# CoqIOA: A Formalization of IO Automata in the Coq Proof Assistant

by

Anish Athalye

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
M. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

# CoqIOA: A Formalization of IO Automata in the Coq Proof Assistant

by

Anish Athalye

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Implementing distributed systems correctly is difficult. Designing correct distributed systems protocols is challenging because designs must account for concurrent operation and handle network and machine failures. Implementing these protocols is challenging as well: it is difficult to avoid subtle bugs in implementations of complex protocols. Formal verification is a promising approach to ensuring distributed systems are free of bugs, but verification is challenging and time-consuming. Unfortunately, current approaches to mechanically verifying distributed systems in proof assistants using deductive verification do not allow for modular reasoning, which could greatly reduce the effort required to implement verified distributed systems by enabling reuse of code and proofs.

This thesis presents CoqIOA, a framework for reasoning about distributed systems in a compositional way. CoqIOA builds on the theory of input/output automata to support specification, proof, and composition of systems within the proof assistant. The framework's implementation of the theory of IO automata, including refinement, simulation relations, and composition, are all machine-checked in the Coq proof assistant. An evaluation of CoqIOA demonstrates that the framework enables compositional reasoning about distributed systems within the proof assistant.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

# Acknowledgments

I am deeply grateful to Frans Kaashoek and Nickolai Zeldovich, who I have been lucky to know since the end of my high school years, for being my mentors and helping me explore my research interests. I would not have been pursuing academic research were it not for their support and the welcoming environment of the Parallel and Distributed Operating Systems group.

I also owe thanks to Frans and Nickolai for their endless guidance, technical insights, and encouragement on the matter of this thesis.

Finally, I thank my parents and my brother, who provide unconditional love and support in everything I do.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Implementing distributed systems correctly is difficult. Designing protocols for distributed systems is challenging because designs must account for concurrent operation and handle network and machine failures. Furthermore, because distributed systems protocols are complicated, it is difficult to avoid subtle bugs in implementations of these protocols.

Production systems under wide use have had subtle correctness bugs. For example, testing has revealed correctness bugs in releases of popular systems such as Cassandra, Consul, ElasticSearch, etcd, Kafka, MongoDB, and others [1].

## 1.1 Implementing correct systems

Unfortunately, tracking down correctness bugs in distributed systems through testing is time-consuming, and furthermore, it is incomplete. No amount of software testing is enough to provably eliminate bugs. Testing is especially hard with distributed systems, where concurrency makes it difficult to catch bugs in tests, even when using fault injection frameworks to simulate network and machine failures. Formal methods provide a much more rigorous means of building high-assurance systems.

Bounded model checking is one approach to verifying designs: given a model of a system, model checkers can exhaustively verify that certain desired properties hold. Systems designers have used model checking to mechanically verify fundamental distributed systems protocols such as the Paxos consensus protocol [2, 3]. A lightweight means of checking correctness, model checking has also seen success outside of academia. For example, Amazon has successfully used model checking to find bugs and verify optimizations in production systems [4].

Approaches using modeling tools are limited due to the limited power of bounded model checking, failing to scale to systems with large or infinite state spaces. Furthermore, relying only on model checking leads to a formality gap: even if a design has been proven correct, the implementation could still be buggy. Often, bugs in production implementations are results of a buggy implementation of a protocol that has been proven correct on paper or using model checking [1].

Deductive verification is a complete approach to verifying both the design and implementation of a distributed system. It provides a machine-checkable proof that code satisfies the specification and is free of bugs. Theory and tools have advanced in recent years, and researchers have succeeded in building provably correct implementations of realistic distributed systems such as Raft [5] on top of the Verdi framework [6] and a replicated state machine and sharded key-value store using the IronFleet methodology [7].

## 1.2 Problem and goal

Prior work in verifying realistic distributed systems represents impressive engineering effort. Unfortunately, there is no straightforward way to reuse this work in building new verified systems, because prior work is not designed for compositional reasoning.

We define compositional reasoning as follows. With an approach to verification that supports compositional reasoning, we should be able to build an implementation on top of specifications of underlying components, prove the system correct with respect to the specifications, and then replace the underlying specifications with their implementations to produce a final system such that we preserve correctness. For example, consider the toy system in Figure 1-1: there is a booking agent, a hotel service, and an airline service, and the booking agent acts as a transaction coordinator

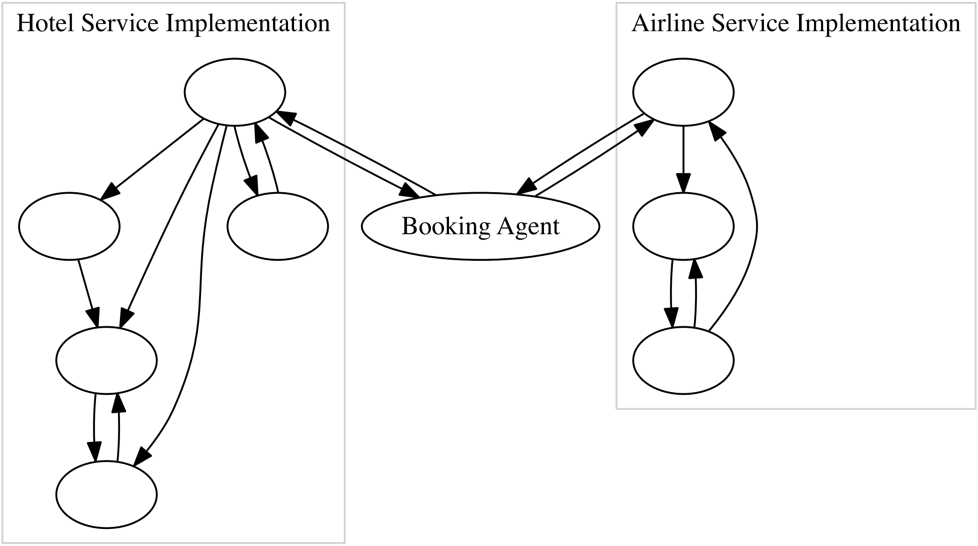

Figure 1-1: An example of a system with logically separate components: a booking agent communicating with a hotel service implementation and an airline service implementation. The booking agent acts as a transaction coordinator to atomically book tickets for both services.

to atomically buy tickets from both the hotel service and the airline service. In the example, both the hotel service and airline service are implemented by complex distributed systems. To build and verify the overall system in a compositional way, we would first write specifications for the hotel and airline services, and we would prove that the implementations satisfy the specifications. Then, we would implement and verify the booking agent with respect to the service specifications, as in Figure 1-2. Finally, we would replace the service specifications with their implementations to produce the final verified system while preserving correctness.

