Programming and Proving with Distributed Protocols

ILYA SERGEY, University College London, UK
JAMES R. WILCOX, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

Distributed systems play a crucial role in modern infrastructure, but are notoriously difficult to implement correctly. This difficulty arises from two main challenges: (a) correctly implementing core system components (*e.g.*, two-phase commit), so all their internal invariants hold, and (b) correctly composing standalone system components into functioning trustworthy applications (*e.g.*, persistent storage built on top of a two-phase commit instance). Recent work has developed several approaches for addressing (a) by means of mechanically verifying implementations of core distributed components, but no methodology exists to address (b) by composing such verified components into larger verified applications. As a result, expensive verification efforts for key system components are not easily reusable, which hinders further verification efforts.

In this paper, we present Disel, the first framework for implementation and *compositional* verification of distributed systems and their clients, all within the mechanized, foundational context of the Coq proof assistant. In Disel, users implement distributed systems using a domain specific language shallowly embedded in Coq and providing both high-level programming constructs as well as low-level communication primitives. Components of composite systems are specified in Disel as *protocols*, which capture system-specific logic and disentangle system definitions from implementation details. By virtue of Disel's dependent type system, well-typed implementations always satisfy their protocols' invariants and *never go wrong*, allowing users to verify system implementations interactively using Disel's Hoare-style program logic, which extends state-of-the-art techniques for concurrency verification to the distributed setting. By virtue of the substitution principle and frame rule provided by Disel's logic, system components can be composed leading to modular, reusable verified distributed systems.

We describe Disel, illustrate its use with a series of examples, outline its logic and metatheory, and report on our experience using it as a framework for implementing, specifying, and verifying distributed systems.

CCS Concepts: • **Theory of computation** → **Logic and verification**; • **Software and its engineering** → **Distributed programming languages**;

Additional Key Words and Phrases: distributed systems, program logics, safety verification, dependent types

ACM Reference Format:
Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (January 2018), 30 pages. https://doi.org/10.1145/3158116

28

1 INTRODUCTION

Real-world software systems, including distributed systems, are rarely built as standalone, monolithic pieces of code. Rather, they are composed of multiple independent modules, which are connected either by the linker or through communication channels. Such a compositional approach enables clean separation of concerns and a modular development process: in order to use one

Authors' addresses: Ilya Sergey, University College London, UK, i.sergey@ucl.ac.uk; James R. Wilcox, University of Washington, USA, jrw12@cs.washington.edu; Zachary Tatlock, University of Washington, USA, ztatlock@cs.washington.edu.

2475-1421/2018/1-ART28
https://doi.org/10.1145/3158116

component within a larger system, one only needs to know *what* it does without requiring details on *how* it works. Unfortunately, the benefits of modular software development are not yet fully realized in the context of verified distributed systems.

Recent work has produced several impressive formal proofs of correctness for implementations of core distributed system components, ranging from consensus protocols to causally consistent key-value stores [Hawblitzel et al. 2015; Lesani et al. 2016; Newcombe et al. 2015; Woos et al. 2016]. These artifacts, while formally verified, are not immediately reusable in the context of larger verified applications. For example, to compose a linearizable database with a causally consistent cache [Ahamad et al. 1995], one would need a framework general enough to express both specifications and reason about their interaction, possibly in the presence of application-specific constraints. Furthermore, existing verified systems entangle implementation details with abstract protocol definitions, preventing independent evolution and requiring extensive refactoring when changes are made [Woos et al. 2016].

Finally, like all software, real-world systems exist in an open world, and should be usable in multiple contexts by various clients, each of which may make different assumptions.

1.1 Towards Modular Distributed System Verification

Recent advances in the area of formal machine-assisted program verification demonstrated that *composition*, obtained by means of expressive specifications and rich semantics, is the key to producing scalable, robust and reusable software artifacts in correctness-critical domains, such as compilers [Kumar et al. 2014; Stewart et al. 2015], operating systems [Gu et al. 2015; Klein et al. 2010] and concurrent libraries [Gu et al. 2016; Sergey et al. 2015]. Following this trend, we identify the following challenges in designing a verification tool to support compositional proofs of distributed systems.

(1) **Protocol-program modularity.** One should be able to define an *abstract model* of a distributed protocol (typically represented by a form of a state-transition system) without tying it to a *specific implementation*. Any purported implementation should then be proven to follow the protocol's abstract model. This separation of concerns supports reuse of existing techniques for reasoning about the high-level behavior of a system, while allowing for optimized implementations, without redefining the high-level interaction protocol.

(2) **Modular program verification.** Once proven to implement an abstract protocol, a *program* should be given a sufficiently expressive declarative *specification*, so that clients of the code never need to be examine the implementation itself. Furthermore, it should be possible to specify and verify programs made up of parts belonging to *different* protocols (horizontal compositionality). This enables decomposing a distributed application into independently specified and proved parts, making verification *scale to large codebases*.

(3) **Modular proofs about distributed protocols.** A single protocol may be useful to multiple different client applications, each of which may exercise the protocol in different ways. For instance, a "core" consensus protocol implementation can be employed both for leader election as well as for a replicated data storage. In this case, the invariants of the core protocol should be proved *once and for all* and then reused to establish properties of composite protocols. These composite protocols often require elaborating the core invariants with client-specific assumptions, but it would be unacceptable to re-verify all existing code under new assumptions for each different use of the core protocol. Instead, clients should be able to prove their elaborated invariants themselves by reasoning about the core protocol after the fact. This also ensures any existing program that follows the protocol is guaranteed to also satisfy the client's new invariant.

This decomposition between core protocols and elaborated client invariants reduces and parallelizes the proof engineering effort: the core system implementor verifies basic properties of the protocol and correctness of the implementation, while the system's client proves the validity of their context-specific invariants.

This paper presents Disel, a mechanized framework for verification and implementation of distributed systems that aims to address these challenges.

1.2 What is Disel?

Disel is a verification framework incorporating ideas from dependent type theory, interactive theorem proving, separation-style program logics for concurrency, resource reasoning, and distributed protocol design.

From the perspective of a distributed protocol designer, Disel is a domain-specific language for defining a protocol $\mathcal{P}$ in terms of its state-space invariants and atomic primitives (*e.g.*, send and receive). These primitives implement specific transitions which synchronize message-passing with changes to the local state of a node. Described this way, the protocols are immediately amenable to machine-assisted verification of their safety and temporal properties [Rahli et al. 2015; Wilcox et al. 2015], and Disel facilitates these proofs by providing a number of higher-order lemmas and libraries of auxiliary facts.

From the point of view of a system implementor, Disel is a *higher-order* programming language, featuring a complete toolset of programming abstractions, such as first-class functions, algebraic datatypes, and pattern matching, as well as *low-level* primitives for message-passing distributed communication. Disel's dependent type system makes programs *protocol-aware* and ensures that *well-typed programs don't go wrong*; that is, if a program c type-checks in the context of one or many protocols $\mathcal{P}_1, \ldots, \mathcal{P}_n$ (*i.e.*, informally, $\mathcal{P}_1, \ldots, \mathcal{P}_n \vdash c$), then it correctly exercises and combines transitions of $\mathcal{P}_1, \ldots, \mathcal{P}_n$.

Finally, for a human verifier, Disel is an expressive higher-order separation-style program logic[1] that allows programs to be assigned declarative Hoare-style specifications, which can be subsequently verified in an interactive proof mode. Specifically, one can check that, in the context of protocols $\mathcal{P}_1, \ldots, \mathcal{P}_n$, a program c satisfies pre/postconditions P and Q, where P constrains the pre-state s of the system, and Q constrains the result res and the post-state s'. The established pre-/postconditions can be then used for verifying larger client programs that use c as a subroutine. Disel takes a *partial correctness* interpretation of Hoare-style specifications, thus focusing on verification of safety properties and leaving reasoning about liveness properties for future work.

We implemented Disel on top of the Coq proof assistant, making use of Coq's dependent types and higher-order programming features. In the tradition of Hoare Type Theory (HTT) by Nanevski et al. [2006, 2008, 2010] and its recent versions for concurrency [Ley-Wild and Nanevski 2013; Nanevski et al. 2014], we give the semantics to effectful primitives, such as send/receive, with respect to a specific abstract protocol (or protocols). Thus, we address challenge **(1)** by ensuring that any *well-typed* program is correct (*i.e.*, respects its protocols) by construction, independently of which and how many of the imposed protocols' transitions are taken and of any imperative state the program might use. This type-based verification method for distributed systems, which was motivated by a recent vision paper by Wilcox et al. [2017], is different from more traditional techniques for establishing *refinement* [Abadi and Lamport 1988; Hawblitzel et al. 2015] between an actual implementation (the code) and a specification (an abstract protocol) via a simulation argument [Lynch and Vaandrager 1995]. In comparison with the refinement-based techniques, the type-based verification method makes it easy to account for horizontal composition of protocols

[1]The framework name stands for Distributed Separation Logic.

(necessary, *e.g.*, for reasoning about remote procedure calls, as we will show in Section 2) and accommodate advanced programming features, such as higher-order functions.

As a program logic, Disel draws on ideas from separation-style logics for shared-memory concurrency [Nanevski et al. 2014; Turon et al. 2014], allowing one to instrument programs with pre/postconditions and providing a form of the *frame rule* [Reynolds 2002] with respect to protocols. For example, assuming that the state-spaces of $\mathcal{P}_1$ and $\mathcal{P}_2$ are disjoint, $\mathcal{P}_1 \vdash c_1$ and $\mathcal{P}_2 \vdash c_2$ together with the frame rule imply $\mathcal{P}_1, \mathcal{P}_2 \vdash C[c_1, c_2]$ for any well-formed program context C. This ensures that the composite program $C[c_1, c_2]$ can "span" multiple protocols, thus addressing challenge **(2)**. The assumption of protocol state-spaces being disjoint might seem overly restrictive, but, in fact, it reflects the existing programming practices. For instance, the local state of a node responsible for tracking access permissions is typically different from the state used to store persistent data.

Disel further alleviates the issue of disjoint state and also addresses challenge **(3)** with two novel logical mechanisms, described in detail in Section 3. The first one supports the possibility of *elaborating protocol invariants* via an inference rule, WithInv, allowing one to strengthen the *assumptions* about a system's state, resulting in the strengthened *guarantees*, as long as these assumptions form an *inductive invariant*. Second, Disel supports "coupling" protocols via *inter-protocol behavioral dependencies*, which allow one protocol restricted *logical* access to state in another protocol, all while preserving the benefits of disjointness, including the frame rule. Dependencies are specified with the novel logical mechanism of inter-protocol *send-hooks*, allowing one to restrict interaction between a core protocol and its clients by placing additional preconditions on certain message sends. For example, a send-hook could disallow certain transitions of the client protocol unless a particular condition holds for the local state associated with the core protocol. These additional preconditions *do not* require re-verifying any core components.

While we do not explicitly model *node failures*, by focusing on establishing *safety* properties, Disel allows one to reason about systems where some of the nodes *can* experience non-Byzantine failures (*i.e.*, stop replying to messages). From the perspective of other participants in such systems, a failed node will be, thus, indistinguishable from a node that just takes "too long" to respond. As customary in reasoning about partial program correctness, this behavior will not violate the established notion of safety, which is termination-insensitive.

To summarize, this paper makes the following contributions:

- Disel, a domain-specific language and the first separation-style program logic for the implementation and compositional verification of message-passing distributed applications for full functional correctness, supporting effectful higher-order functional programming style, as well as custom distributed protocols and their combinations;
- Two conceptually novel logical mechanisms allowing reuse of Hoare-style and inductive invariant proofs while reasoning about distributed protocols: (a) the WithInv rule enabling *elaboration* of the protocol invariant in program specifications, and (b) *send-hooks*, providing a way to modularly verify programs operating in a *restricted product* of multiple protocols.
- A proof-of-concept implementation of Disel as a foundational (*i.e.*, proven sound from first principles [Appel 2001]) verification tool, built on top of Coq, as well as mechanized soundness proofs of Disel's logical rules with respect to a denotational semantics of message-passing distributed programs;
- An extraction mechanism into OCaml and a trusted shim implementation, allowing one to run programs written in Disel on multiple physical nodes;

- A series of case studies implemented and verified in Disel (including the Two-Phase Commit protocol [Weikum and Vossen 2002] and its client application), as well as a report on our experience of using Disel and a discussion on the executable code.

The implementation of Disel, including its mechanized metatheory and proofs of all examples from this paper, is available online: `https://github.com/DistributedComponents/disel`.

2 OVERVIEW

In this section we illustrate the Disel methodology for specifying, implementing, and verifying distributed systems by developing a simple distributed calculator. Disel systems are composed of concurrently running nodes communicating asynchronously by exchanging messages, which, as in real networks, can be reordered and dropped.

In the calculator system, each node n is either a *client* (written $n \in \overline{C}$) or a *server* ($n \in \overline{S}$), and the system is parameterized over some expensive partial function f with domain $\mathsf{dom}(f)$. Given arguments $args \in \mathsf{dom}(f)$, a client can send a request containing $args$ to a server, which will reply with $f(args)$. Fig. 1 depicts an example execution for the calculator system with one server S and two clients, C_1 and C_2. Note that requests and responses may not be received in the order they are sent due to network reordering, and the server may service requests in any order (*e.g.*, due to implementation details such as differing priorities among requests). However, the system should satisfy weak causality constraints, *e.g.*, a client C should only receive a response $f(args)$ if C had previously made a request for $args$. In the remainder of this section we show how Disel enables developers to specify the calculator protocol, implement several versions of server and client nodes that follow the protocol, and prove key invariants of the system.

