Protocol Combinators for Modeling, Testing, and Execution of Distributed Systems

KRISTOFFER JUST ARNDAL ANDERSEN
Google, Aarhus, Denmark
(*e-mail:* `kristoffer@arndalandersen.dk`)

ILYA SERGEY
Yale-NUS College and National University of Singapore, Singapore
(*e-mail:* `ilya.sergey@yale-nus.edu.sg`)

Abstract

Distributed systems are hard to get right, model, test, debug, and teach. Their textbook definitions, typically given in a form of replicated state machines, are concise, yet prone to introducing programming errors if naïvely translated into runnable implementations.

In this work, we present *Distributed Protocol Combinators* (DPC), a declarative programming framework that aims to bridge the gap between specifications and runnable implementations of distributed systems, and facilitate their modeling, testing, and execution. DPC builds on the ideas from the state-of-the art logics for compositional systems verification. The contribution of DPC is a novel family of program-level primitives, which facilitates construction of larger distributed systems from smaller components, streamlining the usage of the most common asynchronous message-passing communication patterns, and providing machinery for testing and user-friendly dynamic verification of systems. This paper describes the main ideas behind the design of the framework and presents its implementation in Haskell. We introduce DPC through a series of characteristic examples and showcase it on a number of distributed protocols from the literature.

This paper extends our preceeding conference publication (Andersen & Sergey 2019a) with an exploration of randomised testing for protocols and their implementations, and an additional case study demonstrating bounded model checking of protocols.

1 Introduction

Distributed fault-tolerant systems are at the heart of modern electronic services, spanning such aspects of our lives as healthcare, online commerce, transportation, entertainment and cloud-based applications. From engineering and reasoning perspectives, distributed systems are amongst the most complex pieces of software being developed nowadays. The complexity is not only due to the intricacy of the underlying protocols for multi-party interaction, which should be resilient to execution faults, packet loss and corruption, but also due to hard performance and availability requirements (Chandra *et al.* 2007).

The issue of system correctness is traditionally addressed by employing a wide range of *whole-system* testing methodologies, with more recent advances in integrating techniques for formal verification into the system development process (Newcombe *et al.* 2015; Hawblitzel *et al.* 2015; Dragoi *et al.* 2016). In an ongoing effort of developing a *verification* methodology enabling the *reuse* of formal proofs about distributed systems in the context of an open world, the DISEL logic, built on top of the Coq proof assistant (Coq Development Team 2020), has been proposed as the first framework for mechanised verification of distributed systems, enabling modular proofs about protocol composition (Wilcox *et al.* 2017; Sergey *et al.* 2018).

The main construction of DISEL is a *distributed protocol* $\mathscr{P}$—an operationally described replicated *state-transition system* (STS), which captures the shape of the state of each node in the system, as well as what it *can* or *cannot* do at any moment, depending on its state. Even though a protocol $\mathscr{P}$ is not an executable program and cannot be immediately run, one can still use it as an *executable specification* of the system, in order to prove the system's intrinsic properties. For instance, reasoning at the level of a protocol, one can establish that a property $I : \mathsf{SystemState} \to \mathsf{Prop}$ is an inductive invariant *wrt.* a protocol $\mathscr{P}$.[1] A somewhat simplified main judgement of DISEL, $\mathscr{P} \vdash c$, asserts that an actual system implementation c will *not* violate the operational specification of $\mathscr{P}$. Therefore, if this holds, one can infer that any execution of a program c, will not violate the property I, proved for protocol $\mathscr{P}$. DISEL also features a full-blown program logic, implemented as a Hoare Type Theory (Nanevski *et al.* 2008), which allows one to ascribe pre- and post-conditions to distributed programs, enforcing them via Coq's dependent types, at the expense of frequently requiring the user to write lengthy proof scripts.

While expressive enough to implement and verify, for instance, a crash-recovery service on top of a Two-Phase Commit (Sergey *et al.* 2018), unfortunately, DISEL, as a systems *implementation* tool, is far from being user-friendly, and is not immediately applicable for rapid prototyping of composite distributed systems, their testing and debugging. Neither can one use it for teaching without assuming students' knowledge of Coq and Separation Logic (O'Hearn *et al.* 2001). Furthermore, system implementations in DISEL must be encoded in terms of low-level $\mathsf{send}/\mathsf{receive}$ primitive, obscuring the high-level protocol design.

In this work, we give a practical spin to DISEL's main idea—disentangling protocol *specifications* from runnable, possibly highly optimised, systems *implementations*, making the following contributions:

- We distil a number of high-level distributed interaction patterns, which are common in practical system implementations, and capture them in a form of a novel family of *Distributed Protocol Combinators* (DPC)—a set of versatile higher-order programming primitives. DPC allow one to implement systems concisely, while still being able to benefit from protocol-based specifications for the sake of testing and specification-aware debugging.
- We implement DPC in Haskell, providing a set of specification and implementation primitives, parameterised by a monadic interface, which allow for multiple interpretations of protocol-oriented distributed implementations.
- We provide a rich toolset for testing, running, and visual debugging of systems implemented via DPC:

[1] Examples of such properties include global-systems invariants, used, in particular, to reason about the whole system reaching a consensus (Pîrlea & Sergey 2018; van Renesse & Altinbuken 2015).

  - visual exploration tools for tracing protocol execution.
  - tools for guided random execution of *implementations*, enabling testing implementations against their protocol specifications as properties in the sense of Claessen & Hughes (2011).
  - a language for expressing protocol invariants, and tools for checking them on the (bounded) state space of the protocol.

- We showcase DPC on a variety of distributed systems, ranging from a simple RPC-based cloud calculator and its variations, to distributed locking (Kleppmann 2016), Two-Phase Commit (Gray 1978), and Paxos consensus (Lamport 1998, 2001).

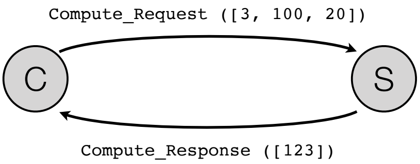
2 Specifying and Implementing Systems with DPC

In this work, we focus on message-passing asynchronous distributed systems, where each node maintains its internal state while interacting with others by means of sending and receiving messages. That is, the messages, which can be sent and received at any moment, with arbitrary delays, drops, and rearrangements, are the only medium of communication between the nodes. DPC takes the common approach of thinking of message-passing systems as shared-memory systems, in which each message in transit is allocated in a virtual shared "message soup", where it lingers until it is delivered to the recipient (Sergey *et al.* 2018; Wilcox *et al.* 2015).

The exact implementation of the *per-node* internal state might differ from one node to another, as it is virtually unobservable by other participants of the system. However, in order for the whole system to function correctly, it is required that each node's behaviour would be at least coherent with some notion of *abstract state*, which is used to describe the interaction protocol.

In the remainder of this section, we will build an intuition of designing a system "top-down". We will start from its specification in terms of a protocol that defines the abstract state and governs the message-passing discipline, going all the way down to the implementation that defines the state concretely and possibly combines several protocols together. For this, we use a standard example of a distributed calculator.

2.1 Describing Distributed Interaction



In a simple cloud calculator, a node takes one of two possible roles: that of a *client* or that of a *server*. A client may send a request along with data to be acted upon to the server (*e.g.*, a list of numbers `[3, 100, 20]` to compute the sum of), and the server in turn responds with the result of the computation, as shown on the diagram on the right. For uniformity of implementation, all message payloads, including the response of the server, are lists of integers. Notice that this description does not restrict *e.g.*, the order in which a server must process incoming requests from the clients, leaving a lot of room for potential optimisations on the implementation side.

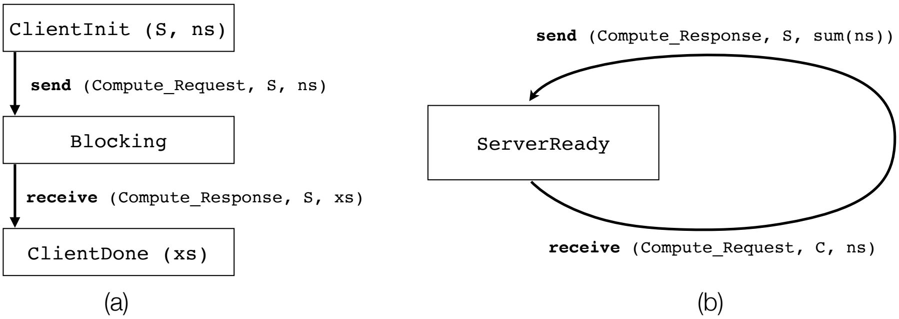

ClientInit (S, ns)
send (Compute_Request, S, ns)
Blocking
receive (Compute_Response, S, xs)
ClientDone (xs)
(a)
send (Compute_Response, S, sum(ns))
ServerReady
receive (Compute_Request, C, ns)
(b)

Fig. 1: State transitions for a client (a) and a server (b) in the calculator protocol.

In order to capture the behavioural contract describing the interaction between clients and servers, we need to be able to outlaw some unwelcome communication scenarios. For instance, in our examples, it would be out of protocol for the server to respond with a wrong answer (in general an issue of safety) or to the wrong client (in general an issue of security). A convenient way to restrict the communication rules between distributed parties is by introducing the *abstract state* describing specific “life stages” of a client and a server, as well as associated messages that trigger changes in this state—altogether forming an STS, a well-known way to abstractly describe and reason about distributed protocols (Lampson 1996; Lamport & Schneider 1985).

Let us now describe our calculator protocol as a collection of coordinated transition systems. The client’s part in the protocol originates in a state `ClientInit` containing the input it is going to send to the server, as well as the server’s identity. From this state, it can send a message to server `S` with the payload `[3, 100, 20]`. It then must wait, in a blocking state, for a response from the server.[2] Upon having received the message, the client proceeds to a third and final state, `ClientDone`. From here, no more transitions are possible, and the client’s role in the protocol is completed. A schematic outline of the client protocol is depicted in Fig. 1 (a).