We need compositional reasoning to make verified distributed systems practical, because compositional reasoning enables us to structure code and proofs to manage complexity and reduce programming effort. With regular non-verified distributed systems, programmers organize code into modules, and programmers often make use of external libraries implementing lower-level protocols. For example, CockroachDB,

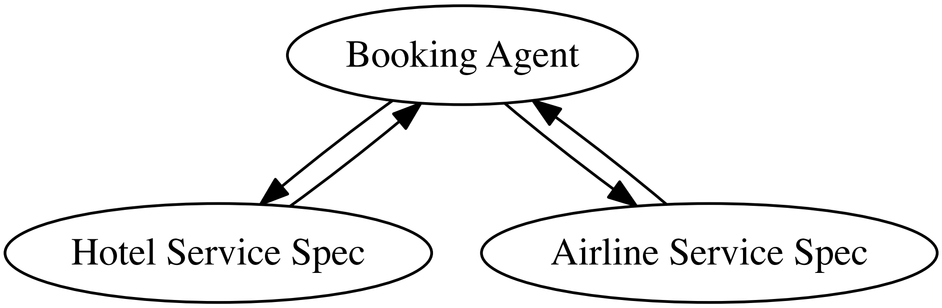

Figure 1-2: The booking agent system building on specifications of the hotel service and the airline service, demonstrating an implementation depending on multiple specifications.

Dgraph, TiKV, and etcd all use CoreOS's well-built and thoroughly tested Raft implementation [8, 9] for distributed consensus. In a similar manner, we need to be able to build up a library of verified components to decrease the burden of building verified distributed systems.

Our goal is to develop a system that enables building reusable modules that are independently verifiable in such a way that reasoning about layering and composition of modules reuses proofs of correctness of individual components. Analogous to the previous real-world example, we would want to enable building a verified Raft library such that we can verify an implementation of a distributed key-value store using only the specification of the Raft library, but still have an end-to-end correctness guarantee for the implementation of the system using the Raft implementation. In the general case, we want to be able to build higher-level components of distributed systems relying only on specifications of lower-level components while producing end-to-end correctness guarantees for system implementations.

Prior work does not support this kind of compositional reasoning. The IronFleet methodology [7] of layered refinement allows for building monolithic distributed systems. The approach does not support composition. Verdi [6] supports a limited form of vertical composition through verified system transformers, but Verdi does not have a way for a system to build on top of multiple dependencies. We describe prior work in more detail in Chapter 2.

## 1.3 Approach

We base our approach on the theory of input/output automata [10], a formal model for reasoning about asynchronous concurrent systems, to reason about distributed systems. The IO automata model is a good theory for reasoning about distributed systems because it enables compositional reasoning.

An IO automaton models a component in a distributed system as a state machine equipped with a transition relation, where transitions are associated with named actions. Actions are classified into one of three types: input, output, or internal. Automata communicate through input and output actions: in a composition, automata responding to the same named action step together synchronously.

Systems are specified as automata, with the behavior of an automaton defined as the set of externally visible execution traces. Implementations are shown to refine specifications by proving that the behavior of the implementation is a subset of the behavior of the specification. Automata in compositions can be substituted with others that refine the original automata, so that the resultant composition refines the original composition. This works even with multiple dependencies as in Figure 1-2: an implementation can build on multiple specifications, and each specification can be swapped for its implementation while preserving correctness. This is what enables compositional reasoning.

We formalize a theory based on IO automata in a proof assistant to enable machine-checked formal reasoning about distributed systems in a compositional way.

## 1.4 Challenges

Prior work has explored reasoning about IO automata within a proof assistant, but the approaches do not support proper compositional reasoning. Work by Bogdanov, which formalizes IO automata in the Larch Prover [11], does not support automata composition at all. Work by Lim implements a translation from a timed IO automata specification language to the PVS prover [12], but it handles compositions in the specification language by recursively inlining the composed automata into a single automaton in PVS, precluding reasoning about individual automata separately from the composition. Work by Nipkow and Slind formalizes IO automata in Isabelle/HOL and supports composition, but it requires that the programmer declare the entire set of possible actions used in a development ahead of time, making compositional reasoning impossible without deciding a priori all the automata that will be composed.

Our main challenge was formalizing IO automata within a proof assistant in a way such that we could perform composition within the proof assistant itself, so that we could separately reason about individual automata within a composition. With our formalization, we can prove high-level theorems about composition such as "if automaton $A'$ refines automaton $A$, then the composition of $A'$ with $B$ refines the composition of $A$ with $B$". These theorems are key to compositional reasoning within the proof assistant.

## 1.5 Thesis contributions

The main contribution of this thesis is a methodology for compositional reasoning about distributed systems in a proof assistant as well as CoqIOA, an implementation of this methodology in the Coq proof assistant [14]. Specifically, the contributions of this thesis are as follows:

1. We formalize input/output automata in the Coq proof assistant, supporting specification, proof, and composition within the proof assistant.

2. We provide machine-checked proofs of the theory of IO automata, including

refinement, simulation relations, and composition.

3. We evaluate the effectiveness of our system in enabling compositional reasoning through a case study of a toy system.

Our current implementation has several limitations. While IO automata theory enables reasoning about both safety and liveness, we reason only about safety. Also, we do not extract executable code from our automata implementations in Coq. Neither of these limitations are inherent, and we aim to address them in future work.

## 1.6 Thesis outline

The rest of this thesis is organized as follows. Chapter 2 discusses related work on distributed systems verification and IO automata. Chapters 3 and 4 describe our system design and formalization in the Coq proof assistant. Chapter 5 evaluates our approach through an example implementation of a toy system. Chapter 6 discusses the limitations of our current work and describes future research directions. Chapter 7 concludes.

# Chapter 2

# Related Work

CoqIOA builds on work done in distributed systems verification and input/output automata.

## 2.1 Distributed systems verification

Prior work in distributed systems verification applies paper proofs, model checking, and deductive verification to prove designs and implementations correct. The approaches trade off between ease of verification and level of rigor.

### 2.1.1 Paper proofs

Works describing distributed systems protocols often contain correctness proofs with varying degrees of formality. The original Paxos paper includes a paper proof of correctness [2]. The Raft consensus protocol has a formal specification written in TLA+, along with a paper proof of correctness [15]. Chord includes paper proofs of correctness as well as probabilistic bounds [16].

Paper proofs are a first step to verifying the correctness of protocols, but paper proofs can contain errors, and paper proofs do not verify implementations.

### 2.1.2 Model checking

Model checking is an approach to mechanically verifying designs. Model checkers work by exhaustively enumerating the state space of a model of a system to verify that desired properties of the model always hold. Model checking has been successful in verifying distributed systems protocols [3, 17] and practical systems designs [4].

Model checking has fundamental limitations: exhaustive enumeration does not scale to complicated specifications due to combinatorial explosion, so model checking is unusable for verifying real systems.

### 2.1.3 Deductive verification

Deductive verification uses theorem-proving software to produce machine-checkable proofs of correctness. Writing machine-checkable proofs involves considerable effort compared to using a model checker, but the approach scales to real systems, and deductive verification allows for proving complex properties of systems. In recent years, researchers have succeeded in applying verification to realistic distributed systems.