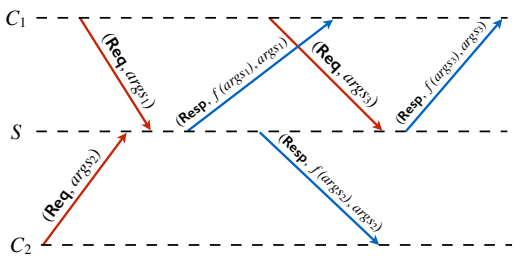

Fig. 1. A communication scenario between a server and two client nodes in a distributed calculator.

2.1 Defining a Calculator Protocol

A protocol in Disel provides a high-level specification of the interface between distributed system components. As with traditional program specifications, Disel protocols serve to separate concerns: implementations can refine details not specified by the protocol (*e.g.*, the order in which to respond to client requests), invariants of the protocol can be proven separately (*e.g.*, showing that calculator responses contain correct answers), and interactions between components within a larger system can be reasoned about in terms of their protocols rather than their implementations. Following the tradition established by Lamport [1978], Disel protocols are defined as *state-transition systems*.

Fig. 2 depicts the state-transition system for the calculator example with two send-transitions and two receive-transition. Each transition is named in the first column: s-transitions are for sending and r-ones for receiving. Their pre- and postconditions (in the form of requires/ensures pairs) are given as assertions in the second and third columns respectively. These assertions are phrased in terms of the message being sent/received, recipient/sender (*to*/*from*), and the protocol-specific state of a node n. For the calculator, the state for node n is a multiset of outstanding requests rs, denoted as $n \rightarrowtail rs$.

Protocol transitions synchronize the exchange of messages with changes in a node's state. Preconditions in send-transitions specify requirements that must be satisfied by the local state of

Send-transitions

τ_s	Requires (m, to)	Ensures
sreq	$n \in \overline{C} \wedge to \in \overline{S} \wedge n \rightarrowtail rs \wedge m = (\mathsf{Req}, args) \wedge args \in \mathsf{dom}(f)$	$n \rightarrowtail (to, args) \uplus rs$
sresp	$n \in \overline{S} \wedge f(args) = v \wedge n \rightarrowtail (to, args) \uplus rs \wedge m = (\mathsf{Resp}, v, args)$	$n \rightarrowtail rs$

Receive-transitions

τ_r	Requires $(m, from)$	Ensures
rreq	$n \in \overline{S}$ && $n \rightarrowtail rs$ && $m = (\mathsf{Req}, args)$	$n \rightarrowtail (from, args) \uplus rs$
rresp	$n \in \overline{C}$ && $n \rightarrowtail (from, args) \uplus rs$ && $m = (\mathsf{Resp}, ans, args)$	$n \rightarrowtail rs$

Fig. 2. Send- and receive-transitions of the distributed calculator protocol with respect to a node n.

node n for it to send message m to recipient to and postconditions specify how n's state must be updated afterward. For example, the *sreq* transition can be taken by a client node $n \in \overline{C}$ to send a request message $(\mathsf{Req}, args)$ to server to where $args \in \mathsf{dom}(f)$ and, after sending, n has added $(to, args)$ to its state. Preconditions in receive-transitions specify requirements that must be satisfied by the local state of node n for it to receive message m from sender $from$ and postconditions specify how n's state must be updated. For example, the *rreq* transition can be taken by a server node n to receive a request message $(\mathsf{Req}, args)$ from node $from$ where, after receiving, n has added $(from, args)$ to its state.

Notice that preconditions in send-transition can be *arbitrary* predicates, while the precondition of receive-transitions must be *decidable* (which we emphasize by using boolean conjunction && instead of propositional $\wedge$). This is because a program's decision to send a message is *active* and corresponds to calling the low-level send primitive (described later in this section); the system implementer *must* prove such preconditions to use the transition. In contrast, receiving messages is *passive* and corresponds to using the low-level recv primitive (also described later in this section) that will react to *any* valid message. A message m sent to node n should trigger the corresponding receive transition only if n's state along with the message satisfies the transition's precondition. To choose such a transition unambiguously, we require that each message's *tag* (*e.g.*, Req and Resp) uniquely identifies a receive-transition that should be run. Combined with the decidability of receive-transition preconditions, this allows Disel systems to automatically decide whether a transition can be executed.

As defined, the calculator protocol prohibits several unwelcome behaviors. For instance, a server cannot send a response without a client first requesting it, since (a) servers only send messages via the *sresp* transition, (b) *sresp* requires $(to, args)$ to be in the multiset of outstanding requests at the server, and (c) $(to, args)$ can only be added to the set of outstanding requests once it has been received from a client. Also note that the precondition of *sreq* requires that when a client sends a request to a server to compute $f(args)$, $args \in \mathsf{dom}(f)$. Similarly, the precondition of *sresp* requires that when a server responds to a client request for $args$, it may only send the correct result $f(args)$. In this case, the initial arguments $args$ are included into the response in order make it possible for the client to distinguish between responses to multiple outstanding requests.

The protocol also leaves several details up to the implementation. For example, the *sresp* transition allows a server to respond to *any* outstanding request, not necessarily the least recently received. This flexibility allows for diverse implementation strategies and enables the implementation $\mathcal{I}$ of a component to evolve without requiring updates to other components which only assume that $\mathcal{I}$ satisfies its protocol.

This state-space and transitions define the calculator protocol $\mathcal{C}$. Protocols are basic specification units in Disel, and, as we will soon see, a single program can "span" multiple protocols. Thus, we will annotate each protocol instance with a unique label ℓ_i (*e.g.*, $\mathcal{C}_{\ell_1}, \mathcal{C}_{\ell_2}$).

2.2 From Protocols to Programs

The transitions in Fig. 2 define functions mapping a state, message, and node id to a new state. We can use these functions as basic elements in building implementations of distributed system components, but first we need to "tie" them to realistic low-level message sending/receiving primitives. We can then combine these basic elements, via high-level programming constructs, into executable programs.

In Disel a programmer can define a new programming primitive based on a send- or receive transition using a library of *transition wrappers*, that decorate send/receive primitives with transitions of protocols at hand. The generic $\mathsf{send}[\tau_s, \ell]$ wrapper from this library takes a send-transition τ_s of a protocol identified by a label ℓ and yields a program that sends a message. For instance, from the description in Fig. 2 and Disel's logic (discussed in Section 3), we can assign the following Hoare type (specification) to a "wrapped" transition $sresp$ run by server n in the context of the protocol $\mathcal{C}_\ell$:

$$\mathcal{C}_\ell \overset{n}{\vdash} \mathsf{send}[sresp, \ell](m, to) : \left\{ \begin{array}{l} n \in \overline{S} \ \wedge n \rightarrowtail ((to, args) \uplus rs) \\ \wedge\ m = (\mathsf{Resp}, f(args), args) \end{array} \right\} \left\{ n \rightarrowtail rs \wedge \mathsf{res} = m \right\} \qquad (1)$$

The assertions in the pre/postconditions of the type (1) quantify implicitly over the *entire* global distributed state s (including previously sent messages), although the calculator protocol only constrains n's local contents in s, which are referred using the "node n's local state points-to" assertion of the form $n \rightarrowtail -$. In particular, the specification ensures that the outstanding request $(to, args)$ is removed from the local state of a node n upon sending a message. As customary in Hoare logic, all unbound variables (*e.g.*, rs, $args$) are universally-quantified and their scope spans both the pre- and post-condition. The return value res, occurring freely in the postcondition of a wrapped send-transition, is the message sent. In most of the cases, we will omit the type of res for the sake of brevity.

Disel's type system ensures Hoare-style pre/postconditions in types are *stable*, *i.e.*, invariant under possible concurrent transitions of nodes *other* than n. Stability often requires manual proving, but is indeed the case in the triple (1), as its pres/posts constrain only *local* state of the node n, which cannot be changed by other nodes. In general, Hoare triples in Disel can refer to state of other nodes as well, as we will demonstrate in Section 4.

Using a wrapper recv for tying a receive-transition to a *non-blocking* receive command is slightly more subtle. In general, we cannot predict which messages from which protocols a node n may receive at any particular point during its execution. To address this, receive wrapper $\mathsf{recv}[T, L]$ specifies a set T of message tags and a set L of protocol labels; and only accept messages whose tag is in T for a protocol whose label is in L.[2] The resulting primitive provides *non-blocking* receive: if there are no messages matching the criteria, it returns None and acts as an idle transition. Otherwise, it returns Some $(from, m)$ for a matching incoming message m from sender $from$, chosen non-deterministically from those available. For example, we can assign the following Hoare type to a wrapper, associated with the tag Req of $\mathcal{C}_\ell$:

[2]Our implementation also allows "filtering" messages to be received with respect to their content.

$$\mathcal{C}_\ell \overset{n}{\vdash} \mathsf{recv}[\{\mathsf{Req}\}, \{\ell\}] \;:\; \{n \in \overline{S} \wedge n \rightarrowtail rs\} \left\{ \begin{array}{l} \textbf{if } \mathsf{res} = \mathsf{Some}\ (from, (\mathsf{Req}, args)) \\ \textbf{then } n \rightarrowtail ((from, args) \uplus rs) \wedge \\ \qquad \langle from, n, \bullet, (\mathsf{Req}, args)\rangle \in MS_\ell \\ \textbf{else } n \rightarrowtail rs \end{array} \right\} \tag{2}$$

The postcondition of the type (2) demonstrates an important feature of Disel's Hoare-style specs: in the case of a received message, it existentially binds its components (*i.e.*, $from$, $args$) in then-branch, and also identifies the message $\langle from, n, \bullet, (\mathsf{Req}, args)\rangle$ in the *message soup* MS_ℓ (which models both the current state and history of the network) of the post-state s' *wrt.* the protocol $\mathcal{C}_\ell$. Messages in Disel's model (described in detail in Section 3.1) are never "thrown away"; instead they are added to the soup, where they remain *active* ($\circ$) until received, at which points they become *consumed* ($\bullet$).[3]

We can now employ the program (2) to write a blocking receive for request messages via Disel's built-in general recursion combinator **letrec** (explained in Section 3), assigning this procedure the following specification:

$$\begin{array}{l} \mathcal{C}_\ell \overset{n}{\vdash} \textbf{letrec } \mathsf{receive\_req}\ (\_ : \mathsf{unit}) \triangleq \\ \qquad r \leftarrow \mathsf{recv}[\{\mathsf{Req}\}, \{\ell\}]; \\ \qquad \textbf{if } \mathsf{res} = \mathsf{Some}\ (from, m) \\ \qquad \textbf{then return } (from, m) \\ \qquad \textbf{else } \mathsf{receive\_req}\ () : \forall u : \mathsf{unit}. \{n \in \overline{S} \wedge n \rightarrowtail rs\} \left\{ \begin{array}{l} n \rightarrowtail ((\mathsf{res}.1, \mathsf{res}.2) \uplus rs) \wedge \\ \langle \mathsf{res}.1, n, \bullet, (\mathsf{Req}, \mathsf{res}.2)\rangle \in MS_\ell \end{array} \right\} \end{array} \tag{3}$$

The Hoare type of receive_req describes it as a function, which takes an argument of type unit and is safe to run in a state, satisfied by its precondition. The pre/postconditions of receive_req are derived from the type (2) by application of a typing (inference) rule for fixpoint combinator, with an assistance of a human prover and according to the inference rules of Disel, described in Section 3.2. Internally, receive_req corresponds to an execution of possibly several idle transitions, followed by one receive-transition. That is, when invoked, it still follows $\mathcal{C}_\ell$'s transitions: otherwise we simply could not have assigned a type to it at all! In other words, a body of receive_req is merely a combination of more primitive sub-programs (namely, the "wrapped" non-blocking receive (2)) that are proven to be protocol-compliant.

2.3 Elaborating State-Space Invariants of a Protocol

```
1  letrec simple_server (_ : unit) ≜
2    (from, args) ← receive_req ();
3    let v = f(args) in
4    send[sresp, ℓ]((Resp, v, args), from);
5    simple_server ()
6  in simple_server ()
```

Let us now use receive_req to implement our first useful component of the system: a simple server, which runs an infinite loop, responding to one request each iteration (on the right). In trying to assign a type to this program in the context of $\mathcal{C}_\ell$ for a node $n \in \overline{S}$, we encounter a problem at line 3. Since f is partially-defined, Disel will emit a verification condition (VC), requiring us to prove that f is defined at $args$. Unfortunately, the postcondition in the spec (3) of receive_req does not allow us to prove the triple: we can only conclude that a message from the soup is consumed, but not that its contents are well-formed, *i.e.*, that $args \in \mathsf{dom}(f)$. The issue is caused by the lack of constraints, imposed by the protocol $\mathcal{C}_\ell$ on the system state s, specifically, on the messages in its soup, which we refer to as $s\#MS_\ell$. The necessary requirement for this

[3] This design choice with respect to message representation is common in state-of-the-art frameworks for distributed systems verification, *e.g.*, IronFleet [Hawblitzel et al. 2015] and Ivy [Padon et al. 2016], as it simplifies reasoning about past events.

```
letrec receive_batch (k : nat) ≜
  if k = k' + 1
  then fargs ← receive_req ();
       rest ← receive_batch k';
       return fargs :: rest
  else return [ ]

letrec send_batch (rs : [(Node, [nat])]) ≜
  if rs = (from, args) :: rs'
  then let v = f(args) in
       send[sresp, ℓ]((Resp, v, args), from);
       send_batch rs'
  else return ()

letrec batch_server (bsize : nat) ≜
  reqs ← receive_batch bsize;
  send_batch reqs;
  batch_server bsize
```

(a)

```
letrec memo_server (mmap : map) ≜
  (from, args) ← receive_req ();
  let ans = lookup mmap args in
  if ans ≠ ⊥
  then
    send[sresp, ℓ]((Resp, ans, args), from);
    memo_server mmap
  else
    let ans = f(args) in
    send[sresp, ℓ](m, (Resp, ans, args));
    let mmap' = update mmap args ans in
    memo_server mmap'
```

(b)

Fig. 3. Batching (a) and memoizing (b) calculator servers defined on top of the protocol $\mathcal{C}'_\ell$.

example, however, could be derived from the following property of a state s:

$$\textsc{Inv}_1(s) \triangleq \forall \mathsf{m} \in s\#MS_\ell,\ \mathsf{m} = \langle from, to, -, (\mathsf{Req}, args)\rangle \implies args \in \mathsf{dom}(f) \tag{4}$$

The good news is that the property $\textsc{Inv}_1$ is an *inductive invariant* with respect to the transitions of $\mathcal{C}_\ell$: if it holds at some initial state s_0, then it holds for *any* state s reachable from s_0 via $\mathcal{C}_\ell$'s transitions. Better yet, since every well-typed program in Disel is composed of protocol transitions, it will *automatically* preserve the inductive invariant and can be given *the same* pre/postconditions, as long as the pre-state satisfies the invariant.

