Compositional and Automated Verification of Distributed Systems

James R. Wilcox

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Zachary Tatlock, Chair

Dan Grossman

Thomas Anderson

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Compositional and Automated Verification of Distributed Systems

James R. Wilcox

Chair of the Supervisory Committee:
Professor Zachary Tatlock
Paul G. Allen School of Computer Science & Engineering

Distributed systems provide the backbone for modern computer systems, from cloud computing to air-traffic control. These complex systems execute concurrently in unreliable environments and are expected to tolerate various faults. Such environments are notoriously difficult to adequately model with testing, but because of the critical importance of these systems, it is essential that they are correct. It thus makes sense to turn to more rigorous methods of ensuring correctness, such as formal verification. Applying formal methods is not a panacea, however, due to the complexity of the systems involved. It is not uncommon, e.g., for a distributed file system to coordinate thousands of machines using a combination of several different protocols to ensure consistency, fault tolerance, and high performance. Verifying such a system requires breaking the problem down into individually verifiable parts, and leveraging automation whenever possible.

This dissertation describes programming languages techniques for verifying distributed systems compositionally and automatically. First, we present Verdi, a framework for verifying distributed systems that reasons about fault tolerance mechanisms as transformers between fault models. Second, we detail Disel, a concurrent separation logic for distributed systems whose key insight is to treat the network as analogous to the heap in sequential programming. Finally, we report on `mypyvy`, a domain-specific language for symbolic transition systems in first-order logic, which supports a variety of automated reasoning tools to analyze systems.

TABLE OF CONTENTS

To Barkley

ACKNOWLEDGMENTS

It takes a village, and without my village, this thesis wouldn't have happened. My parents, Amy and Brian, always encouraged me to pursue my interests, and they put up with seven-year-old me's endless appetite for algebra problems. I've always looked up to my big brother Sam, even though he's two years younger, and to his family, Alexis and Charlie.

Zach: you've been everything I needed in an advisor. I couldn't have done it without you. And a special thank you for putting so much time into building the lab into what it is today. To my reading committee members, Dan and Tom, you are both such great role models for how to be an academic. I learned a ton from watching you work. To my GSR and friend, Geoffrey, my time in your choir was some of the only consistent (and much needed) structure to my schedule those first few years in grad school, and really helped me feel connected to UW and Seattle.

To all the members of the PLSE lab, what an incredible space to learn and hang out! To Doug, I learned so much from you (and your dad). Thanks for teaching me how to impersonate a systems person. Thanks to Colin, Konstantin, Pavel, Eric, Stuart, Calvin, Talia, John, Chandra, Max, and Remy for teaching me so much about PL while disguising it as shooting the breeze. Also a big thanks to all the incredible UW undergraduates I worked with over the years, Steve, Ryan, Pooja, Justin, Miranda, David, Ethan, and Taylor. You all contributed so much to our projects!

One of the best parts about getting a PhD is becoming a member of the international research community. I've had the pleasure of meeting many wonderful people and starting great collaborations with several of them. I met Ilya at the PLDI 2015 banquet, and it wasn't long before we were talking about the basic ideas of Disel. I met Karl at the ASE

2015 banquet, and it wasn't long before we were talking about Verdi and sensor networks. I met Oded at PLDI 2016, and it wasn't long before we were working on yet another proof of Raft's correctness. Oded introduced me to Mooly, Yotam, Sharon, and Marcelo, as well as to Ken and Giuliano. Everyone in this entourage has helped me realize that I am a collaborator at heart. Here's to many more projects together!

I had a beautiful time at my summer internship at Microsoft Research with Jay Lorch and Rustan Leino. To this day, one of my favorite things to do is to make Jay laugh. Rustan taught me pretty much everything I know about building verification tools, especially the insight that compiler engineering principles apply to them. That summer, I also had the chance to get to know Shaz Qadeer, who taught me how to think about refinement proofs and concurrency.

Finishing this thesis was not an easy journey for me. I left UW having defended but not submitted the document in June 2019, and at that point I was feeling pretty ground down and not able to make much progress. I consider it a small miracle that the document will finally be submitted in the summer of 2021. I found a very friendly and welcoming home in the intervening years at Certora, with Mooly and Shelly and the rest of the team. Big thanks to them for hiring me, and for keeping me around part time when I tried to quit. While working at Certora, Daniel Ricketts and Calvin at Oracle were excellent lunch buddies.

I wouldn't have been able to get back to making progress on the thesis without Zach's gentle guidance and project management. "Five minutes a day" was life changing for me a year ago, and I'm so happy to have seen it grow into a full community, first with Chandra and then with `#accountability-coop`. Finally, thanks to Race Condition Running (miles, not pace!), the choirs of Saint Mark's cathedral, and the species *Canis familiaris* for supporting my mental health through this process.

Chapter 1

INTRODUCTION

Distributed systems are widely used in everything from web infrastructure to airplanes, and their correctness is critical. These systems are notoriously hard to implement correctly because they are expected to tolerate execution in harsh environments, where concurrency and partial failure are a fact of life, all while no single node has access to a global view of the system state. Traditional testing techniques are inadequate for exploring the space of possible executions in the presence of concurrency and partial failure because this space includes the exponential number of interleavings of system events and failure events. Furthermore, even considering a particular interleaving of events, one faces all the usual difficulties with testing a sequential program, including the issue of achieving sufficient coverage.

Since testing is not enough for distributed systems, the research community has developed a rich body of work applying formal methods to exhaustively check correctness. Applying formal methods is not a panacea, however, because complex systems have complex proofs of correctness. It is not uncommon, e.g., for a distributed file system to coordinate thousands of machines using a combination of several different protocols to ensure consistency, fault tolerance, and high performance. The primary challenge to applying verification to distributed systems is high system complexity, which leads to high proof complexity.

We seek to address the challenge of proof complexity in verified distributed systems while meeting the following goals.

1. Our solutions should produce *running code*, not just models. (Note that this is different from saying we want to analyze *existing* implementations that were not designed for verification.)

2. Our solutions should support expressing *optimizations* that are essential for good performance in practice, such as efficient data structures and optimized protocols with, e.g., batching and piggybacking.

3. Our solutions should minimize developer effort.

Our approach is to use compositional reasoning to break down the verification problem into smaller parts, and then to apply automated decision procedures to the resulting pieces. In the face of high system complexity, decomposing the problem reduces proof complexity and increases automation. Decomposing proofs leads to two benefits. First, a truly compositional proof imposes no additional proof burden to put the pieces back together. Second, sufficiently decomposed pieces can be analyzed fully automatically using decision procedures.

For example, consider a system that consists of a replicated key-value store and a highly available lock-based resource manager service. (See Figure 1.1.) Both subsystems use consensus for reliability, and the key-value store uses the locking service to manage cross-key transactions. To verify such a system, we can decompose the proof along two dimensions. First, each subsystem's proof can be decomposed vertically, by separating the replication mechanism from the application logic. Second, the entire system's proof can be constructed by horizontally composing the subsystem proofs. Lastly, the application logic is expressed at a high enough level of abstraction that its specification can be analyzed fully automatically.

These considerations lead us to the central claim of this dissertation:

Techniques for composition and automation provide a basis for effective verification of distributed systems implementations.

The remainder of this dissertation addresses describes three aspects of our approach: composing fault-tolerance mechanisms with application logic (Verdi, Chapter 2), composing several protocols to build larger protocols (Disel, Chapter 3), and automating verification of high-level protocols with SMT solvers (`mypyvy`, Chapter 3). These techniques also satisfy

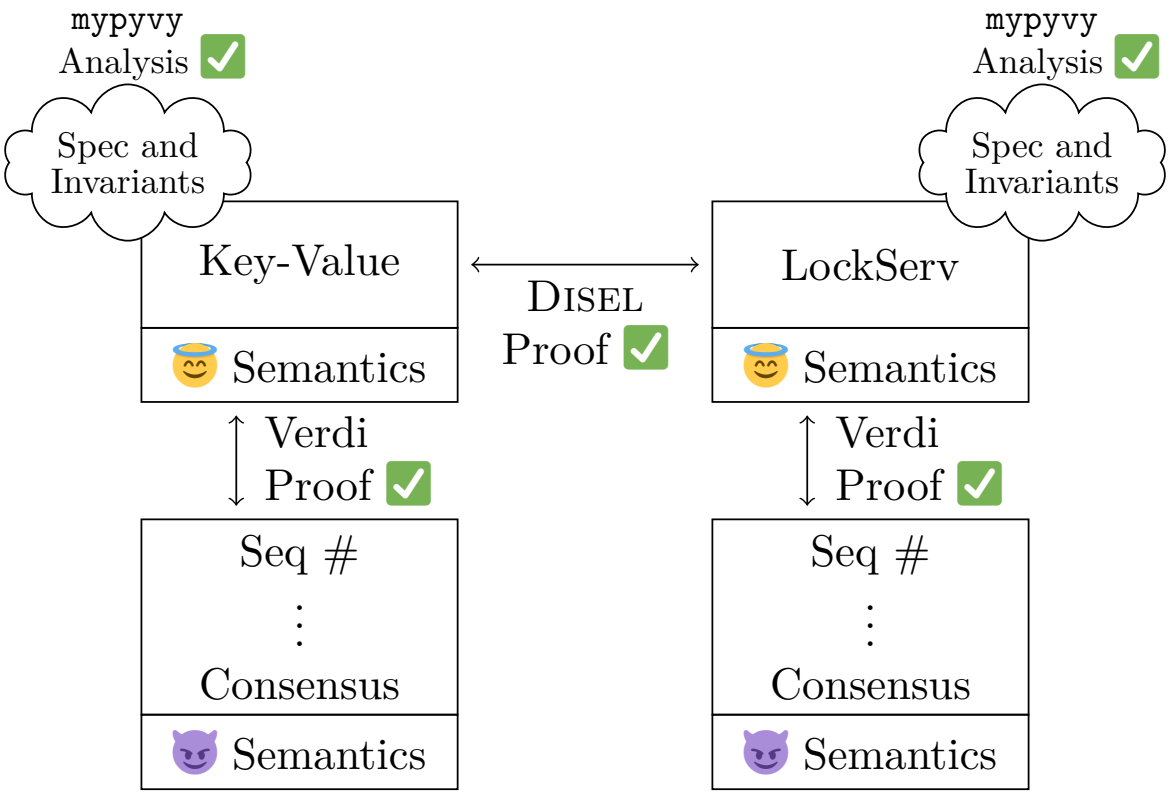


Figure 1.1: The structure of the proof of a verified distributed system using techniques from this dissertation.

our goals described above for working on running code with protocol optimizations while reducing developer effort.

Chapter 2 addresses complexity challenges due to fault tolerance by introducing Verdi, a Coq [17] framework for implementing and verifying distributed systems that supports compositional fault tolerance reasoning through *verified system transformers*. Verdi models system execution using various *network semantics*, each of which encodes assumptions about the environment including possible network faults and machine failures. Network semantics can range from the idealistic to the pessimistic. For example, one might assume that all messages are eventually delivered and that nodes never fail. On the other hand, one might

assume that packets can be dropped and duplicated and that some nodes behave arbitrarily or maliciously. Different systems are designed under different sets of assumptions, and network semantics capture those assumptions. The main point of defining a particular network semantics in Verdi is to verify distributed systems using that semantics as the fault model. Assuming the network semantics accurately describes all possible behaviors of the system's environment, a proof in Verdi guarantees that the system is correct for all executions. For common network semantics, there are typically generic mechanisms that systems use to tolerate faults, e.g., sequence numbering to handle message reordering and duplication. Another key benefit of clearly stating environment assumptions as network semantics is that one can then express these generic fault tolerance mechanisms as transformers between semantics. Using transformers, the engineer can implement and reason about their application in a fault model with relatively few faults, and then automatically transform the system into one that provably works in a more adversarial fault model with relatively more faults. We used Verdi to study the Raft consensus protocol [86], producing the first formal proof of safety and the first verified implementation. The Verdi framework was originally described in a PLDI 15 paper [117], and an extended case study proving the safety of the Raft consensus protocol was published in CPP 16 [118]. These papers were written together with a fantastic team of coauthors: Doug Woos, Pavel Panchekha, Steve Anton, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Chapter 2 is a combined narrative of these two papers with fixed typos and a more coherent description of the Raft proof and less emphasis on the proof engineering difficulties we encountered. The Verdi code is publicly available at `https://github.com/uwplse/verdi`.

Chapter 3 details Disel, a concurrent separation logic for distributed systems. Whereas Verdi separates fault tolerance reasoning from application logic (the vertical dimension of Figure 1.1), Disel separates reasoning about cooperating services by defining interfaces that capture protocol-specific invariants (the horizontal dimension). This supports verifying modern distributed systems which are typically built by composing several services to provide a high-level application. A proof of correctness for a system composed of several services

should similarly compose the guarantees of each individual service. Using the techniques of Verdi alone, such reasoning is not possible. Instead of reasoning across fault models, what we need is to abstract over the low-level details each component uses to provide its guarantee. To achieve this, we take inspiration from modern program logics for concurrent programs that manipulate heap pointers. These logics allow modular reasoning by separating different parts of the heap, each of which can be reasoned about independently. Disel applies this insight to distributed systems by analogizing the network as the heap. Thus, Disel separates the network messages of each protocol from each other, allowing independent reasoning. Disel achieves this through several logical mechanisms that support strengthening the invariants of other services with client-specific facts and capturing the essential interactions between protocols with *hooks*, which allow one protocol's actions to be conditioned on another protocol's state. Disel was originally published in SNAPL 17 and POPL 18 [102, 116], and would not have been possible without my wonderful collaborators, Ilya Sergey, Zachary Tatlock, and Miranda Edwards. Chapter 3 is a version of the POPL 18 paper with just a handful of typos fixed. The Disel code is publicly available at `https://github.com/DistributedComponents/disel`.

Experience in Disel and Verdi showed that the manual effort required to provide strong guarantees about distributed systems doesn't scale with current tooling. In particular, the key sticking point is developing inductive invariants. Deductive verification techniques, such as those used in Disel and Verdi, are highly expressive but require the user to provide a great deal of additional input, including inductive invariants and their proofs. For example, using Verdi to prove an application correct in a relatively nice fault model still requires deriving an application-specific invariant, which simultaneously (1) summarizes all the reachable states of the application, (2) is closed under the transition system's step relation, and (3) ensures the absence of safety violations. The difficulty of deriving these invariants inspired us to investigate techniques for automatically proving and even *inferring* inductive invariants.

Chapter 4 describes `mypyvy`, a tool for automated reasoning about symbolic transition systems in first-order logic. `mypyvy` takes an input file describing a symbolic transition

system and can perform a variety analyses, as requested by the user. Three of the most interesting analyses include inductive invariant checking, invariant inference, and bounded trace reasoning. In all cases, `mypyvy` loads the transition system and compiles it together with the user-requested analysis to a (sequence of) SMT queries, which are dispatched to Z3. The resulting workflow is significantly more automated than using Verdi or Disel to prove inductive invariants, and anecdotally is much more enjoyable to use for verification. On the other hand, the automation is sensitive to the complexity of the protocol being analyzed, so it is essential to *first* decompose the system vertically and horizontally and only *then* to apply `mypyvy`. `mypyvy` was originally developed to support a CAV 19 paper [25], but that paper was not about `mypyvy` per se. Chapter 4 contains previously unpublished material describing `mypyvy` itself and the theory behind it, and will become the primary reference for `mypyvy`. For the purposes of this dissertation, the main contribution of Chapter 4 is methodological: one can use `mypyvy` to automate the verification of distributed protocols once they have been sufficiently decomposed and abstracted. That said, we also have some experience to indicate that `mypyvy` will be useful as a platform for exploring further automated reasoning tasks. The development of `mypyvy` has benefited greatly from many contributors, especially Oded Padon, Yotam Feldman, Sharon Shoham, Mooly Sagiv, Jason R. Koenig, Ken McMillan, Alex Aiken, Giuliano Losa, Daniel Ricketts, Shachar Itzhaky, Lindsey Kuper, William Schultz, and Aaron Weiss. `mypyvy` is actively developed at `https://github.com/wilcoxjay/mypyvy`.

Distributed systems remain a crucial topic with many potential avenues for future work. In Chapter 5, we summarize our plans for extending Verdi, Disel, and `mypyvy` to further improve the verification experience. We are especially excited to live in a world where more users are empowered to verify their distributed systems.

Chapter 2

VERTICAL COMPOSITION: FAULT-TOLERANCE AND APPLICATION LOGIC

2.1 Introduction

This chapter addresses the "vertical" dimension of composition that is our first step on our overarching goal of making it easier to verify distributed systems. By vertical composition, we mean the separation of application logic from generic fault tolerance mechanisms. Our approach is to first describe several different fault models that a distributed system might run in, which we call *network semantics*. We can then describe fault tolerance mechanisms as adapters that allow a system designed for one fault model to run in another fault model. This separation of concerns allows reasoning about application logic in a relatively "nice" fault model (one with fewer failures). We have implemented our approach in a framework called Verdi. Later chapters will further compose several protocols horizontally and automate reasoning about application logic to build an entire system.

There are several design challenges related to vertical composition of verified distributed systems. First, we want to work with executable code rather than mathematical models. For performance reasons, real-world systems often diverge in important ways from their high-level mathematical descriptions [10]. If we were to reason only about models, bugs could creep into the *formality gap* between model and implementation. Second, distributed systems run in a diverse range of environments. For example, some networks may reorder packets, while other networks may also duplicate them. Verdi must support verifying applications against these different fault models. Third, it is difficult to prove that application-level guarantees hold in the presence of faults. Verdi aims to help the programmer separately prove correctness of application-level behavior and correctness of fault-tolerance mechanisms, and to allow these

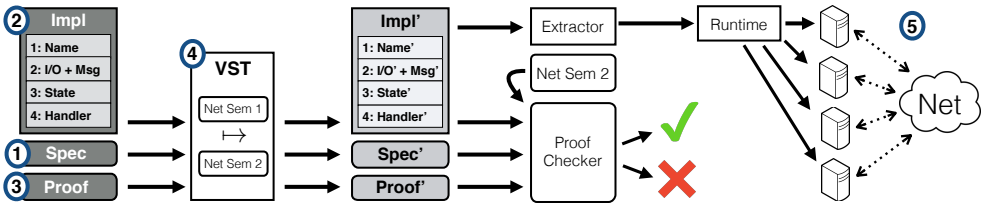


Figure 2.1: Verdi workflow. Programmers provide the dark gray boxes in the left column: the specification, implementation, and proof of a distributed system. Rounded rectangles correspond to proof-related components. To make the proof burden manageable, the initial proof typically assumes an unrealistically simple network model in which machines never crash and packets are never dropped or duplicated. A verified system transformer (VST) transforms the application into one that handles faults, as shown in the column of light gray boxes in the middle column. Note that the programmer does not write any code for this step. Verdi provides the white boxes, including verified systems transformers (VSTs), network semantics encoding various fault models, and extraction of an implementation to an executable. Programmers deploy the executable over a network for execution.

proofs to be easily composed.

Verdi addresses the above challenges with three key ideas. First, Verdi provides a Coq [17] toolchain for writing executable distributed systems and verifying them; this avoids a *formality gap* between the model and the implementation. Second, Verdi provides a flexible mechanism to specify fault models as *network semantics*. This allows programmers to verify their system in the fault model corresponding to their environment. Third, Verdi provides a *compositional* technique for implementing and verifying distributed systems by separating the concerns of application correctness and fault tolerance. This simplifies the task of providing end-to-end guarantees about distributed systems.

To achieve compositionality, we introduce *verified system transformers*. A system transformer is a function whose input is an implementation of a system and whose output is a new system implementation that makes different assumptions about its environment. A verified system transformer includes a proof that the new system satisfies properties analogous to those of the original system. For example, a Verdi programmer can first build and verify a system assuming a reliable network, and then apply a transformer to obtain another version

of their system that correctly and provably tolerates faults in an unreliable network (e.g., machine crashes).

Contributions. This chapter makes the following contributions: (1) Verdi, a toolchain for building provably correct distributed systems, (2) a set of formal network semantics with different fault models, (3) a compositional verification technique using verified system transformers, (4) case studies of implementing, and proving correct, practical distributed systems including a key-value store, a primary-backup replication transformer, and the first formally verified proof of linearizability for the Raft consensus protocol [86], and (5) an evaluation showing that these implementations can provide reasonable performance. Our key conceptual contribution is the use of verified systems transformers to enable modular implementation and verification of systems.

The rest of the chapter is organized as follows. Section 2.2 overviews the Verdi system. Section 2.3 details the small-step operational semantics that specify distributed system behavior in different fault models. Section 2.4 describes how systems in Verdi can be constructed from modular components. Sections 2.5–2.7 describe case studies of using Verdi to implement and verify distributed systems. Section 2.8 evaluates the performance of systems implemented in Verdi. Section 2.9 discusses related work, and Section 2.10 concludes.

2.2 Overview

Figure 2.1 illustrates the Verdi workflow. The programmer ① specifies a distributed system and ② implements it by providing four definitions: the names of nodes in the system, the external input and output and internal network messages that these nodes respond to, the state each node maintains, and the message handling code that each node runs. ③ The programmer proves the system correct assuming a specific baseline network semantics. In the examples in this chapter, the programmer chooses an idealized reliable model for this proof: all packets are delivered exactly once, and there are no node failures. ④ The programmer then selects a target network semantics that reflects their environment's fault model, and applies a verified system transformer (VST) to transform their implementation into one

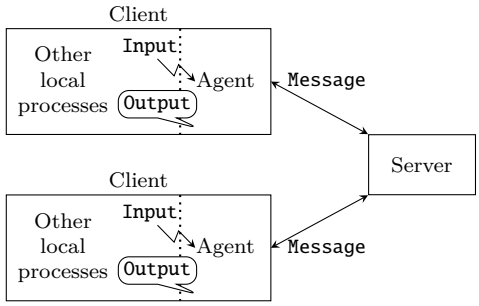


Figure 2.2: Architecture of a lock service application. Boxes represent separate physical nodes, while dotted lines separate processes running on the same node. Each client node runs an Agent process that exchanges input and output with other local processes. The Agent also exchanges network messages with the Server.

that is correct in that fault model. This transformation also produces updated versions of the specification and proof. ⑤ The verified system is extracted to OCaml, compiled to an executable, and deployed across the network.

The rest of this section describes each of these five steps, using a simple lock service as a running example. The lock service manages a single shared lock. Conceptually, clients communicate with the lock service using the following API: a client requests and releases a lock via the `Lock` and `Unlock` input messages, and the lock service grants a lock by responding with a `Grant` output message.

To provide this API, the lock service consists of a central lock Server node, and a lock Agent that runs on every client node, as illustrated in Figure 2.2. That is, each client node runs a lock Agent along with other client processes that access the API through the Agent. Each lock Agent communicates over the network with the central lock server. The Agent requests and releases the lock with the `LockMsg` and `UnlockMsg` network messages, and the server sends a `GrantMsg` network message to notify an Agent when it has received the lock.

```
(* 1 - node identifiers *)
Name := Server | Agent(int)

(* 2 - API, also known as external IO *)
Inp := Lock | Unlock
Out := Grant
(* 2 - network messages *)
Msg := LockMsg | UnlockMsg | GrantMsg

(* 3 - state *)
State (n: Name) :=
  match n with
  | Server => list Name (* head = agent holding lock      *)
                        (* tail = agents waiting for lock *)
  | Agent n => bool     (* true iff this agent holds lock *)

InitState (n: Name) : State n :=
  match n with
  | Server => []
  | Agent n => false

(* 4 - handlers for external input and internal messages *)
HandleInp (n: Name) (s: State n) (inp: Inp) :=
  match n with
  | Server => nop   (* server performs no external IO *)
  | Agent n =>
    match inp with
    | Lock => (* if client requests lock, forward to Server *)
      send (Server, LockMsg)
    | Unlock => (* if client requests unlock and lock held... *)
      if s then (
        (* update state and tell Server lock freed *)
        s := false;;
        send (Server, UnlockMsg))

HandleMsg (n: Name) (s: State n) (src: Name) (msg: Msg) :=
  match n with
  | Server =>
    match msg with
    | LockMsg => (* if lock not held, immediately grant *)
      if s == [] then send (src, GrantMsg);;
      s := s ++ [src] (* add requester to end of queue *)
    | UnlockMsg => (* head of queue no longer holds lock *)
      s := tail s;;
      (* grant lock to next waiting agent, if any *)
      if s != [] then send (head s, GrantMsg)
    | _ => nop (* never happens *)
  | Agent n =>
    match msg with
    | GrantMsg => (* we have the lock *)
      (* update state and notify external listeners *)
      s := true;;
      output Grant
    | _ => nop         (* never happens *)
```

Figure 2.3: A simple lock service application implemented in Verdi, under the assumption of a reliable network. Verdi extracts these definitions into OCaml and links the resulting code with a runtime to send and receive messages over the network.