IronFleet is a methodology for proving practical distributed systems correct [7]. The authors use IronFleet to prove correct a sharded key-value store and IronRSL, a complex Paxos-based replicated state machine library, illustrating that the methodology scales to realistic systems. The approach involves verification in layers: describe a distributed system in a high-level specification, show that it is refined by a distributed protocol, and show that the protocol is refined by an implementation. IronFleet uses TLA-style verification to show a refinement between the protocol layer and the high-level specification, and it uses Hoare logic [18] to show a refinement between the implementation layer and the protocol layer. IronFleet supports building monolithic systems, but it does not support compositional reasoning: the approach does not include a method to build systems on top of already verified implementations. For example, there is no direct way to use the framework to prove a distributed system implementation correct on top of the IronRSL specification.

Verdi is a framework for formally verifying distributed systems [6]. Verdi models multiple forms of network semantics and failure modes of distributed systems, allowing the programmer to design a system for any given model. Verdi introduces the idea of verified system transformers, a mechanism that transforms a system designed for one model to an implementation suitable for another, preserving all properties of the original system. Verified system transformers enable programmers to write and verify an implementation in a simplified model and then transform their implementation to work in a more realistic setting. The authors implement and verify the Raft consensus protocol as a verified system transformer [5], allowing an application programmer to implement and verify a replicated state machine by verifying a state machine for the single-machine case and then using the verified transformer to produce a verified replicated state machine from the single-machine implementation. Verdi offers a more compositional approach than IronFleet. Verified system transformers can be viewed as a limited form of composition, allowing implementations built for a simple network and machine model to have a single dependency on a protocol designed to make systems work with more realistic network and machine semantics. Verdi's composition is not general-purpose, however. For example, Verdi does not support multiple dependencies in a natural way: we cannot use Verdi to verify the system in Figure 1-1 in a compositional way.

## 2.2 Input/output automata

CoqIOA builds on the theory of input/output automata [10], a formal model for reasoning about asynchronous concurrent systems. We discuss the IO automata model in detail in Chapter 3. System designers have used the IO automata model to reason about distributed algorithms and distributed systems protocols [19].

Timed input/output automata (TIOA), a superset of input/output automata, extend the model for timed systems [20]. Lynch et al. have developed the Tempo specification language, a formal language for describing TIOA, along with the Tempo modeling toolkit [21]. The toolkit provides support for syntax checking, simulation, model checking [22], and verification [12].

### 2.2.1 Theorem proving

There are multiple formalizations of input/output automata and timed input/output automata in theorem proving software.

Work by Bogdanov implements a translation tool from an IOA specification language to the Larch prover [11]. The implementation supports reasoning about IO automata, but it does not support composition of automata.

Tempo-PVS implements a similar translation tool, translating timed IO automata specifications from the Tempo language to PVS [12]. The tool does support composition, but it implements composition by recursively inlining composed automata into a single monolithic automaton in the generated PVS code, making compositional reasoning within the theorem prover impossible.

Work by Nipkow and Slind formalizes IO automata in Isabelle/HOL [13]. The approach supports IOA specification and reasoning within the theorem prover, and it also supports composition, but it requires that all automata in a composition share the same set of actions. This requires deciding on a global set of actions a priori, making it impossible to use the system to develop automata in a modular way. It would not be possible to use the approach to develop modular libraries, because library authors do not know the entire set of actions that a client application would want to use.

# Chapter 3

# Design

We formalize input/output automata [10] in the Coq proof assistant, designing the formalization to enable compositional reasoning within the proof assistant.

## 3.1 Input/output automata

An input/output automaton models a component in a distributed system as a state machine equipped with a transition relation, where transitions are associated with named actions. Actions are classified as either external or internal. Automata communicate through external actions: in a composition, automata responding to the same named action step together.

An input/output automaton $A$ has a set of states $\mathit{states}(A)$, a set of external actions $\mathit{ext}(A)$, a set of internal actions $\mathit{int}(A)$, a set of start states $\mathit{start}(A) \subseteq \mathit{states}(A)$, and a transition relation $\mathit{steps}(A) \subseteq \mathit{states}(A) \times (\mathit{int}(A) \cup \mathit{ext}(A)) \times \mathit{states}(A)$.

We encode IO automata in Gallina, Coq's dependently-typed programming language, as shown in Figure 3-1. Automata are records, parameterized by their external action type, containing a state type, internal action type, a set of start states, and a transition relation. This is a straightforward translation of the mathematical definition of IO automata.

As an example, we could model a lossy FIFO channel as an IOA (see figs. 3-2 and 3-

```
Record AutomatonDef (ExternalActionType : Type) :=
  mkAutomatonDef {
      StateType : Type;
      InternalActionType : Type;
      start : StateType → Prop;
      transition : StateType →
                   (InternalActionType + ExternalActionType) →
                   StateType →
                   Prop;
    }.
```

Figure 3-1: Encoding of input/output automata in Coq.

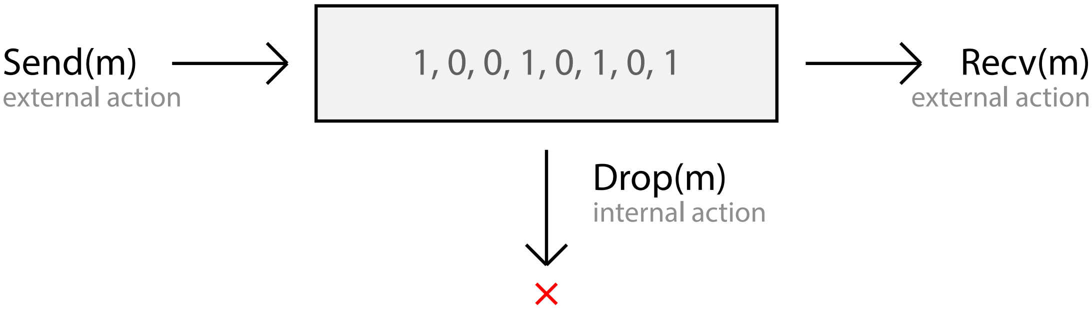

Figure 3-2: A lossy FIFO channel automaton.

3). The state is a list of messages, the external actions are *Send(m)* and *Recv(m)*, and the internal actions are *Drop(m)*. The start state is the empty list. The automata can take the *Send(m)* transition starting from any state, and the result is appending $m$ to the list. The automata can take the *Recv(m)* transition when $m$ is at the head of the list, and the result is removing $m$ from the head of the list. The automata can take the *Drop(m)* transition when $m$ is in the list, and the result is removing $m$ from the list.