To account for this possibility of invariant elaboration, Disel provides a *protocol combinator* WithInv that takes a protocol $\mathcal{P}$ and a state invariant I, proven to be inductive *wrt.* $\mathcal{P}$, and returns a new protocol $\mathcal{P}'$, whose state-space definition is strengthened with I. That is, the pre/postcondition of every transition can be strengthened with I "for free" once I is shown to be an inductive invariant. Therefore, taking $\mathcal{C}'_\ell \triangleq \mathsf{WithInv}(\mathcal{C}_\ell, \textsc{Inv}_1)$, we can reuse all of simple_server's subprograms in the new context $\mathcal{C}'_\ell$. The postcondition on line 3, in conjunction with $\textsc{Inv}_1$ holding over any intermediate states ensures that f is defined at $args$, allowing us to complete the verification of our looping server implementation, assigning it the following type (with the standard False postcondition due to non-termination):

$$\mathcal{C}'_\ell \overset{n}{\vdash} \mathsf{simple\_server}\ () : \{n \in \overline{S} \wedge n \rightarrowtail rs\}\{\mathsf{False}\} \tag{5}$$

Having a server loop assigned a specification (5) ensures that it faithfully follows the protocol's transitions and does not terminate.

2.4 More Implementations for Cheap

With the elaborated protocol $\mathcal{C}'_\ell$, we can now develop and verify a variety of system components, reusing the previously developed libraries and enjoying the compositionality of specs, afforded by Hoare types quantifying over a distributed state and sent/received messages. It is still up to the programmer to verify those implementations in a Hoare style, but writing them does not require changing the protocol, only composing the verified subroutines.

Alternative servers. Fig. 3 presents two alternative looping server implementations. The first one processes requests in batches of a predefined size $bsize$. This batching may cause batch_server to loop for an unbounded period, until $bsize$ requests have been received, but this is perfectly safe. Once this is done, the batch is passed to the second subroutine, send_batch, which delivers the results. Finally, the server loop restarts. Another, more efficient server implementation memo_server uses memoization, implemented by means of store-passing style, in order to avoid repeating computations. It first checks whether the answer for a requested argument list is available in the memoization table $mmap$, and, if so, sends it back to the client. Otherwise, it computes the answer and stores it in the local state, which is then passed to the next recursive call. Both implementations, when invoked with a suitable initial argument (batch size and an empty map, correspondingly), type-check against the same Hoare type as the simple server (5) and are verified directly from the specifications of their components in the context of $\mathcal{C}'_\ell$.

Implementing a calculator client. Let us now build and verify a simple client-side procedure that requests a computation and obtains the result. It can be implemented as shown on the right. The program compute sends a request to a server $serv$ and then runs a blocking procedure receive_resp for a message with the Resp tag, implemented similarly to receive_req, and having, when invoked as a function, the following specification, stating that res is the received response:

```
1  letrec compute (args, serv) ≜
2    send[sreq, ℓ]((Req, args), serv);
3    v ← receive_resp ();
4    return v
```

$$\mathcal{C}'_\ell \overset{n}{\vdash} \mathsf{receive\_resp}\ () \ :\ \begin{array}{l}\{n \in \overline{C} \wedge n \rightarrowtail \{(serv, args)\}\} \\ \{\langle serv, n, \bullet, (\mathsf{Resp}, \mathsf{res}, args)\rangle \in MS_\ell \wedge n \rightarrowtail \emptyset\}\end{array} \tag{6}$$

Unfortunately, this type is not helpful to prove the desired spec of compute, stating that its result is equal to $f(args)$: this dependency is not captured in (6)'s postcondition. In order to deliver a stronger postcondition of receive_resp, we need to elaborate the protocol's state-space assumption even further, proving the following invariant $\textsc{Inv}_2$ inductive:

$$\textsc{Inv}_2(s) \triangleq \forall \mathsf{m} \in s\#MS_\ell,\ \mathsf{m} = \langle n_1, n_2, -, (\mathsf{Resp}, ans, args)\rangle \implies f(args) = ans \tag{7}$$

What is left is to verify the implementation of receive_resp in the context of $\mathcal{C}''_\ell \triangleq \mathsf{WithInv}(\mathcal{C}'_\ell, \textsc{Inv}_2)$. The property $\textsc{Inv}_2$ ensures that any answer carried by a Resp-message is correct *wrt.* the corresponding arguments. Since the client has only one outstanding request at the moment it calls receive_resp, it will only accept a message with an answer to that request. Thus, we can prove the following spec for the RPC compute:

$$\mathcal{C}''_\ell \overset{n}{\vdash} \mathsf{compute}\ (args, serv) \ :\ \begin{array}{l}\{n \in \overline{C} \wedge n \rightarrowtail \emptyset \wedge serv \in \overline{S} \wedge args \in \mathsf{dom}(f)\} \\ \{\mathsf{res} = f(args) \wedge n \rightarrowtail \emptyset\}\end{array} \tag{8}$$

Server as a client. So far, we have only considered programs that operate in the context of a *single* protocol. However, it is common for realistic applications to participate in several systems. Disel accounts for such a possibility by providing an *injection/protocol framing* mechanism, inspired by the FCSL program logic by Nanevski et al. [2014], and allowing one to type-check a program in the context of several protocols with disjoint state-spaces. The disjointness of those *does not* mean the disjointness of the node sets: one node can be a part of several protocols, in which case its local state is divided among them. As an example, let us implement yet another calculator server, this time using an ℓ_1-labelled protocol run by a node n, which, instead of calculating directly, *delegates* to a server n'

```
letrec deleg_server (n' : Node) ≜
  (from, args) ← receive_req_ℓ1 ();
  ans ← compute_ℓ2 (args, n');
  send[sresp, ℓ1]((Resp, ans, args), from);
  deleg_server n'
```

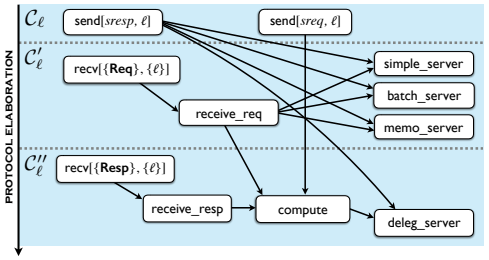

Fig. 4. Components of the calculator system.

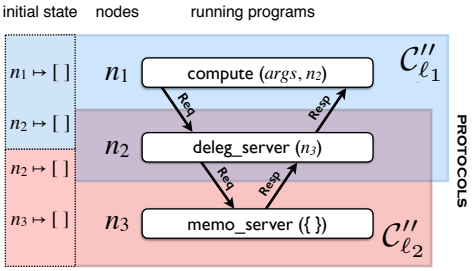

Fig. 5. Initial state and execution with three nodes.

in *another* protocol (labelled with ℓ_2, which we use to annotate the corresponding call to compute to emphasize the protocol it "belongs to"), in which n is a client. The code of deleg_server is almost identical to the code of simple_server and it has the following type in the context of two independent protocols, $\mathcal{C}''_{\ell_1}$ and $\mathcal{C}''_{\ell_2}$:

$$\mathcal{C}''_{\ell_1}, \mathcal{C}''_{\ell_2} \overset{n}{\vdash} \mathsf{deleg\_server}\ (n') : \left\{(n \in \overline{S}_{\ell_1} \wedge n \overset{\ell_1}{\rightharpoonup} rs) * (n \in \overline{C}_{\ell_2} \wedge n' \in \overline{S}_{\ell_2} \wedge n \overset{\ell_2}{\rightharpoonup} \emptyset)\right\} \{\mathsf{False}\} \quad (9)$$

In the precondition, the assertions about the nodes' roles and local state are elaborated for specific constituent protocols, labeled with ℓ_1 and ℓ_2, correspondingly. Furthermore, we use the *separating conjunction* $*$ in order to emphasize the disjointness of the protocol-specific local states, used to handle outstanding requests within two different protocols. As a server, n can have an arbitrary number of "outstanding responses" rs in its local state (hence $n \overset{\ell_1}{\rightharpoonup} rs$), but should start with an empty set of its own outstanding requests, thus $n \overset{\ell_2}{\rightharpoonup} \emptyset$.

Summary of the Disel *methodology.* Our entire development of the calculator-aware applications (*e.g.*, servers and clients) is outlined in Fig. 4. This is a general layout of structuring the development of applications in Disel. In the figure, the top-down direction corresponds to elaborating the protocol invariants (so the specs of programs verified there can be directly reused further down), and the arrows denote dependencies between components.

2.5 Putting It All Together

Disel programs can be extracted into OCaml code, linked with a trusted shim, and run. In order to do so, one needs to assign each participant node a program to run (some nodes might have no programs assigned) and provide an initial distributed configuration that instantiates the local state for each participant in each protocol and satisfies all imposed state-space invariants (*e.g.*, (4) and (7)). The semantics of Hoare types in Disel, defined in Section 3.3, specifies what does it mean for a program to be type-safe (*i.e.*, *correct*) in a distributed setting: postconditions (even those constraining the global state) of well-typed programs are not affected by execution of programs running concurrently on other nodes, and such programs are always *safe* to run when their precondition is stable and satisfied.

As an illustration of one possible finalized protocol/program composition, Fig. 5 depicts the three calculator-based programs, described earlier, running concurrently by three different nodes, n_1, n_2, and n_3, such that n_1 and n_2 communicate according to the protocol $\mathcal{C}''_{\ell_1}$, and n_2 and n_3 follow the protocol $\mathcal{C}''_{\ell_2}$. Solid arrows between nodes denote message exchange, with the time going from left to right. The initial local states for all the nodes/protocols are instantiated with empty lists of requests. Importantly, the code run by the nodes n_1 and n_3 has been verified *separately*, in simpler, smaller contexts, and only the implementation of n_2's program deleg_server has been done in the composite context of two protocols. Our accompanying Coq development provides the complete

State-space components

$$
\begin{aligned}
\mathsf{Node}, \mathsf{Loc}, \mathsf{Mid} &\triangleq \mathbb{N} \\
\mathsf{Lab}, \mathsf{Tag} &\triangleq \mathbb{N} \\
l \in \mathsf{LocState} &\triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val} \\
\mathsf{DistLocState} &\triangleq \mathsf{Node} \xrightarrow{\text{fin}} \mathsf{LocState} \\
MS \in \mathsf{MessageSoup} &\triangleq \mathsf{Mid} \xrightarrow{\text{fin}} \mathsf{Msg} \\
\mathsf{m} \in \mathsf{Msg} &\triangleq \mathsf{Node} \times \mathsf{Node} \times \{\circ, \bullet\} \times \mathsf{MBody} \\
m \in \mathsf{MBody} &\triangleq \mathsf{Tag} \times \mathbb{N}^* \\
d \in \mathsf{Statelet} &\triangleq \mathsf{MessageSoup} \times \mathsf{DistLocState} \\
s \in \mathsf{State} &\triangleq \mathsf{Lab} \xrightarrow{\text{fin}} \mathsf{Statelet}
\end{aligned}
$$

World components

$$
\begin{aligned}
\mathsf{coh} \in \mathsf{Coh} &\triangleq \mathsf{Statelet} \to \mathsf{Prop} \\
\tau_s \in T_s &\triangleq \mathsf{Tag} \times \mathsf{Pre}_s \times \mathsf{Step}_s \\
\tau_r \in T_r &\triangleq \mathsf{Tag} \times \mathsf{Pre}_r \times \mathsf{Step}_r \\
\mathsf{Pre}_s &\triangleq \mathsf{Node} \times \mathsf{Node} \times \mathsf{MBody} \times \mathsf{Statelet} \to \mathsf{Prop} \\
\mathsf{Step}_s &\triangleq \mathsf{Node} \times \mathsf{MBody} \times \mathsf{LocState} \rightharpoonup \mathsf{LocState} \\
\mathsf{Pre}_r &\triangleq \mathsf{Msg} \times \mathsf{LocState} \to \mathsf{bool} \\
\mathsf{Step}_r &\triangleq \mathsf{Msg} \times \mathsf{LocState} \to \mathsf{LocState} \\
\mathcal{P} \in \mathsf{Protocol} &\triangleq \mathsf{Coh} \times T_s^* \times T_r^* \\
h \in \mathsf{hook} &\triangleq \mathsf{LocState} \times \mathsf{LocState} \times \mathsf{MBody} \times \mathsf{Node} \to \mathsf{Prop} \\
H \in \mathsf{Hooks} &\triangleq \mathsf{HkId} \times \mathsf{Lab} \times \mathsf{Lab} \times \mathsf{Tag} \xrightarrow{\text{fin}} \mathsf{hook} \\
C \in \mathsf{Context} &\triangleq \mathsf{Lab} \xrightarrow{\text{fin}} \mathsf{Protocol} \\
W \in \mathsf{World} &\triangleq \mathsf{Context} \times \mathsf{Hooks}
\end{aligned}
$$

Fig. 6. Disel's distributed state and world components.

implementation of the described programs in Disel DSL, their extracted executable counterparts in OCaml, and mechanized proofs of all of the mentioned invariants and specifications.