In our simplified scenario, the protocol for the server (Fig. 1, (b)) can be captured by just one state, `ServerReady`, so that receiving the request and responding to it with a correct result is observed as “atomic” by other parties, and hence, is denoted by a single composite transition. Even though, technically, the server performs two actions (a receive then a send), it is convenient to think of this sequence as of an atomic one, which is the common perception of remote procedure calls. In other words, at the specification level, the server immediately reacts to the request by sending a response.

Notice that the protocol places no demands on the number of clients, servers or unrelated nodes in the network, nor does it restrict the number of instances of the protocol are running in a given network. The specification is ”local” to the parties involved (which in general can number arbitrarily many).

This “request/respond” communication pattern is so common in distributed programming that it is worth making explicit. We will refer to this pattern as a pure *remote proce-*

[2] Remember that this is a specification-level blocking, the implementation can actually do something useful in the same time, just not (observably) related to this protocol!

dure call (RPC) and take it as our first combinator for protocol-based implementation of distributed systems.

2.2 Specifying the Protocol

We can capture the RPC-shaped communication in DPC by first enumerating all possible states of nodes in the protocol in a single data type. For the calculator, the states can be directly translated from the description above to the following Haskell data type:

```
data S = ClientInit NodeID [Int]
       | ClientDone [Int]
       | ServerReady
```

NodeID is a type synonym for Int, but any type with equality would serve. ClientInit contains the name of the server and the list to sum. ClientDone contains the response from the server.[3] Next, we describe the only kind of exchange that takes place in a network of clients and servers communicating by following the RPC discipline. We do so by specifying when a client can produce a request in a protocol, and how the server computes the response, as we will shortly do in code. Perhaps, a bit surprisingly, no more information is needed, as the pattern dictates that clients await responses from servers, and the server responds immediately. This is the reason why we need only enumerate two states for the client, eliding the one for blocking, as per Fig. 1 (a): the framework adds the third during execution by wrapping the states in a type with an additional Blocking constructor.[4] The following definition of compute outlines the specification of the protocol's STSs:

```
compute :: Alternative f ⇒ ([Int] → Int) → Protlet f S
compute h = RPC "compute" clientStep serverStep
  where
    clientStep :: S → f (NodeID, [Int], S)
    clientStep s = case s of
      ClientInit server args → pure (server, args, ClientDone)
      _ → empty
    serverStep :: [Int] → S → f ([Int], S)
    serverStep args s = case s of
      ServerReady → pure ([h args], ServerReady)
      _ → empty
```

As per its type, compute takes a client-provided function of type [Int] → Int, which is used by the server to perform calculations. The result of compute is of type Protlet f S, where S is the data type of our STS states defined just above and f is a type-former encapsulating possible non-determinism in a protocol specification. This is is a standard pattern for programming "with effects" in the pure fragment of Haskell. Later constructions will make integral use of non-determinism to, *e.g.*, decide on the next transition depending on the external inputs, and the parameter f serves to restrict what notion of non-determinism is used in the definition of protocols.[5] For now, the result of compute is entirely deterministic,

[3] One could argue that the client and server states don't have to belong to the same type. However, having experimented with different options, we found it easier to define all states of the same protocol as instances of the same data type. The alternative would increase the implementation overhead when combining parts of the protocol.

[4] See the discussion of executing specification in Section 3.

[5] One can think of any protocol, whose diagram has a fork, as non-deterministic.

but must still be "wrapped" in the constructors of the non-deterministic effect, here `pure` and `empty` indicating a single result and the absence of results, respectively.

Protlets (*aka* "small protocols") are the main building blocks of our framework. Complex protocols from literature decompose into interactions shaped as RPCs, notifications, *etc*, and we manage to capture all of them in protlets. Simply put, for every arrow in a diagram of the network indicating a communication channel, the protocol has a protlet detailing the exchanges occurring across that channel. A distributed protocol can be thought of as a family of protlets, each of which corresponds to a logically independent piece of functionality and can be captured by a fixed interaction pattern between nodes. In a system, each node can act according to one or more protlets, executing the logic corresponding to them sequentially, or in parallel. For this example, there is just the one exchange of messages, so a single protlet makes for the complete protocol description.

Our framework provides several constructors to build protlets from the data type description for the protocol state space and the operational semantics of its transitions. In the example above, `RPC` is a data constructor, which encodes the protlet logic by means of two functions. Its first argument, `clientStep`, prescribes that from `ClientInit` state, a node can send `args` to node `server`, and the response payload is later wrapped via `ClientDone` to form the successor state. The second argument, `serverStep`, says that the state `ServerReady` can serve a request in one step: receiving `args` and responding with `f args` in a singleton list, continuing in the same state. We have now completely captured the above intuitions and transition system of the calculator in less than ten lines of Haskell.

2.3 Executing the Specification

The immediate benefits of having an executable operational specification of a protocol is to be able to run it, locally and without needing full deployment across a network, ensuring that it satisfies basic sanity checks and more complex invariants.

The execution model for protlets is a small-step operational semantics, with the granularity of transitions being that of the involved protlets. We take as machine configurations the entire network of nodes and their abstract states.

In case several protlets of a similar shape are involved (*e.g.*, a node is involved in two or more RPCs—see our Case Study of a Two Phase Commit protocol in Case Study 4.3), we distinguish them by introducing protlet labels, a solution that is standard for program logics for concurrency (Sergey *et al.* 2016; Dinsdale-Young *et al.* 2010). A label is a *name* associated with each protlet instance associated with a node. This solves an implementation detail of maintaining state across several instances of the same protlet over time, local to a node: pure, functional programming cannot discern two structurally equal instances of a protlet unless named. Additionally, having introduced protlet labels to logically partition the local state of each node along the protlet instance space, we can also share the naming across nodes to split the global state into views of each, complete protocol instance. We represent this operational machine configuration as the datatype `SpecNetwork`, which is an instantiation of an abstract structure of a network state `NetworkState`, representing the global environment and a local state for each node in the network. The generality allows code reuse across the framework. For execution, the global environment is a protocol specification for each instance label. The per-node state consists of a protlet state for each protocol instance, and a message queue. The intention is that the operational semantics updates the state of just one protlet of one node at a time.

```
data NetworkState global local = NetworkState {
    _globalState :: global,
    _localStates :: Map NodeID local
  }

type SpecNetwork f s =
  NetworkState (Map Label [Protlet f s])
               (Map Label (NodeState s), [Message])
```

The following describes a network for the calculator protocol with two nodes (identified by `0` and `1`), both running just one protlet (labelled with `0`), for the input for the example from Section 2.1:

```
addNetwork :: Alternative f ⇒ SpecNetwork f S
addNetwork = initializeNetwork nodeStates protocols
  where
    nodeStates = [ (server, [(0, ServerReady)])
                 , (client, [(0, ClientInit server [3, 100, 20])]) ]
    protocols  = [ (0, [compute sum]) ]

    server, client :: NodeID
    (server, client) = (0, 1)
```

Here, `initializeNetwork` is a convenience function to initialize the `SpecNetwork` datastructures from human writeable descriptions in the form of association lists.

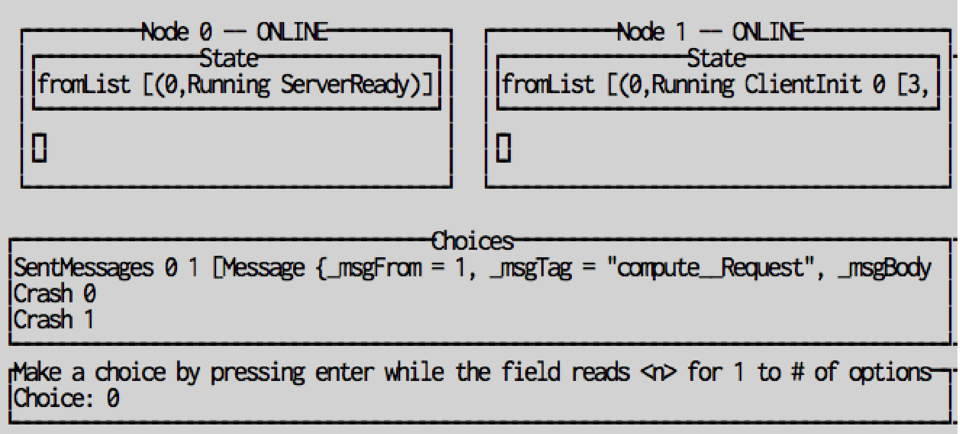
In any given network configuration, many actions can be possible. A node may be ready to initiate an RPC, or it (or another node entirely) might be ready to receive a message—many such actions may be enabled and relevant at once.[6] As the purpose of running the specification is to trace the possible behaviours in the protocol, we choose the next action to execute in the network by leaving the resolution to the *user* of the semantics. To do so, we implement the executable small-step relation as a monad-parameterised function capturing the possibility of non-determinism (hence `Alternative f`). This makes the implementation of the operational semantics simple, yet general, as it just needs to describe an `f`-ary choice or `f`-full collection of possible transitions at each step:

```
step :: (Monad f, Alternative f) ⇒ SpecNetwork f s → f (SpecNetwork f s)
step = applyTransition <$> possibleTransitions network <*> pure network
```

Here, `possibleTransitions` enumerate the possible transitions enabled in `network`, and `applyTransition` computes the subsequent network state through each transition. The network can be "run" by iterating this small-step execution function with a suitable instance of `f`, a standard construction in implementations of non-determinism in monadic interpreters.

For example, we can instantiate the non-determinism to the classic choice of the list monad (Liang *et al.* 1995), which leads to enumerating every possible action. We can then iterate the function `step` by choosing an arbitrary transition, as captured by the `simulateNetworkIO` function used in the following interaction with the library, where we explore the "depth" of a single run of the protocol.