CoqIOA's formalization of IO automata deviates slightly from IO automata as defined by Lynch and Tuttle [10]. Currently, we reason about only safety, so for simplicity, we omit liveness-related mechanisms. CoqIOA does not enforce automata to be input-enabled. In addition, we do not distinguish between input and output

```
Variable T : Type.

Inductive ChannelAPI :=
| Send (m : T)
| Recv (m : T).

Inductive Internal :=
| Drop (m : T).

Definition LossyFifo : AutomatonDef ChannelAPI :=
  mkAutomatonDef
    _
    (list T) (* state: list of messages *)
    Internal
    (fun st ⇒ st = []) (* start state: empty queue *)
    (* transition relation: *)
    (fun st act st' ⇒
      match act with
      | inl (Drop m) ⇒ contains m st ∧ removed m st st'
      | inr (Send m) ⇒ st' = st ++ [m]
      | inr (Recv m) ⇒ st = m :: st'
      end).
```

Figure 3-3: Coq implementation of a lossy FIFO channel automaton. The implementation illustrates defining external and internal actions and then defining an automaton, specifying the state type, start state, and transition relation.

actions at all: we only distinguish between external and internal actions. We also omit mechanisms used to reason about fairness.

## 3.2 Execution

Now that we have a definition of automata, we can describe what it means for an automaton to "run". We define an *execution fragment* of an automaton $A$ as a sequence $[s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi_n, s_n]$ of alternating states $s_i$ and actions $\pi_i$, where $(s_{i-1}, \pi_i, s_i) \in \mathit{steps}(A)$. We define an *execution* of an automaton $A$ as an execution fragment of $A$ where the first state is a start state of $A$. We define a *trace* corresponding to an execution fragment as the sequence with the states and internal actions removed, i.e. just the external actions in the execution fragment. As an example, our lossy FIFO channel automaton with a natural number message type can produce the trace $[\mathit{Send}(1), \mathit{Send}(2), \mathit{Recv}(2)]$. The automaton cannot produce the trace $[\mathit{Send}(1), \mathit{Recv}(5)]$.

We define the *behavior* of an automaton $A$, $\mathit{beh}(A)$, to be the set of all traces of that automaton. This will be our basis for reasoning about specifications and implementations. We say that automaton $A'$ *refines* automaton $A$ if the behavior of $A$ contains the behavior of $A'$: $\mathit{beh}(A') \subseteq \mathit{beh}(A)$. For example, we could have a specification of a lossless FIFO channel, and we could have an implementation over the lossy FIFO channel that uses retransmissions, so that the implementation would behave like a lossless channel, refining the lossless FIFO spec. We formalize refinement in Coq such that we can write `refines A' A` to express this notion when $A$ and $A'$ have the same external action type.

Using this framework, we can formally reason about specifications and implementations as follows. We write specifications of systems as IO automata, and we write implementations of systems as IO automata or collections of IO automata. To show that an implementation satisfies a specification, we prove that the implementation refines the specification.

## 3.3 Composition

In the IO automata mathematical model, in a composition, automata responding to the same named action step together. We cannot directly support this behavior inside a proof assistant with a rigid type system when different automata have overlapping sets of actions. Instead, we provide a mechanism that allows for "wiring" automata together appropriately.

Composing two automata $A$ and $B$ produces a new IO automaton. In CoqIOA, composition is not just a function of the component automata: we must provide wiring information as well. When composing automata $A$ and $B$, we provide a new external action type $T$ for the composed automata, along with two mappings, $T \to ext(A) \cup \{None\}$ and $T \to ext(B) \cup \{None\}$ that describe how the new external action type maps to the external action types of the individual automata. The mappings return $None$ if a given action from $T$ does not correspond with an action for the component automata. Given these, the composition is defined as an IO automaton as follows. The state type is $states(A) \times states(B)$. The external action type is $T$. The internal action type is $int(A) \cup int(B)$. The start states contain $(a, b)$ if and only if $a \in start(A)$ and $b \in start(B)$. The transition relation steps the appropriate component automata for internal actions, and it steps the appropriate component automata (or both) for external actions according to the two mapping functions.

CoqIOA implements this wiring scheme as a `compose` function. In addition to a `compose` operation, CoqIOA also provides a `rename` operation that we can use to hide external actions by reclassifying them as internal actions. The interface is similar to that of the composition operation: the user supplies a new type for external actions, a new type for hidden actions, and functions describing the mapping from these new types to the original external action type. We defer an example of using the `compose` and `refine` machinery to Chapter 5.

## 3.4 Proof techniques

We use a number of different techniques to prove automata correct, i.e. showing that an implementation refines a specification. We use simulation relations in combination with high-level theorems about composition to enable compositional reasoning and proof reuse.

We can only describe automata refining other automata when both automata have the same external action type: it only makes sense to compare automata with the same API.

### 3.4.1 Simulation

Simulation relations are a standard technique for proving correctness of IO automata [23, 24, 25]. We formalize forward simulation and backward simulation in Coq.

A forward simulation between two automata $A$ and $B$ that have the same external action type is a relation $R \subseteq \mathit{states}(A) \times \mathit{states}(B)$ between states of $A$ and $B$ such that:

1. Every $s_A \in \mathit{start}(A)$ is related by $R$ to some $s_B \in \mathit{start}(B)$.

2. For each step $(s'_A, \pi, s_A) \in \mathit{steps}(A)$ and each $s'_B \in \mathit{states}(B)$ where $(s'_A, s'_B) \in R$, there exists a state $s_B \in \mathit{states}(B)$ such that $(s_A, s_B) \in R$ and:

    - If $\pi \in \mathit{ext}(A)$, then there exists some execution fragment of $B$ starting with $s'_B$ and ending with $s_B$ corresponding to the trace $[\pi]$.

    - If $\pi \in \mathit{int}(A)$, then there exists some execution fragment of $B$ starting with $s'_B$ and ending with $s_B$ corresponding to the empty trace.

A backward simulation between two automata $A$ and $B$ that have the same external action type is a relation $R \subseteq \mathit{states}(A) \times \mathit{states}(B)$ between states of $A$ and $B$ such that:

1. Every $s_A \in \mathit{states}(A)$ is related by $R$ to some $s_B \in \mathit{states}(B)$.

```
Theorem forward_simulation :
  forall (E : Type) (A' A : AutomatonDef E)
        (f : (StateType A') → (StateType A) → Prop),
    forward_simulation_relation f →
    refines A' A.
```

Figure 3-4: Statement of the forward simulation theorem in Coq.

2. If a state $s_A \in \mathit{start}(A)$ is related by $R$ to some $s_B \in \mathit{states}(B)$, then $s_B \in \mathit{start}(B)$.

3. For each step $(s'_A, \pi, s_A) \in \mathit{steps}(A)$ and each $s_B \in \mathit{states}(B)$ where $(s_A, s_B) \in R$, there exists a state $s'_B \in \mathit{states}(B)$ such that $(s'_A, s'_B) \in R$ and:

   - If $\pi \in \mathit{ext}(A)$, then there exists some execution fragment of $B$ starting with $s'_B$ and ending with $s_B$ corresponding to the trace $[\pi]$.
   - If $\pi \in \mathit{int}(A)$, then there exists some execution fragment of $B$ starting with $s'_B$ and ending with $s_B$ corresponding to the empty trace.