2.2.1 Specification

A Verdi programmer specifies the correct behavior of their system in terms of *traces*, the sequences of external input and output generated by nodes in the system. For the lock service application, correctness requires mutual exclusion: no two distinct nodes should ever simultaneously hold the lock. This mutual exclusion property can be expressed as a predicate over traces:

$$\begin{aligned}&\texttt{mutex}(\tau) :=\\ &\quad \tau \;=\; \tau_1 \mathbin{+\!\!+} \langle n_1,\ \texttt{Grant}\rangle \mathbin{+\!\!+} \tau_2 \mathbin{+\!\!+} \langle n_2,\ \texttt{Grant}\rangle \mathbin{+\!\!+} \tau_3\\ &\quad \rightarrow \langle n_1,\ \texttt{Unlock}\rangle \in \tau_2\end{aligned}$$

To hold on trace τ, the `mutex` predicate requires that whenever `Grant` is output on node n_1 and then later `Grant` is output on node n_2, there must first be an intervening `Unlock` input from n_1 releasing the lock.

A system implementation satisfies specification Φ in a particular network semantics if for all traces τ the system can produce under that semantics, Φ holds on τ. For the example lock service application, an implementation satisfies `mutex` in a given semantics if `mutex` holds on all the traces produced under that semantics.

2.2.2 Implementation

Figure 2.3 shows the definitions a programmer provides to implement the lock service application in Verdi. (1) `Name` lists the names of nodes in the system. In the lock service application, there is a single Server node and an arbitrary number of Agents. (2) `Inp` and `Out` define the API of the lock service — the external input and output exchanged between an Agent and other local processes on its node. `Msg` defines network messages exchanged between Agents and the central Server. (3) `State` defines the state maintained at each node. Node state is defined as a dependent type where a node's name determines the data maintained locally at that node. In the lock service, the Server maintains a queue of Agent nodes, initially empty, where the head of the queue is the Agent currently holding the lock and the

rest of the queue represents the Agents which are waiting to acquire the lock. Each Agent maintains a boolean, initially false, which is true exactly when that Agent holds the lock. (4) The handler functions `HandleInp` and `HandleMsg` define how nodes respond to external input and to network messages.

This implementation assumes a reliable network where machines never crash and packets may be reordered but are not dropped or duplicated. These assumptions reduce the programmer's effort in both implementing the application and proving it correct. Section 2.2.4 shows how Verdi can automatically transform the lock service application into a version that tolerates faults.

When the system runs, each node listens for events and responds by running the appropriate handler: `HandleInp` for external input and `HandleMsg` for network messages. When an Agent receives an external input that requests to acquire or release the lock, it forwards the request to the Server; in the `Unlock` case, it first checks to ensure that the lock is actually held, and it resets its local state to `false`. Because the network is assumed to be reliable, no acknowledgment of the release is needed from the Server. When the Server receives a `LockMsg` network message, if the lock is not held, the server immediately grants the lock, and always adds the requesting Agent to the end of the queue of nodes. When the Server receives an `UnlockMsg` message, it removes a node from the head of its queue of Agents and grants the lock to the next Agent in the queue, if any. When an Agent receives a `GrantMsg` message, it produces external output (`Grant`) to inform other processes running on its node that the lock is held.

The application will be deployed on some network, and *network semantics* capture assumptions about the network's behavior. For this example, we assume a semantics encoding a reliable network. In a reliable network, each step of execution either (1) picks an arbitrary node and delivers an arbitrary external input, runs that node's input handler, and updates the state, or (2) picks a message in the network, runs the recipient's message handler, and updates the state.

Figure 2.4 shows an execution of the lock service application with two agents. Agents A_1

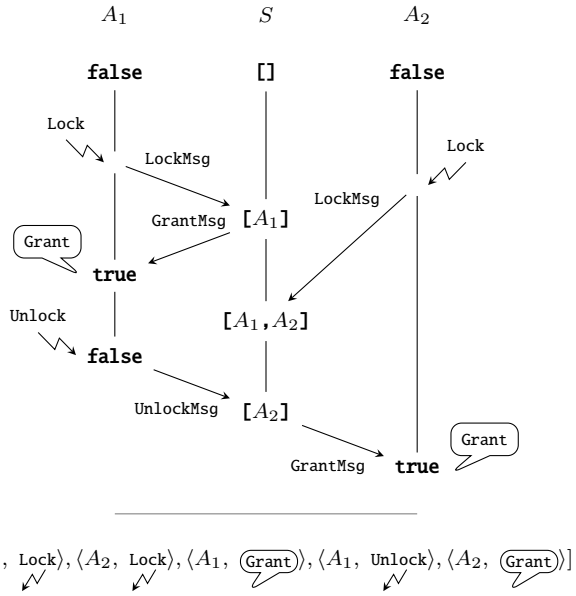


Figure 2.4: The behavior of the lock service application, with one server S and two agents A_1 and A_2. Each agent starts with the state `false`, and the server starts with an empty queue. Time flows downward. In response to external input (drawn with lightning-bolt arrows) and network messages, the nodes exchange messages and update local state. External output is shown as speech bubbles. The trace of this execution is shown at the bottom; note that only externally-visible events (external input and output) appear in the trace.

and A_2 both try to acquire the lock. The service first grants the lock to A_1. Once A_1 releases the lock, the service grants it to A_2. Note that, because our network semantics does not assume messages are delivered in the same order in which they were sent, there is a potential race condition: an agent can attempt to re-acquire the lock before the server has processed its previous release. In that case, the server simply (and correctly) adds the sender to the queue again. Using Verdi, the lock service is guaranteed to behave correctly even in such corner cases.

2.2.3 Verifying the Lock Service Application

We briefly outline the proof of the `mutex` property for the lock service application in the reliable network environment (i.e., no machine crashes nor packet loss/duplication). The

proof that `mutex` holds on all traces of the lock service application consists of three high-level steps: (1) prove an invariant about the reachable node and network states of the lock service application, (2) relate these reachable states to the producible traces, and (3) show that the previous two steps imply `mutex` holds on all producible traces.

The first step proves that all reachable system states satisfy the $\texttt{mutex}_{\texttt{state}}$ property:

$$\begin{aligned}
&\texttt{mutex}_{\texttt{state}}(\Sigma,\ P)\ :=\\
&\quad \forall\ n\ m,\ n \neq m \rightarrow \neg\texttt{hasLock}(\Sigma,\ n) \lor \neg\texttt{hasLock}(\Sigma,\ m)\\
&\texttt{hasLock}(\Sigma,\ n)\ :=\\
&\quad \Sigma(\text{Agent}(n)) = \texttt{true}
\end{aligned}$$

The function Σ maps node names to their state, and P is the set of in-flight packets. The property $\texttt{mutex}_{\texttt{state}}$ ensures that at most one Agent node holds the lock at a time.

A programmer can verify the $\texttt{mutex}_{\texttt{state}}$ property by proving an *inductive state invariant*. A property ϕ is an inductive invariant if both (1) it holds in the initial state, $(\Sigma_0,\ \emptyset)$, where Σ_0 maps each node to its initial state and $\emptyset$ represents the initial, empty network, and also (2) whenever it holds in some state, $(\Sigma,\ P)$, and $(\Sigma,\ P)$ can step to $(\Sigma',\ P')$, then it holds in $(\Sigma',\ P')$.

One inductive state invariant for $\texttt{mutex}_{\texttt{state}}$ is:

$$\begin{aligned}
&(\forall\ n, \texttt{hasLock}(\Sigma,\ n) \rightarrow \texttt{atHead}(\Sigma,\ n))\\
\land\ &(\forall\ p \in P,\ p.\texttt{body} = \texttt{GrantMsg} \rightarrow \texttt{grantee}(\Sigma,\ p.\texttt{dest}))\\
\land\ &(\forall\ p \in P,\ p.\texttt{body} = \texttt{UnlockMsg} \rightarrow \texttt{grantee}(\Sigma,\ p.\texttt{source}))\\
\land\ &\texttt{at_most_one}\ \{\texttt{GrantMsg},\ \texttt{UnlockMsg}\}\ P
\end{aligned}$$

where

$$\begin{aligned} \texttt{atHead}(\Sigma,\ n) \quad &:= \quad \exists\ t,\ \Sigma(\text{Server}) = n :: t \\ \texttt{grantee}(\Sigma,\ n) \quad &:= \quad \texttt{atHead}(\Sigma,\ n) \wedge \neg \texttt{hasLock}(\Sigma,\ n). \end{aligned}$$

The first conjunct above ensures that the Server and Agents agree on who holds the lock. The second and third conjuncts state that `GrantMsg` is never sent to an agent that already holds the lock, and that `UnlockMsg` is never sent from an agent that still holds the lock. Finally, the last conjunct states that there is at most one in-flight message in the set $\{\texttt{GrantMsg}, \texttt{UnlockMsg}\}$; this is necessary to ensure that neither of the previous two conjuncts is violated when a message is delivered. We proved in Coq that this invariant is inductive and that it implies $\texttt{mutex}_{\texttt{state}}$; the proof is approximately 500 lines long.

The second step of the proof relates reachable states to the traces a system can produce:

$$\begin{aligned} &\texttt{trace\_state\_agreement}(\tau,\ \Sigma)\ := \\ &\quad \forall\ n,\ \texttt{lastGrant}(\tau,\ n) \leftrightarrow \texttt{hasLock}(\Sigma,\ n) \\ &\texttt{lastGrant}(\tau,\ n)\ := \exists\ \tau_1\ \tau_2, \\ &\quad \tau = \tau_1 +\!\!+ \langle n,\ \texttt{Grant} \rangle :: \tau_2 \wedge \forall\ m,\ \langle m,\ \texttt{Unlock} \rangle \notin \tau_2 \end{aligned}$$

This property requires that whenever a `Grant` output appears in the trace without a corresponding `Unlock` input, that agent's flag is true (and vice versa). The proof of this property is by induction on the possible behavior of the network.

The third step of the proof shows that together, $\texttt{mutex}_{\texttt{state}}$ and `trace_state_agreement` imply that `mutex` holds on all traces of the lock service application under the reliable semantics. This result follows from the definitions of `mutex`, $\texttt{mutex}_{\texttt{state}}$, and `trace_state_agreement`.

2.2.4 Verified System Transformers

We have proved the `mutex` property for a reliable environment where the network does not drop or duplicate packets and the server does not crash. Assuming such a reliable environment simplifies the proof by allowing the programmer to consider fewer cases. To transfer the property into an unreliable environment with network and machine failures, a programmer uses Verdi's verified system transformers. As illustrated by Figure 2.1 part ④, after verifying a distributed system in one network semantics, a programmer can apply a verified system transformer to produce another version of their system which provides analogous guarantees in another network semantics.

In general, there are two types of transformers in Verdi: *transmission transformers* that handle network faults like packet duplication and drops and *replication transformers* that handle node crashes. Below we describe an example transmission transformer for the lock service application and briefly overview replication transformers, deferring details to Section 2.7.

Tolerating network faults. Figure 2.3's implementation of the lock service application will *not* function correctly in a network where messages can be duplicated. If an `UnlockMsg` message is duplicated but the agent reacquires the lock before the second copy is delivered, the server will misinterpret the duplicated `UnlockMsg` message as releasing the second lock acquisition.

Realistically, most developers would not run into this issue, as correct TCP implementations reject duplicate transmissions. However, some distributed systems need to handle deduplication and retransmission at a higher level, or choose not to trust the guarantees provided by unverified TCP implementations.

As another option, a programmer could rewrite the lock service—for instance, by including a unique identifier with every `GrantMsg` and `UnlockMsg` message to ensure that they are properly paired. The developer would then need to re-prove system correctness for this slightly different system in the semantics that models packet-duplicating networks. This

would require finding a new inductive invariant and writing another proof.

Verdi allows developers to skip these steps. Verdi provides a system transformer that adds sequence numbers to every outgoing packet and ignores packets with sequence numbers that have already been seen. Applying this transformer to the lock service yields both a new system *and a proof* that the new system preserves the `mutex` property even when packets are duplicated by the underlying network. Section 2.4 further details this transformer.

More generally, Verdi decouples the verification of application-level guarantees from the implementation and verification of fault-tolerance mechanisms. Verdi provides a collection of verified system transformers which allow the developer to transfer guarantees about a system in one network semantics to analogous guarantees about a transformed version of the system in another network semantics. This allows a programmer to build and verify their system against an idealized semantics and use a verified system transformer to obtain a version of the system that provably tolerates more realistic faults while guaranteeing system correctness properties.

Tolerating machine crashes. Verdi also provides verified system transformers to tolerate machine crashes via replication. Such replication transformers generally create multiple copies of a node in order tolerate machine crashes. This changes the number of nodes when transforming a system, which we discuss further in Section 2.7. (By contrast, transmission transformers like the one described above generally preserve the number of nodes and the relationships among them when transforming a distributed system.)

2.2.5 Running the Lock Service Application

Now we have a formally verified lock service, written in Coq, that tolerates message duplication faults. To obtain an executable for deployment, a Verdi programmer invokes Coq's built-in extraction mechanism to generate OCaml code from the Coq implementation, compile it with the OCaml compiler, and link it with a Verdi shim. The shim is written in OCaml; it implements network primitives (e.g., packet send/receive) and an event loop that invokes the appropriate event handler for incoming network packets, IO, or other events.

2.2.6 Summary

We have demonstrated how to use Verdi to establish a strong guarantee of the `mutex` property for the lock service application running in a realistic environment. Specifically, a programmer first specifies, implements, and verifies an application assuming a reliable environment. The programmer then applies system transformers to obtain a version of their application that handles faults in a provably correct way.

Verdi's trusted computing base includes the following components: the specifications of verified applications, the assumption that Verdi's network semantics match the physical network, the Verdi shim, Coq's proof checker and OCaml code extractor, and the OCaml compiler and runtime.

Verdi currently supports verifying safety properties, but not liveness properties, and none of Verdi's network semantics currently capture Byzantine fault models. We believe that Verdi could be extended to support these features: liveness properties could be verified by supporting infinite traces and adding fairness hypotheses as axioms as in TLA [59], while Byzantine fault models can be supported by adding more nondeterminism in the network semantics.

2.3 Network Semantics

The correctness of a distributed system relies on assumptions about its environment. For example, one distributed system may assume a reliable network, while others may be designed to tolerate packet reordering, loss, or duplication. To enable programmers to reason about the correctness of distributed systems in the appropriate environment model, Verdi provides a spectrum of *network semantics* that encode possible system behaviors using small-step style derivation rules.

This section presents the spectrum of network semantics that Verdi provides, ranging from single-node systems that do not rely on the network, through models of increasingly unreliable packet delivery (reordering, drops, and duplication), and culminating with a model

$$\frac{H_{\text{inp}}(\sigma,\ i) = (\sigma',\ o)}{(\sigma,\ T) \rightsquigarrow_{\text{s}} (\sigma',\ T +\!\!+ \langle i,\ o \rangle)}\ \textsc{Input}$$

Figure 2.5: Single-node semantics. The derivation rule above encodes possible behaviors of a single-node system that does not rely on the network. When the node is in state σ with input/output trace T, it may receive an arbitrary input i, and respond by running its input handler $H_{\text{inp}}(\sigma,\ i)$, which generates both the next state σ' and a list of outputs o. The Input rule relates the two states of the world $(\sigma,\ T) \rightsquigarrow_{\text{s}} (\sigma',\ T +\!\!+ \langle i,\ o \rangle)$ to reflect that the node has updated its state to σ' and sent outputs o in response to input i. Verifying properties of such single-node systems (i.e., state machines) is useful when they are replicated over a network to provide fault tolerance.

that permits arbitrary node crashes under various recovery assumptions. Each of these semantics is useful for reasoning about different types of systems. For example, the properties of single-node systems can be extended to handle node failures using protocols like Raft, while packet duplication semantics is useful for verifying packet delivery even in the face of reconnection, something that raw TCP does not support.

In Verdi, network semantics are defined as step relations on a "state of the world". The state of the world differs among network semantics, but always includes a trace of the system's external input and output. For example, many semantics include a bag of in-flight packets that have been sent by nodes in the system but have not yet been delivered to their destinations. Each network semantics is parameterized by system-specific data types and handler functions. Below we detail several of the network semantics Verdi currently provides.

Single-node semantics We begin with a simple semantics for single-node systems that do not use the network, i.e., state machines. This semantics is useful for proving properties of single-node systems; these can be extended, using a verified system transformer based on Raft, to provide fault tolerance. The single-node semantics, shown in Figure 2.5, models

$$\frac{H_{\text{inp}}(n,\ \Sigma[n],\ i) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow_{\text{r}} (P \uplus P',\ \Sigma',\ T +\!\!+ \langle i,\ o \rangle)}\ \textsc{Input}$$

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow_{\text{r}} (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}\ \textsc{Deliver}$$

Figure 2.6: Reordering semantics. The derivation rules above encode the behavior of systems running on networks that may arbitrarily reorder packet delivery. The network is modeled as a bag (i.e., a multiset) P of *packets*, which contain source and destination node names as well as a message. The state at each node in the network is a map Σ from node names to system-defined data. The INPUT rule passes arbitrary input i to the input handler H_{inp} for a given node n in state σ, which generates the next state σ', a list of outputs o, and a multiset of new packets P'. The outputs are added to the externally-visible trace, while the packets are added to the network (using the multiset-union operator $\uplus$). The DELIVER rule works similarly, except that instead of running a handler in response to arbitrary input, the network handle H_{net} is run on a packet taken from the network.

systems of a single node that respond to input by modifying their state and producing output. The node's behavior is described by a handler H_{inp}, which takes the current local state and an input and returns the new state and a list of outputs. The state of the world in this semantics is the node's state σ paired with a trace T that records the inputs sent to the system along with the outputs the system generates. The only step, INPUT, delivers an arbitrary input i to the handler H_{inp} and records the results in the next state. The squiggly arrow between two states indicates that a system in the state of the world on the left of the arrow may transition to the state of the world on the right of the arrow when all of the preconditions above the horizontal bar are satisfied. The node's state is updated, and the trace is extended with the input i and the output o.

Reordering semantics The reordering semantics, shown in Figure 2.6, models a system running on multiple nodes where packets are always delivered but may be arbitrarily re-

$$\frac{p \in P}{(P,\ \Sigma,\ T) \rightsquigarrow_{\text{dup}} (P \uplus \{p\},\ \Sigma,\ T)}\ \textsc{Duplicate}$$

Figure 2.7: Duplicating semantics. The duplicating semantics includes all the derivation rules from the reordering semantics, which we elide for space. In addition, it includes the DUPLICATE rule, which duplicates an arbitrary packet in the network. This represents a simple class of network failure in which a network misbehaves by delivering the same packet multiple times.

ordered. This was the "reliable" semantics initially used for the lock service implementation in Section 2.2. Each node communicates with other processes running on the same host via input and output, just as in the single-node semantics. Nodes can also exchange *packets*, which are tuples of the form (source, destination, message), over a network that may reorder packets arbitrarily but does not drop, duplicate, or fabricate them. The behavior of nodes is described by two handler functions. The input handler, H_{inp}, is run whenever a node receives input from another process on the same host. H_{inp} takes as arguments the node on which it is running, the current local state, and the input that was delivered. It returns the new local state and a list of outputs and packets to be processed by the semantics. Similarly, the network handler, H_{net}, is run whenever a packet is delivered from the network. H_{net} takes as arguments the receiver of the packet, the sender of the packet, the local state, and the message that was delivered.

A state of the world in the reordering semantics consists of a bag of in-flight packets P, a map from nodes to their local state Σ, and a trace T. The two rules in the reordering semantics, INPUT and DELIVER, respectively, model input from other processes on the node's host (i.e., the "outside world") and delivery of a packet from the network, where the corresponding handler function executes as described above. Delivered packets are removed from the bag of in-flight packets. Input and output are recorded in the trace; new packets are added to the bag of in-flight packets.

$$\frac{}{(\{p\} \uplus P,\ \Sigma,\ T) \rightsquigarrow_{\text{drop}} (P,\ \Sigma,\ T)}\ \textsc{Drop}$$

$$\frac{H_{\text{tmt}}(n,\ \Sigma[n]) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow_{\text{drop}} (P \uplus P',\ \Sigma',\ T +\!\!+ \langle \text{tmt},\ o \rangle)}\ \textsc{Timeout}$$

Figure 2.8: Dropping semantics. The dropping semantics includes the two rules above in addition to all the derivation rules from the duplicating semantics. The Drop rule allows a network to arbitrarily drop packets. Systems which tolerate dropped packets need to retransmit some messages, so the dropping semantics also includes a Timeout rule, which fires a node's timeout handler H_{tmt}. The Verdi shim implements this by setting system-defined timeouts after every event; if another event has not occurred on a given node before the timeout fires, the system's H_{tmt} handler is executed. Note that the semantics do not explicitly model time and allow timeouts to occur at any step.

Duplicating semantics The duplicating semantics, shown in Figure 2.7, extends the reordering semantics to model packet duplication in the network. In addition to the Input and Deliver rules from the reordering semantics, the duplicating semantics includes the rule Duplicate, which adds an additional copy of an in-flight packet to the network.

Dropping semantics Figure 2.8 specifies a network that drops arbitrary in-flight packets. The Drop rule allows any packet in the in-flight bag P to be dropped. However, simply adding this rule to the semantics would make it very difficult to write working systems, since handler functions only execute when packets are delivered and packets may be arbitrarily dropped. Real networked systems handle the possibility that packets can be dropped by setting timeouts, which execute if a certain amount of time has elapsed without receiving some other input or packet. We model this behavior in the Timeout rule: a timeout can be delivered to any node at any time, and will execute the node's H_{tmt} handler.

Node failure There are many possible models for node failure. Some systems assume that nodes will always return after a failure, in which case node failure is equivalent to a very

$$\frac{n \notin F}{(P,\ \Sigma,\ F,\ T) \rightsquigarrow_{\text{fail}} (P,\ \Sigma,\ \{n\} \cup F,\ T)}\ \textsc{Crash}$$

$$\frac{n \in F \qquad H_{\text{rbt}}(n,\ \Sigma[n]) = \sigma' \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ F,\ T) \rightsquigarrow_{\text{fail}} (P,\ \Sigma',\ F - \{n\},\ T)}\ \textsc{Reboot}$$

Figure 2.9: Failure semantics. The node failure semantics represents a network in which nodes can both stop and start, and adds a set of failed nodes F to the state of the world. The node failure semantics includes all the derivation rules from the dropping semantics in addition to the rules above. The rules from the drop semantics are modified to only run when node n is not in the set of failed nodes F. The Crash rule simply adds a node to the set of failed nodes F. Crashed nodes may re-enter the network via the Reboot rule, at which point their state is restored according to the H_{rbt} function.

long delay. Others assume that nodes will never return to the system once they have failed. Verdi's semantics for node failure, illustrated in Figure 2.9 assumes that nodes can return to the system and that all, some, or none of their state will be preserved (i.e., read back in from non-volatile storage). The state of the world in the node failure semantics includes a set F containing the nodes which have failed. The rules from the drop semantics are included in the failure semantics, but each with an added precondition to ensure that only live nodes (i.e., nodes that are not in F) can receive external input, network packets, or timeouts. A node can fail (be added to F) at any time, and failed nodes can return at any time. When a failed node returns, the H_{rbt} (reboot) function is run on its pre-failure state to determine what state survives the failure.

Low-level details Verdi's network semantics currently elide low-level network details. For example, input, output, and packets are modeled as abstract datatypes rather than bits exchanged over wires, and system details such as connection state are not modeled. This level of abstraction simplifies Verdi's semantics and eases both implementation and proof.

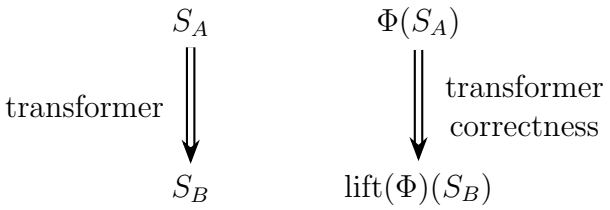


Figure 2.10: A verified system transformer takes a system (S_A) written against some network semantics and returns a new system (S_B) in another semantics. Its correctness property states that for any property Φ of the original system, a lifted version of that property holds on the transformed system.

