that*

$$gmod^{\mathrm{t}}(P, \rho, \tau, ppos, cpos) +^{\#} g \xrightarrow{bio(v,w)} h$$

for some heaps g and h such that $w \in Ty(bio, v)$, $ppos \leq cpos$, and g is dead for $ppos$. Then there exist a process P', positions $cpos'$ and pos', and a heap g' such that $w = \rho(\tau, bio, v)$, g' is dead for $cpos'$, $P \xrightarrow{bio(v,w)} P'$, and $h = gmod^{\mathrm{t}}(P', \rho, \tau \cdot \langle bio(v, w)\rangle, cpos', cpos') +^{\#} g'$.

The following lemma states that transitions with scheduled input never lead to the chaotic state $\bot$.

Lemma A.5 (Transitions with scheduled input). *If $gmod^{\mathrm{t}}(P, \rho, \tau, ppos, cpos) \xrightarrow{bio(v,w)} ho$ for $ppos \leq cpos$ and $w = \rho(\tau, bio, v)$, then $ho \neq \bot$.*

Next, we extend these lemmas from individual transitions to traces.

Lemma A.6 (Canonical heap traces). *Suppose we have*

$$gmod^{\mathrm{t}}(P, \rho_{wit}(\sigma), \tau, ppos, cpos) +^{\#} g \xrightarrow{\tau'} ho,$$

where $\tau \cdot \tau' \leq \sigma$, $ppos \leq cpos$, g is dead for $ppos$, and $ho \in Heap_{\bot}$. Then there exist a process P', place t', heap g', and positions $ppos'$ and $cpos'$ such that $ppos' \leq cpos'$, g' is dead for $ppos'$,

$$P \xrightarrow{\tau'} P', \text{ and } ho = gmod^{\mathrm{t}}(P', \rho_{wit}(\sigma), \tau \cdot \tau', ppos', cpos') +^{\#} g'.$$

Proof. By trace induction using Lemmas A.4 and A.5 for single transitions. □

Now we can prove that each trace of the set of canonical models of P is also a trace of P.

Proposition A.7. $traces^{\mathcal{H}}(can(P)) \subseteq traces(P)$.

Proof. Let $\tau \in traces^{\mathcal{H}}(can(P))$. Then τ is a trace of all canonical heap models of P; hence, in particular, $cmod^{\mathrm{t}}(P, \rho_{wit}(\tau)) \xrightarrow{\tau} ho$ for some $ho \in Heap_{\bot}$. By Lemma A.6, we conclude that $\tau \in traces(P)$. □