We formalize forward simulation in Coq and prove that the existence of a forward simulation from $A'$ to $A$ implies that $A'$ refines $A$. Figure 3-4 shows the statement of this theorem in Coq. We prove a parallel theorem for backward simulation: existence of a backward simulation from $A'$ to $A$ implies that $A'$ refines $A$.

### 3.4.2 Composition theorems

CoqIOA proves high-level theorems about composition that enable compositional reasoning. In the following exposition, we write composition of $A$ and $B$ as $A + B$, and we write $A'$ refines $A$ as $A' \subseteq A$. There notations do not perfectly capture our formalism: our `compose` operator has additional parameters besides the automata themselves, and our definition of `refines` has a requirement that the automata have the same external action type. Still, these notations are useful for developing an intuitive explanation of the composition theorems. CoqIOA takes these additional complexities into account: the composition theorems are proven correct with the

```
Theorem refines_comp_comm :
  forall (EA EB Ext : Type)
         (A :  AutomatonDef EA) (B : AutomatonDef EB)
         (mapA : Ext → option EA) (mapB : Ext → option EB),
    refines
      (compose A B Ext mapA mapB)
      (compose B A Ext mapB mapA).
```

Figure 3-5: Statement of the commutativity of composition theorem in Coq.

```
Theorem refines_comp_subst :
  forall (EA EB Ext : Type)
         (A A' : AutomatonDef EA) (B : AutomatonDef EB)
         (mapA : Ext → option EA) (mapB : Ext → option EB),
    refines A' A →
    refines
      (compose A' B Ext mapA mapB)
      (compose A B Ext mapA mapB).
```

Figure 3-6: Statement of the substitution in composition theorem in Coq.

additional parameters, and a usage of `refines` only type checks if both automata have the same external action type.

The first composition theorem is commutativity of composition: $A + B \subseteq B + A$. Figure 3-5 shows the theorem statement in Coq.

The second composition theorem is a substitution theorem: if $A' \subseteq A$, then $A' + B \subseteq A + B$. Figure 3-6 shows the theorem statement in Coq.

Our formalization of IO automata, along with these theorems, enables compositional reasoning. Chapter 5 demonstrates the use of this machinery.

# Chapter 4

# Implementation

We implement CoqIOA entirely in the Coq proof assistant. Table 4.1 shows the components of the framework, along with the number of lines of code that comprise each component. Only our definitions, comprising about 100 lines of code, are trusted. All other components are mechanically verified by Coq's proof checker: all theorems about IO automata, including theorems on simulation relations and composition, are proven correct in Coq.

We have used the CoqIOA framework to reason about toy systems implemented using IO automata: we describe these examples in Chapter 5. The examples are not included in the lines of code counts shown here.

The CoqIOA prototype currently has one major limitation: we do not have a code extraction mechanism to produce executable code from IO automata descriptions. This limitation is not inherent, and we plan to address it in future work.

All source for CoqIOA and examples is available on GitHub [26].

| **Component** | | **Lines** |
|---|---|---:|
| Definitions (trusted) | IO automaton | 10 |
| | Renaming | 30 |
| | Composition | 60 |
| | Refinement | 20 |
| Proof tools (untrusted) | Proof automation | 300 |
| | Theorems | 750 |

Table 4.1: Lines of code in CoqIOA.

# Chapter 5

# Evaluation

We demonstrate that CoqIOA enables compositional reasoning through a case study of a key-value store. Through the case study, we demonstrate the use of our composition machinery, simulation relation theorems, and composition theorems.

In our example, we have a specification of a key-value store modeled as a single automaton, and we have an implementation of a client communicating with a key-value server over channels that reorder messages. The implementation adds send and receive mediators to the channels to make them effectively implement reliable channels.

We prove that the implementation satisfies the specification, and we use compositional reasoning to construct the proof. First, we prove that a mediated reordering channel implements a reliable channel. Next, we prove a key-value server correct on top of a specification of a reliable channel. Finally, given those two proofs, we use our composition theorems to prove that our system communicating over mediated reordering channels implements a key-value store.

## 5.1 Specification

The key-value store is specified as a single IO automaton. For simplicity, the key-value store implements a mapping between natural numbers. Figure 5-1 shows the API of the key-value store, written as a Coq `Inductive` type: it takes `Put k v` and `Get k`

```
Inductive API :=
| Put (k : nat) (v : nat)
| PutOk
| Get (k : nat)
| GetResult (v : option nat).
```

Figure 5-1: Key-value store API, consisting of input and output actions, specified as an inductive type in Coq.

actions as inputs, and it produces `PutOk` and `GetResult v` actions as outputs. Figure 5-2 shows the Coq code for the specification: the key-value store is nonblocking, but it executes requests sequentially. For example, the specification can produce the trace $[Put(0,1), PutOk, Get(0), Get(5), GetResult(Some(1)), GetResult(None)]$, but the specification cannot produce the trace $[Put(0,1), PutOk, Get(0), GetResult(None)]$.

## 5.2 Implementation

Figure 5-3 shows the components that make up the implementation of the client-server key-value store. We write a modular proof that this implements the key-value store specification given in Figure 5-2. First, we prove that a mediated reordering channel implements a reliable channel. Then, we verify our client-server key-value store on top of the reliable channel. Finally, we invoke our composition theorem to show that the implementation in Figure 5-3 satisfies the specification.

### 5.2.1 Channels

Figure 5-4 defines two generic channel automata implementing a Channel API consisting of `Send(m)` and `Recv(m)` actions. We have a `Reliable` channel that implements a FIFO queue, and a `ReliableReordering` channel that implements a channel that is allowed to reorder messages. The reliable channel is a specification of an ideal channel that does not drop or reorder messages. The reordering channel acts as a simplified model of a real-world network channel: one that can arbitrarily reorder messages but not drop messages.

```
Inductive Request :=
| Req_Put (k : nat) (v :  nat)
| Req_Get (k : nat).
Inductive Response :=
| Resp_Ok
| Resp_Value (v : option nat).
Inductive Internal := Execute.

Record state : Type := mkState {
                        requests : list Request;
                        responses : list Response;
                        data : nat → option nat;
                      }.
Definition start (st : state) : Prop := st = mkState [] [] (fun _ ⇒ None).

Definition step (st : state) (act : Internal + API) (st' : state) : Prop :=
  let (req, res, d) := st in
  match act with
  | inr (Put k v) ⇒ st' = mkState (req ++ [Req_Put k v]) res d
  | inr (Get k) ⇒ st' = mkState (req ++ [Req_Get k]) res d
  | inl Execute ⇒
    exists hd tl, req = hd :: tl ∧
                st' = mkState tl
                             (res ++ [match hd with
                                     | Req_Get k ⇒ Resp_Value (d k)
                                     | _ ⇒ Resp_Ok
                                     end])
                             (match hd with
                              | Req_Put k v ⇒
                                fun k' ⇒ if eq_nat_dec k k' then
                                           Some v else
                                           d k'
                              | _ ⇒ d
                              end)
  | inr PutOk ⇒ exists tl, res = Resp_Ok :: tl ∧ st' = mkState req tl d
  | inr (GetResult v) ⇒ exists tl, res = (Resp_Value v) :: tl ∧ st' = mkState
      req tl d
  end.

Definition KVStore : AutomatonDef API :=
  mkAutomatonDef API state Internal start step.
```

Figure 5-2: Key-value store specified as a single automaton in Coq. The automaton maintains an input queue of requests, an output queue of responses, and the actual mapping from keys to values. Initially, the queues are empty and the mapping is empty. In the transition relation, the automaton can enqueue input, execute an enqueued operation, or send output.