[6] And their abundance is precisely why reasoning about distributed systems is hard.



```
Node 0 — ONLINE
State
fromList [(0,Running ServerReady)]

[]

Node 1 — ONLINE
State
fromList [(0,Running ClientInit 0 [3,

[]

Choices
SentMessages 0 1 [Message {_msgFrom = 1, _msgTag = "compute__Request", _msgBody
Crash 0
Crash 1

Make a choice by pressing enter while the field reads <n> for 1 to # of options
Choice: 0
```

Fig. 2: The interactive exploration tool, loaded with the calculator protocol.

```
> length <$> simulateNetworkIO addNetwork
4
```

This is coherent with the first example we envisioned *wrt.* the protocol: there is (1) the initial state; (2) the state with the client awaiting response, but the message undelivered; (3) the state with the client waiting and the server having sent a response; and finally, (4) a terminal state with the client done.

The non-determinism can be similarly resolved by enumerating all possible paths through a protocol, up to a certain trace length if the execution space is not finite. If the state space of a network is finite, this can yield actual finite-space model checking procedures. In the following subsection, we will explore another alternative to resolving the non-determinism, yielding an unusual yet very useful execution method.

2.4 Interactive Exploration with GUI

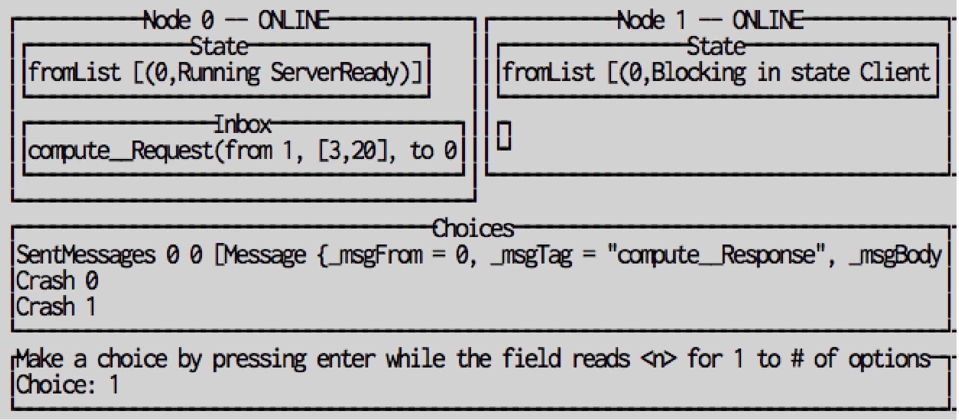
By delegating the decision of which transition to follow to the user of an application that performs this simulation, we can allow the client of the framework to explore the network behaviour interactively. The DPC library provides a command-line GUI application facilitating interactive exploration of distributed networks step-by-step. Provided an initial network specification like the one described previously, one can start the session by typing the following:

```
> runGUI addNetwork
```

This yields the interface displayed in Fig. 2. By choosing specific transitions in sequence, the user can evolve and inspect the network at each step of execution. This is useful for protocol design and debugging, and can help understand the dynamics of a protocol, and the kinds of communication patterns it describes.

For example, in Fig. 3 we show the subsequent prompt after showing the selection of Option 1:

```
SentMessages 0 1 [Message {_msgFrom = 1, _msgTag = "compute__Request",...
```



```
Node 0 — ONLINE
State
fromList [(0,Running ServerReady)]
Inbox
compute__Request(from 1, [3,20], to 0

Node 1 — ONLINE
State
fromList [(0,Blocking in state Client
[]

Choices
SentMessages 0 0 [Message {_msgFrom = 0, _msgTag = "compute__Response", _msgBody
Crash 0
Crash 1

Make a choice by pressing enter while the field reads <n> for 1 to # of options
Choice: 1
```

Fig. 3: Choosing option `1` in the prompt from Fig. 2.

`SentMessages` is a human readable piece of data that represents the option of sending in protocol instance `0`, from node `1` the message with sender `1` of tag `"compute__Request"`. The format chosen is the debug serialization format provided by Haskell's `Show` and `Read` type classes for ease of experimenting: any data of the sort displayed to the user can be directly copied and used in scripts or command prompts. Here, the recipient and message content is elided for issues of screen-space, but as the window is enlarged, so is the depth of information provided to the client of the framework.

The state view then shows that Node `0` now has said message waiting for it in the soup, and Node `1` is now blocking. The user is then presented with subsequent possible choices, here the option for the calculator to receive the request and send the response in one atomic action, as dictated by the protocol.

Additionally, as can be seen in Fig. 2, in the interactive tool we enrich the possible transitions at every step with the possibility of a node to go off-line. In effect, it means it will stop processing messages, modelling a benign (non-byzantine) fault. Other nodes cannot observe this and will "perceive" the node as not responding. It is implemented by eliding the actions performed by the off-line node when computing the set of possible actions. This, however, becomes very useful when we move to explore protocols that allow for partial responses among a collection of nodes, as in the case of crash-resilient consensus protocols. For instance, the Paxos specification of Case Study 4.2 can be readily inspected using this GUI. The example code explores a configuration of 2 proposers and 3 acceptors, but as many actors as the user has screen space can be run.

We concede that the text-based GUI does not scale well beyond 8 actors, but one can imagine richer interfaces than a text-based one. For example, the application is a purely functional program advancing with a small-step operational semantics: the state could be saved on a stack and visualized, like a directory structure familiar from file-systems.

2.5 Protocol-Aware Distributed Implementations

Distributed systems protocols serve as key components of some of the largest software systems in use. The actions taken in the protocol are governed by programs outside the key protocol primitives, so it is vital that implementations can integrate with software components in real general-purpose languages. We here present such a language with primitives for sending and receiving messages as an embedded domain-specific language (EDSL) in Haskell. This allows use of the entire Haskell toolkit in engineering efficient optimised implementations relying on distributed interaction.

Naturally, as implementations "refine" the abstract protocols (in the way they, *e.g.*, implement internal state), we want to ensure that the they still adhere to the protocol as specified. To achieve this, we introduce primitives for *annotating* implementations with protocol-specific assertions. These annotations can be ignored by execution-oriented interpretations aiming for efficiency rather than verification guarantees.

The following code implements a calculator server in plain Haskell using `do`-notation to sequence effectful computations. The effects are described by type class constraints on `m`, the monad used for sequencing: `MessagePassing` provides a `send` and `receive` primitive, and `ProtletAnnotations S` provide the `enactingServer` primitive over the state-space `S`. The type `S` is the data type defined in Section 2.2, and denotes the abstract state space we wish to relate to sub-computations in our implementation, as explained below.

```
addServer :: (ProtletAnnotations S m, MessagePassing m) ⇒ Label → m a
addServer label = loop
 where
  loop = do
   enactingServer (compute sum) $ do
     Message client _ args _ _ ← spinReceive [(label, "Compute__Request")]
     send client label "Compute__Response" [sum args]
   loop
```

By using type classes describing operations, we allow for several different interpretations of this code. For instance, by interpreting the `send` and `receive` as POSIX Socket operations, we obtain a subroutine in the IO Monad, Haskell's effectful fragment, that we can integrate into any larger development with no interpretive overhead. The `spinReceive` operation is defined using recursion and a primitive `receive` operation that attempts to receive an incoming message with a tag from amongst a list of candidate message tags in a non-blocking manner.

The body of `addServer` is annotated with a `(compute sum)` protlet, enforcing that the server responds to the client atomically (in terms of message passing) and to perform the `sum` function (or something observationally equivalent) on the supplied arguments. By bracketing the `receive` and `send` in the `enactingServer` primitive, the implementation declares its intent to conform to the server role of the RPC, as dictated by the protocol. Once we have a client to play the other role in the protocol, we will demonstrate how this intent can be checked dynamically. The message tags that appear in the code are *by convention* the tags used in the RPC protocol, *i.e.*, the name of the protocol with a suffix indicating the role in the RPC that the message plays.

In contrast with DISEL and other static verification frameworks that enforce protocol adherence via (dependent) type systems (embedded in Coq or other proof assistants) (Sergey *et al.* 2018; Krogh-Jespersen *et al.* 2020), we verify protocol properties dynamically. The trade-off is that of coverage versus annotation and proof overhead. We can, through exploiting executable specifications, check that running a program as a system of a program adheres to a protocol. If the abstract protocol specification is violated, a dynamic error will be raised. Notice that addServer is, like the specification of the compute protlet, agnostic in the number and kinds of other nodes in the network. Its behaviour is locally and completely described by its implementation, and is segregated from interfering with unrelated protlets via the label parameter. We refer the reader to the development for a number of client component implementations.

Let us now reap the benefits of protocol-aware distributed programming enabled by DPC and *dynamically check* that the implementations do indeed follow the abstract protocols. We achieve this by interpreting the EDSL into a datatype of abstract syntax trees (AST) that makes it possible to inspect their evaluations at run time. We give a small-step structural operational semantics to this language, and, precisely like the exectuable specifications, lift the evaluation of a single program to that of an entire network of programs, by assigning each program a node identifier in the network, as show below. Here, Node 1 runs the client implementation and Node 0 runs the server.

```
addConf :: (ProtletAnnotations S m, MessagePassing m) ⇒ ImplNetwork m Int
addConf = initializeImplNetwork [
              (1, addClient 0 20 3 0)
            , (0, addServer 0)
            ]
```

The similarity to specification-level configurations is not incidental: ImplNetwork is another instantiation of NetworkState:

```
type ImplNetwork m a = NetworkState [Message] (m a)
```

Here, the global state (of type ImplNetwork m Int, with m constrained as in addServer/addClient) is just the message soup, and the node-local state is the program itself. An interpreter for such configurations is implemented by the following function:

```
traceRoundRobin :: ImplNetwork (AST s) a → [TraceAction s]
```

Here, the AST data type is the AST, in the style of Higher-Order Abstract Syntax (Pfenning & Elliott 1988), for message-passing implementations to be interpreted in. The result of running the network is a (possibly infinite but productive) list of TraceActions. Trace actions describe “events” in the network: messages received and messages sent. We can simulate a full run of the network by using the trace actions to resolve the non-determinism of choices in the operational semantics.