Lower-level semantics could be developed and connected to the semantics presented here via system transformers, as described in the next section. This would further reduce Verdi's trusted computing base and increase our confidence in the guarantees Verdi provides.

2.4 Verified System Transformers

Verdi's spectrum of network semantics enable the programmer to reason about their system running in the fault model corresponding to their environment. However, directly verifying a system in a realistic fault model requires establishing both application-level guarantees and the correctness of fault-tolerance mechanisms simultaneously. Verdi provides verified system transformers to separate these concerns and enable a modular approach to building and verifying distributed systems. The programmer can assume an idealized network while verifying application-level guarantees and then apply a transformer to obtain a system that tolerates more faults while providing analogous guarantees.

For common fault models, the distributed systems community has developed standard techniques to handle failures. For example, as discussed in Section 2.2, by adding a unique sequence number to every message and ignoring previously received messages, systems can handle packet duplication. Verdi supports such standard fault-tolerance mechanisms through verified system transformers, which *transform* systems from one semantics to another while

guaranteeing that analogous system properties are preserved. See Figure 2.10. For example, in the transformer that handles deduplication, any property that holds on the underlying system is true of the transformed system when sequence numbers are stripped away.

System transformers are implemented as wrappers around the system's state, messages, and handlers. Messages and state are generally transformed to include additional fields. Handlers in the transformed system call into underlying handlers and implement additional functionality. The underlying handlers are called with underlying state and underlying messages, capturing the intuition that the underlying handlers are unable to distinguish whether they are running in their original network semantics or the new semantics targeted by the system transformer.

System transformers in Verdi are generally either *transmission transformers*, which tolerate network faults by adding functionality to every node in a system, or *replication transformers*, which tolerate node failures by making several copies of the underlying nodes. The sequence numbering transformer discussed below is an example of a transmission transformer. Sections 2.6 and 2.7 discuss replication transformers.

2.4.1 Sequence Numbering Transformer

Sequence numbering is a technique for ensuring that messages are delivered at most once. Senders tag each outgoing message with a sequence number that is unique among all messages from that sender. Message recipients keep track of all ⟨`number`, `sender`⟩ pairs they have seen. If a message arrives with a ⟨`number`, `sender`⟩ pair that the destination has seen before, the message is discarded.

Figure 2.11 shows the Verdi implementation of the sequence numbering transformer, `SeqNum`. It takes a distributed system S as input and produces a new distributed system that implements sequence numbering by wrapping the message, state, and handler definitions in S. `SeqNum` leaves the `Name`, `Inp`, and `Out` types unchanged. It adds an integer field to each message which is used as a sequence number to uniquely identify messages. `SeqNum` also adds a list of (`Name`, `int`) pairs to the state to track the sequence numbers received from

```
(* S describes a system in the reordering semantics *)
SeqNum (S) :=
  Name := S.Name

  Inp := S.Inp
  Out := S.Out
  Msg := { seqnum: int; underlying_msg: S.Msg }

  State (n: Name) := { seen: list (Name * int);
                       next_seqnum: int;
                       underlying_state: S.State n }

  InitState (n: Name) := { seen := [];
                           next_seqnum := 0;
                           underlying_state := S.InitState n }

  HandleInp (n: Name) (s: State n) (inp: Inp) :=
    wrap_result (S.HandleInp (underlying_state s) inp)

  HandleMsg (n: Name) (s: State n) (src: Name) (msg: Msg) :=
    if not (contains s.seen (src, msg.seqnum)) then
      s.seen := (src, msg.seqnum) :: s.seen;;
      (* wrap_result adds sequence numbers to messages while
         incrementing next_seqnum *)
      wrap_result (S.HandleMsg n (underlying_state s)
                                 src (underlying_msg msg))
```

Figure 2.11: Pseudocode for the sequence numbering transformer.

other nodes in the system, as well as an additional counter to track the local node's current maximum sequence number. The initial state in the wrapped system is constructed by building the initial state for the underlying system and then setting all sequence numbers to zero. To handle messages, the wrapped handler checks the input message to determine if it has previously been processed: if so, the message is simply dropped; otherwise, the message is passed to the message handler of S. Messages sent by the underlying handler are paired with fresh sequence numbers and the sequence number counter is incremented appropriately using the helper function `wrap_result`. The input handler passes input through to the input handler from S and wraps the results.

2.4.2 Correctness of Sequence Numbering

Given a proof that property Φ holds on every trace of an underlying system, the correctness of a system transformer should enable a programmer to easily establish an analogous property

Φ' of traces in the transformed system.

Each verified system transformer T provides a function `transfer` which translates properties of traces in the underlying semantics $\rightsquigarrow_1$ to the target semantics $\rightsquigarrow_2$:

$$\forall\ \Phi\ S, \texttt{holds}(\Phi,\ S,\ \rightsquigarrow_1) \rightarrow$$
$$\texttt{holds}(\texttt{transfer}(\Phi),\ T(S),\ \rightsquigarrow_2)$$

where $\texttt{holds}(\Phi,\ S,\ \rightsquigarrow)$ asserts that a property Φ is true of all traces of a system S under the semantics defined by $\rightsquigarrow$. Crucially, the `transfer` function defines how properties of the underlying system are translated to analogous properties of the transformed system.

For the sequence numbering transformer, $\rightsquigarrow_1$ is $\rightsquigarrow_{\mathrm{r}}$ (the step relation for the reordering semantics) and $\rightsquigarrow_2$ is $\rightsquigarrow_{\mathrm{dup}}$ (the step relation for the duplicating semantics). The `transfer` function is the identity function: properties of externally visible traces are precisely preserved by the transformation. Intuitively, the external output depends only on the wrapped state of the system, and the wrapped state is preserved by the transformer.

We prove that the wrapped state is preserved by *backward simulation*: for any step the transformed system $T(S)$ can take, the underlying system S can take an equivalent step. We specify this using helper functions `unwrap` and $\texttt{dedup}_{\texttt{net}}$. Given the global state of the transformed system, `unwrap` returns the underlying state at each node. Given the global state of the transformed system and the bag of in-flight messages, $\texttt{dedup}_{\texttt{net}}$ returns a bag of packets which includes only those messages which will actually be delivered to the underlying handlers—non-duplicate packets which have not yet been delivered. The simulation is specified as follows, where $\rightsquigarrow^\star_{\mathrm{dup}}$ and $\rightsquigarrow^\star_{\mathrm{r}}$ are the reflexive transitive closures of the duplicating semantics and the reordering semantics, respectively:

$$(\Sigma_0,\ \emptyset,\ \emptyset) \rightsquigarrow^\star_{\mathrm{dup}} (\Sigma,\ P,\ T) \rightarrow$$
$$(\texttt{unwrap}(\Sigma_0),\ \emptyset,\ \emptyset) \rightsquigarrow^\star_{\mathrm{r}} (\texttt{unwrap}(\Sigma),\ \texttt{dedup}_{\texttt{net}}(\Sigma, P),\ T)$$

The proof is by induction on the step relation. For DUPLICATE steps, $\rightsquigarrow_{\mathrm{r}}^{\star}$ holds reflexively, since $\texttt{dedup}_{\texttt{net}}$ returns the same network when a packet is duplicated and the state and trace are unchanged. For DELIVER steps, the proof shows that either the delivered packet is ignored by the destination node, in which case $\rightsquigarrow_{\mathrm{r}}^{\star}$ holds reflexively, or that the underlying handler is run normally, in which case the underlying system can take the analogous DELIVER step. For both the DELIVER and INPUT steps, the proof shows that wrapping the sent packets results in a deduplicated network that is reachable in the underlying system. These proofs require several facts about the internal state of the sequence numbering transformer, such as the fact that all nodes correctly maintain their `next_seqnum` field. These internal state properties are proved by induction on the execution.

2.4.3 Ghost Variables and System Transformers

Many program verification frameworks support *ghost variables*: state which is never read during program execution, but which is necessary for verification (e.g., to provide sufficiently strong induction hypotheses). In Verdi, ghost variables are implemented via a system transformer. Like the sequence numbering transformer, the ghost variable transformer adds information to the system's state while ensuring that the wrapped state is preserved. The system's original handlers are called in order to update the wrapped state and send messages; the new handlers only update the ghost state. The indistinguishability result shows that the ghost transformer does not affect the externally-visible trace or the wrapped state. In this way, ghost state can be added to Verdi systems for free, without requiring any additional proof effort to show that properties verified in the ghost system hold for the underlying system as well.

2.5 Case Study: Key-Value Store

As a case study, we implemented a simple key-value store as a single-node system in Verdi. The key-value store accepts `get`, `put`, and `delete` operations as input. When the system receives input $\texttt{get}(k)$, it outputs the value associated with key k; when the system receives

input $\texttt{put}(k,\ v)$, it updates its state to associate key k with value v; and when the system receives input $\texttt{delete}(k)$, it removes any associations for the key k from its state. Internally, the mapping from keys to values is represented using an association list.

The key-value store's correctness is specified in terms of traces. First, operations on a single key are specified using an interpreter over trace input/output events, which runs each operation and returns the final result. For instance,

$$\texttt{interpret}\ [\texttt{put}\ \text{“foo”},\ \texttt{put}\ \text{“bar”},\ \texttt{get}] = \text{“bar”}$$

Trace correctness is then defined using the interpreter: for every $\langle \texttt{input},\ \texttt{output} \rangle$ pair in the trace, `output` is equal to the value returned by running the interpreter on all operations on that key up to that point. This trace-based specification allows the programmer to change the backing data structure and implementation of each operation without changing the system's specification. Moreover, additional operations can be added to the specification via small modifications to the interpretation function.

We prove the key-value store's correctness by relating its trace to its current state: for all keys, the value in the association list for that key is equal to interpreting all the operations on that key in the trace. The proof is by induction on the execution, and is approximately 280 lines long.

In the next section, we will see how a state-machine replication system can be implemented and verified using Verdi. Combining the key-value store with the replication transformer provides a combined guarantee for a replicated key-value store without requiring the programmer to simultaneously reason about both application correctness and fault tolerance.

2.6 Case Study: Primary-Backup Transformer

In this section, we introduce the primary-backup replication transformer, which takes a single-node system and returns a replicated version of the system in the reordering semantics. A primary node synchronously replicates requests to a backup node: when a request arrives,

```
PB (S) :=
  Name := Primary | Backup

  Msg := Replicate S.Inp | Ack
  Inp := S.Inp
  Out := { request: S.Inp; response: S.Out }
  State (n: Name) = { queue: list S.Inp;
                      underlying_state: S.State }

  InitState (n: Name) = { queue := [];
                          underlying_state := S.InitState n }

  HandleInp (n: Name) (s: State n) (inp: Inp) :=
    if n == Primary then
      append_to_queue inp;;
      if length s.queue == 1 then
        (* if not already replicating a request *)
        send (Backup, Replicate (head s.queue))

  HandleMsg (n: Name) (s: State n) (src: Name) (msg: Msg) :=
    match n, msg with
      | Primary, Ack =>
        out := apply_entry (head s.queue);;
        output { request := head s.queue; response := out };;
        pop s.queue;;
        if s.queue != [] then
          send (Backup, Replicate (head s.queue))
      | Backup, Replicate i =>
        apply_entry i;;
        send (Primary, Ack)
```

Figure 2.12: Pseudocode for the primary-backup transformer. The primary node accepts commands from external input and replicates them to the backup node. During execution, the primary node keeps a queue of operations it has received but not yet replicated to the backup node. The backup node applies operations to its local state and notifies the primary node. Once the primary node receives a notification, it responds to the client.

the primary ensures that the backup has processed it before applying it locally and replying to the client. Whenever a client gets a response, the corresponding request has been processed by both the primary and the backup. Pseudocode for the primary-backup transformer is shown in Figure 2.12.

The primary-backup transformer's correctness is partially specified in terms of traces the primary may produce: any sequence of inputs and corresponding outputs produced by the primary node is a sequence that could have occurred in the original single-node system, and thus any property Φ of traces of the underlying single-node system also holds on all traces

at the primary node in the transformed system. This result guarantees indistinguishability for the primary-backup transformer.

The primary-backup transformer specification also relates the backup node's state to the primary node's state. Because the primary replicates entries synchronously, and one at a time, the backup can fall arbitrarily behind the input stream at the primary. However, the primary does not send a response to the client until the backup has replicated the corresponding request. Thus, the state at the backup is closely tied to that at the primary. In particular, we were able to show that either the primary and the backup have the same state or the backup's state is one step ahead of the primary. This property provides some intuitive guarantees about potential failure of the primary: namely, that manual intervention could restore service with the guarantee that any lost request must not have been acknowledged. It makes sense that manual intervention is necessary in the case of failure: the composed system is verified against the reordering semantics, where the developer assumes that machine crashes require manual intervention.

Once implemented and verified, the primary-backup transformer can be used to construct replicated applications. Applying it to the case study from Section 2.5 results in a replicated key-value store. The resulting system is easy to reason about because of the transformer's indistinguishability result. For example, we were able to show (in about 10 lines) that submitting a `put` request results in a response that correctly reflects the `put`.

2.7 Case Study: Raft Replication Transformer

Fault-tolerant, consistent state machine replication is a classic problem in distributed systems. This problem has been solved with *distributed consensus* algorithms, which guarantee that all nodes in a system will agree on which commands the replicated state machine has executed and in what order, and that each node has a consistent copy of the state machine.

In Verdi, we can implement consistent state machine replication as a system transformer. The consistent replication transformer lifts a system designed for the state machine semantics into a system that tolerates machine crashes in the failure semantics. We implemented the

replication transformer using the Raft consensus algorithm [86]. Our implementation of Raft in Verdi is described in Section 2.7.2.

A Verdi system transformer lifts a safety property of an input system into a new semantics. The consensus transformer provides an indistinguishability result for *linearizability*, which states that any possible trace of the replicated system is equivalent to some valid trace of the underlying system under particular constraints about when operations can be re-ordered. We have proved linearizability in two steps. First, we show that Raft's *state machine safety* property implies linearizability. Second, we verify the (much more difficult) state machine safety property. We discuss these results further in Section 2.7.3.

2.7.1 Raft Background

Raft is a *state machine replication protocol*. The state machine is a deterministic program that specifies the desired behavior of the cluster as a whole. The state machine processes a sequence of *commands*, which are given by the clients of the cluster. External clients interact with the system as if it were a single node running a single copy of the state machine.

Each node in a Raft cluster simulates a copy of the state machine, and the goal of the protocol is to maintain consistency across the copies. Replication allows the system to continue serving clients whenever a majority of machines are available. However, maintaining consistency among replicas is difficult in the presence of asynchrony, network failures (packet drops, duplications, and reordering) and node failures (crashes and reboots). In particular, the combination of asynchrony and failure means that the nodes in the system are never guaranteed to be in global agreement [28].

Since Raft requires that the state machine it replicates is deterministic, the replicas will be consistent as long as the same client commands are executed on each replica's copy in the same order. Raft's main internal correctness invariant, called state machine safety, captures this property.

Proposition 2.7.1 (State Machine Safety)**.** Each replicated copy of the state machine exe-

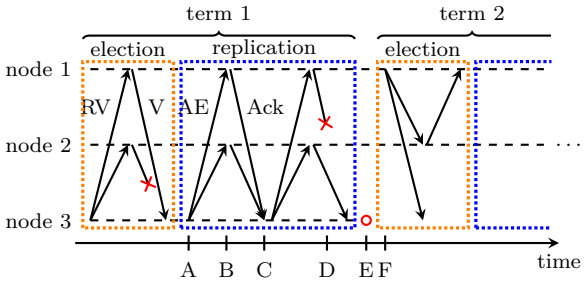


Figure 2.13: Two terms of the Raft protocol, each consisting of a leader election phase (orange) followed by a log replication phase (blue). Node 3 is the leader of the first term, and node 1 is the leader of the second term. Messages are RequestVote (RV), Vote (V), AppendEntries (AE), or Acknowledgment (Ack). × represents a dropped message and ○ represents a crashed node.

cutes the same commands in the same order.

The list of commands to execute on the state machine is kept in the *log*, and the position of a command in the log is called its *index*. Each node has its own copy of the log, and state machine safety reduces to maintaining agreement between all copies of the log.

Figure 2.13 shows an example execution of the Raft protocol.[1] Time is logically divided into *terms*, and each term consists of a leader election phase followed by a log replication phase. During leader election, the cluster chooses a *leader*, who coordinates the cluster and handles all communication with clients during the following log replication phase. Nodes are either leaders, *candidates*, or *followers*. Candidates are in the process of trying to become leader. Followers passively obey the leader of the current term and respond to `RequestVote` messages from candidates.

Leader Election If the leader node crashes or is isolated by a network partition (e.g., node 3 at event E in Figure 2.13), the Raft system elects a new leader. When a node times out waiting to hear from a leader (as node 1 does at event F in Figure 2.13), it becomes

[1] `https://raft.github.io/` has a visualization of Raft in operation.

a candidate.[2] A candidate tries to get itself elected as the new leader by sending messages requesting votes from all other nodes. Once a candidate receives votes from a majority of nodes in the system, it becomes the leader. If no candidate successfully wins the election, a new election will take place following a timeout. Requiring a majority ensures that there is only one leader elected per term.

Proposition 2.7.2 (Election Safety)**.** There is at most one leader per term.

Log Replication During normal operation, the cluster is in the log replication phase. In log replication, when a client sends an input to the leader[3] (e.g., at event A in Figure 2.13), the leader first appends a new *log entry* containing that command to its local log. Then the leader sends an *AppendEntries* message containing the entry to the other nodes in the Raft system. Each follower appends the entries to its log (e.g., at event B in Figure 2.13), and responds to the leader with an acknowledgment. To ensure that follower logs stay consistent with the log at the leader, AppendEntries messages include the index and term of the *previous* entry in the leader's log; the follower checks that it too has an entry at that index and term before appending the new entries to its log. This consistency check guarantees the following property:

Proposition 2.7.3 (Log Matching)**.** If two logs contain entries at a particular index and term, then the logs are identical up to and including that index.

Once the leader learns that a majority of nodes (including itself) have received the new entry (e.g., at event C in Figure 2.13), the leader marks the entry as *committed*.[4] Note that the leader need not receive acknowledgments from all nodes before proceeding (e.g.,

[2]Timeouts are randomized and configured so that candidates rarely compete for leadership. See Ongaro's thesis for more detail on the leader election process [85].

[3]Raft implementations have various mechanisms for clients to locate the leader. In our implementation, clients can send their operations to every node in the cluster until the leader is found.

[4]Committing old entries (those from leaders who failed before completely replicating them) is more complex; see the Raft paper [86] for details.

an acknowledgment is dropped at event D in Figure 2.13, but nodes 2 and 3 constitute a majority). The leader then executes the command contained in the committed entry on the state machine and responds to the client with the output of the command. The followers are also informed that they can safely execute the command on their state machines.

Once an entry is committed, it becomes *durable*, in the sense that its effect will never be forgotten by the cluster. To ensure that leader elections do not violate this property, a new leader must have heard of all committed entries created by the previous leader. Therefore, Raft specifies that a node only votes for candidates whose log is at least as advanced as the voter's. Because a newly elected leader was voted in by a majority, it has a log that is at least as advanced as a majority of the cluster. Since any committed entry is present on a majority, every committed entry is present on at least one node that voted for the candidate. The successful candidate's log thus contains every committed entry.

Proposition 2.7.4 (Leader Completeness)**.** A successfully elected candidate's log contains every committed entry.

Client-facing correctness Clients expect to interact with Raft nodes as if the nodes were collectively a single state machine. More formally, clients see a *linearizable* view of the replicated state [40], i.e., if any node responds to a client command c, all subsequently requested commands will execute on a state machine that reflects the execution of c. Section 2.7.3 gives a precise definition of linearizability.

Proposition 2.7.5. Raft implements a linearizable state machine.

Raft also provides a liveness guarantee: if there are sufficiently few failures, then the system will eventually process and respond to all client commands. To date, we have only verified Raft's safety properties, leaving liveness for future work.

2.7.2 Raft Implementation

```
input :=
  ClientRequest
    (c : cmd) (uid : nat) ...

output :=
  ClientResponse
    (uid : nat) (r : result) ...
| NotLeader

handleInput (i : input) :=
  match i with
  | ClientRequest ...
  end;
  leaderHeartbeat();
  executeEntries()

(* internal Raft messages *)
msg := RequestVote ...
     | Vote ...
     | AppendEntries ...
     | Acknowledgment ...

handleMessage (m : msg) :=
  match m with
  | AppendEntries ...
  | Acknowledgment ...
  | RequestVote ...
  | Vote ...
  end;
  leaderHeartbeat();
  executeEntries()
```

```
handleTimeout :=
   ...; leaderHeartbeat(); executeEntries()

leaderHeartbeat :=
  (* send AppendEntries to followers *)
  (* mark entries as committed *)
  ...

executeEntries :=
  (* execute entries on the local state machine *)
  (* respond to clients if necessary *)

logEntry := { c     : cmd;
              index : nat;
              term  : nat; ... }

nodeType := Leader | Candidate | Follower

data := { log : list logEntry;
          commitIndex : nat;
          term : nat;
          type : nodeType;
          sm : stateMachine; ... }

init : data := { log := [];
                 commitIndex := 0;
                 term := 0;
                 type := Follower;
                 sm := initialStateMachine; ... }
```

Figure 2.14: Signatures of key parts of our Raft implementation.

We implemented Raft as a verified system transformer from a single node semantics with no faults to a multi-node semantics with network and machine faults. To use the transformer, a programmer first implements an algorithm as a (non-distributed) state machine in which a single process responds to input from the outside world. Then, Raft transforms this into a system where the original state machine is consistently replicated across a number of nodes. As a result, the programmer can prove properties about the replicated system by reasoning only about the underlying state machine. The Raft transformer produces a system that is proven correct in an environment in which all messages can be arbitrarily reordered, duplicated, delayed, or dropped, and in which nodes can crash and reboot. We chose this set of faults to be as small as possible, while covering all real world failure scenarios that Raft is designed to tolerate. Other scenarios that are not explicitly included in our model can

be simulated by existing scenarios. For example, a network partition, where a set of nodes cannot send or receive messages from the remaining nodes can be simulated by dropping all such messages. While there is nothing wrong with adding explicit support for network partitions, keeping the set of explicitly modeled failures minimal reduces proof burden.

Figure 2.14 shows signatures for key parts of Raft as implemented in Verdi.[5] There are two classes of messages: *external* messages (inputs from clients) and *internal* messages exchanged between nodes in the system.

Raft has three kinds of external messages. Nodes running Raft receive `ClientRequest` messages from external clients; each such message contains a command of type `cmd`, which is a parameter of the system. These are delivered by the INPUT rule in Figure 2.6. Nodes respond with `NotLeader` to indicate that the client should find the current leader or `ClientResponse`, containing the result of the command, once the system has successfully processed a client command. To ensure that network failures do not cause a single client command to be executed multiple times, each `ClientRequest` includes a unique identifier, shown as `uid` in Figure 2.14. Raft guarantees that a request with a given identifier will only be executed once. Clients can thus repeatedly retry a request; when a client receives a `ClientResponse` with the same `uid`, it knows the command has executed exactly once on the state machine.

Raft has four kinds of internal messages: `AppendEntries` and `Acknowledgment`, used in log replication, and `RequestVote` and `Vote`, used in leader election. These messages correspond directly to the behavior described in Section 2.7.1.

Our Raft implementation consists of event handlers for external messages, internal messages, and timeouts. Each of these handlers begins with some event-specific code and then calls two bookkeeping functions, `leaderHeartbeat` and `executeEntries`. `leaderHeartbeat` performs leader-specific tasks, such as sending `AppendEntries` messages to followers and marking entries as committed. `executeEntries` performs tasks that should be done by every server, such as executing committed entries on the state machine.

---

[5]For more detail, see `raft/Raft.v` at `https://github.com/uwplse/verdi/tree/cpp2015`.

The local state of each Raft node is given in Figure 2.14 by the type `data` and includes the log, the index of the most recently committed entry, the node's current term, the node's type (`Leader`, `Candidate`, or `Follower`), and its copy of the state machine. The log is a list of entries, each of which contains a command to be executed on the state machine, its index (position in the log), and the term in which the entry was initially received by the cluster.

The initial state of each node is given by the value `init`. The log is initially empty, no entries are committed, the current term is 0, every node is a follower (nodes will time out and start an election in term 1 to determine the first leader), and the state machine is in its initial state, having not yet processed any commands.

Our verified implementation of Raft in Coq consists 530 lines of code and 50,000 lines of proof, excluding code from the core Verdi framework. It does not support extensions to Raft which are useful in practice, such as dynamic reconfiguration and log compaction. It also includes more data on `Acknowledgment` messages than is necessary. These limitations are not fundamental, but addressing them would increase the proof burden.