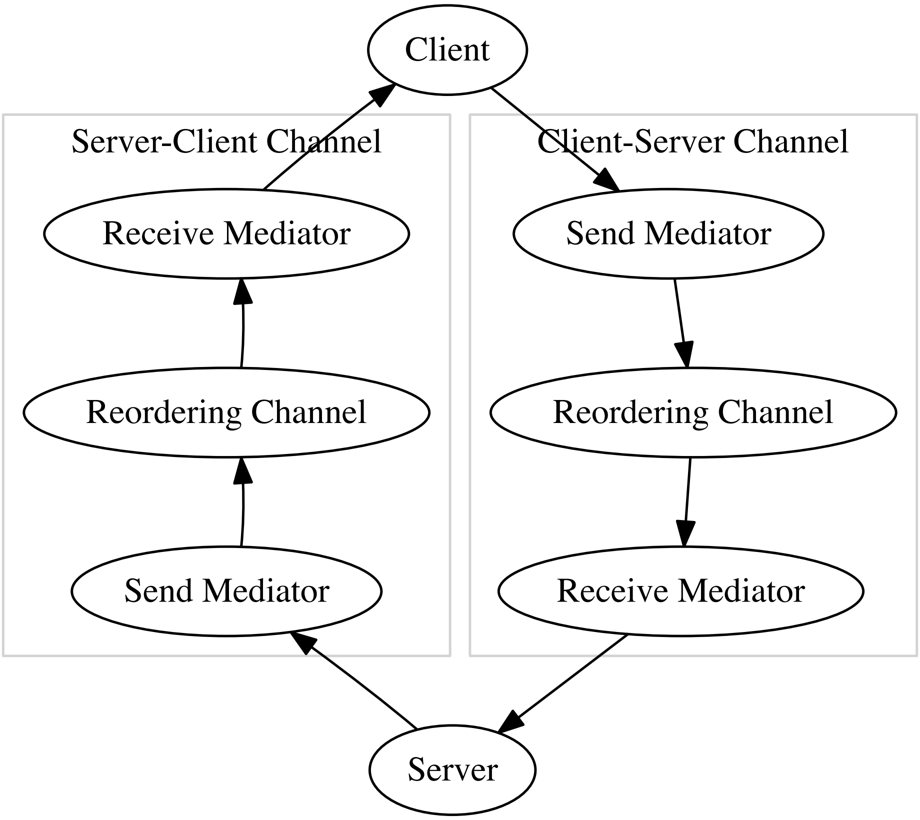

Figure 5-3: Client-server key-value store implementation over reordering channels.

```
Variable T : Type.

Inductive ChannelSenderAPI :=
| Send : T → ChannelSenderAPI.
Inductive ChannelReceiverAPI :=
| Recv : T → ChannelReceiverAPI.
Definition ChannelAPI : Type := ChannelSenderAPI + ChannelReceiverAPI.

Definition Reliable : AutomatonDef ChannelAPI :=
  mkAutomatonDef
    ChannelAPI
    (list T) (* state: list of messages *)
    EmptySet
    (fun st ⇒ st = []) (* start: buffer starts out empty *)
    (* transition relation: *)
    (fun st act st' ⇒
      match act with
      | inl e ⇒ match e with end
      | inr (inl (Send m)) ⇒ st' = st ++ [m]
      | inr (inr (Recv m)) ⇒ st = m :: st'
      end).

Definition ReliableReordering : AutomatonDef ChannelAPI :=
  mkAutomatonDef
    ChannelAPI
    (list T) (* state: list of messages *)
    EmptySet
    (fun st ⇒ st = []) (* start: buffer starts out empty *)
    (* transition relation: *)
    (fun st act st' ⇒
      match act with
      | inl e ⇒ match e with end
      | inr (inl (Send m)) ⇒ st' = st ++ [m]
      | inr (inr (Recv m)) ⇒ removed m st st' (* can receive any message *)
      end).
```

Figure 5-4: Coq code for reliable and reordering channels. A reliable channel is an ideal FIFO queue, while a reordering channel models a channel that is allowed to arbitrarily reorder messages.

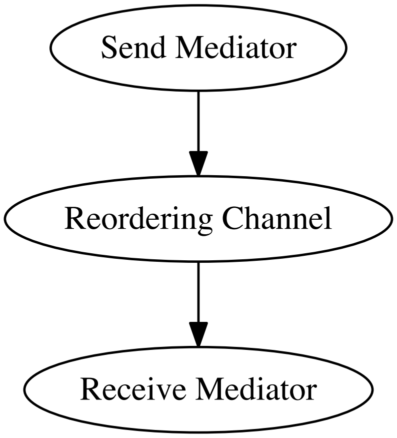

Figure 5-5: Components comprising the mediated reordering channel.

In order to make a reordering channel behave like a reliable channel, we add send and receive mediators. The send mediator adds sequence numbers to messages, and the receive mediator reconstructs the original order of the messages using the sequence numbers. Figure 5-5 shows the components that make up a mediated reordering channel, and Figure 5-6 shows the implementation of the mediators. The send mediator takes input messages of type `T`, and it outputs messages over a channel carrying `nat * T`, tagging messages with sequence numbers. On the other end of the channel, a receive mediator takes the tagged messages, reassembles the messages in the correct order, and delivers the raw messages with the sequence numbers stripped.

To reason about the mediated reordering channel, we first need to compose our send and receive mediators with a reordering channel. Figure 5-7 shows the `compose` operator in action: first, we compose the send mediator with the reordering channel to form `SendMediator_Channel`, and then we compose that with a receive mediator to form `SendMediator_Channel_ReceiveMediator`. We need to perform composition in two steps because CoqIOA's `compose` operator composes two automata at a time. This illustrates us "wiring" automata in the composition.