3 DISTRIBUTED SEPARATION LOGIC

We next describe the formal model of the state and protocols, giving meaning to Disel's Hoare-style specifications in the context of multiple protocols with disjoint state-spaces and possible imposed inter-protocol dependencies.

3.1 State and Worlds

Distributed state and its components. The left part of Fig. 6 defines the components of the state, subject to manipulation by concurrently executing programs run by different nodes. Each global system state s is a finite partial mapping from protocol labels $\ell \in \mathsf{Lab}$ to *statelets*. Each statelet represents a protocol-specific component, consisting of a "message soup" MS and a per-node local state ($\mathsf{DistLocState}$). The former represents a finite partial map from unique message identifiers to messages,[4] each of which carries its sender and recipient node ids, the payload m, which includes a tag, and a boolean indicating whether the message is already received ($\bullet$) or not yet ($\circ$). The per-node local state maps each node id into protocol-specific piece of local state, represented as a mapping from locations (isomorphic to natural numbers) to specific values. For instance, in the calculator system example from Section 2, all local states had the same type and each carried just one value, updated in the course of execution,—a multiset of outstanding requests—so we omitted the only location from assertions in the program specs.

Protocols, hooks and worlds. The right part of Fig. 6 shows the components of Disel protocols and worlds. A protocol $\mathcal{P}$ consists of a state-space coherence predicate coh, which defines the shape of the corresponding statelet (*i.e.*, components of the per-node local state and message soup properties), and two finite sets of send- and receive transitions: T_s and T_r, correspondingly. Each send-transition is defined by a *tag* of a message it can send, a precondition, and a step function. The precondition constrains the sender, the addressee, the message to be sent, and the local state of the sender. The step function, which is partially defined, describes the changes in the local state of the sender, assuming that the state satisfies the precondition. Each receive-transition comes with a tag, which uniquely identifies it in a specific protocol. Its precondition is decidable in order to allow the

[4]The uniqueness constraint is introduced to make the encoding easier in Coq, but our specs and proofs do not rely on it, and the implementation prevents using message ids as values in programs.

$$
\begin{array}{ll}
s \vDash n \overset{\ell}{\rightarrowtail} l & \text{iff } \exists d, s(\ell) = (-, d) \wedge d(n) = l \\
s \vDash P(MS_\ell) & \text{iff } \exists MS, s(\ell) = (MS, -) \wedge P(MS) \\
s \vDash P_1 * P_2 & \text{iff } \exists s_1\ s_2, s = s_1 \uplus s_2 \wedge s_1 \vDash P_1 \wedge s_2 \vDash P_2 \\
s \vDash \mathsf{this}\ s' & \text{iff } s = s'
\end{array}
$$

Fig. 7. Semantics of Disel state assertions.

runtime to check it for applicability. Its step function is totally defined. We will use the notations $\tau.tag$, $\tau.pre$ and $\tau.step$ to refer correspondingly to the tag, precondition and step-components of a transition τ, which might be either send- or receive-one.

A *world* W is represented by a pair $\langle C, H\rangle$, with its first component C being a collection of protocols that are assigned unique labels. For instance, deleg_server from Section 2 was specified in the context of a world with two protocols with disjoint state-spaces, $\mathcal{C}''_{\ell_1}$ and $\mathcal{C}''_{\ell_2}$. The second component of a world H contains client-provided *send-hooks*, used to impose application-specific restrictions on interacting protocols, as we will demonstrate in Section 4. Each hook $h(l_s, l_c, m, to)$ is a predicate, relating a local state of a node l_s, which belongs to a core (or *server*) protocol, a local state l_c of the same node from a *client* protocol, a content of a message m to be sent and a potential recipient to. A hook-map Hooks associates each hook h with a unique id $z \in \mathsf{HkId}$, a core protocol label ℓ_s, a client protocol label ℓ_c and a tag t of a send-transition it applies to. Each send-hook prevents a send-transition τ_s in a particular client protocol from being taken by a node n, unless the hook's predicate holds *wrt.* n's local state in both server and client protocols; in other words hooks allow *strengthening* τ_s's precondition. Hooks are discussed in more detail below. All examples we have seen so far in Section 2 were defined with $H = \emptyset$ (*i.e.*, without any imposed inter-protocol restrictions), but in Section 4 we will show how the mechanism of send-hooks enables modular verification of programs operating in a restricted product of protocols, allowing one to build verified distributed client applications on top of verified core systems.

A world $W = \langle C, H\rangle$ is well-formed *iff* all protocol labels (for servers and clients) in the domain of H are also in the domain of C. A state s is coherent *wrt.* a world $W = \langle C, H\rangle$ ($W \Vdash s$) *iff* (a) both C and s are defined on the same set of unique labels, and (b) $\forall \ell \in \mathsf{dom}(C), C(\ell).\mathsf{coh}(s(\ell))$, *i.e.*, each statelet in s is coherent with respect to the corresponding protocol in C. When defining a protocol, it is a programmer's responsibility to show that all its transitions preserve the global protocol-specific state coherence, a fact that can be then used freely in the proofs about programs.

3.2 Language, Specifications and Selected Inference Rules

The programming language of Disel, embedded shallowly into Coq, features pure, strictly normalizing, *expressions* (*i.e.*, those of Gallina), such as **let**-expressions, tuples, variables and literals, ranged over by e (with v being a fully reduced value), and *commands* c, whose effect is distributed interaction, reading from local state and divergence, due to general recursion. The meta-variable F ranges over possibly recursive procedures. Non-interpreted effectful procedures are ranged over by a functional symbol f. Non-Hoare types are ranged over by a meta-variable $\mathcal{T}$. The syntax of Disel commands is given below:

$$
\begin{array}{rcl}
c & ::= & \mathsf{send}[\tau_s, \ell](e_m, e_{to}) \mid \mathsf{recv}[T, L] \mid \mathsf{read}_\ell(v) \mid x \leftarrow c_1; c_2 \mid \mathbf{return}\ e \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid F(e) \\
F & ::= & f \mid \mathbf{letrec}\ f(x : \mathcal{T}) \triangleq c
\end{array}
$$

Commands include send, receive and read actions, decorated with the corresponding protocol labels and transition tags. A decorated receive takes a set of tags T and a set of protocol labels L to identify the messages to react to. The $\mathsf{read}_\ell(v)$ command is used to examine the contents of a location v of a local state with respect to the protocol labelled ℓ, at the corresponding node

executing the command. The commands also include the standard monadic **return** e that returns the value of e, a sequential composition $x \leftarrow c_1; c_2$, implemented as a monadic bind (x may be omitted if not used in c_2), a conditional statement, and an application $F(e)$.

Program specifications. Fig. 7 provides the semantics of the assertions with respect to a distributed system state that we have used in the examples in Section 2, referring to particular component of the state constrained by pre- and postconditions of the corresponding Hoare specs. Specifically, a local state assertion $n \overset{\ell}{\rightarrowtail} l$ allows one to refer to a specific component l of a local state of a node n (which might be different from the one running the code), with respect to a protocol labelled ℓ. The *message soup selector* MS_ℓ allows one to make statement about message soup of a specific protocol. Finally, the separating conjunction ($*$), allows one to decompose assertions in the presence of a composite state s, which can be represented as a disjoint union of sub-states $s_1 \uplus s_2$. The separating conjunction allows one to combine separately proved specifications *wrt.* multiple involved protocols, as we did when assigning the type (9) to deleg_server. As is customary in Separation Logic [Reynolds 2002], the $*$ operator distributes over plain conjunction for assertions that do not constrain state. this s' allows one to assert that the immediate state is equal to a certain fixed state s'.

A command c run by a node n in a world W satisfies a spec $W \overset{n}{\vdash} c : \{P\}\{Q\}$ if it is safe to execute c from a global system state s satisfying P, concurrently with programs on other nodes, c respects the protocols and hooks from W, and returns a result value res, leaving the system in a state s', such that $s' \vDash Q$ holds. Here and below, we assume that res occurs freely in Q. All other unbound variables in Q and P are considered to be logical variables, whose scope spans both pre- and postcondition of the specification, with logical variables in Q (except for res) being a subset of those in P. In order to describe an effect of an uninterpreted and potentially recursive procedure $f(x : \mathcal{T})$, we employ the following notation for parameterized Hoare specs: $W \overset{n}{\vdash} f(x) : \forall x : \mathcal{T}.\{P\}\{Q\}$, where x may occur freely in P and Q. The Hoare-style logic of Disel will ensure that all intermediate program-level assertions, describing the global state from a perspective of a node n, which runs the code being verified, are *stable* [Jones 1983; Vafeiadis and Parkinson 2007], *i.e.*, closed under observable changes performed by all other nodes, involved into execution of the protocol, and, thus, captured by its definition.

Logic judgements and inference rules. The top part of Fig. 8 shows selected inference rules of Disel. In order to account for typed free program variables and functional symbols f, Disel's judgements are stated in the presence of a typing context Γ, defined as follows:

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, x : \mathcal{T} \mid \Gamma, f : \langle W, \forall x : \mathcal{T}.\{P\}\{Q\}\rangle$$

Typing entries for procedures f include the world W in which their specification was derived. The top two rules, Bind and Letrec, demonstrate the use of typing contexts.

The next two rules, SendWrap and ReceiveWrap, are crucial for program verification in Disel, as they allow one to assign Hoare specifications to atomic decorated send- and receive-commands, instrumented with the suitable protocol annotations. Both rules require user-assigned pre/postconditions to be *stable* with respect to interference imposed by the protocols in the world W. The net effect of sending or receiving a message atomically is captured by the two auxiliary assertion tuples Sent and Received, defined at the bottom of Fig. 8, which relate the states s and s' (captured via free logical variables) immediately before and after sending and receiving a message correspondingly.

Specifically, Sent ensures that the precondition of the corresponding send-transition τ_s, holds over the pre-state s, as well as all of the hook statements imposed by H, which is ensured by the auxiliary predicate HooksOk defined below in the same figure. The immediate post-state s' is the

$$\textsc{Bind}\quad \frac{\Gamma; W \overset{n}{\vdash} c_1 : \{P\}\{Q \wedge \mathsf{res} : \mathcal{T}\} \qquad \Gamma, x : \mathcal{T}; W \overset{n}{\vdash} [x/\mathsf{res}]c_2 : \{Q\}\{R\} \quad x \notin \mathsf{FV}(R)}{\Gamma; W \overset{n}{\vdash} x \leftarrow c_1; c_2 : \{P\}\{R\}}$$

$$\textsc{Letrec}\quad \frac{\Gamma, x : \mathcal{T}, f : \langle W, \forall x : \mathcal{T}.\ \{P\}\{Q\}\rangle; W \overset{n}{\vdash} c : \{P\}\{Q\}}{\Gamma; W \overset{n}{\vdash} \mathbf{letrec}\ f(x : \mathcal{T}) \triangleq c : \forall x.\mathcal{T}.\ \{P\}\{Q\}}$$

$$\textsc{SendWrap}\quad \frac{P, Q \text{ are } W\text{-stable} \quad W = \langle C, H\rangle \quad \tau_s \in C(\ell).T_s \qquad \mathsf{Sent}(\tau_s, \ell, n, m, to, H) \sqsubseteq (P, Q)}{\Gamma; W \overset{n}{\vdash} \mathsf{send}[\tau_s, \ell](m, to) : \{P\}\{Q\}}$$

$$\textsc{ReceiveWrap}\quad \frac{P, Q \text{ are } W\text{-stable} \quad W = \langle C, H\rangle \qquad \mathsf{Received}(T, L, C) \sqsubseteq (P, Q)}{\Gamma; W \overset{n}{\vdash} \mathsf{recv}[T, L](m, to) : \{P\}\{Q\}}$$

$$\textsc{Read}\quad \frac{P, Q \text{ are } W\text{-stable} \quad W = \langle C, H\rangle \qquad \left(\begin{array}{l}\mathsf{this}\ s \wedge \mathsf{coh}\ s \wedge \\ v \in \mathsf{dom}(s(\ell)(n))\end{array}\ ,\ \begin{array}{l}\mathsf{this}\ s \wedge \mathsf{coh}\ s \wedge \\ \mathsf{res} = s(\ell)(n)(v)\end{array}\right) \sqsubseteq (P, Q)}{\Gamma; W \overset{n}{\vdash} \mathsf{read}_\ell(v) : \{P\}\{Q\}}$$

$$\textsc{Frame}\quad \frac{\Gamma; W \overset{n}{\vdash} c : \{P\}\{Q\} \qquad \mathsf{NotHooked}(W, H) \quad R \text{ is } C\text{-stable}}{\Gamma; W \uplus \langle C, H\rangle \overset{n}{\vdash} c : \{P * R\}\{Q * R\}}$$

$$\textsc{WithInv}\quad \frac{\Gamma; \langle \ell \mapsto \mathcal{P}_\ell \uplus W, H\rangle \overset{n}{\vdash} c : \{P\}\{Q\} \qquad I \text{ is inductive } \textit{wrt.}\ \mathcal{P}_\ell \qquad \mathcal{I} \triangleq \forall s, \mathsf{this}\ s \Rightarrow I(s)}{\Gamma; \langle \ell \mapsto \mathsf{WithInv}(\mathcal{P}_\ell, I) \uplus W, H\rangle \overset{n}{\vdash} c : \{P \wedge \mathcal{I}\}\{Q \wedge \mathcal{I}\}}$$