2.7.3 Raft Proof

The behavior of a Verdi system is described by *traces*, which record the interaction between the system and its clients. Internal messages sent between nodes of the system are *not* included in the trace, as they are not observable by clients of the cluster. For example, if Raft is used to replicate a simple key-value store, a valid trace of the resulting system might be:

```
[ClientRequest (Put "x" "hello") 1;
 ClientResponse 1 "";
 ClientRequest (Get "x") 2;
 ClientResponse 2 "hello"].
```

In this execution, a client first sends a `ClientRequest` containing a command to set the key `"x"` to the value `"hello"`; this request is assigned the unique identifier 1. The system then

sends a response containing the empty string as its result, which serves as an acknowledgment that the `Put` has taken place. The client then sends a request to read the value of the key `"x"`; the request is assigned the unique identifier 2. Finally, the system responds with the value `"hello"`.

The correctness of a system *transformer* such as Raft is a *relation* that must hold between the traces generated by the transformed system and those generated by the original system. In Figure 2.10, this relation is called *lift*.

The relational specification of the Raft transformer is that the traces it generates *linearize* (see below) to traces generated by the single-node state machine. Intuitively, linearizability means that once the Raft cluster sends a `ClientResponse` for a command c, the execution of all subsequently issued commands will reflect the execution of c. More precisely, a trace of a replicated system linearizes to a trace of the underlying system if its operations can be reordered to match the underlying trace without moving an incoming command before a previously acknowledged command. For example, in Raft, the system can reorder concurrently issued client requests, but if a request is received after a previous request is acknowledged, then the system must respect that ordering.

We formalize the linearizes-to relation as follows.[6]

Definition 2.7.6 (Linearizes-to)**.** Let τ be a trace of inputs and outputs, where each input-output pair is given a unique key. Then τ *linearizes to* a sequence of state machine commands σ if the events of τ can be reordered into a trace τ' such that

1. τ' is sequential, i.e., it consists of alternating inputs and outputs with matching keys;

2. τ' agrees with σ, i.e., they consist of the same sequence of commands, and each output in τ' equals the result given by the corresponding command in σ; and

3. if an output o appears in τ before an input i, then o also appears before i in τ'.

[6]The relevant Coq development is `raft/Linearizability.v` at `https://github.com/uwplse/verdi/tree/cpp2015`.

Note that this definition requires τ and σ to contain the same set of commands. Thus, we can define linearizability:

Definition 2.7.7 (Linearizability)**.** A trace τ is linearizable if there exists a sequence σ of state machine commands such that τ linearizes to σ.

This definition captures the notion of linearizability, but establishing it directly for Raft would be difficult because it would require strengthening it to be an inductive invariant of the system. Instead, we proved Raft linearizable by relating the system's trace to the local state of each node and the set of packets in the network.

First, we related the trace of the system to each node's local copy of the state machine via state machine safety (Proposition 2.7.1 from Section 2.7.1). Proving linearizability from state machine safety required proving each of the conditions in Definition 2.7.6 by reducing each to an internal property of Raft.

Theorem 2.7.8. State machine safety implies linearizability.[7]

Proof. Given an execution trace τ of Raft, we must find σ such that τ linearizes to σ. There is an obvious choice for σ: it is just the sequence of commands executed by the nodes on their local state machines. State machine safety guarantees that the nodes agree on this sequence, so our choice is well defined.

It remains to show that τ linearizes to σ. In other words, we must find τ' such that the conditions of Definition 2.7.6 are satisfied. Let τ' be the sequential input–output trace corresponding to σ, i.e., for each command of σ, τ' contains an input immediately followed by the corresponding output for that command. Then τ' is sequential and agrees with σ by construction, and it remains to show that τ' is a permutation of τ that respects the ordering condition (item 3) of Definition 2.7.6. Each of these is established as a separate invariant by induction on the execution. □

[7] This argument is formalized in `raft/RaftLinearizableProofs.v`, along with the lemmas imported by that file.

This result fits into the framework of verified system transformers, as described in Section 2.4. The remainder (and vast majority) of our Raft verification effort establishes state machine safety. Since each node executes commands on its state machine as entries become committed in the node's log, state machine safety requires that nodes never disagree about committed entries. The proof of State Machine Safety requires the use of *ghost variables*. Ghost variables are components of system state that are tracked for the purposes of verification but not needed at run time. This state is therefore not tracked in the extracted implementation.

Theorem 2.7.9 (State Machine Safety)**.** State machine safety holds for every reachable state of the system.[8]

Proof Sketch. First strengthen the induction hypothesis to quantify over ghost state and appropriately constrain each node's history. Next proceed by induction on the step relation, and in each case show that the strengthened hypothesis is preserved. □

The proof of State Machine Safety requires several ghost variables on local data, as well as one on messages. Figure 2.15 shows pseudocode for the local data ghost state, including the ways in which it is updated in response to incoming messages. Intuitively, each ghost variable stores part of the system's *history*, which is not tracked in the actual implementation but which is necessary for proofs. For example, a node in the system does not actually need to keep a record of every vote that it has every cast; it is sufficient to track only the vote for its current term. However, in order to prove that only one leader is elected per term, the proof uses the `votes` ghost variable. We use the ghost state to establish the Election Safety and Leader Completeness properties, from which we then prove State Machine Safety. As an example, we show how Election Safety follows using these ghost variables.[9]

---

[8]The top-level proof is in `raft-proofs/StateMachineSafetyProof.v`. The ghost variables required are specified in `raft/RaftRefinementInterface.v` and `raft/RaftMsgRefinementInterface.v`.

[9]The proof of Leader Completeness is available in `raft-proofs/LeaderCompletenessProof.v`.

```
ghostData := {
  (* list of term, candidate this node voted for,
     log at time of vote *)
  votes : list (nat * name * list logEntry);

  (* term -> list of nodes who voted for
     this node in that term *)
  cronies : nat -> list name;

  (* term, log when this node became leader *)
  leaderLogs : list (nat * list logEntry);

  (* list of term, entry:
     all entries ever present in log at this node*)
  allEntries : list (nat * logEntry)
}

ghostHandleMessage (m : msg) :=
  match m with
  | AppendEntries ... =>
    (* If entries added to log, add to allEntries
       and tag with current term *)
  | RequestVote ...
    (* If voting, add the current term,
       the candidate's name,
       and the current log to votes *)
  | Vote ...
    (* Add sender to cronies at current term *)
    (* If node becomes leader, add current term
       and log to leaderLogs *)
  end
```

Figure 2.15: Ghost variables used in the verification of Raft

Theorem 2.7.10 (Election Safety)**.** Election safety is true in every reachable state of the system.[10]

Proof Sketch. If a node is a leader, then it has a majority of nodes in its `cronies` for that term. A node h does not appear in `cronies` at a node h' unless h' is in `votes` at h for the same term. A node only votes for one leader for each term. If there are two leaders for one term, at least one node h must be in `cronies` at both leaders since they each have a majority. That node must have voted for both of them at that term, so they must be the same node. Therefore, Election Safety holds. □

2.8 Evaluation

This section aims to answer the following questions:

- How much effort was involved in building the case studies discussed above?
- To what extent do system transformers mitigate proof burden when building modular verified distributed applications?
- Do Verdi applications correctly handle the faults they are designed to tolerate?
- Can a verified Verdi application achieve reasonable performance relative to analogous unverified applications?

2.8.1 Verification Effort

Table 2.1 shows the size of the specification, implementation, and proof of each case study. The Verdi row shows the number of lines in the shim, the network semantics from Section 2.3, and proofs of reusable, common lemmas in Verdi. The KV+PB row shows the *additional* lines of code required to state and prove a simple property of the key-value store with

[10] See `raft-proofs/OneLeaderPerTermProof.v`.

Table 2.1: Verification effort: size of the specification, implementation, and proof, in lines of code (including blank lines and comments).

System	Spec.	Impl.	Proof
Sequence numbering	20	89	576
Key-value store	41	138	337
Primary-backup	20	134	1155
KV+PB	5	N/A	19
Raft (Linearizability)	170	520	4144
Raft (SMS)	47	N/A	50719
Verdi	148	220	2364

the primary-backup transformer applied. This line shows that verified system transformers mitigate proof burden by preserving properties of their input systems.

2.8.2 Verification Experience

While verifying the case studies, we discovered several serious errors in our system implementations. The most subtle of these errors came from our implementation of Raft: servers could delete committed entries when a complex sequence of failures occurred. Such a sequence is unlikely to arise in regular testing, but proving Raft in Verdi forced us to reason about all possible executions. The Raft linearizability property we proved prevents such subtle errors from going unnoticed.

2.8.3 Verification and Performance

We applied the consensus transformer described in Section 2.7 to the key-value store described in Section 2.5; we call the composed system `vard`.[11] We performed a simple evaluation of its performance. We ran our benchmarks on a three-node cluster, where each node had eight 2.0 GHz Xeon cores, 8 GB main memory, and 7200 RPM, 500 GB hard drives. All the nodes were connected to a gigabit switch and had ping times of approximately 0.1 ms. First,

[11] Pronounced *var-DEE*.

Table 2.2: A performance comparison of etcd and our `vard`.

	Throughput (req./s)	**Latency** `get` (ms)	`put` (ms)
`etcd`	38.9	205	198
`vard`	34.3	232	232

we ran the composed system and killed the leader node; the system came back as expected. Next, we measured the throughput and latency of the composed system and compared it to etcd [18], a production fault-tolerant key-value store written in the Go language which also uses Raft internally. We used a separate node to send 100 random requests using 8 threads; each request was either a put or a get on a key uniformly selected from a set of 50 keys.

As shown in Table 2.2, `vard` achieves comparable performance to etcd. We believe that etcd has slightly better throughput and latency because of better data structures and because requests are batched. `vard` is not feature complete with respect to etcd, which uses different internal data structures and a more complex network protocol. Nonetheless, we believe this benchmark shows that a verified Verdi application can achieve roughly equivalent performance compared to existing, unverified alternatives.

2.9 Related Work

This section relates Verdi to previous approaches for building reliable distributed systems.

Proof assistants and distributed systems. EventML [92] provides expressive primitives and combinators for implementing distributed systems. EventML programs can be automatically abstracted into formulae in the Logic of Events, which can then be used to verify the system in NuPRL [16]. The ShadowDB project implements a total-order broadcast service using EventML [99]. The implementation is then translated into NuPRL and verified to correctly broadcast messages while preserving causality. A replicated database is implemented on top of this verified broadcast service. Unlike `vard` (described in Section 2.8),

the database itself is unverified.

Bishop et al. [6] used HOL4 to develop a detailed model and specification for TCP and the POSIX sockets API, show that their model implements their specification, and validate their model against existing TCP implementations. Rather than verifying the network stack itself, in Verdi we chose to focus on verifying high-level application correctness properties against network semantics that are assumed to correctly represent the behavior of the network stack. These two lines of work are therefore complementary.

Ridge [97] verified a significant component of a distributed message queue, written in OCaml. His technique was to develop an operational semantics for OCaml which included some basic networking primitives, encode those semantics in the HOL4 theorem prover, and prove that the message queue works correctly under those semantics. Unlike in Verdi, the proofs for the system under failure conditions were done only on paper.

Verdi's system transformers enable decomposing both systems and proofs. This allows developers to establish correctness guarantees of the implementation of their distributed systems, from the low-level network semantics to a high-level replicated key-value store, while retaining flexibility and modularity. The system transformer abstraction could be integrated into these other approaches; for example, ShadowDB's consensus layer could be implemented as a system transformer along the lines of Verdi's Raft implementation.

Ensemble. Ensemble [39] layers simple *micro protocols* to produce sophisticated distributed systems. Like Ensemble micro protocols, Verdi's system transformers implement common patterns in distributed systems as modular, reusable components. Unlike Ensemble, Verdi's systems transformers come with correctness theorems that translate guarantees made against one network semantics to analogous guarantees against another semantics. Unlike Verdi, Ensemble enables systems built by stacking many layers of abstraction to achieve efficiency equivalent to hand-written implementations via cross-protocol optimizations. These micro protocols are manually translated to IO automata and verified in NuPRL [41, 69]. In contrast, Verdi provides a unified framework that connects the implementation and the formalization, eliminating the formality gap.

Verified SDN. Formal verification has previously been applied to software-defined networking, which allows routing configurations to be flexibly specified using a simple domain specific language (see, e.g., Verified NetCore [36]). As in Verdi, verifying SDN controllers involves giving a semantics for OpenFlow, switch hardware, and network communication. The style of formalization and proof in Verified NetCore and Verdi are quite similar and address orthogonal problems. Verified NetCore is concerned with correct routing protocol configuration, while Verdi is concerned with the correctness of distributed algorithms that run on top of the network.

Specification Reasoning. There are many models for formalizing and specifying the correctness of distributed systems [29, 90, 98]. One of the most widely used models is TLA, which enables catching protocol bugs during the design phase [60]. For example, Amazon developers reported their experience of using TLA to catch specification bugs [81]. Another approach of finding specification bugs is to use a model checker. For example, Zave applied Alloy [44] to analyzing the protocol of the Chord distributed hash table [123]. Lynch [72] describes algorithm transformations which are similar to Verdi's verified system transformers.

On the other hand, Verdi focuses on ensuring that implementations are correct. While this includes the correctness of the underlying algorithm, it goes further by also showing that the actual running system satisfies the intended properties.

Model checking and testing. There is a rich literature in debugging distributed systems. Run-time checkers such as Friday [30] and D^3S [70] allow developers to specify invariants of a running system and detect possible violations on the fly or offline. Model checkers such as Mace [49, 50], MoDist [120], and CrystalBall [119] explore the space of executions to detect bugs in distributed systems. These tools are useful for catching bugs and easy to use for developers, as they only need to write invariants. On the other hand, Verdi's proofs provide correctness guarantees.

For example, Mace provides a full suite of tools for building and model checking distributed systems. Mace's checker has been applied to discover several bugs, including *liveness* violations, in previously deployed systems. Mace provides mechanisms to explicitly break

abstraction boundaries so that lower layers in a system can notify higher layers of failures. Verdi does not provide liveness guarantees nor mechanisms to break abstraction boundaries, but enables stronger guarantees via full formal verification.

Verification. Several major systems implementations have been verified fully formally in proof assistants. The CompCert C compiler [64] was verified in Coq and repeatedly shown to be more reliable than traditionally developed compilers [61, 121]. Our system transformers are directly inspired by the translation proofs in CompCert, but adapted to handle network semantics where faults may occur.

The Reflex framework [96] provides a domain-specific language for reasoning about the behavior of reactive systems. By carefully restricting the DSL, the authors were able to achieve high levels of proof automation. Bedrock [13] and Ynot [79] are verification frameworks based on separation logic and are useful for verifying imperative programs in Coq, but also consider only the behavior of a single node and do not model faults.

2.10 Conclusion

This chapter presented Verdi, a framework for building formally verified distributed systems. Verdi's key conceptual contribution is the use of verified system transformers to separate concerns of application correctness and fault tolerance, which simplifies the task of implementing and verifying distributed systems. This modularity is enabled by Verdi's encoding of distinct fault models as separate network semantics. We demonstrated how to apply Verdi to writing and verifying several practical applications, including the Raft state replication library and the `vard` fault-tolerant key-value store, with the help of verified system transformers. These applications provide strong correctness guarantees and acceptable performance while imposing reasonable verification burden. From the perspective of this dissertation as a whole, Verdi provides vertical decomposition of application logic, fault models, and fault tolerance mechanisms, reducing overall proof complexity.

Chapter 3

HORIZONTAL COMPOSITION: SYSTEMS BUILT FROM MANY PROTOCOLS

3.1 Introduction

The previous chapter described techniques for building a single verified distributed system by decomposing the application logic from the fault-tolerance mechanisms. Real-world systems, however, rarely consist of just a single monolithic application. Rather, they are composed of multiple independent modules, which are then linked together in some way. This compositional software development approach enables clean separation of concerns and a modular development process: in order to use one component within a larger system, one only needs to know *what* it does without requiring details on *how* it works. This chapter describes techniques to bring this modularity into verified distributed systems using *horizontal composition*.

There are several design constraints on such horizontal reasoning. For example, to compose a linearizable database with a causally consistent cache [2], 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, composable verified systems must disentangle implementation details from abstract protocol definitions, allowing independent evolution of components without extensive refactoring [118]. 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.

3.1.1 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 [54, 104], operating systems [34, 51] and concurrent libraries [35, 101]. 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 implementer 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 chapter presents DISEL, a mechanized framework for verification and implementation of distributed systems that aims to address these challenges.

3.1.2 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 [93, 117], 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 implementer, 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 [17] proof assistant, making use of Coq's dependent types and higher-order programming features. In the tradition of Hoare Type Theory (HTT) by Nanevski, Morrisett, and Birkedal [78], Nanevski et al. [79], and Nanevski, Vafeiadis, and Berdine [80] and its recent versions for concurrency [66, 77], we give the semantics to effectful primitives, such as $\mathsf{send}/\mathsf{receive}$, 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 is different from more traditional techniques for establishing *refinement* [1, 38] between an actual implementation (the code) and a specification (an abstract protocol) via a simulation argument [73]. In comparison with the refinement-based techniques, the type-based verification method makes it easy to account for horizontal composition of protocols (necessary, e.g., for

[1]The framework name stands for DIstributed SEparation Logic.

reasoning about remote procedure calls, as we will show in Section 3.2) 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 [77, 110], allowing one to instrument programs with pre/postconditions and providing a form of the *frame rule* [95] with respect to protocols. Thus, there is an analogy between the heap in shared-memory concurrency and the set of protocols in DISEL. 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 3.3. 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 chapter 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 [3]) 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 [115] and its client application), as well as a report on our experience of using Disel and a discussion on the executable code.

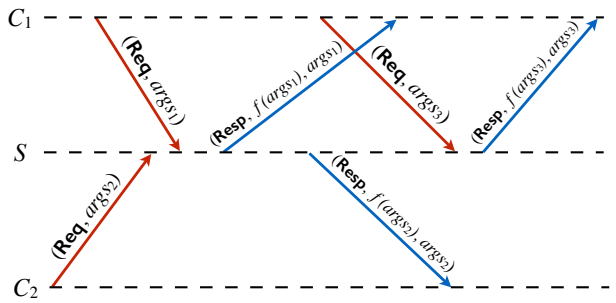


Figure 3.1: A communication scenario between a server and two client nodes in a distributed calculator.

3.2 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 $f(args)$. Figure 3.1 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.

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	$n \in \overline{C}$ && $n \rightarrowtail (from, args) \uplus rs$ && $m = (\mathsf{Resp}, ans, args)$	$n \rightarrowtail rs$

Figure 3.2: Send- and receive-transitions of the distributed calculator protocol with respect to a node n.

3.2.1 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 [57], DISEL protocols are defined as *state-transition systems*.

Figure 3.2 depicts the state-transition system for the calculator example with two send-transitions and two receive-transitions. 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 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 $(\mathsf{Req}, \mathit{args})$ to server *to* where $\mathit{args} \in \mathsf{dom}(f)$ and, after sending, n has added $(\mathit{to}, \mathit{args})$ 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 $(\mathsf{Req}, \mathit{args})$ from node *from* where, after receiving, n has added $(\mathit{from}, \mathit{args})$ 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 $(\mathit{to}, \mathit{args})$ to be in the multiset of outstanding requests at the server, and (c) $(\mathit{to}, \mathit{args})$ 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 $f(\mathit{args})$, $\mathit{args} \in \mathsf{dom}(f)$. 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., $\mathcal{C}_{\ell_1}, \mathcal{C}_{\ell_2}$).

3.2.2 From Protocols to Programs

The transitions in Figure 3.2 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 Figure 3.2 and DISEL's logic (discussed in Section 3.3), 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 3.4.

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.

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

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 $\langle \mathit{from}, n, \bullet, (\mathsf{Req}, \mathit{args}) \rangle$ 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 3.3.1) 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 3.3), assigning this procedure the following specification:

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

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

[3]This design choice with respect to message representation is common in state-of-the-art frameworks for distributed systems verification, e.g., IronFleet [38] and Ivy [89], as it simplifies reasoning about past events.

```
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 ()
```

Figure 3.3: A simple server that responds to one request in each iteration of an infinite loop.

rules of DISEL, described in Section 3.3.2. 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.

3.2.3 Elaborating State-Space Invariants of a Protocol

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 (see Figure 3.3). 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 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 the specification (5) ensures that it faithfully follows the protocol's transitions and does not terminate.

3.2.4 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.

```
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)

Figure 3.4: Batching (a) and memoizing (b) calculator servers defined on top of the protocol $\mathcal{C}'_\ell$.

Alternative servers. Figure 3.4 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 $\mathcal{C}'_\ell$.

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

Figure 3.5: A client in the calculator protocol that asks the server to compute its answer.

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 in Figure 3.5. 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:

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

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