The resultant automata has the wrong API: the details of the inner reordering

```
Variable T : Type. (* message type *)

(* input from world, output to channel *)
Definition ReorderingSendMediator :
  AutomatonDef (ChannelSenderAPI T + ChannelSenderAPI (nat ∗ T)) :=
  mkAutomatonDef
    _
    (nat ∗ list (nat ∗ T)) (* state: next sequence number, message buffer *)
    EmptySet
    (fun st ⇒ st = (0, [])) (* start: seq=0, empty buffer *)
    (fun st act st' ⇒
      match act with
      | inl e ⇒ match e with end
      | inr (inl (Send m)) ⇒
        (* enqueue message tagged with sequence number in local buffer *)
        st' = (fst st + 1, snd st ++ [(fst st, m)])
      | inr (inr (Send (c, m))) ⇒
        (* send buffered message over channel *)
        fst st' = fst st ∧ removed (c, m) (snd st) (snd st')
      end).

(* input from channel, output to world *)
Definition ReorderingReceiveMediator :
  AutomatonDef (ChannelReceiverAPI (nat ∗ T) + ChannelReceiverAPI T) :=
  mkAutomatonDef
    _
    (nat ∗ list (nat ∗ T)) (* state: next sequence number, message buffer *)
    EmptySet
    (fun st ⇒ st = (0, [])) (* start: seq=0, empty buffer *)
    (fun st act st' ⇒
      match act with
      | inl e ⇒ match e with end
      | inr (inl (Recv (c, m))) ⇒
        (* enqueue message *)
        st' = (fst st, snd st ++ [(c, m)])
      | inr (inr (Recv m)) ⇒
        (* deliver message with the current sequence number *)
        fst st' = fst st + 1 ∧ removed (fst st, m) (snd st) (snd st')
      end).
```

Figure 5-6: Coq code for send and receive mediators. The send mediator takes input messages and tags them with sequence numbers. The receive mediator takes input messages tagged with sequence numbers and delivers them in the correct order.

channel are exposed. We want the automata to only expose the send mediator's input and the receive mediator's output. Figure 5-8 fixes this using the `rename` operator to only expose a `ChannelAPI T` while hiding the actions corresponding to the inner channel by reclassifying them as internal actions. `MediatedReliableReordering` is the result of this hiding operation.

We prove that the mediated reordering channel implements a reliable channel. In particular, we prove `refines (MediatedReliableReordering T) (Reliable T)`. We use forward simulation to prove the mediated reordering channel correct: the proof involves reasoning about the interactions between the send and receive mediator and showing that the receive mediator correctly reassembles reordered messages.

### 5.2.2 Client-server key-value store

We prove our client-server key-value store correct on top of reliable channels, as shown in Figure 5-9, rather than directly reasoning about the key-value store built on reordering channels. In this proof, we reason about communication between the client and server, but we do not have to reason about messages being reordered. This separation of concerns greatly simplifies the proof. We use forward simulation to prove the implementation correct, and the proof ends up being quite simple: we show that the client buffer, channel, and server buffers act like one big queue in each direction, preserving the order and identity of operations invoked at the client and results coming from the server.

### 5.2.3 System

Given the proofs of correctness of our mediated reordering channel and our key-value store implemented on top of reliable channels, we invoke our composition theorem to produce an end-to-end proof of correctness of the original system as described in Figure 5-3. Our composition theorem makes this easy: because we have shown that mediated reordering channels implement reliable channels, we can swap the reliable channel specification for its implementation while preserving correctness. This

```
Definition SendMediator_Channel :=
  compose
    ReorderingSendMediator
    (ReliableReordering (nat ∗ T))
    (ChannelSenderAPI T + ChannelAPI (nat ∗ T))
    (fun act ⇒ match act with
                | inl act’ ⇒ Some (inl act’)
                | inr act’ ⇒ match act’ with
                              | inl act” ⇒ Some (inr act”)
                              | inr _ ⇒ None
                              end
                end)
    (fun act ⇒ match act with
                | inl _ ⇒ None
                | inr act’ ⇒ Some act’
                end).

Definition SendMediator_Channel_ReceiveMediator :=
  compose
    SendMediator_Channel
    ReorderingReceiveMediator
    (ChannelSenderAPI T + ChannelAPI (nat ∗ T) + ChannelReceiverAPI T)
    (fun act ⇒ match act with
                | inl act’ ⇒ Some act’
                | inr _ ⇒ None
                end)
    (fun act ⇒ match act with
                | inl act’ ⇒ match act’ with
                              | inl _ ⇒ None
                              | inr act” ⇒ match act” with
                                             | inl _ ⇒ None
                                             | inr act”’ ⇒ Some (inl act”’)
                                             end
                              end
                | inr act’ ⇒ Some (inr act’)
                end).
```

Figure 5-7: Composition of components of a mediated reordering channel. The code demonstrates “wiring” automata in a composition.

```
Definition MediatedReliableReordering : AutomatonDef (ChannelAPI T) :=
  rename
    SendMediator_Channel_ReceiveMediator
    (ChannelAPI T)
    (fun act ⇒ match act with
                | inl act' ⇒ inl (inl act')
                | inr act' ⇒ inr act'
                end)
    (ChannelAPI (nat ∗ T))
    (fun act ⇒ inl (inr act)).
```

Figure 5-8: Hiding the internal details of the mediated reordering channel using the rename operator. The result is a mediated channel that has the same API as the reliable channel and the reordering channel.

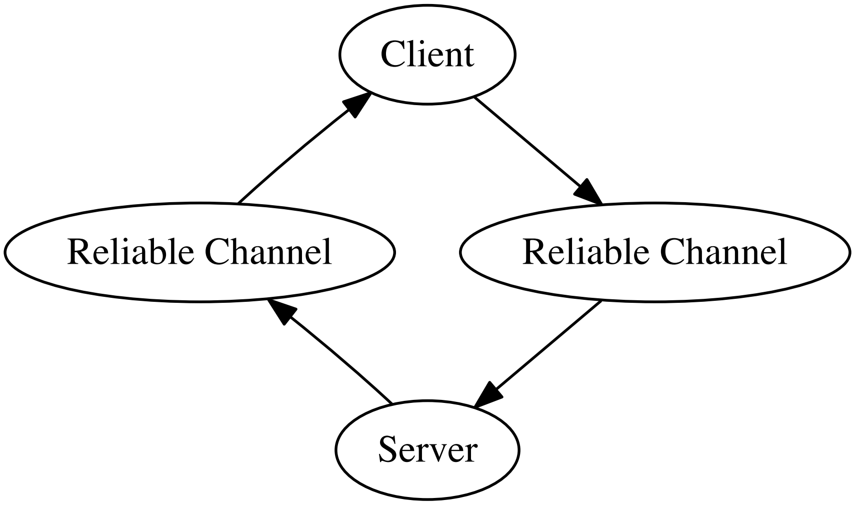

Figure 5-9: Client-server key-value store implementation over reliable channels.

| **Component** | | **Lines** |
|---|---|---|
| Channel | Specification (trusted) | 40 |
| | Implementation (untrusted) | 80 |
| | Proof | 250 |
| Key-Value Store with Reliable Channels | Specification (trusted) | 50 |
| | Implementation (untrusted) | 150 |
| | Proof | 80 |
| Key-Value Store with Reordering Channels | Proof | 10 |

Table 5.1: Lines of code in the key-value store implementation.

correctness result only requires a couple additional lines of proof, invoking our rename-related and composition-related theorems to reuse the proofs of correctness of the individual components.