Auxiliary definitions

$$\mathsf{Sent}(\tau_s, \ell, n, m, to, H) \triangleq \left(\begin{array}{l}\mathsf{this}\ s \wedge \mathsf{coh}\ s \wedge \\ \tau_s.pre(n, to, m, s(\ell)) \wedge \\ \mathsf{HooksOk}(H, \tau_s, \ell, n, m, to)\end{array}\ ,\ \begin{array}{l}\mathsf{this}\ s' \wedge \mathsf{coh}\ s' \wedge \mathsf{res} = m \wedge \\ s' = (s[\ell, n] \mapsto \tau_s.step(to, m, s(\ell)(n))) \wedge \\ s'\#MS_\ell = s\#MS_\ell \uplus \langle n, to, \circ, (\tau_s.tag, m)\rangle\end{array}\right)$$

$$\mathsf{Received}(T, L, C) \triangleq \left(\begin{array}{l}\mathsf{this}\ s \wedge \\ \mathsf{coh}\ s\end{array}\ ,\ \begin{array}{ll}\mathsf{this}\ s' \wedge \mathsf{coh}\ s' \wedge \mathbf{if}\ \mathsf{res} = \mathsf{Some}\ (from, m) & \\ \mathbf{then}\ \exists \ell \in L, t \in T, MS', \tau_r \in C(\ell).T_r,\ t = \tau_r.tag & \wedge \\ \qquad s\#MS_\ell = MS' \uplus \langle from, n, \circ, (t, m)\rangle & \wedge \\ \qquad s'\#MS_\ell = MS' \uplus \langle from, n, \bullet, (t, m)\rangle & \wedge \\ \qquad \tau_r.pre(\mathsf{m}, s(\ell)(n)) & \wedge \\ \qquad s' = (s[\ell, n] \mapsto \tau_r.step(m, s(\ell)(n))) & \\ \mathbf{else}\ s = s' & \end{array}\right)$$

$$\mathsf{HooksOk}(H, \tau_s, \ell_c, s, n, m, to) \triangleq \forall \ell_s\ h\ z, H(z, \ell_s, \ell_c, \tau_s.tag) = h \implies h(s(\ell_s)(n), s(\ell_c)(n), m, to)$$
$$\mathsf{NotHooked}(W, H) \triangleq \exists C,\ W = \langle C, -\rangle \wedge \forall (z, \ell_s, \ell_c, t) \in \mathsf{dom}(H), \ell_c \notin \mathsf{dom}(C).$$

Fig. 8. Selected logic inference rules of Disel and auxiliary predicates.

same as s, except for the local state of node $s(\ell)(n)$ of the node n *wrt.* the protocol ℓ, which is updated with the effect of the state transition $\tau_s.step$ (we use the notation $s(\ell)(n)$ to refer directly to the local state of n of in the second component of $s(\ell)$). Finally, the new message is added to the ℓ-related message soup MS_ℓ of s'. In contrast with sending, receiving messages does not impose any non-trivial preconditions, but in case of a successfully received message (*i.e.*, res is not None), it allows one to learn a number of facts about the pre-state, as captured by the assertions of Received. For instance, the tag t of a received message corresponds to the tag of the corresponding triggered receive-transition τ_r of the ℓ-labelled protocol, so the transition has changed the local state of n accordingly, and also "consumed" the received message in the message soup MS_ℓ. In conjunction with the protocol invariants, relating local state and message soup properties, this allows one to infer global assertions about the state of the network, as we have shown in Section 2.3.

The premises of these rules rely on the following definition of *Hoare ordering* $\sqsubseteq$, allowing one to strengthen the precondition $P_2 \Rightarrow P_1$ and weaken the postcondition $Q_1 \Rightarrow Q_2$, while accounting for the local scope of free logical variables in the assertions [Kleymann 1999].

Definition 3.1 (Hoare ordering). For the given pairs preconditions P_1, P_2 and postconditions Q_1, Q_2, possibly containing free logical variables, we say $(P_1, Q_1) \sqsubseteq (P_2, Q_2)$ *iff* $\forall s\ s',\ (s \vDash \exists \overline{x}_2.P_2 \Rightarrow s \vDash \exists \overline{x}_1.P_1) \wedge ((\forall \overline{x}_1\ \mathsf{res}.\ s \vDash P_1 \Rightarrow s' \vDash Q_1) \Rightarrow (\forall \overline{x}_2\ \mathsf{res}.\ s \vDash P_2 \Rightarrow s' \vDash Q_2)$, where $\overline{x}_i$ are the free logical variables of both P_i and Q_i correspondingly.

The rule Read is similar to the rules for sending and receiving messages, but it does not modify the local state in any way, observable by other nodes, which is what is ensured by the "atomic specification" in its premise, which expresses that the pre/post-states are the *very same* state this s, modulo W-interference, tolerated by pre/postconditions P and Q .

The rule Frame is the key to horizontal compositionality with respect to involved protocols. It allows one to add a "framed in" world part $\langle C, H\rangle$ (with the corresponding assertion R, quantifying over components of C-relevant state) to a specification, assuming that all involved assertions are stable. This rule is inherently asymmetric due to the "hooking" component H. Specifically, it allows any additions $\langle C, H\rangle$ as long as hooks in H cannot invalidate preconditions of send-transitions of W's protocols. This check, captured by the NotHooked auxiliary predicate defined at the bottom of Fig. 8, can be done *syntactically* on the domains of W and H, just by checking the "intersection" of their "footprints", very much in the spirit of ordinary Separation Logic.[5] Furthermore, if $H = \emptyset$, the rule Frame becomes symmetric and can be used to combine any two worlds that do not have mutual inter-protocol restrictions, which is what we did in Section 2.4 when implementing a delegating server. Typically, the world W contains a number of core protocols (*e.g.*, for locking or replication), whereas the addition $\langle C, H\rangle$ comes with client-specific protocols and restrictions imposed by the state *wrt.* W, so client applications have to be verified in a joint "large-footprint" world $W \uplus \langle C, H\rangle$. Here, $\uplus$ is a pointwise disjoint union of labeled protocols and hooks, so the rule only applies when the result of $\uplus$ is defined. In Section 4, we will demonstrate how to make such efforts reusable by exploiting Coq's higher-order definitions and abstract predicates.

Finally, the rule WithInv allows one to elaborate the context assumptions *wrt.* a specific protocol $\mathcal{P}_\ell$ and also the corresponding state assertions for any invariant I, which is $\mathcal{P}_\ell$-inductive, *i.e.*, it, as an assertion, over the global network state, is preserved while any node invokes any allowed send- or receive-transitions of $\mathcal{P}_\ell$.[6] Internally, the *protocol combinator* $\mathsf{WithInv}(\mathcal{P}_\ell)$ replaces the coherence predicate coh of the protocol $\mathcal{P}_\ell$ with a new one, elaborated with the inductive I. Applying this rule corresponds to proving *whole-system* properties, which is complementary to Hoare-style specifications, local for specific nodes.

The remaining rules, such as the rule of conjunction, function application, specification weakening *etc*, are standard and thus omitted.

3.3 Program Semantics and Logic Soundness

The semantics of programs and the soundness result in Disel are closely tied to the notion of *protocol-aware network semantics*. This is a non-deterministic small-step operational semantics, and its two transition rules are shown in Fig. 9 (ignore the gray boxes for now). All free variables in the rules other than s, n and W are existentially quantified. That is, the SendStep-rule will fire for a node n in a world $W = \langle C, H\rangle$ if there is a protocol $\mathcal{P}_\ell$ in C and there is a send-transition τ_s in $\mathcal{P}_\ell$, such that the corresponding local state of the sender n and the message m satisfy its precondition and also all W's hooks constraining τ_s are satisfied. The resulting state will thus have its n-entry

[5] This definition of NotHooked is a syntactic approximation of "framing *wrt.* transitions" that suffices for our purposes. More elaborated checks could be devised for tracking fine-grained dependencies between the core and the client protocols by considering the "transition footprint" instead of a "protocol footprint".

[6] The formal definition of inductive invariants is with respect to the protocol-aware network semantics, defined in Section 3.3, and is available in the accompanying Coq development.

$$\textsc{SendStep}\quad \frac{\begin{array}{c} W = \langle C, H\rangle \\ W \Vdash s \quad \ell \in \mathsf{dom}(C) \quad \mathcal{P}_\ell = C(\ell) \quad (MS, d) = s(\ell) \quad \{n, to\} \subseteq \mathsf{dom}(d) \quad \tau_s \in \mathcal{P}_\ell.Ts \\ \tau_s.pre(n, to, m, d) \quad \mathsf{HooksOk}(H, \tau_s, \ell, s, n, m, to) \quad MS' = MS \uplus \langle n, to, \circ, (\tau_s.tag, m)\rangle \end{array}}{s \rightsquigarrow^{n}_{W} s[\ell \mapsto (MS', d[n \mapsto \tau_s.step(to, m, d(n))])]}$$

$$\textsc{ReceiveStep}\quad \frac{\begin{array}{c} W = \langle C, H\rangle \quad W \Vdash s \quad \ell \in \mathsf{dom}(C) \quad \mathcal{P}_\ell = C(\ell) \quad (MS, d) = s(\ell) \\ \tau_r \in \mathcal{P}_\ell.Tr \quad MS = MS' \uplus \mathsf{m} \quad \mathsf{m} = \langle from, n, \circ, (\tau_r.tag, m)\rangle \quad \{from, n\} \subseteq \mathsf{dom}(d) \quad \tau_r.pre(\mathsf{m}, d(n)) \\ MS'' = MS' \uplus \langle from, n, \bullet, (\tau_r.tag, m)\rangle \end{array}}{s \rightsquigarrow^{n}_{W} s[\ell \mapsto (MS'', d[n \mapsto \tau_r.step(\mathsf{m}, d(n))])]}$$

Fig. 9. Transition rules of the network semantics.

wrt. $\mathcal{P}_\ell$ updated correspondingly, and a new message added to the soup MS with a fresh logical message id (omitted here for brevity). The rule ReceiveStep is similar in that it looks for an active message m in the soup MS of a arbitrarily chosen protocol $\mathcal{P}_\ell$, such that n is its addressee, and its tag corresponds to a specific receive-transition τ_r of $\mathcal{P}_\ell$. It then checks the precondition of τ_r at n's local state, and executes it, updating s's local state and soup correspondingly.

One can notice the similarity between the network semantic rules SendStep and ReceiveStep and the inference rules SendWrap and ReceiveWrap from Fig. 8. This should not come as a surprise: indeed, the two mentioned inference rules provide a way to symbolically account for corresponding local executions of send- receive-transtions by a specific node, consistend with the network semantics.

We build the semantics of programs in Disel with respect to a specific node n and a world W. To do so, we provide the semantics of wrappers for transitions via the following semi-formal definitions (the formal ones are in our Coq code), accompanied by the natural adequacy result (Lemma 3.4).

Definition 3.2 (Send-wrapper). The semantics of a send-wrapper call $w = \mathsf{send}[\tau_s, \ell](m, to)$ is defined by fixing the grayed elements in the rule Send to be the wrapper's arguments τ_s, m, ℓ, and to. The wrapper *precondition* $w.pre$ is $\tau_s.pre$ and its result is m.

Definition 3.3 (Receive-wrapper). The semantics of a receive-wrapper call $\mathsf{recv}[T, L]$ is defined by fixing the grayed elements in the rule Recv such that $\ell \in L$ and $\tau_r.tag \in T$ are chosen non-deterministically. The precondition $w.pre$ is True and the result is the pair Some $(from, m)$ from m, if side conditions of Recv are satisfied and there is a message in the soup matching some tag $t \in T$ and a label $\ell \in L$, or None otherwise.

We use the notation $s \overset{w,n}{\rightsquigarrow_W} s'$ to indicate the effect of a wrapper w, executed by a node n in a global system state s, such that $s \Vdash W$, resulting in a new state s'.

Lemma 3.4 (Wrappers obey the network semantics). *Let w be a send- or receive-wrapper call at a node n in a world W, instantiated with valid arguments. Then for any global state s, such that $W \Vdash s$, the resulting state s' of a wrapper execution $s \overset{w,n}{\rightsquigarrow_W} s'$ is computable from s and w, and $s \overset{n}{\rightsquigarrow_W} s'$ holds.*

A program execution in Disel can be thought of as a sequence of wrapper calls. Indeed, in a distributed system, every such execution at a specific node takes place *concurrently* with executions on other nodes, which will typically result in multiple possible outcomes for the global state s. To account for all such behaviors experienced by a program e running locally, we adopt the

trace-based approach for semantics of sequentially-consistent concurrent programs [Brookes 2007]. We define a *denotational* semantics of a Disel command c as a (possibly infinite) set of finite *partial execution traces* $[\![c]\!] = \{\mathsf{t}_\kappa \mid \mathsf{t} = [w_1, \ldots, w_n]\}$, where each element w_i of a trace t is a transition wrapper call or an idle step (corresponding to reading local state) as it occurs during a single, potentially incomplete, sequential execution of c, and $\kappa \in \{\bot, \mathsf{done}\ v\}$, where $\bot$ indicates an incomplete execution of c, and $\mathsf{done}\ v$ stands for a complete execution returning a result value v. Thus, a trace t is generated by a program running at a node, so each of its element corresponds to a single, possible idle, transition, changing the global system state. Since all composite commands in Disel preserve monotonicity in the complete lattice of sets of traces, the semantics of a recursive procedure is defined as the least fixed point of the corresponding functional by the Knaster-Tarski theorem. That is, Disel programs are not directly executable within Coq, but are rather extracted into the corresponding OCaml definitions, as we will outline in Section 5.