```
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'
```

Figure 3.6: Delegating calculator server that forwards all requests to an existing server.

can prove the following spec for the RPC compute:

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

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. [77], 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' 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 DISEL implementation of such a server is shown in Figure 3.6. 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 $\mathcal{C}''_{\ell_2}$:

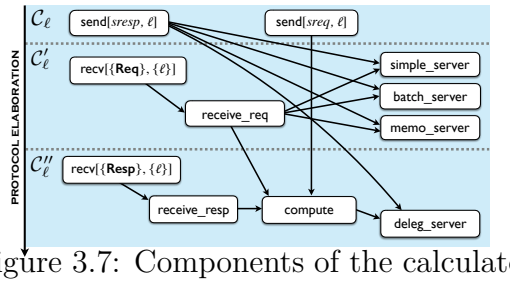


Figure 3.7: Components of the calculator system.

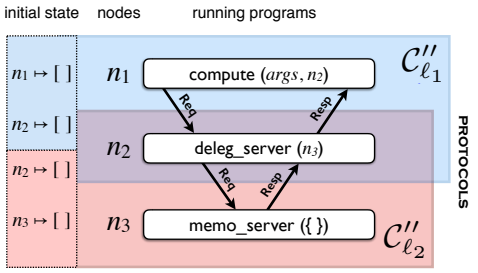


Figure 3.8: Initial state and execution with three nodes.

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

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 $n \overset{\ell_1}{\rightarrowtail} rs$), but should start with an empty set of its own outstanding requests, thus $n \overset{\ell_2}{\rightarrowtail} \emptyset$.

Summary of the DISEL methodology. Our entire development of the calculator-aware applications (e.g., servers and clients) is outlined in Figure 3.7. 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.

3.2.5 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 3.3.3, specifies what it means 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, Figure 3.8 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 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. [4]

3.3 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.

---

[4]Our GitHub repository `https://github.com/DistributedComponents/disel` contains a `README.md` file that describes how to check the proofs and run the code.

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

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

Figure 3.9: DISEL's distributed state and world components.

3.3.1 State and Worlds

Distributed state and its components. The left part of Figure 3.9 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 (DistLocState). The former represents a finite partial map from unique message identifiers to messages,[5] 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 3.2, 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.

[5]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.

Protocols, hooks and worlds. The right part of Figure 3.9 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 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 3.2 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 3.4. 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 $z \in$ 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 3.2 were defined with $H = \emptyset$ (i.e., without any imposed inter-protocol restrictions), but in Section 3.4 we will show how the mechanism of

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

Figure 3.10: Semantics of DISEL state assertions.

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 then be used freely in the proofs about programs.

3.3.2 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:

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

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. Figure 3.10 provides the semantics of the assertions with respect to a distributed system state that we have used in the examples in Section 3.2, 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 [95], 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* [45, 112], 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 Figure 3.11 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 Figure 3.11, 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 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

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

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

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

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

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

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

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

Auxiliary definitions

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

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

$$\mathsf{HooksOk}(H, \tau_s, \ell_c, s, n, m, to) \triangleq \forall \ell_s\ h\ z, H(z, \ell_s, \ell_c, \tau_s.tag) = h \implies h(s(\ell_s)(n), s(\ell_c)(n), m, to)$$

$$\mathsf{NotHooked}(W, H) \triangleq \exists C,\ W = \langle C, -\rangle \wedge \forall (z, \ell_s, \ell_c, t) \in \mathsf{dom}(H), \ell_c \notin \mathsf{dom}(C).$$

Figure 3.11: Selected logic inference rules of Disel and auxiliary predicates.

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 3.2.3.

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 [52].

Definition 3.3.1 (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 Figure 3.11, 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.[6] 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 3.2.4 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 3.4, 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 $\mathcal{P}_\ell$.[7] 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.

3.3.3 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

[6] This definition of $\mathsf{NotHooked}$ is a syntactic approximation of "framing *wrt.* transitions" that suffices for our purposes. More elaborate 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".

[7] The formal definition of inductive invariants is with respect to the protocol-aware network semantics, defined in Section 3.3.3, and is available in the accompanying Coq development.

SendStep

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

ReceiveStep

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

Figure 3.12: Transition rules of the network semantics.

semantics, and its two transition rules are shown in Figure 3.12 (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 *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 Figure 3.11. 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-transitions by a specific node, consistent 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 3.3.4).

Definition 3.3.2 (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 3.3.3 (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 $\mathsf{Some}\ (from, m)$ 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 3.3.4 (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 [8]. We define a *denotational* semantics of a DISEL command c as a (possibly infinite) set of finite *partial execution traces* $[\![c]\!] = \{\mathfrak{t}_\kappa \mid \mathfrak{t} = [w_1, \ldots, w_n]\}$, where each element w_i of a trace $\mathfrak{t}$ 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 $\mathfrak{t}$ 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 3.5.

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 [66, 77], 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 3.3.5. 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 3.3.6. A trace $\mathfrak{t}_\kappa$ is *post-safe* for n, s and Q *iff* either

- $\mathfrak{t} = [\,]$, $\kappa = \mathsf{done}\ v$ and $\forall s', s \overset{\neg n*}{\rightsquigarrow}_W s' \implies s' \vDash [v/\mathsf{res}]Q$, or
- $\mathfrak{t} = w :: \mathfrak{t}'$, 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 $s' \overset{w.n}{\rightsquigarrow}_W s''$, $\mathfrak{t}'_\kappa$ is post-safe for n, s'' and Q.

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

Definition 3.3.7 (Hoare Type Semantics)**.** $W \overset{n}{\vdash} c : \{P\}\{Q\}$ *iff* for *any* s, such that $s \vDash P$, and for *any* trace $\mathfrak{t}_\kappa \in [\![c]\!]$, such that $\kappa = \mathsf{done}\ v$, the trace $\mathfrak{t}_\kappa$ is post-safe for n, s and Q.

Definition 3.3.7 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.* Figure 3.11) 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 3.3.8 (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 3.3.7.

Definition 3.3.7 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 a non-stable precondition P *wrt.* a node n means an inconsistent *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 3.2.4, whose preconditions mention only the node-local state and are, thus, stable.

3.4 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`) [115, Chapter 19]. `TPC` is widely used in

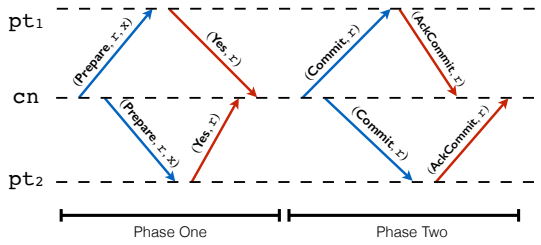


Figure 3.13: One round of the Two-Phase Commit.

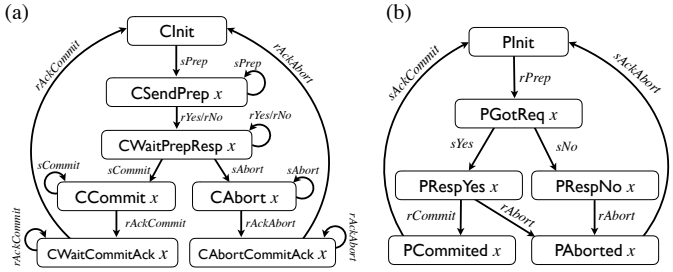


Figure 3.14: States of a coordinator (a) and a participant (b).

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.

3.4.1 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 Figure 3.13. 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.

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 Figure 3.14(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 partici-

```
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.
```

Figure 3.15: Send and receive transitions of a coordinator in a DISEL definition of the `TPC` protocol.

pant sends messages to or receives messages from only the coordinator before changing its state.

Figure 3.15 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.

3.4.2 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 Figure 3.16. 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 parameterized 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 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

```
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).
```

Figure 3.16: 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)).
```

Figure 3.17: Coordinator spec elaborated with `TPCInv`.

the connective $\mathsf{cn} \overset{\mathsf{TPC}}{\mapsto} \ldots$ from Figure 3.10, 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.

3.4.3 Protocol Consistency and Inductive Invariant

The spec given to `coordinator_round` in Figure 3.16 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 [89] and `mypyvy` (Chapter 4) make the process of finding an inductive invariant much more pleasant by providing automatic assistance in debugging and correcting invariants, or even inferring invariants automatically. For now, though, we proceed to manually construct an invariant.

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. Figure 3.17 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 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.

3.4.4 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 [115] and, with a rare exception [65], almost never a focus of major verification efforts [38, 93, 118], which, therefore cannot be reused in any larger verified context.

```
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).
```

Figure 3.18: 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').
```

Figure 3.19: Hook definition and abstract predicates.

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). Figure 3.18 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 3.2: 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 Figure 3.18. 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 Figure 3.19.

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 Figure 3.19 to provide specifications for querying procedures from Figure 3.18, 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 Figure 3.18 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 parameterization of querying programs with abstract predicates and hypotheses from Figure 3.19, we can compose them with any other instance of a consensus protocol, e.g., Paxos [58] or Raft [86], thus, reusing the proofs of their core invariants.

3.5 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 [33, 75, 100] as well as on the implementation of partial finite maps and heap theory by Nanevski, Vafeiadis, and Berdine [80].

Table 3.1 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

Component	Defs/Specs	Impl	Proofs	Build
Calculator (§3.2)				
protocol (§3.2.1) INV$_1$ (§3.2.3) INV$_2$ (§3.2.4)	239	-	243	4.8
simple_server (§3.2.3) batch_server (§3.2.4) memo_server (§3.2.4)	192	43	153	8.6
compute (§3.2.4)	120	24	99	4.8
deleg_server (§3.2.4)	75	7	49	2.4
Two-Phase Commit (§3.4.1–§3.4.3)				
protocol (§3.4.1)	465	-	231	3.9
coordinator (§3.4.2)	236	35	440	18
participant (§3.4.2)	163	24	198	10
TPCInv (§3.4.3)	997	-	2113	25
Query/TPC (§3.4.4)				
protocol	169	-	115	2.1
querying procedures	326	18	707	19
`run_and_query`	76	5	89	2.6

Table 3.1: Statistics for implemented systems: sizes of protocol definitions/specs, programs, proofs of protocol axioms/invariants/specs (LOC), and build times (sec).

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 [13] or inductive invariant proofs [89], we consider these results encouraging for future development.

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 3.3.3 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 [38, 89, 117].

2. The shim implementation follows the operational rules from Figure 3.12 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 3.2.4, 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. [4].

3.6 Related and Future Work

3.6.1 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 [22, 77, 91, 106, 110, 111] provides a "localized" version of more traditional Rely/Guar-

antee obligations [45], which, in their original formulation, are not modular [26, 27, 112]. The two closest to DISEL logics employing protocols to reason about interference are FCSL by Nanevski et al. [77] and GPS by Turon, Vafeiadis, and Dreyer [110]. 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* [84], 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 3.4, "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 [46, 47] 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* [95]. 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.* Figure 3.11), which enabled modular invariant proofs and distributed systems composition.

A related logic by Villard, Lozes, and Calcagno [114] 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 Figure 3.17) 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 [105] 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* a 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.

3.6.2 Types for Distributed Systems

Session Types [42] are traditionally used to ensure that distributed parties follow a predefined communication protocol *wrt.* a specific channel. While the *multiparty* [43] and *multirole* [20] Session Types enable a form of system composition and role-play, and *dependent session types* allow one to quantify over messages [109], 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. [107], although that work did not reason about inductive invariants separately from implementations, neither did it address composition of systems with inter-protocol dependencies.

3.6.3 Verification of Large Systems

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

IronFleet [38] supports proving *liveness* in addition to safety, all embedded in Dafny [62]. 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 [68], 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. [35], 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 [40] 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 parameterized Hoare types and abstract predicates, as shown in Section 3.4.4.

The Chapar framework by Lesani, Bell, and Chlipala [65] 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 [89]. PSync by Drgoi, Henzinger, and Zufferey [23] is a DSL allowing one to prove inductive invariants of consensus algorithms in networks with

potential faults, operating in a synchronous round-based model [24]. This assumption enables efficient proof automation, but prohibits low-level optimizations, such as, e.g., batching. Mace by Killian et al. [49] and DistAlgo by Liu et al. [71] 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 [93]. 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 chapter.

Arguably, our Two-Phase Commit implementation is a relatively small case study when compared to the systems verified in Verdi (Chapter 2) or in other frameworks including IronFleet and EventML. Nevertheless, given enough time and effort, we are confident we could conduct safety proofs of Raft [86] and MultiPaxos [94] 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.

3.7 Conclusion

This chapter presented Disel, a concurrent separation logic for distributed systems. Disel's key contribution to this dissertation is *horizontal protocol composition*, made possible by our frame rule, with hooks to express protocol dependencies. We showed examples of composing protocols, including using two-phase commit mediate access to a distributed resource. More generally, Disel allows us to compose separately designed and verified protocols to build more complex systems from existing pieces. Together with the vertical decomposition provided by Verdi, one can systematically break down a large distributed system into independently verifiable protocols that add up to a verified whole.

Chapter 4

AUTOMATIC VERIFICATION WITH TRANSITION SYSTEMS

4.1 Introduction

Our experience with Verdi and Disel has been that the most challenging aspect of distributed systems verification is discovering inductive invariants. An inductive invariant is a property of system states that is true in all initial states and preserved by all steps the system can take. Such a property is guaranteed to be true in all reachable states of the system by a simple inductive argument on executions. A common verification task is to prove that there is no reachable state that violates some specified *safety property*. The basic approach to this task is to find an inductive invariant that implies the safety property, which is sufficient because any reachable state satisfies the inductive invariant, so it must also satisfy the safety property. The challenge in this approach is finding such an inductive invariant, which has led researchers to investigate techniques for automatically inferring inductive invariants, given a description of a system and its safety property.

This chapter describes `mypyvy` a tool for automated reasoning about symbolic transition systems in first-order logic that supports a variety of automated reasoning techniques to analyze systems. `mypyvy` takes an input file describing a symbolic transition system and performs the analysis requested by the user. Three of the most interesting analyses include inductive invariant checking, inductive invariant inference, and bounded trace reasoning, including bounded model checking. In all cases, `mypyvy` loads the transition system and compiles it together with the user-requested analysis to a (sequence of) SMT queries, which are dispatched by Z3.

`mypyvy` is not just the sum of the analyses currently available; it is a platform for doing

research in automated verification. We have ongoing projects that use the `mypyvy` foundation to build several new invariant inference techniques and user interfaces for verification and exploration. In this context, `mypyvy` can be seen as a sort of intermediate language that captures the right level of abstraction for implementing invariant inference techniques. Higher level languages can access the invariant inference capabilities of `mypyvy` by compiling their problems into `mypyvy`'s query language. For example, IVy [89] is a modular high level imperative-logical language for verifying implementations or models of concurrent and distributed systems. A typical IVy program consists of many verification tasks of the form described above, at least one per module in the program. Each task could be translated to `mypyvy` to find an inductive invariant, and the result could be translated back to IVy. We are working with collaborators to implement such translations for IVy. There is much exciting work to be done here, but this chapter is more modest in scope, describing just the core of `mypyvy` in theory and implementation.

4.2 Background on Transition Systems

4.2.1 The robot example informally and pictorially

Imagine a robot in a 2-dimensional world. The robot starts at position $(0, 5)$, and can take steps to integer grid points either due north or diagonally southeast of its current position. The world has a circular hole of radius 3 centered at the origin. Can the robot ever fall in the hole?[1]

The initial situation is depicted in Figure 4.1. The smaller black circle represents the robot's initial position, and the larger shaded red circle represents the hole at the origin. In its first step, shown in Figure 4.2, the robot could either move north one square, or diagonally southeast one square. These two possibilities are drawn as thick black arrows. So far, the robot manages to avoid falling in.

As the robot progresses, more and more positions are reachable, each with a sequence

[1]Thanks to Jon Howell for this example.

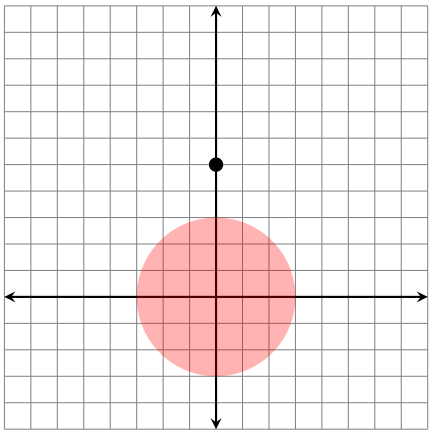


Figure 4.1: Robot initial configuration.

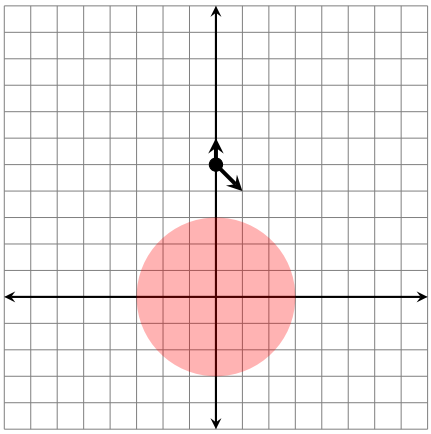
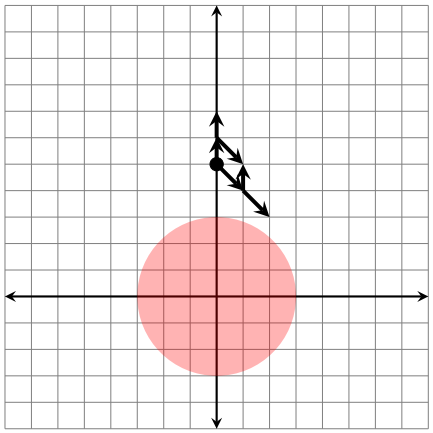


Figure 4.2: Robot's first steps.



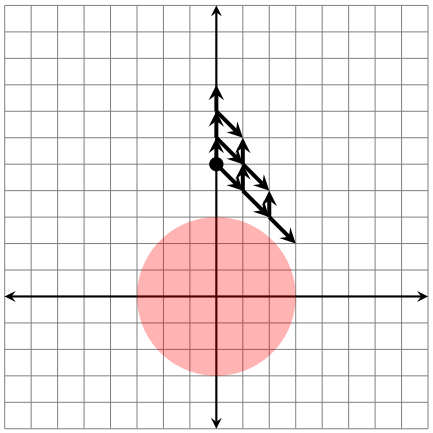


Figure 4.3: Robot's second and third steps.

of moves due north or diagonally southeast. The set of possible positions after two and three steps are shown in Figure 4.3. Despite the growing number of reachable positions, no sequence of steps causes the robot to fall in.

We start to see a pattern emerging. No matter what happens, the robot will never be able to move west of the y axis, nor will it be able to move southwest of the diagonal line $x + y = 5$. This region is shown shaded in green in Figure 4.4. Visually, this green polygonal region does not overlap with the red circle, which "proves" that the robot never falls in.

In fact, this green region exactly characterizes the set of possible positions that the robot

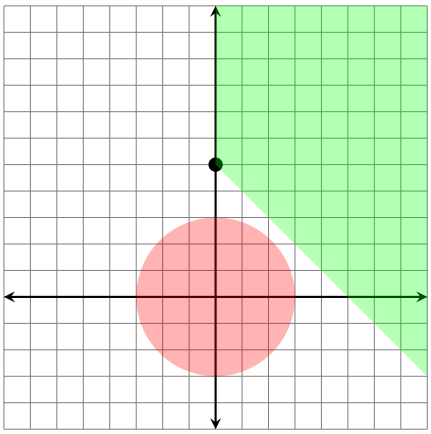


Figure 4.4: Exact characterization of the robot's reachable positions.

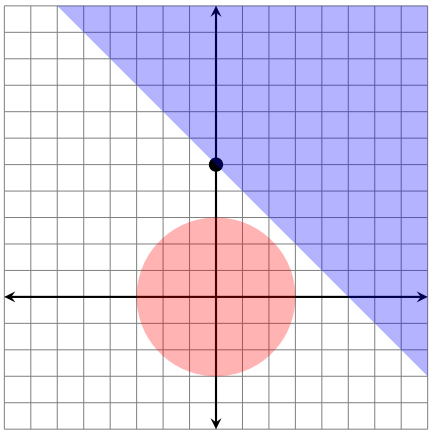


Figure 4.5: A simple overapproximation to the set of reachable positions.

could reach through some sequence of moves. (We call such positions "reachable".) Given any point in the green region, the robot can reach it by first moving southeast over and over until it is vertically below the target point, and then moving due north over and over until the point is reached.

Our proof that the robot never falls in can be simplified slightly be noticing that we never used the fact that the robot always remains east of the y axis. Instead, it is enough to convince ourselves that the robot is always northeast of the line $x + y = 5$. This region is shown in blue in Figure 4.5. Again, it does not visually overlap with the circle, so it constitutes a visual "proof" that the robot never falls in.

This time, though, not every position in the blue polygonal region is reachable, since we have already realized that no position west of the y axis is reachable. The unreachable positions from this region are shown in Figure 4.6. This second proof demonstrates that we can prove the robot never falls in even without exactly characterizing the set of reachable positions.

Reflecting on our two admittedly informal proofs so far, we can boil them each down into three essential proof steps: (1) identify a set of positions I that (2) contains all reachable positions; and that (3) does not intersect the red circle. Each proof step is important. Proof

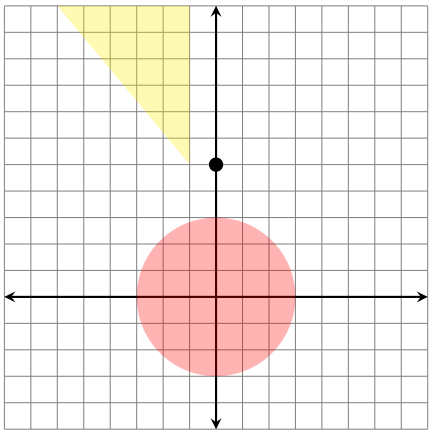


Figure 4.6: Positions from Figure 4.5 that are unreachable.

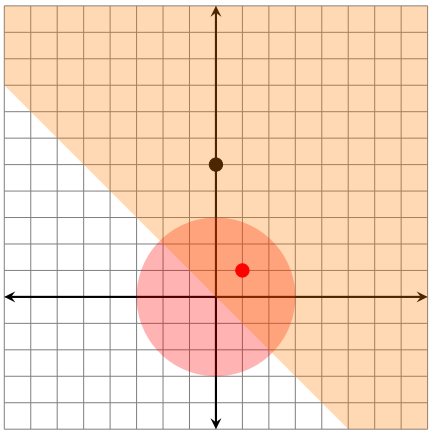


Figure 4.7: The orange region is unsafe because it intersects the red circle.

step (1) requires some creativity and foresight to pick a set that will make proof steps (2) and (3) possible. Proof step (3) has been fairly straightforward so far, because we can visually analyze our set and the red circle to determine if they overlap. If we wanted to be more precise about proof step (3), we could do some algebra to show that any position (x, y) that satisfies the linear inequalities that define the polygonal regions from Figures 4.4 and 4.5 always fall outside the red circle, i.e. $\sqrt{x^2 + y^2} > 3$.

Proof step (2) is more subtle. We have claimed informally that the robot will never be able to move southwest of the line $x+y = 5$, nor west of the y axis, but what would constitute a more detailed proof of this fact? These claims are claims of *invariance*, i.e., that the set I from proof step (1) is an invariant, where an invariant is a set that contains all reachable positions. The key to proving invariance is an *inductive* argument, which first shows (2.1) that the initial position is in I and then shows (2.2) that, from any position in I, all steps the robot could take lead to new positions that are also in I. By induction, this implies that all reachable positions are contained in I. For the invariants represented in Figures 4.4 and 4.5 we could make proof steps (2.1) and (2.2) more precise by showing that the initial position satisfies the linear inequalities for the polygonal regions and by showing that if (x, y) satisfies

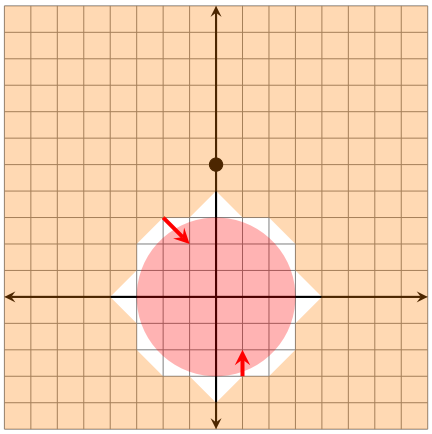


Figure 4.8: The region outside the circle is invariant but not inductive.

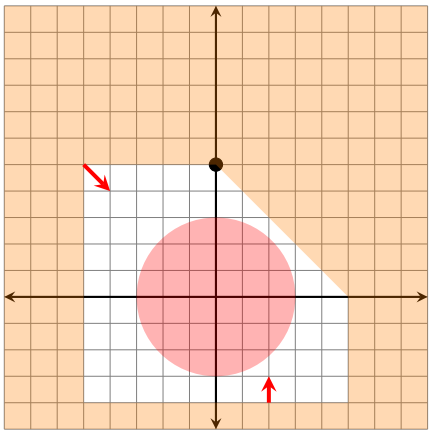


Figure 4.9: This simplified region is also invariant but not inductive.

the inequalities, then so do both $(x, y+1)$ (moving due north) and $(x+1, y-1)$ (moving southeast).

It is instructive to see what happens when the set I selected in proof step (1) fails due to proof steps (2) or (3). When proof step (3) fails, I intersects the red circle, as in Figure 4.7. In this case we call the set I "unsafe". The small opaque red dot shows an example position that is both in the candidate set I drawn as the orange region and the shaded red circle at the origin. When proof step (2) fails, there is a position in I from which the robot can step to a position outside I, as in Figures 4.8 and 4.9. The red arrows show examples of positions in the orange regions that can step out of orange regions. We call such positions "counterexamples to inductiveness", or CTIs.

In the robot example, we can rephrase inductiveness by saying: if a position (x, y) is *not* in I, then neither is any state due south or diagonally northwest of (x, y). Since our goal in any proof of the robot's safety is to come up with a set I that is inductive and does not intersect the red circle, we know the states in the red circle must not be in I. Using the rephrasing above, we can conclude that I must also not contain any state due south or diagonally northwest of the red circle. Continuing to rule out states in this way, we can find the *largest* set I that will make the proof go through, shown in Figure 4.10. To convince

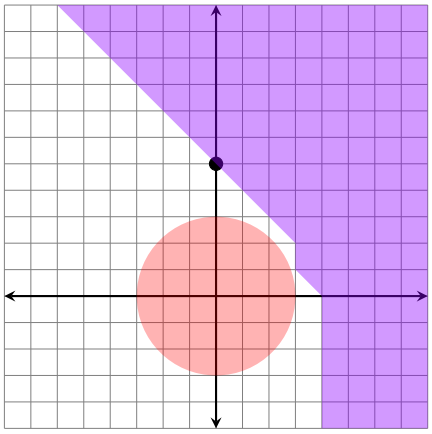


Figure 4.10: The largest safe inductive invariant for the robot.

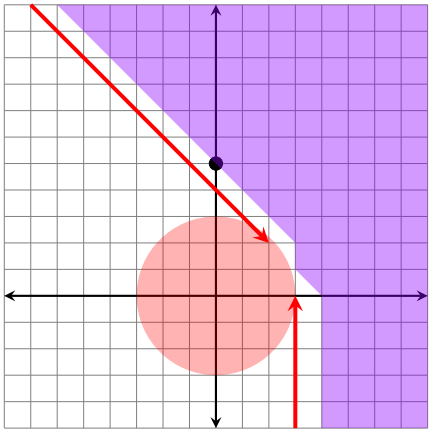


Figure 4.11: Proof that every position outside the purple region is backward reachable.

ourselves that this region I is the largest safe inductive invariant, it is enough to show that every position not in I is *backward reachable*, by which we mean, there is a sequence of steps starting from the position and ending somewhere in the red circle. Figure 4.11 demonstrates that the positions just outside the purple region are all backward reachable, using sequences of steps that follow the red arrows to the red circle. Every other position outside the purple shaded region is either due south or diagonally northwest of one of these two red arrows, and so is also backward reachable.

Because the robot example is so simple, we are able to exactly characterize the set of reachable states (Figure 4.4) and the set of backward reachable states (Figure 4.11). In more complex examples, this will be much more difficult, but luckily it will always be sufficient to find an overapproximation to the set of reachable states, such as Figure 4.5, which proves safety without exactly characterizing reachability or backward reachability.

4.2.2 The robot example in set theory

Let's formalize the robot example from the previous section using the language of set theory.

A *transition system* $\tau = (S, S^0, \rightarrow)$ consists of a set of states S, a set of initial states

$S^0 \subseteq S$, and a binary transition relation $\rightarrow \subseteq S \times S$. We write $s \rightarrow s'$ to mean $(s, s') \in \rightarrow$.

We can formalize the robot example as a transition system as follows.

$$\begin{aligned}
\tau_{\mathsf{robot}} &= (S_{\mathsf{robot}}, S^0_{\mathsf{robot}}, \rightarrow_{\mathsf{robot}}) \\
S_{\mathsf{robot}} &= \{(x, y) \mid x, y \in \mathbb{Z}\} \\
S^0_{\mathsf{robot}} &= \{(0, 5)\} \\
\rightarrow_{\mathsf{robot}} &= \{((x, y), (x, y + 1)) \mid x, y \in \mathbb{Z}\} \cup \\
&\qquad \{((x, y), (x + 1, y - 1)) \mid x, y \in \mathbb{Z}\}
\end{aligned}$$

We write $s \rightarrow^* s'$ to mean that there is a (possibly empty) sequence of steps from s to s'. Using this notation, we can say that a state s is *reachable* if there is an initial state $s_0 \in S^0$ such that $s_0 \rightarrow^* s$. We write $\mathsf{reach}(\tau)$ for the set of reachable states of the transition system τ.

In the robot example, we characterized the set of reachable states in Figure 4.4. We can express the set precisely as

$$\mathsf{reach}(\tau_{\mathsf{robot}}) = \{(x, y) \mid x \geq 0 \text{ and } x + y \geq 5\}$$

The proof of this fact follows the informal discussion from the previous section. First, the set on the right hand side contains all reachable states, which can be shown with an inductive argument (which we spell out below). Second, every state in this set is reachable, which we can show by first showing that everything on the diagonal $x + y = 5 \wedge x \geq 0$ is reachable (by a sequence of southeast moves from the initial state), and then showing that every state above the diagonal is reachable (by a subsequent sequence of north moves).

In the robot example, our goal was to prove the robot never falls in the red circle. In general, the kinds of goals we will be interested in will be *safety properties*, specifically, that a transition system avoids a certain set of *bad states* $B \subseteq S$ that is claimed not to intersect the set of reachable states. In the robot example, the bad states are the ones in the red

circle:

$$B_{\mathsf{robot}} = \left\{(x, y) \mid \sqrt{x^2 + y^2} \leq 3\right\}.$$

The claim that the robot is *safe* means that B_{robot} does not intersect $\mathsf{reach}(\tau_{\mathsf{robot}})$, which is visually apparent in Figure 4.4.

In simple systems, such as the robot example, $\mathsf{reach}(\tau)$ can be characterized directly, but in more complex systems, it is neither practical nor useful to obtain an exact characterization of reachable states. Instead, one often uses *overapproximations* to reachability, known as invariants. We say that a set I is an *invariant* if it contains every reachable state, that is, if $\mathsf{reach}(\tau) \subseteq I$. Thus, another way to talk about safety properties is to say that the complement of the set of bad states is an invariant, i.e. $\mathsf{reach}(\tau) \subseteq S \setminus B$. Indeed, another way to set up the safety verification problem is to specify $S \setminus B$ directly instead of B, and to claim that $S \setminus B$ is an invariant. We call this the *positive phrasing of the safety problem*.

When $\mathsf{reach}(\tau)$ has a direct characterization, one can prove that I is an invariant directly from the definition. But in complex systems where no useful direct characterization of $\mathsf{reach}(\tau)$ exists, one instead uses an inductive argument to establish invariance. We say that I is an *inductive invariant* if: (1) $S^0 \subseteq I$; and (2) for any $s \in I$, if $s \to s'$, then $s' \in I$.

Both the green region from Figure 4.4 and the blue region from Figure 4.5 are invariants, and in fact are inductive invariants. (Indeed $\mathsf{reach}(\tau)$ is always an inductive invariant in any transition system.) However, not all invariants are inductive invariants. For example, the orange regions from Figures 4.8 and 4.9 are both invariants (because they contain the green region from Figure 4.4) but neither are inductive. If we wanted to use an inductive argument to show that these orange regions are invariants, we would need to first *strengthen* them.

In general, suppose we want to prove that I is an invariant of some transition system τ, and that we don't have a useful direct characterization of $\mathsf{reach}(\tau)$ handy. Further, suppose that I is not inductive. To show that I is an invariant, it is enough to find another set J such that: (1) $J \subseteq I$; and (2) J is inductive. Since J is inductive, J is an invariant, i.e. $\mathsf{reach}(\tau) \subseteq J$. But $J \subseteq I$, so I is also an invariant. In the robot example, we could use the

blue region from Figure 4.5 as an inductive strengthening to prove that either of the orange regions is an invariant. Most of the art and skill of working with transition systems is in the ability to look at a non-inductive property and see what kinds of strengthenings of it might be good ideas to get it to be inductive.

4.2.3 First-order logic for transition systems

We are on a journey to more and more formally specify the robot example and to prove its safety more and more automatically. Our next stop is first-order logic, which we assume the reader is at least passingly familiar with. (See Section 4.4 for a formal presentation.) The basic idea is to make the states of a transition system *first-order structures* over *state variables*, and to use formulas to describe the initial states, the transition relation, the bad states, and the inductive invariant.

For the robot, the state variables are x and y, both of sort $\mathbb{Z}$. A first order structure is just an assignment of variables to values of the correct sort. In this case, a structure will be an integer for x and an integer for y, that is, a pair of integers, or, in yet other words, a point in the plane.

The robot only has one initial state, $(5, 0)$. We can describe this as a logical formula over the variables x and y as

$$Init_{\mathsf{robot}} \quad = \quad x = 0 \wedge y = 5.$$

Taking the positive phrasing of the robot example's safety problem, we want to show that the set of states with distance strictly greater than 3 from the origin is an invariant of the transition system. We can express this safety invariant as follows (avoiding square roots by squaring both sides)

$$Safe_{\mathsf{robot}} \quad = \quad x^2 + y^2 > 9.$$

Since the initial condition and the safety condition of a transition system are *sets* of states, they are encoded in logic as formulas over *one* copy of the state variables. The transition relation, on the other hand, is a *binary relation*, so it is encoded in logic as a

formula over *two* copies of the state variables. (We refer to formulas over two copies of the state variables as "2-vocabulary" or "2-state" formulas, and we call the second copy of the variables "primed" and write them as x', for example.) In the robot example, we can write the transition relation as

$$Tr_{\mathsf{robot}} \quad = \quad (x' = x \wedge y' = y + 1) \vee (x' = x + 1 \wedge y' = y - 1).$$

We think of the primed variables x' and y' as holding the values *after* the step occurs, while the unprimed versions hold the values from before the step. The transition formula says:

> A step is possible from (x, y) to (x', y') if either x stays the same and y is incremented by one, or x is incremented by one and y is decremented by one.

Next, we can state the blue region from Figure 4.5 as a formula

$$Inv_{\mathsf{robot}} \quad = \quad x + y \geq 5.$$

If we are trying to prove *Safe* using *Inv*, there are two things to check. First, *Inv* must be strong enough to prove *Safe*, that is,

$$Inv \Rightarrow Safe.$$

In the robot example, this amounts to the fact that the blue region from Figure 4.5 does not intersect the red circle at the origin,

$$x + y \geq 5 \Rightarrow x^2 + y^2 > 9.$$

Second, *Inv* must be inductive, which is phrased logically using two formulas, one to check that the initial condition implies the invariant,

$$Init \Rightarrow Inv,$$

and one to check that the invariant is preserved by the transition relation,

$$Inv \wedge Tr \Rightarrow Inv'.$$

The first of these two formulas is over a single copy of the state. But the second formula is a 2-vocabulary formula, both because it contains the 2-vocabulary formula Tr, and because we use Inv' to mean "Inv with all the state variables changed to their primed copy". In the robot example, there is only one initial state, so the check on initial states amounts the fact that the initial state is in the blue region or, logically,

$$x = 0 \wedge y = 5 \Rightarrow x + y \geq 5.$$

To check that the blue region is preserved by the robot's steps, we need to observe that there is no way to start in a state in the blue region and take a step outside of it, or, logically,

$$x + y \geq 5 \wedge ((x' = x \wedge y' = y + 1) \vee (x' = x + 1 \wedge y' = y - 1)) \Rightarrow x' + y' \geq 5.$$

This formula is valid, since either $y' > y$ and $x' = x$ in the first kind of step, or $x+y = x'+y'$ in the second kind of step. While a bit tedious, this is perhaps the first time in the robot example that we haven't had to hand-wave around the fact that one of the regions is inductive. It follows from the tedium of checking this formula's validity.

4.3 The Robot in `mypyvy`

It's finally time to write some `mypyvy`. Let's take a tour of the syntax using our robot example. A `mypyvy` program is a list of *declarations*. There are two broad kinds of declarations: those that define the transition system and those that make *queries* over the transition system.

Here are two declarations for the state variables of the robot.