## 5.3 Effort

Table 5.1 shows the components of the key-value store, along with the number of lines of code comprising each component. The final proof of the key-value store with unreliable channels required minimal effort: it required only 10 lines of code invoking our renaming and composition theorems. This was only possible because we were able to use our composition theorems to reuse the proofs of correctness of the individual components.

## 5.4 Discussion

We demonstrate that CoqIOA enables compositional reasoning about systems in a proof assistant, which was not possible with prior work. We show that our general-purpose composition theorems allow us to build and verify implementations on top of specifications and then swap out the underlying specifications for implementations while preserving correctness of the implementation.

# Chapter 6

# Future Work

CoqIOA in its current form has limitations that we aim to address in future work.

**Code extraction.** CoqIOA provides a framework for reasoning about IO automata, but it does not provide a mechanism for executing IO automata on real machines. In order to build certified executable implementations of distributed systems, we plan on building code extraction machinery for CoqIOA.

Prior work has explored translating IO automata specifications to executable code. Musial implemented a translator from IO automata specs to Java code [27], compiling IOA specifications from a subset of the Tempo specification language to executable code.

It would be difficult to automatically translate IO automata specified in CoqIOA to executable code due to the way our automata are specified. Automata can behave nondeterministically, and the transition relation does not even need to be decidable. This is desirable for modeling and specification, but it makes automatic code extraction difficult. An approach to code extraction from CoqIOA would likely involve defining a type of restricted IO automaton designed for code extraction. For these restricted automata, we could limit nondeterminism and structure the automata so that effects of actions are decidable, simplifying code extraction.

**Larger examples.** Chapter 5 demonstrates that CoqIOA is capable of compositional reasoning through a toy example, but we have not yet implemented realistic large-scale distributed systems in our framework.

Prior work has used IO automata to reason about a large class of distributed systems, including complex designs such as Byzantine fault tolerance [28]. We could formalize such paper proofs in CoqIOA, though it will likely involve considerable effort to fill in the details in the paper proofs to produce machine-checkable versions.

**Liveness.** CoqIOA supports reasoning only about safety. The IOA mathematical model does support reasoning about liveness. It should be possible to extend CoqIOA's model of IO automata to include the liveness-related components and then formalize liveness-related theorems about IO automata in Coq.

# Chapter 7

# Conclusion

This thesis contributes a methodology for compositional reasoning about distributed systems in a theorem prover and CoqIOA, an implementation of the methodology in the Coq proof assistant. The CoqIOA framework formalizes the theory of IO automata in a way that enables compositional reasoning about distributed systems within the proof assistant. CoqIOA includes a formalization of IO automata along with theorems about automata refinement, simulation, and composition, all of which are mechanically verified using Coq's proof checker.

We implemented and verified a toy key-value store in a modular way using the CoqIOA framework. The evaluation demonstrates that the CoqIOA framework enables compositional reasoning and is a promising approach for building modular verified distributed systems.

# Bibliography

[1] K. Kingsbury, "Jepsen analyses," http://jepsen.io/analyses.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[3] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "Verifying safety properties with the TLA+ Proof System," in *Proceedings of the 5th International Conference on Automated Reasoning*, ser. IJCAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 142–148. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14203-1_12

[4] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699417

[5] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the Raft consensus protocol," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2016. New York, NY, USA: ACM, 2016, pp. 154–165. [Online]. Available: http://doi.acm.org/10.1145/2854065.2854081

[6] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *PLDI 2015: Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, Portland, OR, USA, Jun. 2015, pp. 357–368.

[7] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "IronFleet: Proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 1–17. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815428

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX

Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666

[9] CoreOS, "CoreOS Raft library: Notable users," https://github.com/coreos/etcd/tree/master/raft#notable-users.

[10] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, pp. 219–246, 1989.

[11] A. Bogdanov, "Formal verification of simulations between i/o automata," Master's thesis, Massachusetts Institute of Technology, Sep. 2001.

[12] H. Lim, "Translating timed I/O automata specifications for theorem proving in PVS," Master's thesis, Massachusetts Institute of Technology, Feb. 2006.

[13] T. Nipkow and K. Slind, *I/O automata in Isabelle/HOL*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 101–119. [Online]. Available: http://dx.doi.org/10.1007/3-540-60579-7_6

[14] Coq development team, *Coq Reference Manual, Version 8.6*, INRIA, Dec. 2016, https://coq.inria.fr/distrib/V8.6/refman/.

[15] D. Ongaro, "Consensus: Bridging theory and practice," Ph.D. dissertation, Stanford University, Aug. 2014.

[16] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.808407

[17] L. Lamport, "Byzantizing Paxos by refinement," in *Proceedings of the 25th International Conference on Distributed Computing*, ser. DISC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 211–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=2075029.2075058

[18] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[19] S. J. Garland and N. Lynch, "Foundations of component-based systems," G. T. Leavens and M. Sitaraman, Eds. New York, NY, USA: Cambridge University Press, 2000, ch. Using I/O Automata for Developing Distributed Systems, pp. 285–312. [Online]. Available: http://dl.acm.org/citation.cfm?id=336431.336455

[20] D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager, *The Theory of Timed I/O Automata, Second Edition*, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010. [Online]. Available: http://dx.doi.org/10.2200/S00310ED1V01Y201011DCT005

[21] N. A. Lynch, S. J. Garland, D. Kaynar, L. Michel, and A. Shvartsman, *The Tempo Language User Guide and Reference Manual*, Massachusetts Institute of Technology, Oct. 2011, http://www.veromodo.com/resources/Tempo_Guide.pdf.

[22] *The Tempo Model Checker User Guide and Reference Manual*, Veromodo, Inc., May 2010, http://www.veromodo.com/resources/MC_manual.pdf.

[23] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, May 1991. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(91)90224-P

[24] N. Lynch and F. Vaandrager, *Forward and backward simulations for timing-based systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 397–446. [Online]. Available: http://dx.doi.org/10.1007/BFb0032002

[25] J. F. Søgaard-Andersen, S. J. Garland, J. V. Guttag, N. A. Lynch, and A. Pogosyants, *Computer-assisted simulation proofs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 305–319. [Online]. Available: http://dx.doi.org/10.1007/3-540-56922-7_25

[26] "CoqIOA code," https://github.com/anishathalye/coqioa, 2017.

[27] P. M. Musial, *Using Timed Input/Output Automata for Implementing Distributed Systems*, http://www.veromodo.com/resources/Tempo2JavaGuide.pdf.

[28] M. Castro and B. Liskov, "A correctness proof for a practical Byzantine-Fault-Tolerant replication algorithm," Massachusetts Institute of Technology, Tech. Rep., 1999.