For utility we here bake in a round robin schedule to ensure fair execution, but provide more general interpreters parametrised by a schedule and returning richer results, e.g.

```
runWithSchedule :: [NodeID] →
                   ImplNetwork (AST s) a →
                   [(TraceAction s, ImplNetwork (AST s) a)]
```

We can verify that our implementation indeed adheres to the desired protocol by the trace produced by traceRoundRobin on a network configuration, ensuring that (a) every observable action is compatible with the state that the node is supposed to be in, and (b) checking the messages expected from these states. For this, we implement yet another operational semantics, where the machine configuration is a protocol state for every node id, and the program is a trace of primitive actions. We here call it checkTrace. The interpreter faults if the current action is not applicable to the state, or sends or receives messages not prescribed by the specification. We can run the adherence checker on a *prefix* (*e.g.*, of length 15) of the infinite trace as follows:

```
> checkTrace addNetwork $ take 15 $ traceRoundRobin addConf
Right ()
```

The result of `Right ()` indicates success: the trace did indeed conform to the protlet annotations of the program, assuming the initial state of the implementations in `addConf` conformed to an initial abstract state corresponding to the the network state of `addNetwork`. This implementation defines our notion of finite *protocol adherence*: a node adheres to a protocol when it sends and receive messages in the order of, and of the structure prescribed by, a specification. We illustrate below that the implementation can identify both out of order messaging, and malformed or "functionally" incorrect messages in the correct order.

What happens if we introduce a mistake in the implementation? For instance, what if we run the client implementation *twice*? We can illustrate this by altering `addConf` to demand the `addClient` to be run twice in succession:

```
addConf = initializeImplNetwork [
    (1, addClient 0 3 20 0 >> addClient 0 100 11 0)
  , (0, addServer 0)
  ]
```

The checker then reports an error, as this is not allowed by the protocol: the client would have brought itself to the terminal state `ClientDone` by the first RPC, and, hence, cannot proceed.

```
> checkTrace addNetwork $ take 30 $ traceRoundRobin addConf
*** Exception: Node 1 expected to initiate rpc compute
               Node is in state: ClientDone [23]
```

In a different scenario, if we erroneously annotate the server as intending to serve a `product` function (instead of `sum`), we will fail protocol adherence, because the specification does not agree on the content of the messages.

```
addServer :: (ProtletAnnotations S m, MessagePassing m) ⇒ Label → m a
addServer label = loop
 where
   loop = do
     enactingServer (compute product) {- !!ERROR!! -} $ do
       ...

> checkTrace addNetwork $ take 15 $ checkTrace addConf
*** Exception: The server response did not follow the protocol from state
               ServerReady
                 Expected: [60]
                 Got: [23]
```

Of course, here we only observe the error because our *single* instantiation of the client's payload, `[20, 3]`, happened to disagree on the `sum` and `product` function. What if the payload was `[1]`?

By enriching dynamic testing with protocol adherence checks we believe we can achieve greater assurances of the correctness of our implementations without resorting to use full-blown verification frameworks (Hawblitzel *et al.* 2015; Sergey *et al.* 2018).

2.6 Introducing Randomised Testing to Distributed Systems

Naturally, the dynamic testing demonstrated in the preceeding section is only as good as the creativity and insight of the developer. The originators of QuickCheck observed that pure functional programs with executable specifications acting on first-order data is a natural setting for exploiting randomised testing: instead of carefully crafting pathological cases to demonstrate absence of errors, define a generator for random program input and write more programs to decide whether the program under test performs as expected (Claessen & Hughes 2011).

DPC is a natural fit for this approach: we have an executable specification of a protocol, along with a re-interpretable DSL for implementing these protocols. This suggests that we generate random input data for the implementations, and use our executable specification as ground truth for correctness of implementations.

This is in general as complicated as the type of the input data: in this, and all other protocols in this paper, we are acting on simple messages of lists of integers. Generators for this is readily available in the QuickCheck libary.

First, we parameterise the initial configurations of the specification and implementation execution configurations by the numbers that the client want operated on.

```
addNetwork :: Int → Int → SpecNetwork f S
addConf :: Int → Int → ImplNetwork m [Int]
```

With this in hand we can formulate universally quantified boolean properties (indexed boolean-valued expressions) that can then be evaluated on randomly generated inputs. At its most basic, we wish the evaluation of the implementation to conform to the specification, a property here formulated using `checkTrace` as described in Section 2.5:

```
prop_simpleAddNetwork :: Int → Int → Bool
prop_simpleAddNetwork x y =
  let trace = take 100 $ traceRoundRobin (addConf x y) in
    checkTrace (addNetwork x y) trace == Right ()
```

The prefix of `prop_` is a convention that allows for discovery of properties by the QuickCheck toolset. The function `traceRoundRobin` is a pure interpreter for the implementation language that schedules nodes in a fair round-robin fashion. QuickCheck can now help us exorcise bugs of the class previously identified as problematic for unit testing:

```
> quickCheck prop_simpleAddNetwork
*** Failed! (after 2 tests and 2 shrinks):
Exception:
  The server response did not follow the protocol from state: ServerReady
    Expected: [0]
    Got: [1]
0
1
```

QuickCheck reports that on payload `[0, 1]`, the server implementation violated the server specification: it replied to the client with 1 rather than 0 as (erroneously) dictated by the specification.

QuickCheck can also help us with the problem of systematically testing a class of errors unique to the setting of non-deterministic concurrent computation via message-passing, namely that of programs not accounting for all possible schedules. As the number of in-

Semantics Name	Machine Configuration	Step Function Signature
Executable Specifications	`SpecNetwork`	`SpecNetwork →f (SpecNetwork)`
Pure, Tracing Semantics	`(Schedule, ImplNetwork AST)`	`ImplNetwork → NodeID → (ImplNetwork AST, TraceAction)`
Trace Verification	`([TraceAction], SpecNetwork)`	`TraceAction → SpecNetwork → Either Error SpecNetwork`

Table 1: Operational semantics for distributed protocol.

structions per process increases, the number of possible schedules grows exponentially, and aggressively so. Corner cases are also difficult to foresee, so in lieu of formal verification, the possibility of randomly exercising possible schedules is worth pursuing. A schedule arises from the non-deterministic interleaving of threads, but a concrete schedule can be represented by a sequence of integers, in Haskell a value of type `[NodeID]`, indicating the order of execution of the nodes in the distributed system. We generalize `roundRobinTrace` to `traceSchedule`, parametrised by the specific schedule to use.

```
arbitraryScheduleFor :: [NodeID] → Gen [NodeID]
arbitraryScheduleFor s = infiniteListOf (elements s)

prop_simpleAddNetworkArbSchedule :: Int → Int → Property
prop_simpleAddNetworkArbSchedule x y =
    forAll (arbitraryScheduleFor (nodes conf)) $ λschedule →
      let trace = take 100 $ fst <$> runWithSchedule schedule conf in
          checkTrace (addNetwork x y) trace == Right ()
  where
    conf = addConf x y
```

We use the `forAll` combinator to build a `Property`, in essence a generalization of a boolean valued expression to a function taking a random seed (and some additional configuration controlling the generation process). Here, `forAll` is used to explicitly supply the generator `arbitraryScheduleFor` to be used for generating traces as opposed to the implicit inference of appropriate generators for `x` and `y` via type classes (The existing instance for `[Int]` simple generates a finite list of random integers). QuickCheck uses the convention of naming generators 'arbitrary'. This now let us exercise the implementation for bugs arising due to pathological execution orders of each node in the network. It appears robust to arbitrary interleavings of execution:

```
> quickCheck (withMaxSuccess 10000 prop_simpleAddNetworkArbSchedule)
+++ OK, passed 10000 tests.
```

We believe we here have illustrated the applicability of randomised testing to build a discipline of testing for distributed systems. By exploiting that the specification language of DPC is executable, we leverage existing technologies to give us a light-weight process for writing convincingly correct implementations of distributed components.

2.7 Multiple Semantics for Distributed Systems Executions

The versatilty of our approach to designing and running distributed systems is enabled by the ability to execute the composed protocols using three different structural operational semantics, summarised in Table 1.

The first is a semantics for the specification language acting at the level of protlets, as demonstrated in Section 2.3. A `SpecNetwork` is an assignment of protocol states to every node in a network along with a collection of protlets over those nodes. The stepping function is parametrised by a notion of non-determinism, `f`, that we instatiate with, *e.g.*, `IO` to produce the GUI-driven exploration tool (Section 2.4).

The second semantics, showcased in Section 2.5, is an "implementation". It acts on a network configuration where each node has a program in the pure monadic language of Haskell extended with message passing operations, here reified as `AST` as explained in the corresponding section of the paper. This is then interpreted in a straight forward manner. A part of the configuration is a `Schedule`, an infinite list of `NodeIDs`. The step relation then computes the corresponding `TraceAction` (Send, Receive or No-Op) of the next node in the schedule, and advances the network state.

The third semantics also acts on a `SpecNetwork`, but drives the evolution by applying the step relation of the first, and verifying that a supplied trace action can produce *at least one* of the "`f`-many" choices. In this particular case `f` is instantiated with `[]`, the list `Alternative` used to represent finite determinism. This semantics is used for verification, as in the randomised testing examples (Section 2.6).