```
mutable constant x: int
```

```
mutable constant y: int
```

In mypyvy, we spell "variable" as "**mutable constant**", following the logical terminology of a constant symbol.[2] We discuss mutability in more detail in the next section. All the state declarations together define the state space of the transition system.

The initial conditions are declared with the **init** keyword.

```
init x = 0
init y = 5
```

The conjunction of all the **init** declarations in a mypyvy program is the initial condition of the resulting transition system.

The transition relation is declared with several **transition** declarations.

```
transition north()
  modifies y
  new(y) = y + 1

transition south_east()
  modifies x, y
  & new(x) = x + 1
  & new(y) = y - 1
```

Each **transition** has a name and takes parameters in parentheses, which are existentially quantified. The **modifies** clause is a comma-separated list of state components that are changed by this transition; all other state components are implicitly constrained to not change. For example, in the **north** transition, there is an implicit constraint $x' = x$, which was explicit when we were writing directly in first-order logic in the previous section. In mypyvy syntax, instead of primed symbols, we use the **new** keyword, so **new(y)** refers to y'. mypyvy also allows conjunction and disjunction symbols (**&** and **|**) to appear *before* the first

[2]Logicians use the terminology of relations, constants, and functions, and mypyvy has both immutable and mutable versions of each of these. Unfortunately, the combined terminology clashes badly, and we end up with the somewhat strange sounding phrase "**mutable constant**".

conjunct or disjunct, for the sole reason that it makes it easy to vertically align the lines of a long formula. The global transition relation of the transition system is the *disjunction* of all the transition declarations in the program.

The safety property is declared using the **safety** keyword.

```
safety [no_fall_in] x * x + y * y > 9
```

A safety property can be given a name by including it in square brackets. In this case the name is `no_fall_in`. `mypyvy` will use this name to refer to this invariant. If no name is given, `mypyvy` will refer to a line number instead. If the program has more than one **safety** declaration, the safety property is the *conjunction* of all the declarations. The **safety** keyword is our first example of a declaration that is a *query*. When processing the program, `mypyvy` will attempt to prove that the safety property is true in all reachable states of the system. As we have discussed throughout this chapter, the primary technique for proving a safety property is to find an inductive invariant.

These invariants are declared using the **invariant** keyword.

```
invariant x + y >= 5
```

Invariants can also be named, but here we choose to not name the invariant. All the **invariant** declarations in a program are implicitly conjoined.[3]

That completes the robot example in `mypyvy`. Running `mypyvy` on the resulting file causes it to issue queries to the solver to verify that the invariants are inductive and imply the safety property. For this program, the verification succeeds. `mypyvy`'s output can be seen in the top half of Figure 4.13. Commenting out the **invariant** declaration causes the verification to fail because the safety property is not itself inductive. `mypyvy`'s output in this case can be seen in the bottom half of Figure 4.13. `mypyvy` shows a counterexample to

[3]One could imagine wanting to check that two different invariants are separately inductive. We have thus far chosen not to support this use case in `mypyvy` because: (1) it seems more common that one cares about *invariance* rather than *inductiveness*, and there is no such thing as "invariance, separately", unlike inductiveness; and (2) supporting separate checking would require a more verbose syntax for verification queries, whereas the current approach allows a `mypyvy` file to contain a single implicit verification query of inductiveness of the whole invariant.

```
1   mutable constant x: int
2   mutable constant y: int
3
4   init x = 0
5   init y = 5
6
7   transition north()
8     modifies y
9     new(y) = y + 1
10
11  transition south_east()
12    modifies x, y
13    & new(x) = x + 1
14    & new(y) = y - 1
15
16  # don't fall in!
17  safety [no_fall_in]
18    x * x + y * y > 9
19
20  invariant x + y >= 5
```

Figure 4.12: The complete robot example in `mypyvy`.

```
checking init:
  implies invariant no_fall_in... ok.
  implies invariant on line 20... ok.
checking transition north:
  preserves invariant no_fall_in... ok.
  preserves invariant on line 20... ok.
checking transition south_east:
  preserves invariant no_fall_in... ok.
  preserves invariant on line 20... ok.
all ok!
----------------------------------------
checking init:
  implies invariant no_fall_in... ok.
checking transition north:
  preserves invariant no_fall_in...
state 0:
x = 1
y = -3
state 1:
x = 1
y = -2
error robot.pyv:17:1: invariant no_fall_in
    is not preserved by transition north
error robot.pyv:7:1: this transition does
 not preserve invariant no_fall_in
program has errors.
```

Figure 4.13: `mypyvy` output when running on two versions of the robot.

induction (CTI), which is a pair of states related by the transition relation (in this case, the `north` transition) such that the first state satisfies the candidate invariant but the second state does not. The CTI in Figure 4.13 corresponds to the red arrow pointing north in Figure 4.8.

4.4 Background on first-order logic

In this section, we return to the basics of first-order logic. The ideas of first-order logic are not complicated, but they are tedious to set up formally. We will do our best. There are many other presentations of first-order logic in the literature, see, e.g., Libkin's book on

$$
\begin{array}{rcl}
u & \in & \text{String} \\
s & ::= & u \mid \mathsf{bool} \\
t & \in & \text{FunctionType} \\
t & ::= & (s, \ldots, s) \to s \\
\sigma & \in & \text{String} \rightharpoonup \text{FunctionType} \\
e & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg e \mid e \wedge e \mid e \vee e \mid e \Rightarrow e \mid \\
 & & e = e \mid f(e, \ldots, e) \mid x \mid \forall x : s.\, e \mid \exists x : s.\, e
\end{array}
$$

Figure 4.14: Syntax of pure, multi-sorted first-order logic.

finite model theory [67].

Figure 4.14 describes the syntax of first-order logic. A *sort* s is either bool or an uninterpreted symbol u. A *function type* t is some number of argument sorts and a return sort. A *vocabulary* is a map from function names to function types. We write $\mathsf{usorts}(\sigma)$ for the set of all uninterpreted sort symbols used in the type of any function symbol in σ. We distinguish several special classes of function types. If a function c has zero arguments, then we say that c is a *constant*, and in expressions we abbreviate $c()$ as c. If a function R has return type bool, then we say that R is a *relation*.

An *expression* is one of: a boolean constant, the application of a unary or binary boolean operation to subexpressions, an equation between expressions, the application of a function to some number of arguments, a variable, or a quantified expression. There is a straightforward type system that assigns a sort (either an uninterpreted sort or bool) to each expression, in the context of a vocabulary and a local context that binds variables to sorts.[4] The type system also enforces that functions are applied to the correct number of arguments, that function arguments have types corresponding to the functions declared type in the vocabulary, and that both sides of an equation have the same type. We omit the definition of this type

[4]Our approach to first-order logic is somewhat nonstandard, in that we treat bool as "just another sort", rather than having a separate syntactic category for formulas and terms. This setup is somewhat more parsimonious in the presence of multiple uninterpreted sorts, and it also has the advantage of allowing quantifying over booleans. In any case, the fundamental expressivity of the logic is not changed by these choices.

system, and implicitly assume that all expressions are well typed. We use the word *formula* to mean "an expression of type bool".

A *first-order structure* A over a vocabulary σ is (1) a set A_u for every uninterpreted sort $u \in \mathsf{usorts}(\sigma)$ and (2) for every function symbol $f \in \mathsf{dom}(\sigma)$ with type $(s_1, \ldots, s_n) \to s$, a corresponding mathematical function $A_f : A_{s_1} \times \cdots \times A_{s_n} \to A_s$. (In case one of the sorts is bool, we define $A_{\mathsf{bool}} = \mathbb{B}$, i.e., every first-order structure interprets bool as the set $\{\top, \bot\}$.) We call $\bigcup_{u \in \mathsf{usorts}(\sigma)} A_u$ the *universe* of A.

Given a vocabulary σ, a first-order structure A over σ, an environment $\rho \in \mathrm{String} \rightharpoonup \bigcup_{u \in \mathsf{usorts}(\sigma)} A_u$, and a first-order expression e of sort s, we define $[\![e]\!]^\rho_A$, the interpretation of e in A, as follows.

$$
\begin{array}{rcl}
[\![e]\!]^\rho_A & \in & A_s \\
[\![\mathsf{true}]\!]^\rho_A & = & \top \\
[\![\mathsf{false}]\!]^\rho_A & = & \bot \\
[\![\neg e]\!]^\rho_A & = & \neg [\![e]\!]^\rho_A \\
[\![e_1 \wedge e_2]\!]^\rho_A & = & [\![e_1]\!]^\rho_A \wedge [\![e_2]\!]^\rho_A \\
[\![e_1 \vee e_2]\!]^\rho_A & = & [\![e_1]\!]^\rho_A \vee [\![e_2]\!]^\rho_A \\
[\![e_1 \Rightarrow e_2]\!]^\rho_A & = & [\![e_1]\!]^\rho_A \Rightarrow [\![e_2]\!]^\rho_A \\
[\![e_1 = e_2]\!]^\rho_A & = & [\![e_1]\!]^\rho_A = [\![e_2]\!]^\rho_A \\
[\![f(e_1, \ldots, e_n)]\!]^\rho_A & = & A_f([\![e_1]\!]^\rho_A, \ldots, [\![e_n]\!]^\rho_A) \\
[\![x]\!]^\rho_A & = & \rho(x) \\
[\![\forall x : s.\, e]\!]^\rho_A & = & \forall a \in A_s.\, [\![e]\!]^{\rho[x \mapsto a]}_A \\
[\![\exists x : s.\, e]\!]^\rho_A & = & \exists a \in A_s.\, [\![e]\!]^{\rho[x \mapsto a]}_A
\end{array}
$$

If $e : \mathsf{bool}$ and $[\![e]\!]^\rho_A$ holds, then we say that A is a *model* of e, and we write $A \models e$. We say that e is *satisfiable* if there exists a first order structure A over σ such that $A \models e$. We say that e is *valid* if for every first order structure A over σ, $A \models e$. A formula e is valid if and only if $\neg e$ is unsatisfiable (i.e., not satisfiable). If $e_1 : \mathsf{bool}$ and $e_2 : \mathsf{bool}$ are two formulas

over σ, then e_1 is *semantically equivalent* to e_2 if for every structure A, $A \models e_1$ if and only if $A \models e_2$.

Let A be a first order structure over σ, and for each $u \in \mathsf{usorts}(\sigma)$ let $B_u \subseteq A_u$. Then define $B_f = A_f|_{B_{s_1} \times \cdots \times B_{s_n}}$, for each $f \in \mathsf{dom}(\sigma)$ of type $(s_1, \ldots, s_n) \to s$. Then in order for B to be a first order structure, we need it to be "closed" under all function symbols, i.e.,

$$B_f(B_{s_1}, \ldots, B_{s_n}) \subseteq B_s.$$

(In the case that $s = \mathsf{bool}$, there is nothing to show here, because $B_{\mathsf{bool}} = \{\top, \bot\} = A_{\mathsf{bool}}$.) We call B the "restriction of A to the B_u". It is sometimes convenient to gloss over the distinction between the different sorts u and to speak of "the restriction of A to the set X", where $X \subseteq \bigcup A_u$. We say that B is a *substructure* of A if B is the restriction of A to $\bigcup B_u$. (This really just amounts to saying that B is a first-order structure over σ whose sets are subsets of A and whose interpretation of every function symbol agrees with A's interpretation.)

Substructures agree on the interpretation of quantifier free expressions, as shown by the following lemma.

Lemma 4.4.1. Let e_0 be quantifier free (not necessarily of sort bool), and let A and B be first-order structures over σ such that B is a substructure of A. Also, let $\rho \in \mathrm{String} \rightharpoonup \bigcup_{s \in \mathsf{usorts}(\sigma)} B_s$. Then $[\![e]\!]^\rho_B = [\![e]\!]^\rho_A$.

Intuitively, a quantifier-free formula can only "see" elements of the structures A and B if they are in the image of the interpretation of some function symbol, or if they are in the environment ρ. Since substructures agree on all function interpretations, the interpretation of e_0 can't change as we pass to the substructure B.

Proof. Induction on e using the definition of $[\![\cdot]\!]$. □

A formula $e : \mathsf{bool}$ is *quantifier free* if its syntax does not contain any uses of $\forall$ or $\exists$. A formula $e : \mathsf{bool}$ is called *universal* if it is equivalent to a formula of the form $\forall^* e_0$, where

e_0 is quantifier free. (Here we use $\forall^*$ to abbreviate some (possibly empty) sequence of $\forall$ quantifiers over unknown sorts.)

Universal formulas have a special relationship with substructures, as shown by the following lemma.

Lemma 4.4.2. Let e be universal and suppose $A \models e$ and that B is a substructure of A. Then $B \models e$.

Intuitively, e says something about "all" elements of A, and there are no elements of B that are not also elements of A, so e also says the same thing about "all" elements of B.

Proof. Write $e = \forall y_1 : s_1, \ldots y_n : s_n . e_0$, where e_0 is quantifier free. Since $A \models e$, we know that for all $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$, $[\![e_0]\!]_A^{[y_1 \mapsto a_1, \ldots, y_n \mapsto a_n]}$ holds. Now let $b_1 \in B_{s_1}, \ldots, b_n \in B_{s_n}$ be arbitrary. We need to show that $[\![e_0]\!]_B^{[y_1 \mapsto b_1, \ldots, y_n \mapsto b_n]}$ holds.

Since B is a substructure of A, we also have $b \in A_s$, and so we have $[\![e_0]\!]_A^{[y_1 \mapsto b_1, \ldots, y_n \mapsto b_n]}$ holds. Then by Lemma 4.4.1, this is the same as $[\![e_0]\!]_B^{[y_1 \mapsto b_1, \ldots, y_n \mapsto b_n]}$, which completes the proof. □

If a vocabulary σ consists of only relations and constants, then we call it "function free". If σ is function free and e : bool over σ, then we say that e is *effectively propositional for satisfiability* if e is equivalent to a formula of the form $\exists^* \forall^* e_0$, where e_0 is some quantifier-free formula. Taking the negation, we also obtain *effectively propositional formulas for validity*, which have the form $\forall^* \exists^* e_0$. The fragment of first-order logic consisting of only effectively propositional formulas is known as "effectively propositional reasoning" or EPR. When it is clear from context whether we are talking about satisfiability or validity, we allow ourselves to use the abbreviation "e is in EPR" to mean one of the two notions above.

The reasons for defining these classes of formulas is that there are decision procedures for satisfiability and validity.

Lemma 4.4.3 (Small model property)**.** Let σ be function free, and let e : bool over σ be effectively propositional for satisfiability. Then there exists a natural number n (depending on σ and e) such that e is satisfiable if and only if it has a model of size at most n.

Intuitively, since e is EPR, after removing existential quantifiers, we are left with a universal formula. So given a model of e we can construct a substructure consisting of only the interpretations of the existential variables and the constants of σ. Since the remainder of the formula is universal, it also holds in this substructure. Finally, the size of the substructure is bounded by the number of constant symbols in σ plus the number of existential variables of e.

Proof. Since e is in EPR, we can write $e = \exists x_1, \ldots, x_m. \forall y_1, \ldots, y_k. e_0$, where one or both of m and k might be zero, and e_0 is quantifier free. Let l be the number of constant symbols in σ, say $c_1, \ldots, c_l$, and set $n = m + l$. We need to show e is satisfiable iff it has a model of size at most n. The backwards direction follows directly.

In the forwards direction, suppose $A \models e$ for some (possibly infinite) first-order structure A. Then peeling apart the interpretation of e in A, we find that there must exist elements $a_1, \ldots, a_m$ of A such that interpreting the x_i as a_i causes e to come out true, i.e.,

$$[\![\forall y_1, \ldots, y_k. e_0]\!]_A^{[x_1 \mapsto a_1, \ldots, x_m \mapsto a_m]}$$

must hold.

Now let B be the restriction of A to the set $X = \{a_1, \ldots, a_m, A_{c_1}, \ldots, A_{c_l}\}$. We claim B is a model of e. First, we must show B is closed under all function symbols. Since σ is function free, we need only consider relations and constants. If $r \in \mathsf{dom}(\sigma)$ is a relation, then there is nothing to show, since we never restrict the interpretation of the boolean sort. If $c \in \mathsf{dom}(\sigma)$ is a constant, then $B_c() = A_c \in X$, since X contains the interpretation of every constant symbol.

By Lemma 4.4.2,

$$[\![\forall y_1, \ldots, y_k. e_0]\!]_B^{[x_1 \mapsto a_1, \ldots, x_m \mapsto a_m]}$$

holds, and the result follows. □

Decidability of EPR follows as a corollary, after the following lemma.

Lemma 4.4.4. Given a fixed (finite!) vocabulary σ (not necessarily function free), there are only finitely many first-order structures of any particular universe size.

Proof. Fix a particular universe size n. Then for each constant symbol c in σ, there are at most n choices of how to interpret c in the universe. For each relation symbol R, there are at most 2^{nk} choices for how to interpret R, where k is the arity of R. And for each function symbol f, there are at most n^{nk}, where k is there arity of f. Taking the product of all these choices across all symbols in σ gives us an upper bound on the number of first order structures over σ with universe size n. □

Theorem 4.4.5. Let σ be function free, and let e : `bool` over σ be effectively propositional for satisfiability. Then it is decidable whether e is satisfiable.

Proof. By Lemma 4.4.3, it suffices to check for a model of up to size n, where n is the number of constants in σ plus the number of existentially quantified variables of e. Then by Lemma 4.4.4, there are finitely many structures of size at most n. We can enumerate them and check for each one whether it is a model of e. If we find a model, e is satisfiable. If we exhaust all structures of size at most n, then we know no model of e exists of any size (even infinite!). □

4.5 Expressing Transition Systems in `mypyvy`

`mypyvy` consists of a language for expressing transition systems (this section) and a tool for answering queries about such systems (Section 4.6). The `mypyvy` language closely corresponds to the theoretical development of first-order logic from Section 4.4.

A `mypyvy` file consists of a sequence of declarations, which together define a transition system. Recall that a transition system consists of a state space, a set of initial states, and a transition relation. There are declarations for all of these things. In addition, there are declarations to record other properties of interest, such as inductive invariants and safety properties. While not officially part of the definition of the transition system, it makes sense to record these properties in the same file, so that queries about the system can make easy

use of them. For example, there is a query to check that all the invariants in a file actually are invariants.

A transition system consists of a state space, a set of initial states, and a transition relation. `mypyvy` has one or more declarations corresponding to each of these components.

To define the state space, `mypyvy` supports declaring uninterpreted sorts, constants, relations, and functions. The **`sort`** keyword declares the name of an uninterpreted sort. Uninterpreted sorts can be used in the types of other declarations in the file. The state consists of constants, relations, and functions, which we refer to as "state variables". Each state variable can by **`immutable`** or **`mutable`**. An **`immutable`** state variable cannot be changed by the transition relation throughout an execution of the system, while a **`mutable`** state variable can evolve over time. There are keywords **`constant`**, **`relation`**, and **`function`** for each category of state variable. Here are some examples of sort declarations and state variable declarations.