To give semantics for the Hoare types and formulate a type soundness result, we need several auxiliary definitions, relating program traces and system states. Those are directly inspired by modern concurrency logics [Ley-Wild and Nanevski 2013; Nanevski et al. 2014], and we refer the reader to our Coq code for fully formal definitions. We first define *interference-reachable* states from a system state s with respect to a node n:

Definition 3.5. A state s' is interference-reachable from s *wrt.* a node n (denoted by $s \overset{\neg n*}{\rightsquigarrow}_W s'$) *iff* $s = s'$ or there exist s'', $n' \neq n$, such that $s \overset{n'}{\rightsquigarrow}_W s''$ and $s'' \overset{\neg n*}{\rightsquigarrow}_W s'$.

We next define Q*-satisfying safe* traces *wrt.* a node n, state s, and an assertion Q, as traces executing from s to the end under interference, so the final state and the result satisfy Q:

Definition 3.6. A trace t_κ is *post-safe* for n, s and Q *iff* either

- $\mathsf{t} = [\,]$, $\kappa = \mathsf{done}\ v$ and $\forall s', s \overset{\neg n*}{\rightsquigarrow}_W s' \implies s' \vDash [v/\mathsf{res}]Q$, or
- $\mathsf{t} = w :: \mathsf{t}'$, and for any s', such that $s \overset{\neg n*}{\rightsquigarrow}_W s'$, the state s' satisfies $w.pre$, and for any s'', such that $s' \overset{w,n}{\rightsquigarrow}_W s''$, t'_κ is post-safe for n, s'' and Q.

Finally, we define *well-typed* programs via our denotational semantics and post-safe traces.

Definition 3.7 (Hoare Type Semantics). $W \overset{n}{\vdash} c : \{P\}\{Q\}$ *iff* for *any* s, such that $s \vDash P$, and for *any* trace $\mathsf{t}_\kappa \in [\![c]\!]$, such that $\kappa = \mathsf{done}\ v$, the trace t_κ is post-safe for n, s and Q.

Definition 3.7 implicitly incorporates *fault-avoidance* (safety) into the semantics of a type: if a program can be assigned a type, it will safely run from a state satisfying its precondition till the end or diverge, with each wrapper in its trace being able to execute, and the final state satisfying the postcondition. Our implementation comes with a number of lemmas, allowing one to reduce a derivation of a Hoare type for a composite program c to those of its components, corresponding precisely to inference rules (*cf.* Fig. 8) in program logics. The proofs of those lemmas with respect to the denotational semantics $[\![\cdot]\!]$ of specific programming constructs deliver the soundness result of Disel as a logic:

Theorem 3.8 (Soundness of Disel logic). *If the type* $\emptyset; W \overset{n}{\vdash} c : \{P\}\{Q\}$ *can be derived in* Disel*, the program* c *satisfies the spec* $W \overset{n}{\vdash} c : \{P\}\{Q\}$ *according to Definition 3.7.*

Definition 3.7 of a type incorporates interference, hence the stability obligations in the premises of the rules for the basic commands, such as SendWrap, ReceiveWrap. While the logic does not enforce the stability of a precondition imposed by the client at each proof rule (as those can be strengthened arbitrarily), it is *impossible* to prove an unstable postcondition (as those can be only weakened). Since having an non-stable precondition P *wrt.* a node n means an inconsistent

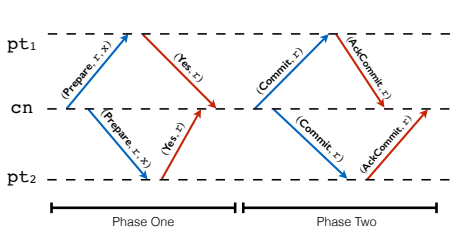

Fig. 10. One round of the Two-Phase Commit.

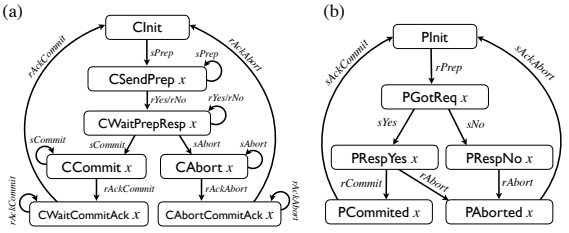

Fig. 11. States of a coordinator (a) and a participant (b).

specification (*i.e.*, $s \vDash P \wedge s \rightsquigarrow^{\neg n *}_{W} s' \wedge s' \vDash P \Rightarrow \mathsf{False}$), it will not be possible to invoke a subroutine with a non-stable precondition within any large consistently specified program context. In order to avoid unsoundness with respect the "topmost" calls, which are extracted and executed on a shim as the end programs in a trusted (*i.e.*, unverified) environment, we require the user to establish stability of their preconditions, which should hold over the initial state, used to initialize the network. For instance, this is the case for the Hoare specifications of the calculator servers from Section 2.4, whose preconditions mention only the node-local state and are, thus, stable.

4 CASE STUDY: TWO-PHASE COMMIT AND ITS CLIENT APPLICATION

We now present a case study: an implementation and verification in Disel of the basic distributed Two-Phase Commit algorithm (TPC) [Weikum and Vossen 2002, Chapter 19]. TPC is widely used in distributed systems to implement a centralized consensus protocol, whose goal is to achieve agreement among several nodes about whether a transaction should be committed or aborted (*e.g.*, as part of a distributed database). Since the system may execute in an asynchronous environment where message delivery is unreliable and machines may experience transient crashes, achieving agreement requires care.

The goal of conducting this exercise for us was twofold: (a) to show that the protocol properties established for systems in the distributed systems community (*e.g.*, consensus) are useful for Hoare-style reasoning about program composition and (b) to demonstrate that Disel's protocols with disjoint state-space and hooks are sufficient for conducting modular proofs about core algorithms (*e.g.*, TPC) and their client applications. To give a better taste of Disel-style programming and verification, in this section we abandon mathematical notation and show fragments of our development taken, with cosmetic adjustments, from our code.

4.1 The Protocol: Intuition and Formalization

The Two-Phase Commit protocol designates a single node as the *coordinator*, which is in charge of managing the commit process; other nodes participating in the protocol are *participants*. The protocol proceeds in a series of rounds, each of which makes a single decision. Each round consists of two phases; an example round execution is shown in Fig. 10. In phase one, the coordinator begins processing a new transaction by sending Prepare messages to all participants. Each participant responds with its local decision Yes or No. In the figure, both participants vote Yes, so the coordinator enters phase two by sending Commit messages to all participants, informing them of its decision to commit. If some participant had voted No, the coordinator would instead send Abort messages. In either case, participants acknowledge the decision by sending AckCommit or AckAbort to the coordinator. When the coordinator receives all acknowledgments, it knows that all nodes have completed the transaction.

```
Definition c_send_step (r: round) (cs: CState)
          (log: Log) (to: node) := match cs with
  (* Sending prepare-messages *)
  | CSentPrep x tos ⇒ if (* sent all messages *)
       (* switch for receiving responses *)
       then (r, CWaitPrepResp x [::], l)
       (* keep sending requests *)
       else (r, CSentPrep x (to :: tos), l)
  (* ...more cases depending on cs and to... *)
  end.
```

```
Definition c_recv_step (r : round) (cs : CState)
      (log : Log) (tag : nat) (mbody : seq nat) :=
  match cs with
  (* Waiting for prepare-responses *)
  | CWaitPrepResp x ⇒ if (* received all votes *)
      then (r, if (* all votes yes *)
               then CCommit x
               else CAbort x, log)
      else (r, CWaitPrepResp, log)
  (* ...more cases depending on cs, tag, mbody... *)
  end.
```

Fig. 12. Send and receive transitions of a coordinator in a Disel definition of the TPC protocol.

The component of the coherence predicate constraining the local state `l` (expressed via Coq/Ssreflect predicate notation `[Pred l | ...]`) of each node `n` depending on its role, coordinator or a participant, is defined as follows:

```
Definition localCoh (n: nid) := [Pred l |
  if n == cn then ∃(r: round) (s: CState) (log: Log), l = st ↦ (r, s) ⊎ lg ↦ log
  else if n ∈ pts
       then ∃(r: round) (s: PState) (log: Log), l = st ↦ (r, s) ⊎ lg ↦ log else True].
```

According to the predicate `localCoh`, the local state of the coordinator (`cn` is a parameter bound at the level of the protocol description) consists of two globally defined locations, `st` and `lg`, which together store a round number `r`, a *coordinator status* `s`, and a `log`. The state of a participant (`n ∈ pts`) is similar, except that its status is a *participant status*. Finally, any node which is not the coordinator or a participant (*e.g.*, a node participating only in other protocols) may have an arbitrary local state with respect to `TPC`.

The coordinator's status can be in any of the seven states shown in shown in Fig. 11(a). Between rounds, the coordinator waits in the CInit state. From the initial state, the coordinators enters the CSentPrep phase and remains in it until all prepare-requests are sent, after which it switches into the receiving state CWaitPrepResp x for the data x. Upon receiving all response message to the prepare-requests, the coordinator changes either to the commit-state or to the abort-state, notifying all of the participants about the decision and collecting the acknowledgements, eventually returning to the CInit state with an updated log. The participants follow a similar pattern to the coordinators's, except that a participant sends messages to or receives messages only from the coordinator before changing its state.

Fig. 12 shows how to encode a few of the coordinator's transitions. Recall that Disel transitions are computable functions that describe how to update the local state of the node when executing the transition. The figure shows the snippets of Disel code related to sending a prepare-request messages and receiving a corresponding response message from participants. In the latter case, depending on the responses, once all of them are collected, the coordinator switches to either CCommit or CAbort state.

4.2 Program Specification and Implementation

With the protocol in hand, we can now proceed to build programs that implement the coordinator and participant and assign them useful Hoare-style specifications. An implementation of a single round of the coordinator and its Hoare type are shown in Fig. 13. The function `coordinator_round` takes as an argument the transaction data to be processed in this round. The type `{r log} DHT [cn, TPC] (...)` represents a Hoare spec, whose logical variables are `r` and `log`. The spec is parametrized by the dedicated coordinator node id `cn` and a world with a single protocol instance `TPC`, with no hooks. The pre/postconditions (in parentheses) are encoded as Coq

```
Definition coordinator_round (d : data) :
{r log}, DHT [cn, TPC]
(fun s ⇒ loc cn s = st ↦ (r, CInit) ⊎ lg ↦ log,
 fun res s' ⇒
 loc cn s' = st↦(r+1, CInit) ⊎ lg↦(log++[(res, d)]))
  := Do (r ← read_round;
         send_prep_loop r d;;
         res ← receive_prep_loop r;
         b ← read_resp_result;
         (if b then send_commits r d;;
                    receive_commit_loop r
               else send_aborts r d;;
                    receive_abort_loop r);;
          return b).
```

Fig. 13. Spec and code of a coordinator round.

```
Definition run_coordinator (data_seq : seq data) :
DHT [cn, _]
(fun s ⇒ s = loc cn s = st ↦ (0, CInit) ⊎ lg ↦ [::]
 fun _ s' ⇒ ∃ (choices : seq bool),
   let r := size data_seq in
   let lg := zip choices data_seq in
   loc cn s' = st ↦ (r, CInit) ⊎ lg ↦ log ∧
   ∀ pt, pt ∈ pts →
         loc pt s' = st ↦ (r, PInit) ⊎ lg ↦ log)
 := Do (with_inv TPCInv (coordinator data_seq)).
```

Fig. 14. Coordinator spec elaborated with TPCInv.

functions `fun s ⇒...` and `fun res s' ⇒...`, correspondingly, so the immediate pre/post-states `s`/`s'` are made explicit, similarly to using the connective this s.

The precondition, which makes use of the *local state getter* `loc cn s = ...`, equivalent to the connective $\mathsf{cn} \overset{\mathsf{TPC}}{\longmapsto} \ldots$ from Fig. 7, requires that the coordinator is in the CInit state, with an arbitrary round number and log. The postcondition ensures that the local state has returned to CInit, the round number has been incremented, and the return value accurately reflects the decision made on the data, which is also reflected in the updated log. The code proceeds along the lines required by the protocol: it reads the round number from the local state, sends requests, collects the responses and then, depending on the locally stored result `b`, sends commit/abort messages, collecting the acknowledgements from participants.

4.3 Protocol Consistency and Inductive Invariant

The spec given to `coordinator_round` in Fig. 13 only constrains the local state `loc` of the coordinator, but in fact the protocol maintains stronger global invariants. For example, we might like to conclude that between rounds, all logs are in agreement. This strong global agreement property is not implied by the coherence predicate given above, so we must prove an inductive invariant that implies it. Finding such inductive invariants is the art of verification, and the process typically requires several iterations before converging on a property that is inductive and implies the desired spec. Tools such as Ivy [Padon et al. 2016] make the process of finding an inductive invariant much more pleasant by providing automatic assistance in debugging and correcting invariants, and it would be interesting to connect Disel to Ivy, which we leave to the future work.