3 Framework Internals

3.1 The Specification Language

A full distributed system specification consists of a collection of nodes, each assigned a unique node identifier, and a collection of protlets for each instance label. A node owns local state, partitioned according to protocol instance labels. A protlet describes one exchange pattern between parties. A collection of protlets over the same state space then describe an entire protocol.

In the overview we saw the simplest protlet, the pure RPC, but through exploration of examples and case studies, we have discovered a number of such patterns. These are implemented as extensions to the `Protlet` data type. One such is the broadcast protlet, integral for describing multi-party protocols. We elide the other protlet constructors, which can be found in our implementation.

```
data Protlet f s =
    | RPC          String (ClientStep s) (ServerStep s)
    | Broadcast    String (Broadcast s)  (Receive f s) (Send f s)
    | ...
```

The component functions of the protlets reuse a number of common type abbreviations, here `ClientStep`, `Send` etc. All are at work in the above listing. This common structure unifies their implementation in the operational semantics. The expansion of, *e.g.*, the `Broadcast` synonym is as follows:

```
type Broadcast s = s → Maybe ([(NodeID, [Int])], [(NodeID, [Int])] → s)
```

This models a "partial" function on states `s`, saying under which conditions a node can initiate a broadcast, by enumerating the recipients and the body of the messages to them, along with a continuation processing the received answers with their associated senders.

This continuation is stored in the implicit blocking state during actual execution of the specification.

The specification language is given a non-deterministic operational semantics as described in Section 2.3. Recall the network step function:

```
step :: (Monad f, Alternative f) ⇒ SpecNetwork f s → f (SpecNetwork f s)
```

It is implemented by computing an `f`-full of possible transitions for every node in the network and combining the result of taking all possible transitions on the current network. The key operation of `step` is a dispatch on the current protocol state of a node:

```
case state of
  BlockingOn _ tag f nodeIDs k →
    resolveBlock label tag f nodeID inbox nodeIDs k
  Running s → do
    protlet ← fst <$> oneOf (_globalState Map.! label)
    stepProtlet nodeID s inbox label protlet
```

The constructors `BlockingOn` and `Running` are supplied by the framework. The first is used to track the terms under which a node is blocking: what message(s) it needs to continue and from whom. `resolveBlock` computes whether the conditions are met for the current node to continue.

Here, `_globalState` is the mapping of collections of protlets (*i.e.*, a protocol) from instance labels. We then choose between protlets using `oneOf::[a] → f a`. The function `stepProtlet` dispatches control based on a case distinction on the protlet constructor: for example, here is the branch for the `Broadcast` protlet:

```
stepProtlet :: (Monad m, Alternative m) ⇒
  NodeID → s → [Message] → Label → Protlet m s →  m (Transition s)
stepProtlet nodeID state inbox label protlet = case protlet of
  ...
  Broadcast name broadcast receive respond →
    tryBroadcast label name broadcast nodeID state inbox <|>               -- (1)
    tryReceive label (name ++ "__Broadcast") receive nodeID state inbox <|>  -- (2)
    trySend label respond nodeID state inbox                                -- (3)
  ...
```

A node attempting to advance a protocol using the `Broadcast` protlet can do so if it is (1) a client ready to perform a broadcast; (2) a server ready to receive such a broadcast; or (3) a server that is ready to respond to a broadcast. The `try` functions all follow the same structure: check that the user-provided protlet component functions apply, and if so, generate an appropriate transition. The result of each call is combined using <|>, the choice operator for the `Alternative` instance for `m`. For instance, here is the signature of one such function for `Broadcast`:

```
tryBroadcast :: Alternative f ⇒ Label → String → Broadcast s →
                NodeID → s → [Message] → f (Transition s)
```

Interpretations of Protocols. As described in Section 2.3, the operational semantics of protocols can be instantiated to obtain different interpretations. We here look at bounded model checking mentioned in passing in the overview. We can use the `List` monad to enumerate all execution paths in a breadth-first manner:

```
simulateNetworkTraces :: SpecNetwork [] s → [[SpecNetwork [] s]]
```

This yields a list-of-lists where the *n*th list contains all possible states after *n* steps of execution, in a breadth first enumeration of the state space. Each constituent list of states is necessarily finite, but the list-of-lists need not be in the case of infinite network executions. By virtue of Haskell's lazy evaluation, such a computational object is easy to construct and to manipulate, without worrying about its size upon creating it. We can write a procedure that, given a trace, applies a boolean predicate at every step of the trace.

```
checkTraceInvariant :: Invariant m s Bool → m → [SpecNetwork f s] →
                       Maybe Int
```

The `Invariant` data type is an abbreviation for a boolean predicate on the type `s` that additionally takes some "meta-data" `m`, like "roles" in a protocol, needed to express the invariant. The procedure `checkTrace` returns `Nothing` to signify that there were no violations of the invariant, while it returns `Just n` to report that the `n`th state was the first state to violate the invariant. With this language of predicates we can build invariants and with the aforementioned checking procedure we can perform (bounded) checking that an invariant is in fact inductive (*i.e.*, holds for each state). In the case of a finite state space, this amounts to real verification of inductive invariants. The most sophisticated example we have successfully specified is an inductive invariant for a Two-Phased Commit protocol (Sergey *et al.* 2018), for which we refer the curious reader to the implementation.

3.2 The Implementation Language

The monadic language for message-passing programs is implemented as an EDSL in Haskell. This has the benefit of providing all the standard tools for writing Haskell programs; all the abstraction mechanisms and organisational principles are at hand to write sophisticated software, including lazy evaluation, higher-order functions, algebraic data types and more. By virtue of the modularity offered by the approach of EDSLs, it is straightforward to give multiple interpretations of such programs.

The described DPC's implementation (Andersen & Sergey 2019b) fragment comes with three interpretations of the monadic interface:

1. The `AST` monad used for dynamic verification of implementation adherence of the implementations to protocols, and covered in detail in Section 2.5.
2. A shared-memory based interpretation where nodes are represented as threads, and message passing is performed by writing to shared message queues using non-blocking concurrency primitives.
3. An interpretation for distributed message passing.

In the third case (true distribution), we give an interpretation into `IO` computations performing message passing through POSIX Sockets. For this, each computation needs an "address book" mapping `NodeIDs` to physical addresses (concretely, IP addresses and ports). Additionally, each program will have access to a local mailbox, represented by a message buffer being filled by a local thread whose only function is to listen for messages. These two pieces of data are collected in a record of type `NetworkContext`. Computations running in such a context are idiomatically captured in a type synonym over the `ReaderT` monad transformer:

```
newtype SocketRunnerT m a = SocketRunnerT {
    runSocketRunnerT :: ReaderT NetworkContext m a }
```

What follows is the implementation of the `send` primitive in this particular instance of the message-passing interface:

```
instance MonadIO m ⇒ MessagePassing (SocketRunnerT m) where
  send to lbl tag body = do
    thisID ← this
    let p = encode $ Message thisID tag body to lbl
    peerSocket ← (!to) <$> view addressBook
    void . liftIO $ Socket.send peerSocket p mempty
```

The code for sending messages is, thus, implemented in a form of a `Reader`-like computation over an `IO`-capable monad `m` as indicated by the `MonadIO` constraint. It starts by building a `Message` containing the supplied tag, body, receiver (`to`) and label, along with the executing nodes ID, as supplied by another primitive, `this`. It then uses `encode` to serialize this message into bytestring `p`. Then, `p` is sent to the appropriate `peerSocket`, as resolved by the `addressBook`, using the `System.Socket.Send` operation from the POSIX Socket library for Haskell. The monadic glue code (and the rest of the Haskell toolkit) is interpreted by choosing an appropriate base monad for the interpretation, *e.g.*, the `IO` monad. Ultimately, we build the following function for running the system:

```
defaultMain :: NetworkDescription → NodeID → SocketRunner a → IO ()
```

It takes a `NetworkDescription`, which maps `NodeID`s to physical addresses, a `NodeID` with which to identify this node, and a computation in the above described interpretation of message passing programs. The result is an `IO ()` computation that establishes (if run on each machine) a fully connected mesh network with every node in the supplied network description, and then proceeds to run the supplied computation, passing messages accordingly. This interpretation can be used to facilitate integration of DPC-based implementations with real Haskell code once they have been assured to comply with their protocols.

4 Evaluation

The implementation of DPC is publicly available online for extensions and experimentation.[7] We now report on our experience of using DPC for implementing and validating some commonly used distributed systems.

4.1 More Examples

In order to evaluate the framework, we have encoded a number of textbook distributed protocols, translating their specifications to the abstractions of DPC. By doing so, we were aiming to answer the following research questions:

1. Are our `Protlet`-based combinators sufficiently expressive to capture a variety of distributed systems from the standard literature in a natural way?

[7] The latest version is available at `https://github.com/kandersen/dpc`. The version at the time of publication can be found at (Andersen & Sergey 2019b)

Protocol	Impl	Protlets	LOC	RPC	ARPC	Notif	Broad	Quorum
Calculator	✓	1	10	✓	✓			
Lock Server		4	73	✓	✓	✓		
Concurrent Database		3	23	✓				
Two-Phase Commit		2	43				✓	
Paxos	✓	2	42					✓

Table 2: A summary for implemented systems: protocol, runnable implementation, count of constituent protlets, size of encoding (lines of code), employed combinators.

2. Is it common to have realistic protocols that require *more than one* combinator, *i.e.*, can be readily expressed decomposed as multiple `Protlets`?
3. What is the implementation burden for encoding systems using DPC?

The statistics for our experiments is summarised in Table 2.

The framework has been shaped by the explorations of protocols that we have made, but we believe that the answer to Q1 is affirmative, supported by the variety of protocols we have so far explored. The answer to Q2 is also affirmative. For instance the two-phase protocols like Paxos and Two-Phase Commit (2PC) naturally decompose into two broadcast/quorum phases, while more asymmetric protocols like distributed locking (Kleppmann 2016) requires as many as four protlets.