```
sort A
immutable constant source: A
mutable relation r(A)
immutable function f(A): A
```

Each state variable declaration starts with its mutability, then its category, followed by its name. After the name, the sorts of the arguments (if any) are given, followed by a return type (if any; relations always return `bool`).

To define the initial conditions, `mypyvy` provides the **`init`** keyword. An **`init`** declaration consists of this keyword followed by a 1-state expression of type `bool`. For example, the following declaration says that initially the relation `r` contains only the element `source`.

```
init forall X:A. r(X) <-> X = source
```

If a file contains more than one **`init`** declaration, they are all conjoined together to form the initial condition for the transition system.

To define the transition relation, `mypyvy` provides the **`transition`** keyword. A **`transition`** declaration consists of this keyword followed by a name and some number of comma-

separated parameters in parentheses, then a modifies clause, and then a 2-state expression of type `bool`. For example, the following transition says that for any element currently in the `r` relation, you can add its image under the function `f` to `r` as well.

```
transition step(x: A)
  modifies r
  & r(x)
  & (forall X. new(r(X)) <-> r(X) | X = f(x))
```

The transition is named `step` and takes one parameter called `x` of type `A`. Parameters are implicitly existentially quantified in the global transition relation. The modifies clause declares which mutable state components are changed by the transition. For any mutable state component *not* in the modifies clause, `mypyvy` implicitly adds a conjunct to the transition saying that that component does not change. If there is more than one **`transition`** declaration in the file, then their *disjunction* forms the transition relation of the transition system.

The **`invariant`** declaration takes a 1-state expression and conjoins it to the inductive invariant for the transition system. `mypyvy` can check that the resulting conjunction is actually an inductive invariant. Some invariants can be marked with the The **`safety`** keyword instead, which indicates that these conjuncts are the "specification" of the transition system, and that the non-**`safety`** conjuncts are only there to make the invariant inductive. For example, consider the following two **`safety`** declarations.

```
# oops! not true initially
safety forall X. r(X) -> exists Y. X = f(Y)

# true and inductive
safety forall X. r(X) -> X = source | exists Y. X = f(Y)
```

In the transition system example of this section, the first declaration says that every member of the relation `r` is in the image of the function `f`. This is not true in the initial state, though, since `source` might not be in the image of `f`. The second declaration fixes this problem by

saying that every member of the relation `r` is either equal to `source` or is in the image of the function `f`.

The immutable symbols of a transition system form a sort of "background theory" for the system. `mypyvy` provides the **`axiom`** declaration, which takes a 0-state formula and adds it as an axiom of the background theory. For example, we can use axioms to model the existence of a total order on a sort.

```
sort A
immutable relation le(A, A)
axiom le(X, X)
axiom le(X, Y) & le(Y, X) -> X = Y
axiom le(X, Y) & le(Y, Z) -> le(X, Z)
axiom le(X, Y) | le(Y, X)
```

Since the immutable relations are never updated by the transition relation, these axioms are true in every state.

The **`theorem`** declaration takes a k-state formula and ensures its validity given the background theory. This can be used to prove that two alternate formulations of an invariant are equivalent, for example, or that one invariant or safety property implies another. For example, given the background theory about `le` above, we could state a simple corollary that `le` is "3-transitive".

```
zerostate theorem
  le(X, Y) & le(Y, Z) & le(Z, W) -> le(X, W)
```

`mypyvy` can check the validity of these theorems. See the next section.

4.5.1 k-state expressions

The expressions of `mypyvy` closely follow those of first-order logic. When reasoning about a transition system, one often has two copies of the state components in scope, those from the pre-state of a transition, and those from the post-state. These two copies of the state

components share a single copy of the immutable components; only the mutable components are duplicated. An expression can refer to the post-state copy of a mutable symbol using the **new** operator, as in **new**(r(X)). We call such expressions 2-state expressions. For example, the body of each transition is a 2-state expression. In contrast, many other expressions are more accurately referred to as 1-state expressions, meaning they only have one copy of the mutable symbols in scope. For example, safety properties, invariants, and initial conditions all contain 1-state expressions. Additionally, some expressions have *no* copies of the mutable symbols in scope. We call these 0-state expressions. Axioms are the primary example of 0-state expressions; they can only refer to the immutable symbols.

In fact, further generalizations are possible to k-state expressions for any natural number k. For example, one could write a 3-state expression, which describes a sequence of 3 states, where the first state's components are referred to directly, the second state's components use **new**, and the third state's component use two applications of **new**. Internally, `mypyvy` fully supports k-state expressions, but their only user-facing appearance is in trace queries, which allow a customized form of bounded model checking.

A k-state formula can be thought of as a first-order formula over an extended vocabulary with k copies of the mutable symbols. A model for a k-state formula is a model for this first-order formula, i.e., a first-order structure that assigns interpretations to k copies of the mutable symbols. Models of k-state formulas often arise in the answers to queries over transition systems, described in the next section.

4.6 Queries on Transition Systems

`mypyvy` supports several solver-aided queries over transition systems.

Inductiveness checking. The most common query is to check that the invariant specified in the file is an inductive invariant of the transition system. `mypyvy` performs this query on a file when run with the `mypyvy verify` subcommand. As described at a high level in

Section 4.2.3, checking inductiveness consists of proving validity of the two formulas:

$$Init \Rightarrow Inv,$$

and

$$Inv \wedge Tr \Rightarrow Inv'.$$

In the context of `mypyvy`, *Init* is the conjunction of all **`init`** declarations, *Inv* is the conjunction of all **`invariant`** declarations, and *Tr* is the *disjunction* of all **`transition`** declarations. Also, the prime symbol ($'$) is written **`new(...)`** in `mypyvy`. These two high-level queries can easily be directly expressed to the underlying solver. As usual, to prove validity using a satisfiability solver, we negate the formula and hope for an `unsat` result. In fact, `mypyvy` performs one small optimization, which is to expand all top-level disjunctions outside the solver. Instead of issuing a single query for the entire transition relation, `mypyvy` issues separate queries for each transition. Similarly, though perhaps less obviously, since the right-hand side of each query's implication is a conjunction, after the validity-to-satisfiability negation, this also becomes a disjunction at the solver level, and `mypyvy` issues separate queries for each conjunct in the invariant. So in total, if there are n invariants and m transitions, `mypyvy` issues n 1-state queries to check that the initial states satisfy the invariant, plus mn 2-state queries to check that all invariants are preserved by all transitions. In our anecdotal experience, splitting these disjunctions outside the solver improves performance and reliability, and, best of all, when something is taking a long time (usually because something is not inductive), the user can see what transition and invariant conjunct the solver is stuck on, improving transparency of the tool.

Bounded model checking. Given a transition system and a safety property, bounded model checking (BMC) asks, "Is there a counterexample to safety in up to k transitions?" `mypyvy` expresses this query to the underlying solver by successively "unrolling" up to k copies of the transition relation and asking whether there is a violation in exactly that many

steps. More precisely, to check whether there is a counterexample in exactly i transitions, `mypyvy` sets up $i + 1$ copies of the mutable state variables such that: (1) the first copy satisfies the initial conditions; (2) the last copy *violates* safety; (3) all adjacent pairs of states are related by the transition relation. If the solver finds a model for this query, it can be interpreted as an execution trace that starts in an initial states, and takes some sequence of transitions until it reaches a state that violates safety. Such an example indicates a problem with either the transition system or the safety property. To check for counterexamples *up to* k transitions, `mypyvy` separately checks for counterexamples with exactly i steps for each i from 1 to k.

The naïve approach to bounded model checking described here does not scale as k grows and as the transition relation becomes more and more disjunctive (i.e., more transitions). Partially to workaround these scalability problems, `mypyvy` also supports "trace queries."

Trace queries. A trace query is "just" a k-state formula, whose satisfiability encodes some kind of restricted reachability property that the user is interested in. For example, in a model of a distributed system with many actions, naïve BMC will only scale to a small depth, say 5 actions, in a reasonable amount of time, but many interesting behaviors of the system may not occur until a much greater depth, say 10 or 15 actions. Of course, to gain confidence about all reachable states, no matter how deep, the user should prove an inductive invariant that implies safety. But during testing, it can be convenient to explore deeper into the state space without needing to come up with an invariant. The user can write a trace query which pares down the set of transitions to explore at each step. For example, if the distributed system starts by electing a leader before accepting further requests, the user can create a trace query, which lists a sequence of transitions that result in a leader being elected, followed by a sequence of transitions to accept a request. At the end of this sequence, the user can include an assertion, such as "the final state reached on this trace violates safety." The resulting k-state expression can be checked for satisfiability. If it is unsatisfiable, then there is no execution violating safety. Trace queries can be solved at a much higher depth than

BMC because each step of the execution is constrained to execute a single kind of transition (or just a handful of transitions), rather than the disjunction of all possible transitions. This helps the solver not get confused enumerating many different possible combinations.

In addition to trace queries that are expected to be unsatisfiable, it is also useful to make trace queries that are expected to be satisfiable, to check, e.g., that there is at least one execution where a leader gets elected successfully. This latter kind of query is especially useful for detecting vacuity bugs, where a typo in the formal model causes one or more transitions to be equivalent to false. Vacuity bugs are impossible to detect by proving inductive invariants, because every invariant is preserved by false! So trace queries become a crucial tool to ensure a transition system accurately reflects its author's understanding and intentions.

Theorems. `mypyvy`'s **`theorem`** declaration checks the validity of a k-state formula in the background theory of the transition system, as discussed in Section 4.5. Although the syntax is quite different from trace queries, the implementation is very similar. A trace query is a k-state formula that the user declares to be "expected satisfiable" or "expected unsatisfiable". On the other hand, a theorem is k-state formula that the user declares is "expected valid", i.e., its negation is expected to be unsatisfiable.

Together, trace queries and the **`theorem`** declaration cover three out of the four possibilities of the satisfiable/unsatisfiable and negated/not-negated dimensions. The remaining possibility would be a formula whose negation is expected to be satisfiable. A reasonable name for this idea would be "nonvacuous," and it seems in principle that it could be as useful as satisfiable trace queries are. A future version of **`mypyvy`** may unify trace queries and theorems into one syntactic declaration form, and allow for "nonvacuous" queries as well.

Universal Property-Directed Reachability (PDR$^\forall$). `mypyvy` includes an implementation of PDR$^\forall$, which can infer universally quantified inductive invariants in pure first-order logic (i.e., without arithmetic) [48]. PDR$^\forall$ takes a transition system and a desired safety property and tries to construct an inductive invariant which implies safety. If it succeeds, it

returns the inductive invariant. If it does not succeed, PDR$^\forall$ can either loop forever or return a "relaxed counterexample". A relaxed counterexample *proves* that there is no universally quantified inductive invariant that implies safety. A relaxed counterexample consists of a sequence of interleaved transitions from the transition system and "relaxation steps", which are steps where some elements of some sorts get deleted.

The space of possible invariants inferred by PDR$^\forall$ are conjunctions of universally quantified clauses of pure literals. By pure literal, we mean either a pure atom or its negation, and by pure atom we mean either a relational fact over variables, an equation between a constant and a variable, or an equation between a function applied to variables and a variable. The reason for this seemingly obscure search space is due to the way PDR$^\forall$ constructs candidate invariants, which is by finding backwards reachable states and then "blocking" them by computing a "forbidden sub-state" that rules out all states with a certain pattern of relation/constant/function facts. By default, when asked to perform PDR$^\forall$, `mypyvy` uses the invariants in the file marked **`safety`** as the target property. `mypyvy`'s implementation of PDR$^\forall$ has been a useful baseline for other invariant inference research to compare against, see, e.g., Phase-PDR$^\forall$ [25].

Answers to queries, or, how to read counterexamples. While each of the above queries is quite different, they all occasionally return answers that involve models of k-state formulas. For example, when inductiveness checking fails, it returns either a 1-state model demonstrating a violation of safety in an initial state, or a 2-state model demonstrating a counterexample to induction. As another example, when bounded model checking finds an execution that violates safety, it returns a k-state model witnessing the executions trace. Queries often have other possible outputs besides k-state models, e.g., "verified" or "no trace found" or, for PDR$^\forall$, a formula that is an inductive invariant proving safety. But these other outputs are either simple or have well understood existing output formats.

We have already seen one example of `mypyvy`'s output format for displaying k-state models: the bottom half of Figure 4.13 shows a CTI (i.e., a 2-state model) for the robot

system. In that example, the 2-state model is printed by showing the values of all the variables in state 0 and in state 1. More generally, for a model with k states, `mypyvy` first displays the values of all the *immutable* symbols, and then, for each state, the values of the mutable symbols in that state. For relational symbols, by default `mypyvy` only prints positive literals, i.e., the tuples that are in the relation.

Annotations, plugins, and custom printers for states. `mypyvy` answers queries by calling Z3, and `mypyvy` prints whatever model Z3 returns, translated into `mypyvy` syntax. Often, solver models are strange and hard to read. For example, if a transition system uses a total order relation on one of its sorts, it would make sense to always print that with names that are ordered by the total order, but Z3 is unlikely to do this by chance. To improve the readability of the model format, `mypyvy` supports custom formatting via printer plugins.

Every declaration in `mypyvy` can be tagged with *annotations*, which have no inherent meaning, but can be detected by plugins to cause things to be printed differently. Annotations come after the declaration they are attached to, are written with an at-sign (`@`), and can take arguments in parentheses, as in the following example.