In this case, an invariant that closely follows the intuitive execution of the protocol (its formulation can be found in our Coq files) suffices to prove the global log agreement property. For example, when the coordinator is in the CSendCommit state, the invariant ensures that all participants are either waiting to hear about the decision, have received the decision but not acknowledged it, or have acknowledged the decision and returned to the initial state. The invariant also implies a simple statement of *global log agreement*, shown below:

```
Lemma cn_log_agreement d r log pt : loc cn d = st ↦ (r, CInit) ⊎ lg ↦ log →
       coh d → TPCInv d → ∀ pt, pt ∈ pts → loc pt d = st ↦ (r, PInit) ⊎ lg ↦ log.
```

In other words, a coordinator `cn` in the CInit state and a round `r` can conclude that *all* participants `pt ∈ pts` have also reached the current round `r` and have logs equal to its own.

Putting the inductive invariant to work. We can freely use the elaborated invariant in proofs of programs. Fig. 14 shows a coordinator program that executes a series of rounds based on a given list `data_seq` of data elements. Its postcondition asserts that all participants have finished

```
Program Definition run_and_query (ds : seq data) pt :
  {reqs resp}, DHT [cn, (TPC ⊎ Query, QHook)]
  (fun s ⇒ loc s = st ↦ (0, CInit) ⊎ lg ↦ [::] ∧
            pt ∈ pts ∧ query_init s],
   fun (res : nat * Log) s' ⇒ ∃ (chs : seq bool),
     let d := (size ds, zip chs ds) in
     loc s' = st ↦ (d.1, CInit) ⊎ lg ↦ d.2 ∧
     query_init s' ∧ res = d)
  := Do (run_coordinator ds;;
         rid ← generate_fresh_request_id pt;
         send_request rid pt;;
         res ← receive_responce rid pt;
         return res).
```

Fig. 15. Querying after the TPC coordinator.

```
Parameter core_state : Data → LocState → Prop.
Parameter local_indicator : Data → LocState → Prop.

Definition QHook := (1, lab_c, lab_q, resp) ↦
 fun lc lq m to ⇒
 ∀ rid data, m = rid :: serialize data →
             core_state data lc.

Hypothesis core_state_inj :
  ∀ l d d', core_state d l →
            core_state d' l → d = d'.

Hypothesis core_state_step : ∀ data s s' n1 n2,
  n1 != n2 → local_indicator data (loc lab_c n1 s)
  → network_step (lab_c ↦ pc, ∅) n2 s s'
  → core_state data (loc lab_c n2 s').
```

Fig. 16. Hook definition and abstract predicates.

the round and have logs agreeing with the one of the coordinator. The proof of this specification is by a straightforward application of the WithInv rule, making use of the elaborated invariant `TPCInv` as well as the lemma `cn_log_agreement`. Importantly, the postcondition is stable, because each round of the Two-Phase Commit begins with a coordinator's move, hence no participant can change its state from the "initial" one while the coordinator's status is CInit.

4.4 Composing Two-Phase Commit with a Querying Application using Hooks

Even though core consensus protocols, such as TPC, are not designed to exist in isolation, but rather to be used in a context of larger applications (*e.g.*, for crash recovery), formal reasoning about *client-specific properties* (*i.e.*, properties of applications relying on certain characteristics of a "core" distributed protocol) is only barely covered in classical textbooks [Weikum and Vossen 2002] and, with a rare exception [Lesani et al. 2016], almost never a focus of major verification efforts [Hawblitzel et al. 2015; Rahli et al. 2015; Woos et al. 2016], which, therefore cannot be reused in any larger verified context.

We now demonstrate how to employ Disel's logical mechanisms for restricted composition of protocols in order to prove, in a modular fashion, properties of client code from a core protocol's invariants. To do so, we verify a composite application, which uses TPC for building a replicated log of data elements, and a *side-channel protocol* for sending independent queries about the state of TPC participants (*e.g.*, for the purpose of implementing recovery after a coordinator's failure). Fig. 15 shows a program that first calls the coordinator program `run_coordinator`, and then uses the side protocol to query the local state of a participant `pt`, which the program then returns as its final result `res`. Ignoring the `query_init` part in the pre/postcondition for now, notice that the postcondition asserts that `res` is *equal to* the pair `d` (round, log) stored in the local state of the coordinator (which did not crash this time)!

Establishing such validity of the query *wrt.* TPC-related state is, however, not trivial at all, given how the querying protocol is defined. The protocol `Query` is very similar to the calculator from Section 2: any node n_1 in it can send a request to any other node n_2, to which n_2 may respond with *any* arbitrary message (the details of the formal protocol definition can be found in our Coq code). This protocol definition is intentionally made very weak: while it allows one to prove some interesting inductive invariants (*e.g.*, no request is answered twice), it leaves all other interaction aspects for the final client to specify. In particular, it *does not* enforce any specific shape of data being sent in a response to a request.

Thus, without imposing the additional restriction that the protocol `Query` *can only transmit the local state of a node wrt.* `TPC`, we will not be able to prove the spec in Fig. 15. The necessary restriction is provided by a send-hook entry `QHook` that is used when composing the protocols `TPC` and `Query` in the spec of `run_and_query`, and is defined in Fig. 16.

In order to make the client verification effort reusable in the context of *any* consensus protocol, not just `TPC`, we formulate the hook statement in terms of an abstract type `Data` and an *abstract predicate* `core_state`, which we will later instantiate specifically for `TPC`, both afforded by Coq's higher-order programming capabilities. The hook enforces that any message `m` containing a request id `rid` and serialized `data` adequately encodes the current local state (storing `data`) of the sender node, at the moment of sending `m`, with respect to the protocol with label `lab_c`. The abstract predicate `core_state d lc`, capturing precisely this "adequacy of the encoding", is supplied with the injectivity hypothesis `core_state_inj` (to be proved by each consensus implementation), which ensures that the abstract data representation is unambiguous.

We also declare an abstract predicate `local_indicator` and the corresponding hypothesis `core_state_step`, which essentially corresponds to *irrevocability* of consensus and should be proved for each consensus implementation (in particular, for `TPC`), ensuring that if a local state of a node `n1` is of certain shape `data`, the local state of `n2`, captured by `core_state data` will be remaining *the same* under interference (`network_step`) *wrt.* the core `lab_c`-labelled protocol `pc`—precisely what is ensured by the lemma `cn_log_agreement` of `TPC`.

Finally, we can use the abstract predicates from Fig. 16 to provide specifications for querying procedures from Fig. 15, stating `query_init` in terms of assertions involving `local_indicator` and `query_state`, in the context *parameterized* over a "core" consensus protocol `pc` and restricted with `QHook`. To verify the program in Fig. 15 against the desired spec we only need to instantiate the predicates as follows and prove the corresponding hypotheses for `TPC`, which follow from the invariant `TPCInv` and Lemma `cn_log_agreement`:

```
(* For TPC, abstract Data type is instantiated with a round number (nat) and Log. *)
Definition Data := nat * Log.
Definition local_indicator (d : Data) l := l = st ↦ (d.1, CInit) ⊎ log ↦ d.2.
Definition core_state (d : Data) l := l = st ↦ (d.1, PInit) ⊎ log ↦ d.2.
```

The rest of the proof is via the Frame rule with $W = \langle \texttt{TPC}, \emptyset \rangle$, $C = \texttt{Query}$ and $H = \texttt{QHook}$. Since `QHook` does not restrict the transitions of `TPC`, NotHooked holds. Thanks to the parametrization of querying programs with abstract predicates and hypotheses from Fig. 16, we can compose them with any other instance of a consensus protocol, *e.g.*, Paxos [Lamport 1998b] or Raft [Ongaro and Ousterhout 2014], thus, reusing the proofs of their core invariants.

5 IMPLEMENTATION AND EXPERIENCE

Disel combines two traits that rarely occur in a single tool for reasoning about programs. First, thanks to the representation of Hoare types by means of Coq's dependent types, the soundness result of Disel scales not just to a toy core calculus, but to the entirety of Gallina, the *programming language* of Coq, enhanced with general recursion and message-passing primitives. Second, Disel programs are immediately *executable* by means of extracting them into OCaml, which provides the features that Gallina lacks: general fixpoints, mutable state, and networking constructs, enabled by our trusted shim implementation.

Formal development and proof sizes. The size of our formalization of the metatheory, inference rules and soundness proofs is about 4500 LOC. Our development builds on well-established Ssreflect/MathComp libraries [Gonthier et al. 2009; Mahboubi and Tassi 2017; Sergey 2014] as well as on the implementation of partial finite maps and heap theory by Nanevski et al. [2010].

Table 1 summarizes the proof effort for the calculator, `TPC/Query` systems. The Defs/Specs column measures all *specification* components, including, *e.g.*, auxiliary predicates, whereas Impl reports the sizes of actual Disel programs. Due to the high degree of code reuse, it is difficult to provide separate metrics in some cases; for those parts we only report the joint numbers. Although Disel is not yet a production-quality verification tool, safety proofs of interesting systems can be obtained in it in a reasonably short period of time and with moderate verification effort (*e.g.*, the full development of the core `TPC` system took nine person-days of work). Given that the current version of Disel employs no advanced proof automation, beyond what is offered by Coq/Ssreflect, for discharging program-level verification conditions [Chlipala 2011] or inductive invariant proofs [Padon et al. 2016], we consider these results encouraging for future development.

Table 1. Statistics for implemented systems: sizes of protocol definitions/specs, programs, proofs of protocol axioms/invariants/specs (LOC), and build times (sec).

Component	Defs/Specs	Impl	Proofs	Build
Calculator (§2)				
protocol (§2.1) Inv$_1$ (§2.3) Inv$_2$ (§2.4)	239	-	243	4.8
simple_server (§2.3) batch_server (§2.4) memo_server (§2.4)	192	43	153	8.6
compute (§2.4)	120	24	99	4.8
deleg_server (§2.4)	75	7	49	2.4
Two-Phase Commit (§4.1–§4.3)				
protocol (§4.1)	465	-	231	3.9
coordinator (§4.2)	236	35	440	18
participant (§4.2)	163	24	198	10
`TPCInv` (§4.3)	997	-	2113	25
Query/TPC (§4.4)				
protocol	169	-	115	2.1
querying procedures	326	18	707	19
`run_and_query`	76	5	89	2.6

Extraction and execution. Disel's logic reasons about programs in terms of their denotational semantics as traces, but each primitive also has a straightforward operational meaning. For example, executing a wrapped send transition should actually send the corresponding network message. Thus it is relatively straightforward to extract Disel programs by providing OCaml implementations of the primitive operations in a trusted shim. Our shim consists of about 250 lines of OCaml, including primitives for sending and receiving messages and general recursion. The local state of each node is implemented as a map from protocol labels to heaps, where a heap is implemented as a map from locations to values. Since Disel does not draw a distinction between real and auxiliary state so far, both are manifested at run time. In the future, we plan to allow users to mark state as auxiliary to improve performance. Due to artifacts of the extraction process, a Disel program that appears tail-recursive at the Coq source level does not extract to a tail-recursive OCaml program. This causes long running loops (such as those typically used to implement blocking receive) to quickly blow the OCaml stack. To circumvent this issue, we added a while-loop combinator to Disel, which is encoded using the general fixpoint combinator, but is extracted to an efficient OCaml procedure that uses constant stack space. Our implementations of the calculator and `TPC` use this while-loop combinator to implement blocking receive.

In this work, our goal was not to extract high-performance code for Disel programs, but rather show that, with a careful choice of low-level primitives with precise operational meaning, such extraction is feasible and requires a very small trusted codebase.

Adequacy of the extraction. What is the correspondence between our denotational semantics, presented in Section 3.3 and the operational one implemented by our shim? While in this work we do not state a fully formal correspondence, as the shim is written in OCaml and uses operating system and network components, which have no formal semantics, we argue that the extraction is adequate *wrt.* the denotational semantics for the following reasons:

(1) Our denotational semantics is simply a trace-collecting operational semantics for interleaved, asynchronous, message-passing concurrency, with the shared message soup being the only communication medium. Such an operational representation is widely considered adequate for modelling distributed systems and has been employed and evaluated (also, without verifying the extraction) in previous works [Hawblitzel et al. 2015; Padon et al. 2016; Wilcox et al. 2015].

(2) The shim implementation follows the operational rules from Fig. 9 verbatim, and protocol transitions are encoded in Disel as *functions* on the local state, so they are easy to extract and execute. The shim, thus, provides an accurate implementation of the protocol-aware network semantics.

Our fixpoint definition (available in our Coq sources) admits non-terminating executions, "approximating" them iteratively by sets of incomplete post-safe traces. It is extracted into OCaml's general fixpoint operator, with a somewhat ad-hoc tail-call optimisation described above in this Section. This means that our logic proves only partial correctness: verified programs may loop at runtime, but they will never violate the protocol.

Information hiding and separation. One might wonder, whether we can *hide* implementation-specific parts of local state from the clients, *e.g.*, when reasoning about other nodes' implementations? At the moment any mutable state in Disel should be manifested *in a protocol definition* (and, thus, *known* to all its users) and can be only altered by sending/receiving. This is why in the examples, such as the memoizing calculator from Section 2.4, we model hidden state by passing a functional argument. However, what the framework *does allow* one to do is to encode an auxiliary protocol implementing a mutable storage, which, once joined (via $\uplus$) with its client protocol (*e.g.*, calculator), *does not* have to be exposed to the clients of the latter one, similarly to how it is done in the delegating calculator example.

To support a version of a "proper" hidden local mutable state (*i.e.*, a heap with mutable pointers) we would need to formulate a nested program logic with the corresponding low-level semantics for state-manipulating programs—a direction we consider as interesting future work, with an idea of adopting for this role Verifiable C by Appel et al. [2014].