Regarding Q3, the lines of code versus complexity of protocol are indicative of a positive relationship between complexity and effort to encode a protocol, which is desirable. That is, a lot of complexity is encapsulated by the treatment of combinators, so the coding effort in the framework is very light.

The nature of the verification that the framework enables is naturally not strictly sound (as it is dynamic), but techniques like bounded model checking are readily explorable. With it, we have been able to validate, *e.g.*, correctness for the 2PC protocol (Sergey *et al.* 2018), whose formal proof poses a significant proof burden.

In terms of real time expenses of verification, individual runs of verification, trace generation *etc* is perceptually instantaneous for the scale of the experiments here. Verification via randomised testing naturally scales with the number of samples tested. Likewise, bounded checking scales with the depth of traces. Below, we provide some rudimentary timings, obtained on a Late 2013 MacBook Pro, 2.6 GHz i5 CPU, 8 GB of 1600 MHz DDR3 Memory. We first illustrate testing time versus number of samples:

```
*DPC.Examples.PADL> quickCheck (withMaxSuccess 1000 prop_simpleAddNetwork)
+++ OK, passed 1000 tests.
(0.87 secs, 403,673,944 bytes)

*DPC.Examples.PADL> quickCheck (withMaxSuccess 10000 prop_simpleAddNetwork)
+++ OK, passed 10000 tests.
(7.07 secs, 4,036,093,952 bytes)
```

Secondly, time versus depth of verification:

```
*DPC.Examples.PADL> quickCheck (withMaxSuccess 1000
                               (prop_simpleAddNetworkWithDepth 10))
+++ OK, passed 1000 tests.
(0.34 secs, 68,176,840 bytes)
```

```
*DPC.Examples.PADL> quickCheck (withMaxSuccess 1000
                                (prop_simpleAddNetworkWithDepth 100))
+++ OK, passed 1000 tests.
(0.70 secs, 403,681,864 bytes)

*DPC.Examples.PADL> quickCheck (withMaxSuccess 1000
                                (prop_simpleAddNetworkWithDepth 1000))
+++ OK, passed 1000 tests.
(6.68 secs, 3,701,282,896 bytes)
```

The framework also affords exploration in other directions than we have mentioned so far. We have experimented with enriching the message passing language with operations for shared-memory concurrency and thread-based parallelism. The database example in the table uses *node-local* threads to maintain a database that is served by two different threads. Our approach to dynamic checking of protocol adherence scales to concurrency, and we have a concurrent Calculator server serving *multiple* arithmetic functions *in parallel*.

4.2 Case Study: Constructing and Running Paxos Consensus

For a representative exploration of the capabilities of DPC we turn to a study of the Paxos Consensus (Lamport 1998; García-Pérez *et al.* 2018; Chandra *et al.* 2007). Paxos solves a problem of reaching a consensus on a single value agreed upon across multiple nodes, of which a subset acts as proposers (who suggest the values) and another, complementary subset acts as acceptors (who reach an agreement). The nature of the Paxos algorithm lends itself well to interactive exploration and the specification should be robust to issues that appear specifically in distributed systems, like arbitrary interleaving of messages, message reorderings, and nodes going offline. The tools we have developed so far are enough to explore these aspects of the protocol.

We can specify this protocol in DPC with relatively little code. We further generalise the `Broadcast` combinator to "quorums" — broadcasts that await only a certain number of responses before proceeding. We introduce another entry in our `Protlet` datatype for capturing this pattern.

```
data Protlet f s = ...
  | Quorum String Rational (Broadcast s) (Receive s) (Send f s)
```

The `Quorum` protlet is and acts identical to the Broadcast protlet, but it is further instrumented by a rational number indicating the number of responses to await before proceeding. We encode the dissection of nodes into proposers and acceptors directly in the state of the protocol, similar to how we dissected the state space of the cloud server along Client/Server lines.

The state-space diagram of Paxos is given in Figure 4. The proposer starts in `(ProposerInit b v as)` with the desire to propose to acceptors as the value `v` with priority (*ballot*) `b`. We encode this with a quorum protlet:

```
prepare :: Alternative f ⇒ Label → Int → Protlet f PState
prepare label n = Quorum "prepare" ((fromIntegral n % 2) + 1) propositionCast ...
  where
    propositionCast = λcase
      ProposerInit b v as → Just (zip as (repeat [b]), propositionReceive b v as)
      _ → Nothing
```

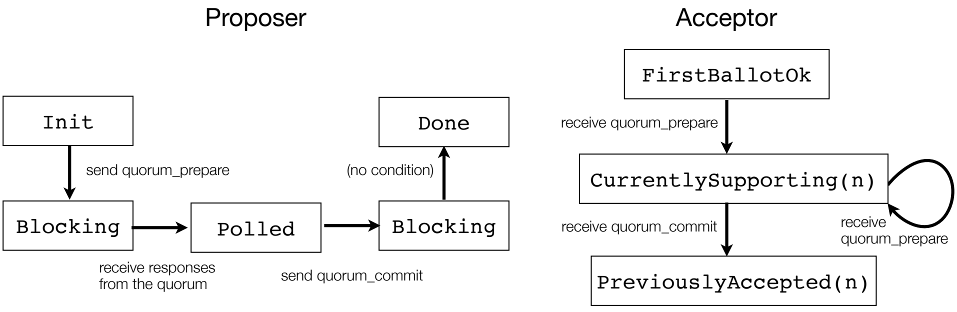


Fig. 4: State-Transition Diagram of Paxos Consensus Protocol.

Here, `prepare` is parameterised by the number of participants. Hence, the protlet dictates we should wait for a majority quorum, to avoid ties in the system. The listing shows the initiation of the first broadcast as representative of the rest of the implementation. The proposer starts in a `ProposerInit` state, in which it initiates a broadcast poll of all `as` acceptors, sending its ballot `b`.

The second phase of the protocol is encoded as another `Quorum` protlet, where the proposers react to the outcome of the responses on the first polling. The phase ordering is not explicitly defined, but a deliberate consideration is made in the design of the state space: the initial state of the second phase protlet is precisely the terminal state of the first phase protlet. Phase ordering emerges from this mechanism. The interactive exploration tool can be used to explore, for instance, the robustness of the protocol with respect to crashing participants versus crashing proposers, and why a quorum size of $\left(\frac{n}{2}+1\right)$ acceptors is sufficient for reaching consensus.

The explored implementation demonstrates use of the *state* monad to organise the acceptor as an effectful program, and a *callback* to provide the ballot to the proposer, using features of Haskell, while retaining the benefits of the framework. Neither effect is possible to express at the protocol specification level.

4.3 Case Study 2: Specifying and Model Checking Two-Phased Commit

For a representative of the verification capabilities of DPC we turn to a study of the Two-Phase Commit protocol encoded in DISEL (Sergey *et al.* 2018). There, it was properly formalized in the DISEL framework, so it translates readily to DPC. This case serves as a study of the same work done using a light-weight, formally-guided approach, as opposed to a fully formal framework.

The Two-Phase Commit state-transition diagram is given in Figure 5. The DISEL encoding of the 2PC protocol as described by Weikum & Vossen (2002) assumes a single *coordinator*, often known as a "proposer" in similar treatments, and a static collection of participants, or acceptors. The coordinator asks the participants to agree or disagree with a particular transaction, and the consensus is communicated back to the acceptors once the coordinator has tallied all votes. The state space of the nodes in the protocol is the most complicated we have studied so far:

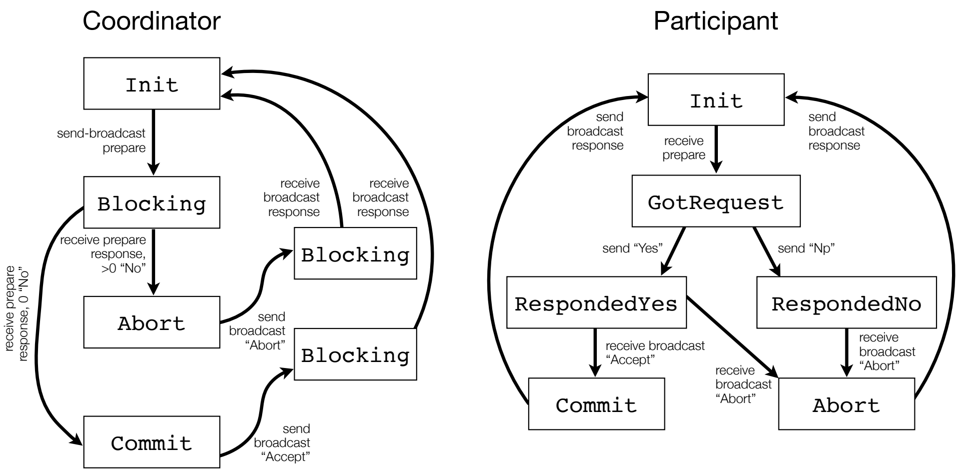


Fig. 5: State-Transition Diagram of the Two-Phase Commit Protocol.

```
data State = CoordinatorInit [NodeID]
           | CoordinatorCommit [NodeID]
           | CoordinatorAbort [NodeID]
           | ParticipantInit
           | ParticipantGotRequest NodeID
           | ParticipantRespondedYes NodeID
           | ParticipantRespondedNo NodeID
           | ParticipantCommit NodeID
           | ParticipantAbort NodeID
```

The coordinator maintains a list of participants to poll for acceptance, and each participant records the server to respond to upon contact. The server proceeds from `CoordinatorInit` to either `CoordinatorCommit` or `CoordinatorAbort` depending on the outcome of the polling round, and from there back to `CoordinatorInit` upon sending the result of the poll to all participants. Each participant starts in the `ParticipantInit` state, and proceeds to `ParticipantGotRequest` upon being polled by a coordinator. From here it can accept or reject the request as desired, this is left up to the implementaion: it is however *specified* that it must move to `ParticipantRespondedYes` or `ParticipantRespondedNo`, respectively. From there it moves to `ParticipantCommit` or `ParticipantAbort` as appropriate when learning of the outcome of the poll at large from the coordinator.