```
sort node @printed_by(ordered_by_printer, le)
```

This annotation tells `mypyvy` that the sort `node` should be printed in the order given by the total order `le`. The annotation name `printed_by` is detected by `mypyvy`'s model printing logic, and when such an annotation is found, it calls a plugin named by the first argument in the annotation, in this case `ordered_by_printer`. That plugin uses the second expression as a relation name to look up the total order in the model, which is then used to order the elements of sort `node` before printing them.

`mypyvy` supports several other custom printers, including one for printing sorts that represent sets of elements coming from another sort. Users can also define their own printers by defining a printing plugin. `mypyvy`'s proof of the Raft protocol comes with a custom printer for displaying Raft's logs in a readable way.

`mypyvy` also supports a handful of other annotations. `@no_print` instructs `mypyvy` not

to print a sort, relation, constant, or function at all, which can be useful either because a custom printer for another symbol also handles printing this symbol, or temporarily because the model is large and the symbol is irrelevant to the current debugging session. `@no_minimize` is used to instruct `mypyvy`'s model minimizer not to minimize elements of a certain sort or relation.

The annotation framework is extensible, and we expect other uses of annotations will come up in the future. We would like to thank Daniel Ricketts for his contributions to this aspect of `mypyvy`.

4.7 Using `mypyvy`

We have used `mypyvy` as a platform to support several research projects in the general area of automated verification and invariant inference.

Another proof of Raft. In Section 2.7, we discussed a Coq proof of the safety of Raft consensus protocol. That effort involved over 50000 lines of proof for about 500 lines of code. Using the automation of `mypyvy`, and based on ideas from our collaborators on how to encode the proofs of Paxos and MultiPaxos/Raft in first-order logic (in fact, in EPR) [88, 108], we were able to produce another proof of Raft's safety in `mypyvy`. The `mypyvy` proof is under 500 lines total, including both the model of the protocol and its invariants.[5] The primary difficulty that remains in the proof is just to *state* the inductive invariant that implies safety. We hope to continue to improve the state of the art here through work on invariant inference, which would allow the proof to get even shorter and easier.

There are some important differences and tradeoffs between the Coq proof of Raft from Chapter 2 and the `mypyvy` proof presented here. First, the Coq proof yielded an executable implementation of the protocol via extraction to OCaml, which we were able to run and benchmark. In contrast, the `mypyvy` proof is merely about a logical model of the system.

[5] `https://github.com/wilcoxjay/mypyvy/blob/5b5cb9f3be8a990ed14837638d541f5eebdfe88f/examples/raft.pyv`

Neither is clearly a better approach in all circumstances, given the massive effort-guarantee tradeoff present. Second, the Coq proof is more *foundational*, meaning that any needed theory is developed from first principles. For example, we developed a theory of linearizability and used it to prove Raft linearizable. In contrast, the `mypyvy` proof *bakes in* many assumptions about linearizability, induction, and safety proofs. Finally, the specifications are somewhat different. The Coq proof proves linearizability, which is a property of the set of traces that make up the behaviors of the system. In contrast, `mypyvy` "only" proves what the Raft paper calls "State Machine Safety". This safety property is still a deep result about the protocol, but it is somewhat simpler than the trace reasoning required to prove linearizability. Overall, the proofs make different tradeoffs in the effort-results space, and we believe both are valuable. Speaking anecdotally, we learned much from both efforts. The fact that the Coq proof yielded an implementation meant that we were forced to consider practical concerns of efficiency, ensuring we didn't cut too many corners. On the other hand, the `mypyvy` proof gave us higher-level insight into "why" the protocol works, because the edit-verify-debug cycle was so much tighter, we were able to optimize our proof and really understand its guts.

Discussion of the small-model property. We encoded our proof of Raft in the EPR fragment of first-order logic (see Section 4.4). Since EPR satisfies the small model property (Lemma 4.4.3), whenever there is a counterexample to safety, there must be a counterexample of some bounded size, where the bound is a function of the syntax of the safety formula. In the concrete case of Raft, where we have uninterpreted sorts for nodes, terms, values, logs, etc., this means that all counterexamples are guaranteed to have a bounded number of nodes, although the naïve bound is quite bad (roughly doubly exponential in the size of the safety formula). Thus, the small model property of EPR hints at two interesting benefits for verification in practice. First, it is the crucial step in proving decidability, and often for SMT solvers, decidability is a decent proxy for tractability. Having tractable automated solving of verification queries makes the user experience much nicer, since users have a clear job: encode

their queries in EPR. If they do this, they can typically expect reasonable performance from the solvers. The second benefit of the small model property is that it is a sort of formalization of the "small world hypothesis," which is a heuristic often discussed in the context of bounded model checking. The idea is that if a protocol is wrong, then we expect there to be a relatively small example demonstrating the bug. The small world hypothesis is an empirical pattern of the kinds of systems that humans design—they aren't pathological. On the other hand, the small model property says that the small world hypothesis is just *true* for EPR formulas. In other words, it lifts the small world hypothesis from an empirical pattern to a theorem. Returning to the proof of Raft, the amazing thing about the small model property is that it guarantees safety *for any number of nodes*, by showing that it suffices to check up to a bounded number of nodes.

Phase Invariants. Several projects have used `mypyvy`'s implementation of PDR$^\forall$ as a baseline which can be extended and compared against in the hopes of improving invariant inference. In one such project, we extended PDR$^\forall$ with the idea of a *phase automaton*, a finite state machine whose states are labeled with invariants and whose edges are synchronized with transitions of the underlying transition system [25]. Such an automaton captures the "phase structure" of the protocol, e.g., first elect a leader, then commit some log entries. A labeled phase automaton is *inductive* if, roughly speaking, the transitions on all edges in the automaton preserve the invariants on their incident nodes. An *unlabeled* or partially (noninductively) phase automaton poses a generalization of the invariant inference problem, where the goal is to infer a strengthening of the labels to make the automaton inductive. This label inference problem can be solved by a generalization of PDR$^\forall$ that simultaneously solves for the invariants of each node, passing proof obligations between them. By providing a partial phase automaton, the user can help make the invariant inference problem easier, thus improving performance.

First-order Quantified Separators. Another project which uses `mypyvy` as a platform to build improved invariant inference techniques, but which the author was not directly involved in, is Koenig et al.'s FOL-IC3 algorithm [53]. PDR$^\forall$ extends the propositional IC3/PDR algorithms to work on universal formulas. Similarly, FOL-IC3 extends the same propositional algorithms to work on arbitrary quantified formulas. The key insight of this work is to reduce the invariant inference problem to a *separation* problem, which asks for a (quantified) first-order formula that can distinguish two given sets of example structures.

In addition to the three projects mentioned above, we have several ongoing pieces of work that use `mypyvy` as a platform. So far, it seems to be working out well!

4.8 Related Work

`mypyvy` is directly inspired by IVy [89].[6] The IVy tool supports specification, implementation, and verification of systems, including distributed and concurrent systems. Systems are expressed as a set of *actions*, each written in a simple imperative programming language over state variables from a first-order vocabulary. The verification queries IVy asks of the underlying solver are carefully designed to land in a decidable fragment of logic, increasing the efficiency, reliability, and predictability of the verification. When verification fails, concrete counterexample traces are shown to the user demonstrating the violation. IVy also has a powerful module system that supports so-called "circular" assume-guarantee reasoning, where all modules get to assume all other modules' invariants, but are under the obligation to show that they do not violate their own invariants. (This reasoning is not actually circular, but sound, because "nobody violates their invariants first" implies "nobody violates their invariants".)

One can view `mypyvy` as similar to a hypothetical intermediate language in IVy's pipeline to the solver. IVy compiles the modular imperative program into a set of purely logical transition systems, each of which must be verified. One could imagine making this connection

[6] IVy's code is available on Github at `https://github.com/kenmcmil/ivy`. See also the IVy website at `http://microsoft.github.io/ivy/`.

explicit, by using `mypyvy` as a "backend" for IVy, translating the transition systems into `mypyvy` syntax. This would have the advantage of making `mypyvy`'s invariant inference algorithms available to IVy programs. Indeed, many of the examples and benchmarks used by `mypyvy` were manually translated from IVy. We have begun work on such a translator, and hope to continue to work more closely with IVy in the future.

Dafny is a programming language designed from the ground up for verification [62]. Dafny is built on top of the Boogie intermediate verification language [63], which itself uses the Z3 SMT solver [19]. Dafny has an imperative object-oriented programming language with objects, statements, loops, arrays, and heap-allocated data structures, and has a rich Hoare logic for reasoning about these programs. It also has a purely functional expression-oriented programming language with first-class and higher-order functions, recursion, lists and logical quantifiers. These logical features can be used express the specification of the imperative code, or they can be used by themselves, turning Dafny into more of a proof assistant than a programming language. Dafny enjoys a high degree of proof automation, since all obligations are eventually sent to Z3. However, these queries can have complex quantifier structure to them, meaning that they typically do not fall into any decidable fragment of first-order logic. Dafny instructs Z3 to use syntactic heuristics based on E-matching to manage quantifier instantiation process [21, 76]. This approach achieves good performance in practice, but it means that the solver cannot return counterexamples, and that the user must have a basic understanding of E-matching in order to be an expert user of quantifiers in Dafny.

Previous chapters of this thesis used the Coq proof assistant [17]. Coq is an interactive theorem prover and purely functional programming language based on dependent type theory. Its design gives it essentially limitless expressiveness, but this comes at the cost of manual proof effort. Coq is the perfect tool for the job of building a new logic, like we did in Chapter 3 with Disel. Also, Coq has support for building domain-specific proof automation, as promoted by Chlipala [12], so with careful design, one can greatly reduce manual proof effort. `mypyvy` places itself at a very different point in the design space, constraining what

the programmer can write to a first-order transition system, and in return, giving essentially full automation. My own journey as a researcher has taken me through many points on this spectrum. I have seen the benefits of being able to transliterate your mathematical theorem statements directly into a Coq proposition, but I have also seen the beauty of not having to write any proofs. In any particular domain, it often makes sense for the community to begin by building tools in very expressive frameworks, because we don't yet know what we will need. After this initial step, researchers can start the process of figuring out exactly what needs to be expressed, and what can be traded away in exchange for better automation.

TLA$^+$ is a specification language for modeling systems developed by Lamport [56, 59]. It comes with a suite of tools to analyze models, including tools for model checking and deductive theorem proving. Lamport advocates for a style of using TLA$^+$ where most of the benefit of the process is gained just from formally expressing the model of the system, because the user is forced to carefully think through the details. Users can derive additional benefit by model checking their systems, using the TLC bounded model checker [122]. TLC is an explicit-state model checker that exhaustively explores a finite version of the system (e.g., all Raft executions where there at most 5 nodes, at most 2 commands, and at most 3 terms, etc.). Users can go even further by using the TLA$^+$ proof system (TLAPS) to prove their systems correct [11]. TLAPS uses a hierarchical (treelike) proof structure, where the leaves of the tree can be dispatched by automated solvers. The most interesting point of comparison for `mypyvy` is TLA$^+$'s language for specifying systems. TLA$^+$ is much more expressive than `mypyvy`, allowing users to write arbitrary temporal logic formulas to describe their system. While sometimes needed, this expressiveness makes analysis and proof more challenging, and users often stay within an idiomatic subset that constructs a transition relation as the finite disjunction of parameterized actions. One way to view `mypyvy` is as a codification of this idiom into a language. By restricting the way users write their system specifications, `mypyvy` is able to completely automatically analyze the safety problem for these systems. On the other hand, `mypyvy` does not currently support liveness reasoning, so users of TLA$^+$ would miss having access to that kind of reasoning. We plan to investigate

liveness in future work, encoding the queries in first-order logic following the approach of Padon et al. [87].

nuSMV and nuXmv are symbolic model checkers, originally based on BDDs and SAT solvers, and later adopted to infinite-domain systems by using SMT solvers [9, 14, 15]. These tools are "workbenches" that implement many different techniques to attack the model checking problem. This is similar in spirit to `mypyvy`'s goal of providing a framework to implement several approaches. One key difference between nuXmv and mypyvy is that nuXmv's support for infinite-state systems is based on integer and real number types, while `mypyvy`'s primary way to reason about such systems is based on pure first-order uninterpreted sorts. In our experience, most distributed systems do not need specific interpreted operations over numbers, and instead are naturally expressed over uninterpreted sorts using axioms for, e.g. a total order.

Historically, many model checkers intended for hardware have worked on the bit level, i.e., treating each bit as a separate boolean variable. This encoding allows the use of off-the-shelf SAT solvers, but loses mujch of the high-level structure present in the original problem. Recently, the community has developed several *word-level* techniques, that maintain this high-level information during solving. AVR is one such word-level symbolic model checker that has shown promise in the hardware model checking community [32]. At its core, AVR uses syntax-guided abstraction to compute a word-level model of the system in first-order logic, which can then be analyzed with an implementation of IC3 on top of several SMT backend solvers to infer an inductive invariant [7]. Most closely related to `mypyvy` is the subsequent tool I4 [74], which builds on top of AVR's ability to efficiently verify finite-domain systems to analyze infinite-state systems such as distributed protocols. I4 works by constructing finite instances of the protocol, analyzing them with AVR to get an inductive invariant for the finite protocol, and then generalizing this invariant to get a candidate invariant for the original protocol. This candidate must then be verified using unbounded techniques, such as IVy [89], and if a counterexample is obtained, a larger finite instance of the protocol must be analyzed. I4 reads protocols written in the IVy input language.

Verification modulo theories (VMT) [83, 113] is an extension of the SMT-LIB standard [103] to support reasoning about symbolic transition systems. Transition systems are defined by annotating a standard SMT-LIB function definition with a special keyword marking it as the definition of the initial conditions or transition relation. Since transition relations are 2-state formulas, as discussed in Section 4.5, VMT again uses special keywords to declare that a SMT-LIB variable is the "next" variable corresponding to another variable. Both safety properties (of the form $\Box\varphi$) and liveness properties (of the form $\Diamond\Box\varphi$) can also be specified in the VMT description of the transition system, but other queries (such as bounded reachability queries or `mypyvy`-style "trace" queries) cannot be specified.

`mypyvy`'s k-state semantics for formulas is related to interpretations of Linear Temporal Logic (LTL) over finite traces [31]. A key difference in the `mypyvy` setting is that each state is a first-order structure, rather than a propositional truth assignment. It would be interesting to extend `mypyvy` with explicit temporal operators that are "unrolled" when translating to the solver.

Btor2 is a language for specifying word-level hardware model checking problems over bitvectors and arrays [82]. It is an extension of the well-known bit-level format AIGER [5], which is used in the hardware model checking competition [37]. Both AIGER and Btor2 are tailored to the case of reasoning about finite-state systems, especially those derived from hardware designs. Thus, neither supports infinite or unbounded sorts, which is the focus of `mypyvy`.

4.9 Conclusion

In this chapter, we described `mypyvy`, a language and tool for analyzing symbolic transition systems expressed in first-order logic. `mypyvy` has already outgrown its original purpose and has had a surprisingly successful beginning to its life. There are several directions for future work, both internally facing and external.

Future Work. On the internals side, there are conceptual design choices that should be revisited in the interests of cleanliness. For example, `mypyvy` currently separately represents constants, relations, and functions as distinct kinds of objects, but it would be cleaner to represent everything as a function. A similar kind of conceptual simplification seems possible with queries, since essentially every query (except for invariant inference queries) can be phrased as asking a satisfiability question about a k-state formula. It would be nice to capture this insight into a more general query language that unifies inductiveness checking, background theory theorem proving, bounded model checking, and trace queries. Another minor internal nit is that `mypyvy` currently assumes that there is one global transition system under analysis. This assumption makes sense in the context of a command line tool that is asked a simple query about a single file. But in more sophisticated cases, the assumption breaks down. For example, any kind of program transformation of the transition system itself causes there to be two transition systems in scope, which is currently difficult to work with, and we end up hacking around it. It would be nice to remove this limitation.

Another aspect of `mypyvy`'s design worth revisiting is its interface to the underlying SMT solver. `mypyvy` was originally designed to use Z3's Python API, which was initially convenient and allowed rapid prototyping, but this has the downside that we cannot easily use other SMT solvers. We have actually hacked in support for CVC4, but the mechanism for doing so it very unsavory: we use the Z3 API to build the query and then ask Z3 to print it in SMT-LIB format, which we then ship to CVC4. A better solution would be to introduce some kind of mid-level solver interface, probably based on SMT-LIB, which would allow us to use multiple solvers easily. An important design constraint here is that we don't want to give up the performance benefits of using the API for expensive queries such as $\text{PDR}^\forall$, which builds many ASTs with sharing efficiently using the API.

On the external side, there are endless possibilities for tools to build using `mypyvy`. For example, we have several ongoing projects related to invariant inference and improving $\text{PDR}^\forall$. More generally, we are interested in building an "IDE for transition systems," which is able to give real-time feedback on the model as the user develops it. Relatedly, we'd like

to make invariant inference more interactive. Currently, the execution model of PDR$^{\forall}$ is a batch processing job, and a rather expensive one at that. When PDR$^{\forall}$ fails to find an invariant, it often seems to loop forever, giving no intermediate results or feedback to the user. Instead, it would be nice to have an approach that could be stopped midway through inference and have its state inspected by the user, who could then guide the search.

Besides our own interests, there are also many other possible tools to build on top of `mypyvy`. We encourage readers to join us in the quest for user-friendly symbolic reasoning about transitions that is fast and works on interesting and practical examples. These are hard problems, and there is much work to do. We need your help!

Chapter 5

CONCLUSION

Chapter 2 and Chapter 3 presented techniques for decomposing proofs of distributed systems, both along the "vertical axis" of fault models and fault tolerance mechanisms, and along the "horizontal axis" of separating cooperating protocols. Once suitably decomposed, the verification of the high-level logic of each protocol, executing in a relatively benign fault model, can be automated (or at least partially automated) using a tool such as `mypyvy`, presented in Chapter 4. We have demonstrated that this decompose-and-automate workflow can be applied to distributed systems of roughly the complexity that we would ask an undergraduate to implement (but not verify!) in a senior level course.

Almost two decades ago, Lamport [55] propounded the thesis **Composition: a way to make proofs harder**, favoring mathematical models over implementations for real system verification, while eschewing composition: "*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*". Lamport was certainly right in the sense that in the intervening 20+ years, it has not become common for engineers to reason about compositional specifications. That said, the results of this dissertation indicate that a compositional approach is key to reasoning about systems of any realistic complexity, especially when faced with verifying implementations, not just models. With the compositional, implementation-producing tools and techniques presented here, it is now possible to begin to prove Lamport wrong about the next 15 years.

We are optimistic about the role of formal methods in supporting sophisticated system building, but for the time being, these methods appear inaccessible to those without graduate

education in verification.

One of our primary goals for future work is to make these methods more accessible. For the compositional aspects of our work, we believe programmers are already well equipped to understand mechanisms for modularity, as these are familiar from similar mechanisms in mainstream languages. For the automated aspects, we believe there is a much more significant gap in creating accessible tools, which boils down to a philosophical difficulty. Many automated tools attempt to be *complexity-hiding*, in the sense that they try to solve the entire problem automatically in all cases, but lack good support for the failure case where the problem can't be solved. When these tools do fail, they typically do so completely opaquely, with debugging output only useful to those who implemented the tool. Instead, we believe that the correct approach to building automated tools is to *embrace complexity*, to use a phrase coined by Jean Yang.[1] Such tools start by acknowledging that not all instances of the problem will be solved automatically, and they are designed to give the programmer sufficient information to understand the failure and recourse to address it. This requires programmers to have a basic mental model for how the tool works, what kinds of problems it can solve, and how to respond to its failures by adjusting the input verification query.

For example, our implementation of PDR$^\forall$ in `mypyvy` is currently best described as hiding complexity. Either it finds an inductive invariant, or it doesn't. And when it doesn't, the debugging information produced is difficult to understand and turn into a concrete remedy, and there is no indication if a partial inductive invariant was found. But in fact, the PDR$^\forall$ algorithm is well positioned for building a complexity embracing tool. Any inductive facts discovered could be reported to the user, even though they don't fully verify the safety property. And there is especially nice theory about PDR$^\forall$ that tells us that its failures can often be explained in terms of an abstract trace of the system that demonstrates the failure explicitly. We are excited to build such a complexity-embracing version of PDR$^\forall$, and to explore other invariant inference techniques from this point of view as well.

---

[1] See `https://twitter.com/jeanqasaur/status/1389645922183696384`.

BIBLIOGRAPHY

[1] Martn Abadi and Leslie Lamport. The Existence of Refinement Mappings. In: *LICS*. IEEE Computer Society, 1988, pp. 165–175.

[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal Memory: Definitions, Implementation, and Programming. In: *Distributed Computing* 9.1 (1995), pp. 37–49.

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

[4] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN: 9781107256552.

[5] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2011.

[6] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with Logic: HOL Specification and Symbolic-Evaluation Testing for TCP Implementations. In: *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*. Charleston, SC, Jan. 2006, pp. 55–66.

[7] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Vol. 6538. Lecture Notes in Computer Science. Springer, 2011, pp. 70–87.

[8] Stephen Brookes. A semantics for concurrent separation logic. In: *Th. Comp. Sci.* 375.1-3 (2007).

[9] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 334–342.

[10] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In: *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. Portland, OR, Aug. 2007, pp. 398–407.

[11] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. The TLA$^+$ Proof System: Building a Heterogeneous Verification Platform. In: *Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC)*. Ed. by Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock. Vol. 6255. Lecture Notes in Computer Science. Springer, 2010, p. 44.

[12] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, Dec. 2013.

[13] Adam Chlipala. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. In: *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, June 2011, pp. 234–245.

[14] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NUSMV 2: An opensource tool for symbolic model checking. In: *International Conference on Computer Aided Verification*. Springer. 2002, pp. 359–364.

[15] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Checker. In: *Int. J. Softw. Tools Technol. Transf.* 2.4 (2000), pp. 410–425.

[16] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T Sasaki, and S. F. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice Hall, 1986.

[17] Coq development team. *Coq Reference Manual*. `http://coq.inria.fr/distrib/current/refman/`. INRIA. 2020.

[18] *etcd: A highly-available key value store for shared configuration and service discovery*. `https://github.com/coreos/etcd`. 2014.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In: *TACAS*. 2008, pp. 337–340.

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

[21] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. In: *J. ACM* 52.3 (2005), pp. 365–473.

[22] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In: *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*. Maribor, Slovenia, 2010, pp. 504–528.

[23] Cezara Drgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, Jan. 2016, pp. 400–415.

[24] Tzilla Elrad and Nissim Francez. Decomposition of Distributed Programs into Communication-Closed Layers. In: *Sci. Comput. Program.* 2.3 (1982), pp. 155–173.

[25] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring Inductive Invariants from Phase Structures. In: *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 405–425.

[26] Xinyu Feng. Local Rely-Guarantee Reasoning. In: *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*. Savannah, GA, Jan. 2009, pp. 315–327.

[27] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In: *Proceedings of the 16th European Symposium on Programming (ESOP)*. Braga, Portugal, 2007, pp. 173–188.

[28] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. In: *J. ACM* 32.2 (1985).

[29] Stephen J. Garland and Nancy Lynch. Using I/O Automata for Developing Distributed Systems. In: *Foundations of Component-based Systems*. Ed. by Gary T. Leavens and Murali Sitaraman. Cambridge University Press, 2000.

[30] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In: *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*. Cambridge, MA, Apr. 2007, pp. 285–298.

[31] Giuseppe De Giacomo and Moshe Y. Vardi. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. Ed. by Francesca Rossi. IJCAI/AAAI, 2013, pp. 854–860.

[32] Aman Goel and Karem A. Sakallah. AVR: Abstractly Verifying Reachability. In: *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by Armin Biere and David Parker. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 413–422.

[33] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Tech. rep. 6455. Microsoft Research – Inria Joint Centre, 2009.

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

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

[36] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-Verified Network Controllers. In: *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, June 2013, pp. 483–494.

[37] *Hardware Model Checking Competition*. `http://fmv.jku.at/hwmcc20/`. 2020.

[38] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015, pp. 1–17.

[39] Mark Hayden. The Ensemble System. PhD thesis. Cornell University, 1998.

[40] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492.

[41] Jason J. Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and Proofs for Ensemble Layers. In: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. York, UK, Mar. 2009, pp. 119–133.

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

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

[44] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Feb. 2012.

[45] Cliff B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. In: *ACM Trans. Program. Lang. Syst.* 5.4 (1983), pp. 596–619.

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

[47] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In: *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, Jan. 2015, pp. 637–650.

[48] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-Directed Inference of Universal Invariants or Proving Their Absence. In: *J. ACM* 64.1 (2017), 7:1–7:33.

[49] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language Support for Building Distributed Systems. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Diego, CA, June 2007, pp. 179–188.

[50] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In: *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*. Cambridge, MA, Apr. 2007, pp. 243–256.

[51] 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. seL4: formal verification of an operating-system kernel. In: *Commun. ACM* 53.6 (2010), pp. 107–115.

[52] Thomas Kleymann. Hoare Logic and Auxiliary Variables. In: *Formal Asp. Comput.* 11.5 (1999), pp. 541–566.

[53] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In: *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, UK, June 2020.

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

[55] Leslie Lamport. Composition: A Way to Make Proofs Harder. In: *Compositionality: The Significant Difference (COMPOS)*. Bad Malente, Germany, 1997, pp. 402–423.

[56] Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, July 2002.

[57] Leslie Lamport. The Implementation of Reliable Distributed Multiprocess Systems. In: *Computer Networks* 2 (1978), pp. 95–114.

[58] Leslie Lamport. The Part-Time Parliament. In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169.

[59] Leslie Lamport. The Temporal Logic of Actions. In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923.

[60] Leslie Lamport. *Thinking for Programmers*. http://channel9.msdn.com/Events/Build/2014/3-642. Apr. 2014.

[61] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. In: *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, UK, June 2014, pp. 216–226.

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

[63] K. Rustan M. Leino. *This is Boogie 2*. 2008.

[64] Xavier Leroy. Formal verification of a realistic compiler. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115.

[65] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, Jan. 2016, pp. 357–370.

[66] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In: *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*. Rome, Italy, Jan. 2013, pp. 561–574.

[67] Leonid Libkin. *Elements of finite model theory*. Springer, 2013.

[68] Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. In: *Commun. ACM* 18.12 (1975), pp. 717–721.

[69] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert L. Constable. Building Reliable, High-Performance Communication Systems from Components. In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*. Kiawah Island, SC, Dec. 1999, pp. 80–92.

[70] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^{3}S: Debugging Deployed Distributed Systems. In: *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, CA, Apr. 2008, pp. 423–437.

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

[72] Nancy A. Lynch. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN: 1558603484.

[73] Nancy A. Lynch and Frits W. Vaandrager. Forward and Backward Simulations: I. Untimed Systems. In: *Inf. Comput.* 121.2 (1995), pp. 214–233.

[74] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2019, pp. 370–384.

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

[76] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT Solvers. In: *Proceedings of the 21st International Conference on Automated Deduction (CADE)*. Vol. 4603. Lecture Notes in Computer Science. Springer, 2007, pp. 183–198.

[77] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In: *Proceedings of the 23rd European Symposium on Programming (ESOP)*. Grenoble, France, 2014, pp. 290–310.

[78] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare Type Theory. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Portland, OR, Sept. 2006, pp. 62–73.

[79] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent Types for Imperative Programs. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Victoria, British Columbia, Canada, Sept. 2008.

[80] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In: *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain, Jan. 2011, pp. 261–274.

[81] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. In: *Commun. ACM* 58.4 (2015), pp. 66–73.

[82] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, BtorMC and Boolector 3.0. In: *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 587–595.

[83] *nuXmv User Manual*. `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`. 2014.

[84] Peter W. O'Hearn. Resources, concurrency, and local reasoning. In: *Th. Comp. Sci.* 375.1-3 (2007), pp. 271–307.

[85] Diego Ongaro. Consensus: Bridging Theory and Practice. PhD thesis. Stanford University, Aug. 2014.

[86] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In: *Proceedings of the 2014 USENIX Annual Technical Conference*. Philadelphia, PA, June 2014, pp. 305–319.

[87] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 26:1–26:33.

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

[89] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In: *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, June 2016, pp. 614–630.

[90] James L. Peterson. Petri Nets. In: *ACM Computing Surveys* (Sept. 1977), pp. 223–252.

[91] Azalea Raad, Jules Villard, and Philippa Gardner. CoLoSL: Concurrent Local Subjective Logic. In: *Proceedings of the 24th European Symposium on Programming (ESOP)*. London, UK, 2015.

[92] Vincent Rahli. Interfacing with Proof Assistants for Domain Specific Programming Using EventML. In: *Proceedings of the 10th International Workshop On User Interfaces for Theorem Provers*. Bremen, Germany, July 2012.

[93] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. In: *Proceedings of the 15th International Workshop on Automated Verification of Critical Systems (AVoCS)*. Edinburgh, UK, Sept. 2015.

[94] Robbert van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. In: *ACM Comput. Surv.* 47.3 (2015), 42:1–42:36.

[95] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In: *Proceedings of the 17th Symposium on Logic in Computer Science (LICS)*. Copenhagen, Denmark, July 2002, pp. 55–74.

[96] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating Formal Proofs for Reactive Systems. In: *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, UK, June 2014, pp. 452–462.

[97] Thomas Ridge. Verifying Distributed Systems: The Operational Approach. In: *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*. Savannah, GA, Jan. 2009, pp. 429–440.

[98] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. New York, NY, USA: Cambridge University Press, 2001. ISBN: 0521781779.

[99] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. Developing Correctly Replicated Databases Using Formal Tools. In: *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Atlanta, GA, June 2014, pp. 395–406.

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

[101] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized Verification of Fine-grained Concurrent Programs. In: *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, June 2015, pp. 77–87.

[102] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 28:1–28:30.

[103] *SMT-LIB standard*. `http://smtlib.cs.uiowa.edu/language.shtml`. 2016.

[104] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In: *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, Jan. 2015, pp. 275–287.

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

[106] Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In: *Proceedings of the 23rd European Symposium on Programming (ESOP)*. Grenoble, France, 2014, pp. 149–168.

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

[108] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In: *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Philadelphia, PA, June 2018.

[109] Bernardo Toninho, Lus Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In: *PPDP*. ACM, 2011, pp. 161–172.

[110] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In: *Proceedings of the 2014 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR, Oct. 2014, pp. 691–707.

[111] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In: *Proceedings of the 40th*

ACM Symposium on Principles of Programming Languages (POPL). Rome, Italy, Jan. 2013, pp. 343–356.

[112] Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In: *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR)*. Lisbon, Portugal, 2007, pp. 256–271.

[113] *Verification Modulo Theories*. `http://www.vmt-lib.org/`. 2014.

[114] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving Copyless Message Passing. In: *APLAS*. Vol. 5904. LNCS. Springer, 2009, pp. 194–209.

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

[116] James R. Wilcox, Ilya Sergey, and Zachary Tatlock. Programming Language Abstractions for Modularly Verified Distributed Systems. In: *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*. Asilomar, CA, 2017, 19:1–19:12.

[117] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In: *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, June 2015, pp. 357–368.

[118] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for change in a formal verification of the Raft Consensus Protocol. In: *Proceedings of the 2016 International Conference on Certified Programs and Proofs (CPP)*. Saint Petersburg, FL, Jan. 2016, pp. 154–165.

[119] Maysam Yabandeh, Nikola Kneevi, Dejan Kosti, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In: *Proceed-*

ings of the 5th Symposium on Networked Systems Design and Implementation (NSDI). San Francisco, CA, Apr. 2008, pp. 229–244.

[120] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In: *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, Apr. 2009, pp. 213–228.

[121] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In: *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, June 2011, pp. 283–294.

[122] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In: *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*. Ed. by Laurence Pierre and Thomas Kropf. Vol. 1703. Lecture Notes in Computer Science. Springer, 1999, pp. 54–66.

[123] Pamela Zave. Using Lightweight Modeling To Understand Chord. In: *ACM SIGCOMM Computer Communication Review* 42.2 (Apr. 2012), pp. 49–57.