6 RELATED AND FUTURE WORK

6.1 Program Logics for Concurrency

Disel builds on many ideas from modern program logics for compositional concurrency reasoning. The notion of protocols (often called *regions*) in shared-memory concurrency logics [Dinsdale-Young et al. 2010; Nanevski et al. 2014; Raad et al. 2015; Svendsen and Birkedal 2014; Turon et al. 2014, 2013] provides a "localized" version of more traditional Rely/Guarantee obligations [Jones 1983], which, in their original formulation, are not modular [Feng 2009; Feng et al. 2007; Vafeiadis and Parkinson 2007]. The two closest to Disel logics employing protocols to reason about interference are FCSL by Nanevski et al. [2014] and GPS by Turon et al. [2014]. Besides those being logics for *shared-memory*, rather than *message-passing* concurrency, protocols in FCSL and GPS are tailored for the notion of *ownership transfer* [O'Hearn 2007], as a way to express exclusivity of access to shared resources. Due to the lack of *immediate* synchronization between nodes in a message-passing setting, we consider the notion of ownership to be of less use for most of the systems of interest. That said, even though Disel does not feature explicit ownership transfer, it can be easily encoded on a per-protocol basis, by defining a suitable local state and transitions.

Composition of modular proofs about protocols is a problem that has not received much attention in modern concurrency logics. In FCSL, which tackles a similar challenge, in order to constrain inter-protocol interaction, a user must set up her protocols with a very specific foresight of how they are

going to be composed with other protocols, defining *intrinsic* "ownership communication channels" for *all* involved components, thus, effectively prohibiting unforeseen interaction scenarios. This is not the case in Disel: as we have shown in Section 4, "core" and "client" protocols (*e.g.*, `TPC` and `Query`) can be developed and verified independently and then composed in joint applications via *extrinsic* client-specified send-hooks.

The recent logical framework Iris [Jung et al. 2016, 2015] suggests to express protocols as a specific case of resources, represented, in general, by partial commutative monoids, viewing *state reachability* as a specific instance of *framing* [Reynolds 2002]. This generality does not buy much for verifying distributed applications, as the resulting proof obligations are the same as when proving inductive invariants. Having an *explicit* notion of protocols in the logic, though, allowed us to provide the novel protocol-tailored rules WithInv and Frame (*cf.* Fig. 8), which enabled modular invariant proofs and distributed systems composition.

A related logic by Villard et al. [2009] only considers protocols associated with specific message-passing channels, rather than entire distributed systems. In Villard et al.'s logic, messages do not carry any payload: they are simply *tags*, indicating ownership transfer of a certain heap portion in the *same* shared memory space. It is not immediately obvious how to use Villard et al.'s specifications for *locally* asserting *global* properties of stateful distributed systems (e.g., the agreement of `TPC` in Fig. 14) without considering all involved processes. In addition to that, Villard et al.'s logic does not provide a mechanism for establishing inductive contract invariants. A recent framework *Actor Services* by Summers and Müller [2016] provides abstractions similar to our protocol transitions, but only allows to state *local* actor invariants, and lacks a formal metatheory and soundness proof.

To the best of our knowledge, none of the existing concurrency logics features *both* foundational soundness proof (*i.e.*, the proof that the entire logic, not just its toy subset, is sound as a verification tool), *and* a mechanism to extract and run verified applications.

6.2 Types for Distributed Systems

Session Types [Honda et al. 1998] are traditionally used to ensure that distributed parties follow a predefined communication protocol *wrt.* a specific channel. While the *multiparty* [Honda et al. 2008] and *multirole* [Deniélou and Yoshida 2011] Session Types enable a form of system composition and role-play, and *dependent session types* allow one to quantify over messages [Toninho et al. 2011], session types do not allow quantification over the global system state and reasoning out of inductive invariants, neither do they allow restricted composition of protocols.

We believe that Disel's combination of Hoare types and protocols provides the necessary level of expressivity to capture rich safety properties of distributed applications. A similar approach has been explored in F* by Swamy et al. [2011], although that work did not reason about inductive invariants separately from implementations, neither did it address composition of systems with inter-protocol dependencies.

6.3 Verification of Large Systems

Recent work has verified implementations of core pieces of distributed systems infrastructure, both by using specialized models and DSLs.

IronFleet [Hawblitzel et al. 2015] supports proving *liveness* in addition to safety, all embedded in Dafny [Leino 2010]. IronFleet focuses on layered verification of standalone monolithic systems. In those systems, each layer is a state-transition system (STS) specifying the system's behavior at a certain abstraction level, with the top-most layer expressing how a collection of nodes together implement a high-level (*e.g.*, shared-memory) specification, and the actual implementation, run by the nodes, at the bottom. Adjacent layers are connected by establishing refinement between their STSs via reduction [Lipton 1975], which often involves proving inductive invariants, similar to

what we have proven in Disel. In our understanding, such specifications do not allow for horizontal composition, *i.e.*, reasoning about interaction with separately verified systems in a client code. Such an interaction has been, however, explored *wrt. shared-memory* concurrency by Gu et al. [2016], who built a series of abstraction layers in a verified concurrent OS kernel. That work has shown that establishing a refinement between a spec STSs and a *family* of interacting lower-level STSs is possible, although the proofs are usually quite complex, as they involve reasoning about *semantics* of a restricted product of STSs. In contrast with those systems, Disel's logic does not provide machinery to establish STS refinement, but rather explicitly identifies valid linearization points [Herlihy and Wing 1990] in the implementations, as they correspond precisely to taken protocol transitions. Abstract specifications and the corresponding system properties, usable by client code, such as consensus, are encoded in Disel via parametrized Hoare types and abstract predicates, as shown in Section 4.4.

Verdi [Wilcox et al. 2015; Woos et al. 2016] provides a form of *vertical compositionality* by means of verified system transformers, which allow systems to be decomposed into layers of functionality (*e.g.*, sequence numbers or state machine replication). The Chapar framework by Lesani et al. [2016] is tailored to causally consistent key-value stores, and also provides verified model checking for client programs using the verified KV stores. Ivy is a tool to assist users in iteratively discovering inductive invariants by finding counterexamples to induction [Padon et al. 2016]. PSync by Dragoi et al. [2016] is a DSL allowing one to prove inductive invariants of consensus algorithms in networks with potential faults, operating in a synchronous round-based model [Elrad and Francez 1982]. This assumption enables efficient proof automation, but prohibits low-level optimizations, such as, *e.g.*, batching. Mace by Killian et al. [2007] and DistAlgo by Liu et al. [2012] adopt an asynchronous protocol model, similar to ours. Mace provides a suite of tools for generating and model checking distributed systems, while DistAlgo allows extraction of efficient implementation from a high-level protocol description. EventML is another DSL for verifying monolithic distributed systems, based on compiling to the Logic of Events in Nuprl [Rahli et al. 2015]. None of these frameworks tackles the challenges of modular reasoning about horizontally composed systems **(2)** and elaborated protocols **(3)**, stated in the introduction of this paper.

Arguably, our Two-Phase Commit implementation is a relatively small case study when compared to the systems verified in IronFleet, Verdi, and EventML. Nevertheless, we are sure that, given enough time and manpower, we can conduct safety proofs of Raft [Ongaro and Ousterhout 2014] and MultiPaxos [van Renesse and Altinbuken 2015] in Disel, as their implementations and invariants are based on the same semantic primitives and reasoning principles that were employed for TPC. We believe, though, that compositionality, afforded by Disel's logical mechanisms, is a key to make the results of future verification efforts reusable for building even larger verified distributed ecosystems.

6.4 Future Work

We consider Disel as just the beginning of our journey towards building modularly verified and highly reusable distributed implementations. Our next steps are to investigate more protocol combinators, in addition to WithInv, establishing refinement, in the spirit of certified abstraction layers [Gu et al. 2016], of the higher-level distributed models (*e.g.*, round-based register by [Boichat et al. 2003]) by current Disel's protocols, formulated in terms of send/receive transitions. We are also going to expand the language fragment for local node implementations with more imperative features, such as exceptions and concurrency. We are planning to incorporate the ideas from program logics to reason, in a modular way, about local system faults [Ntzik et al. 2015] and liveness properties under fairness assumptions [Liang and Feng 2016]. Finally, we are going to investigate possibilities for automating proofs of inductive invariants [Padon et al. 2017, 2016].

7 CONCLUSION

Almost two decades ago, Lamport [1998a] propounded the thesis **Composition: a way to make proofs harder**, favoring mathematical models over program logics for real system verification: "*in 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build.* [...] *It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years*". He was right: it took two decades of active research in rigorous program verification, combining the strengths of mathematical models (protocols) and program logics, to make compositional verification of open-world systems today's reality.

ACKNOWLEDGMENTS

We thank Philippa Gardner, Alexey Gotsman, Yoichi Hirai, Ranjit Jhala, Luke Nelson, Karl Palmskog, Daniel Ricketts, Doug Woos, and Nobuko Yoshida for their comments on earlier drafts of this paper. We are grateful to the PLDI'17 reviewers, especially Reviewers C and E, for their feedback regarding insufficient support for modularity in an earlier version of Disel, which forced us to revise the approach and introduce the notion of send-hooks. We wish to acknowledge the feedback by the OOPSLA'17 reviewers on the presentation. We thank the POPL'18 PC and AEC reviewers for the careful reading and many constructive suggestions on the paper and the implementation. Finally, we thank Éric Tanter for his dedication to bring out the best of the paper as our shepherd, and Andrew C. Myers for his efforts as POPL'18 PC chair.

Sergey's research was supported by EPSRC First Grant EP/P009271/1 "Program Logics for Compositional Specification and Verification of Distributed Systems". Tatlock's research was supported by a generous gift from Google.

REFERENCES

Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *LICS*. IEEE Computer Society, 165–175.

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing* 9, 1 (1995), 37–49.

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *LICS*. IEEE Computer Society, 247–256.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press.

Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. 2003. Deconstructing paxos. *SIGACT News* 34, 1 (2003), 47–67.

Stephen Brookes. 2007. A semantics for concurrent separation logic. *Th. Comp. Sci.* 375, 1-3 (2007).

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. ACM, 234–245.

Coq Development Team. 2017. *The Coq Proof Assistant Reference Manual - Version 8.6.*

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *POPL*. ACM, 435–446.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP (LNCS)*, Vol. 6183. Springer, 504–528.

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*. ACM, 400–415.

Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.* 2, 3 (1982), 155–173.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. ACM, 315–327.

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *ESOP (LNCS)*, Vol. 4421. Springer, 173–188.

Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2009. *A Small Scale Reflection Extension for the Coq system.* Technical Report 6455. Microsoft Research – Inria Joint Centre.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM, 595–608.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*. USENIX Association, 653–669.

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 *SOSP*. ACM, 1–17.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (LNCS)*, Vol. 1381. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. ACM, 256–269.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650.

Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *PLDI*. ACM, 179–188.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115.

Thomas Kleymann. 1999. Hoare Logic and Auxiliary Variables. *Formal Asp. Comput.* 11, 5 (1999), 541–566.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. ACM, 179–192.

Leslie Lamport. 1978. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks* 2 (1978), 95–114.

Leslie Lamport. 1998a. Composition: A Way to Make Proofs Harder. In *Compositionality: The Significant Difference, International Symposium (LNCS)*, Vol. 1536. Springer, 402–423.

Leslie Lamport. 1998b. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS)*, Vol. 6355. Springer, 348–370.

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *POPL*. ACM, 357–370.

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *POPL*. ACM, 561–574.

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *POPL*. ACM, 385–399.

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *OOPSLA*. ACM, New York, NY, USA, 395–410.

Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233.

Assia Mahboubi and Enrico Tassi. 2017. *Mathematical Components*. Available at https://math-comp.github.io/mcb.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP (LNCS)*, Vol. 8410. Springer, 290–310.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and separation in Hoare Type Theory. In *ICFP*. ACM, 62–73.

Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *ICFP*. ACM Press, 229–240.

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *POPL*. ACM, 261–274.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73.

Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. 2015. Fault-Tolerant Resource Reasoning. In *APLAS (LNCS)*, Vol. 9458. Springer, 169–188.

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Th. Comp. Sci.* 375, 1-3 (2007), 271–307.

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*. 305–319.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL* 1, OOPSLA (2017), 108:1–108:31.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. ACM, 614–630.

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *ESOP (LNCS)*, Vol. 9032. Springer.

Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. 2015. Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. In *AVOCS*. EASST.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

Ilya Sergey. 2014. *Programs and Proofs: Mechanizing Mathematics with Dependent Types*. Lecture notes with exercises. Available at http://ilyasergey.net/pnp.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-grained Concurrent Programs. In *PLDI*. ACM, 77–87.

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*. ACM, 275–287.

Alexander J. Summers and Peter Müller. 2016. Actor Services - Modular Verification of Message Passing Programs. In *ESOP (LNCS)*, Vol. 9632. Springer, 699–726.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS)*, Vol. 8410. Springer, 149–168.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *ICFP*. ACM, 266–278.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *PPDP*. ACM, 161–172.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, 691–707.

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*. ACM, 343–356.

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR (LNCS)*, Vol. 4703. Springer, 256–271.

Robbert van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3 (2015), 42:1–42:36.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2009. Proving Copyless Message Passing. In *APLAS (LNCS)*, Vol. 5904. Springer, 194–209.

Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann.

James R. Wilcox, Ilya Sergey, and Zachary Tatlock. 2017. Programming Language Abstractions for Modularly Verified Distributed Systems. In *SNAPL (LIPIcs)*, Vol. 71. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 19:1–19:12.

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 *PLDI*. ACM, 357–368.

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 *CPP*. ACM, 154–165.