A safety specification of the protocol is traditionally given in a form of *inductive invariant*—a property that is satisfied by the initial state of the system and is preserved by each modification it undergoes—ultimately implying that "nothing bad happens". Finding an invariant strong enough, so it would adequately capture the relevant properties system, is a work of art, and inevitably requires a human prover's assistance (Padon *et al.* 2016). However, once an invariant is defined, it can be checked mechanically. Invariants are usually defined by conjoining global predicates on the states of each node without regard for the specification of the protocol. That is, they declaratively express the legal states of the *entire system* without mention of legal ways to get there.

Invariants in DPC are expressed as predicates on the specification level state space, and in particular, it was straight forward to adapt the Coq-formulated inductive invariant of the 2PC system to a Haskell function deciding the same property for a 2PC specification.

The invariant checker enabled by DPC appears to be a very useful tool. In this particular case we can borrow an invariant from DISEL formalization that expresses full correctness of the 2PC protocol, but it is conceivable that "smaller" properties might be of interest. For local examples, that a particular state is never entered, or that the payload of a particular state satisfies a predicate. As an example of system-wide properties, spanning multiple nodes, one can assert that no two nodes are in a particular state at once.

The entire invariant is specified as a disjunction:

```
tpcInvariant :: TPCInv
tpcInvariant = everythingInit <||> phaseOne <||> phaseTwo
```

The invariant asserts that all nodes in the system are either in the initial state, the started phase one or all participants have been polled and the system is in phase two. The type synonym `TPCInv` simply wraps the `Invariant` type introduced in Section 3.1, instantiated by the `State` type of this particular case study. The operator <||> simply lifts boolean disjunction to the `Invariant` type.

Let us illustrate the implementation details of the invariant looking at the implementation the `phaseOne` invariant and its components.

```
phaseOne :: TPCInv
phaseOne =
  forCoordinator coordinatorPhaseOne <&&>
  forallParticipants participantPhaseOne
```

We here use domain specific combinators that have been built to make it convenient to refer to the nodes in the state space by their roles as opposed to their `NodeID`. The predicate `participantPhaseOne` is a large disjunction enumerating that a particular node is either in the initial state because it hasn't been polled by the coordinator; or it is in the `GotRequest` state because it has been polled exactly once; or it has responded yes or no, sending the appropriate messages to the coordinator:

```
participantPhaseOne :: NodeID → TPCInv
participantPhaseOne pt = do
  cn ← getCoordinator
  foldOr [
    runningInState ParticipantInit pt
    <&&> noMessageFromTo pt cn
    <&&> messageAt pt "Prepare__Broadcast" [] cn,

    runningInState (ParticipantGotRequest cn) pt
    <&&> noOutstandingMessagesBetween pt cn,

    runningInState (ParticipantRespondedYes cn) pt
    <&&> noMessageFromTo cn pt
    <&&> messageAt cn "Prepare__Response" [1] pt,

    runningInState (ParticipantRespondedNo cn) pt
    <&&> noMessageFromTo cn pt
    <&&> messageAt cn "Prepare__Response" [0] pt
    ]
```

The utility predicates like `noMessageFromTo` are "primitives" provided by the `Invariant` library that are reusable across specifications. They express general properties like state of the message soup pertaining to a particular node or set of nodes.

With an invariant like this in hand, we can check that the specification satisfies this property at every step, *i.e.*, that the invariant is *inductive*:

```
> checkInvariantTraces tpcInvariant initNetworkMetadata . take 15 $
                       simulateNetworkTraces initNetwork

Nothing
```

What we see is exhaustive bounded model checking of the specification: the trace enumeration via `simulateNetworkTraces` evolves the network in a breadth-first manner, returning a list of frontiers, while `checkInvariantTraces` iterates through this "tree" and ensures that the supplied invariant is never violated. Additionally, it is supplied with protocol specific meta-data, some global context accessible to the invariant, which in this case includes the assignment of roles in the protocol to the node identifiers used in `initNetwork`.

Here, we look 15 frontiers deep, a grossly exponential number of states, but enough for the protocol to have run at least once.

While by no means a wild feat of engineering, this brings hard verification to a very lightweight toolkit at very little cost to developers of algorithms. By comparison, the equivalent *formal* verification in DISEL is more than 2,000 lines of definitions and well-formedness property proofs, not counting the invariant itself, *before* the formalization begins any proofwork. Here, we start exploring the behaviour and nuances of the protocol of interest in as little as 75 lines in the case of 2PC.

5 Related Work

Declarative programming for distributed systems. In the past five years, several works were published proposing mechanised formalisms for verification of distributed protocols, both in synchronous (Dragoi *et al.* 2016) and asynchronous setting (Sergey *et al.* 2018; Wilcox *et al.* 2015). All those frameworks allow for executable implementations, yet the encoding overhead is prohibitively high, and no abstractions for specific interaction patterns are provided in any of them. Most of the DSLs for distributed systems we are aware of are implemented by means of extracting code rather than by means of a shallow DSL embedding (Killian *et al.* 2007; Liu *et al.* 2012; Leonini *et al.* 2009). MACE (Killian *et al.* 2007), a C++ language extension and source-to-source compiler, provides a suite of tools for generating and model checking distributed systems. DISTALGO (Liu *et al.* 2012) and SPLAY (Leonini *et al.* 2009) extract implementations from protocol descriptions.

In a recent work, Brady (2017) has described a discipline for protocol-aware programming in IDRIS, in which adherence of an implementation to a protocol is ensured by the host language's dependent type system, similarly to DISEL, but in a more lightweight form. Brady's approach allows for static verification of distributed interactions by using dependent types for constraining *peer-to-peer* communication. However, this design also makes it difficult to provide dedicated combinators for specific *one-to-many* or *many-to-many* communications patterns, *e.g.*, broadcasts or quorums, which would retain the same static safety guarantees. Those combinators are possible to implement in DPC due to our framework's less restrictive typing discipline and focus on run-time verification.

Similarly to our DPC-based language for defining protocol combinators, the P programming language by Desai *et al.* (2013) has been introduced as a way to facilitate modular construction of distributed systems. In P, a program is a collection of machines. Machines communicate with each other asynchronously through events. In order to implement a protocol, the programmer must specify the structure of the machines and events. P programs can be verified via the built-in PTESTER tool, and are compiled to C as a executables. Therefore, P approach introduces a gap between the verified and the executable artifacts.

More recently, the MODP system (Desai *et al.* 2018) has been built on top of it. MODP is an extension of P, which allows for more complex programs to be built. In MODP, users can implement systems as individual modules, and combine them into a larger module *horizontally*, *i.e.*, by means of DISEL-style Rely-Guarantee-based composition (Jones 1983). While similar in spirit to DPC, MODP's composition framework appears to be more *coarse-grained* then what we have described. For instance, even though MODP has been used to define and test a version of Paxos, it is not clear how to implement a in it a combinator such as our `Quorum`, which can be reused across multiple protocols.

Behavioural Types and Dynamic Contracts. Related to our work are the recent approaches for the dynamic verification of message-passing distributed systems based on session types, which make use of contracts and runtime monitors (Melgratti & Padovani 2017; Gommerstadt *et al.* 2018). For instance, the *chaperone contracts* by Melgratti & Padovani (2017) allow to impose dynamic checks, in the style of Findler & Felleisen (2002), on the contents of the messages transmitted between the processes, providing a corresponding *blame calculus* to detect faulty processes at runtime. The work by Gommerstadt *et al.* (2018) defines dynamic contracts that are more expressive with regard to a number of properties that can check for a message-passing communication. Similarly to monitors, they can establish that the communication abides by the constraints imposed by the protocol by inspecting the contents of the messages. Instrumented with internal state Gommerstadt *et al.*'s concurrent contracts are even capable of dynamically checking properties of data being transferred, for instance, checking that the integer responses to a series of requests come in an ascending order. While DPC take an intrinsic approach to defining the behaviour, enforcing the crucial state invariants *by construction*, the concurrent contracts take a more extrinsic perspective, being ascribed to an already defined implementation as additional checks. Furthermore, in their dynamic checks, concurrent contracts and monitors are limited to the data, which is locally available to a process. In contrast, for the purpose of debugging and model checking, DPC allows to declare and check global invariants of the entire system, which span the state of multiple concurrently operating nodes (as in the case study from Section 4.3).

In the nomenclature by Ancona *et al.* (2016), DPC provide a *top-down* approach for specification and implementation of correct-by-construction distributed systems, similarly to multi-party session types and choreographies (Hüttel *et al.* 2016). That is, once a global description of the system's behavior is specified the local implementations of individual nodes are derived from it. An opposite *bottom-up* approach, in which the system's properties are derived from the definitions of its individual components (Lange & Tuosto 2012), corresponds to deductive verification (Hawblitzel *et al.* 2015), and is typically adopted for the systems, whose implementation is, to the large extent fixed and is not evolving.

While the DISEL's take on verification combines both top-down and bottom-up approaches, DPC focuses on the former one as a way to ensure the resulting system's correctness by construction. Our rationale for following the top-down approach was to encourage the system's designers to think about the system's properties and invariants upfront, before implementing the low-level details. By taking a bottom-up approach for our goals (by *e.g.*, deriving specifications from implementations that are still in the development) we would risk to introduce noticeable overheads due to the need to revise already implemented components that do not compose well.

Relation to DISEL. DPC's protlets adapt DISEL's protocols, that are phrased exclusively in terms of *low-level* send/receive commands, which should be instrumented with protocol-specific logic for each new construction. While it is possible to derive DPC's protlets in DISEL, extracting them and ascribing them suitable types requires large annotation overhead. To wit, only the protocol description for Two-Phase Commit in DISEL takes nearly 400 LOC of Coq, while the *entire* protocol, implementation *and* invariant for 2PC in DPC take only 243 LOC of Haskell. We believe that providing a concise reusable specification to advanced DPC protlets, such as `Quorum`, allowing for verification of, *e.g.*, Paxos, would be an interesting research challenge by itself.

The idea of exploiting random exploration of process interleavings in asynchrounous settings in general is not a new one. For instance, generating and controlling schedules of execution have been central in lines of work surrounding concurrency errors in web applications (Adamsen *et al.* 2017). We here similarly demonstrate the applicability of the approach in a lightweight framework inspired by a program logic.

6 Conclusion and Future Work

Declarative programming over distributed protocols is possible and, we believe, can lead to new insights, such as better understanding on how to structure systems implementations. Even though there are several known limitations to the design of DPC (for instance, in order to define new combinators, one needs to extend `Protlet`), we consider our approach beneficial and illuminating for the purposes of prototyping, exploration, and teaching distributed system design. In the future, we are going to explore the opportunities, opened by DPC, for randomised protocol testing and lightweight verification with refinement types.

Acknowledgements. We thank the JFP referees for their many helpful suggestions that helped to improve the presentation of the paper. Ilya Sergey's work has been supported by the grant of Singapore NRF National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) and by Crystal Centre at NUS School of Computing.

References

Adamsen, Christoffer Quist, Møller, Anders, Karim, Rezwana, Sridharam, Manu, Tip, Frank, & Sen, Koushik. (2017). Repairing event race errors by controlling nondeterminism. *Pages 289–299 of: ICSE*. ACM.

Ancona, Davide, Bono, Viviana, Bravetti, Mario, Campos, Joana, Castagna, Giuseppe, Deniélou, Pierre-Malo, Gay, Simon J., Gesbert, Nils, Giachino, Elena, Hu, Raymond, Johnsen, Einar Broch, Martins, Francisco, Mascardi, Viviana, Montesi, Fabrizio, Neykova, Rumyana, Ng, Nicholas, Padovani, Luca, Vasconcelos, Vasco T., & Yoshida, Nobuko. (2016). Behavioral types in programming languages. *Foundations and trends in programming languages*, **3**(2-3), 95–230.

Andersen, Kristoffer Just Arndal, & Sergey, Ilya. (2019a). Distributed protocol combinators. *Pages 169–186 of: PADL*. LNCS, vol. 11372. Springer.

Andersen, Kristoffer Just Arndal, & Sergey, Ilya. (2019b). *Distributed protocol combinators: Implementation*. `https://doi.org/10.5281/zenodo.3902686`.

Brady, Edwin. (2017). Type-driven development of concurrent communicating systems. *Computer science (AGH)*, **18**(3).

Chandra, Tushar, Griesemer, Robert, & Redstone, Joshua. (2007). Paxos made live: an engineering perspective. *Pages 398–407 of: PODC*. ACM.

Claessen, Koen, & Hughes, John. (2011). Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, **46**(4), 53–64.

Coq Development Team. (2020). *The Coq Proof Assistant Reference Manual*. Available from `http://coq.inria.fr`.

Desai, Ankush, Gupta, Vivek, Jackson, Ethan K., Qadeer, Shaz, Rajamani, Sriram K., & Zufferey, Damien. (2013). P: safe asynchronous event-driven programming. *Pages 321–332 of: PLDI*. ACM.

Desai, Ankush, Phanishayee, Amar, Qadeer, Shaz, & Seshia, Sanjit. (2018). Compositional Programming and Testing of Dynamic Distributed Systems. *PACMPL*, **2**(OOPSLA), 159:1–159:30.

Dinsdale-Young, Thomas, Dodds, Mike, Gardner, Philippa, Parkinson, Matthew J., & Vafeiadis, Viktor. (2010). Concurrent Abstract Predicates. *Pages 504–528 of: ECOOP*. LNCS, vol. 6183. Springer.

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

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Pages 48–59 of: ICFP*. ACM.

García-Pérez, Álvaro, Gotsman, Alexey, Meshman, Yuri, & Sergey, Ilya. (2018). Paxos Consensus, Deconstructed and Abstracted. *Pages 912–939 of: ESOP*. LNCS, vol. 10801. Springer.

Gommerstadt, Hannah, Jia, Limin, & Pfenning, Frank. (2018). Session-typed concurrent contracts. *Pages 771–798 of: ESOP*. LNCS, vol. 10801. Springer.

Gray, James N. (1978). Notes on data base operating systems. *Pages 393–481 of: In Operating Systems*. Springer.

Hawblitzel, Chris, Howell, Jon, Kapritsos, Manos, Lorch, Jacob R., Parno, Bryan, Roberts, Michael L., Setty, Srinath T. V., & Zill, Brian. (2015). IronFleet: proving practical distributed systems correct. *Pages 1–17 of: SOSP*. ACM.

Hüttel, Hans, Lanese, Ivan, Vasconcelos, Vasco T., Caires, Luís, Carbone, Marco, Deniélou, Pierre-Malo, Mostrous, Dimitris, Padovani, Luca, Ravara, António, Tuosto, Emilio, Vieira, Hugo Torres, & Zavattaro, Gianluigi. (2016). Foundations of session types and behavioural contracts. *ACM comput. surv.*, **49**(1), 3:1–3:36.

Jones, Cliff B. (1983). Tentative steps toward a development method for interfering programs. *Acm transactions on programming languages and systems*, **5**(4), 596–619.

Killian, Charles Edwin, Anderson, James W., Braud, Ryan, Jhala, Ranjit, & Vahdat, Amin M. (2007). Mace: Language support for building distributed systems. *Pages 179–188 of: PLDI*. ACM.

Kleppmann, Martin. 2016 (Feb). *How to do distributed locking*. https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html.

Krogh-Jespersen, Morten, Timany, Amin, Ohlenbusch, Marit Edna, Gregersen, Simon Oddershede, & Birkedal, Lars. (2020). Aneris: A mechanised logic for modular reasoning about distributed systems. *Pages 336–365 of: ESOP*. LNCS, vol. 12075. Springer.

Lamport, Leslie. (1998). The Part-Time Parliament. *ACM toplas*, **16**(2), 133–169.

Lamport, Leslie. (2001). *Paxos made simple*.

Lamport, Leslie, & Schneider, Fred B. (1985). Formal foundation for specification and verification. *Pages 203–285 of: Distributed Systems: Methods and Tools for Specification, An Advanced Course*. LNCS, vol. 190. Springer.

Lampson, Butler W. (1996). How to build a highly available system using consensus. *WDAG*.

Lange, Julien, & Tuosto, Emilio. (2012). Synthesising Choreographies from Local Session Types. *Pages 225–239 of: CONCUR*. LNCS, vol. 7454. Springer.

Leonini, Lorenzo, Riviere, Etienne, & Felber, Pascal. (2009). SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). *Pages 185–198 of: NSDI*. USENIX Association.

Liang, Sheng, Hudak, Paul, & Jones, Mark P. (1995). Monad transformers and modular interpreters. *Pages 333–343 of: POPL*. ACM Press.

Liu, Yanhong A., Stoller, Scott D., Lin, Bo, & Gorbovitski, Michael. (2012). From clarity to efficiency for distributed algorithms. *Pages 395–410 of: OOPSLA*. ACM.

Melgratti, Hernán C., & Padovani, Luca. (2017). Chaperone contracts for higher-order sessions. *Proc. ACM program. lang.*, **1**(ICFP), 35:1–35:29.

Nanevski, Aleksandar, Morrisett, Greg, Shinnar, Avi, Govereau, Paul, & Birkedal, Lars. (2008). Ynot: Dependent types for imperative programs. *Pages 229–240 of: ICFP*.

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

O'Hearn, Peter W., Reynolds, John C., & Yang, Hongseok. (2001). Local reasoning about programs that alter data structures. *CSL*. LNCS, vol. 2142. Springer.

Padon, Oded, McMillan, Kenneth L., Panda, Aurojit, Sagiv, Mooly, & Shoham, Sharon. (2016). Ivy: safety verification by interactive generalization. *Pages 614–630 of: PLDI*. ACM.

Pfenning, Frank, & Elliott, Conal. (1988). Higher-order abstract syntax. *Pages 199–208 of: PLDI*. ACM.

Pîrlea, George, & Sergey, Ilya. (2018). Mechanising blockchain consensus. *Pages 78–90 of: CPP*. ACM.

Sergey, Ilya, Nanevski, Aleksandar, Banerjee, Anindya, & Delbianco, Germán Andrés. (2016). Hoare-style Specifications as Correctness Conditions for Non-linearizable Concurrent Objects. *Pages 92–110 of: OOPSLA*. ACM.

Sergey, Ilya, Wilcox, James R., & Tatlock, Zachary. (2018). Programming and proving with distributed protocols. *PACMPL*, **2**(POPL), 28:1–28:30.

van Renesse, Robbert, & Altinbuken, Deniz. (2015). Paxos made moderately complex. *ACM comp. surv.*, **47**(3), 42:1–42:36.

Weikum, Gerhard, & Vossen, Gottfried. (2002). *Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann.

Wilcox, James R., Woos, Doug, Panchekha, Pavel, Tatlock, Zachary, Wang, Xi, Ernst, Michael D., & Anderson, Thomas E. (2015). Verdi: a framework for implementing and formally verifying distributed systems. *Pages 357–368 of: PLDI*. ACM.

Wilcox, James R., Sergey, Ilya, & Tatlock, Zachary. (2017). Programming Language Abstractions for Modularly Verified Distributed Systems. *Pages 19:1–19:12 of: SNAPL*.