# Process-Algebra Proofs for Distributed, Message-Passing Cryptographic Code

by

Mahmoud Sobier

S.B. Computer Science and Engineering and Philosophy, MIT, 2022

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

September 2024

| | |
|---|---|
| Authored by: | Mahmoud Sobier<br>Department of Electrical Engineering and Computer Science<br>August 13, 2024 |
| Certified by: | Adam Chlipala,<br>Professor of Computer Science, Thesis Supervisor |
| Accepted by: | Katrina LaCurts,<br>Chair, Master of Engineering Thesis Committee |

# Process-Algebra Proofs for Distributed, Message-Passing Cryptographic Code

by

Mahmoud Sobier

Submitted to the Department of Electrical Engineering and Computer Science on August 13, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

## ABSTRACT

The translation of a cryptographic protocol from specification to code has long been a source of bugs and vulnerabilities. In particular, erroneous state-machine logic has undermined the security of many cryptographic implementations. Inspired by a high-profile case where faulty control flow in TLS state-machine code led to a severe vulnerability, we develop modular reasoning principles for proving the correctness of a low-level state-machine implementation against a high-level specification of a distributed message-passing protocol. We introduce a compact and expressive representation for modeling concurrent state machines in Coq and connect it to the Interactive Probabilistic Dependency Logic, a process calculus that comes coupled with an equational logic for simplifying cryptographic proofs. In the style of process-algebra proof techniques, we simplify verification of concurrent state-machine implementations by stating algebraic laws by which proofs of multithreaded programs are derived from those of their component threads.

Thesis supervisor: Adam Chlipala,
Title: Professor of Computer Science

# Acknowledgments

I would like to thank Adam Chlipala, my thesis advisor, for his guidance and mentorship throughout my time as an MEng student. Additionally, I would like to express my gratitude towards Andres Erbsen for volunteering his time during the early stages of the project to help formulate project ideas. I also want to thank Joshua Gancher for answering my queries about IPDL via email, and Samuel Guetter for providing me a language definition file for displaying the Coq snippets in this document.

I want to thank the countless friends and associates I've made during my time at MIT who have encouraged me to not only enrich my technical skills further but also my character and sense of justice. Last but certainly not least, I thank my family for their enduring love and support, without which I would not be where I am today.

# Contents

# Chapter 1

# Introduction

For as long as cryptographic protocols have been deployed, there has always been a glaring gap between the worlds of theory and practice. While a protocol may sport formal guarantees of security subject to certain assumptions, it is usually another story entirely as to whether it retains that security when laid out into code by error-prone authors. In light of this concern, it would be desirable to be able to write out cryptographic protocols in a high-level specification source language and demonstrate that a low-level reference implementation preserves the semantics and proven security of the protocol. To this end, we present proof techniques to verify low-level state machine code, mechanized in the Coq theorem prover.

## 1.1 Motivating Example: TLS Handshake Vulnerability

To illustrate the problem at hand, we turn our attention to a particularly grievous TLS vulnerability that plagued OpenSSL 1.0.1g and earlier [1]. During the TLS handshake, the server and client engage in a stateful interaction to establish cipher settings and session-specific keys to be used in the encrypted session to follow. In the affected versions, OpenSSL accepted a `ChangeCipherSpec` message before the master secret was generated, causing session-specific keys to be derived from a zero-length master secret [2]. With this in mind, an attacker in the middle of the connection can inject a `ChangeCipherSpec` message in both directions to cause both parties to generate identical, deterministic session keys. From there, the attacker would know both parties' shared secret and thus have the ability to decrypt and modify all encrypted traffic.

The vulnerability arises from a faulty interpretation of the TLS state-machine control flow. In a correct TLS implementation, a `ChangeCipherSpec` should be accepted only when a new cipher is ready to be used. However, in the affected code, the boolean flag `new_cipher` tracking this information is actually set as soon as a new cipher type is decided, not when the cipher parameters have actually been set [2].

## 1.2 Contributions

To remedy the sort of issues described in the prior section, we provide the following contributions:

- We present a small-step operational semantics for Interactive Probabilistic Dependency Logic (IPDL), a high-level language for specifying distributed, message-passing cryptographic protocols that comes coupled with an equational logic for mechanizing security proofs.

- We construct a compact representation for state-machine programs, aiding in simplifying involved control flow into straight-line code and making more explicit transitions between individual program states.

- We develop a labeled transition system for state machines, prove trace inclusion between state machines and IPDL, and develop a notion of *simulation* for equating the behavior of IPDL programs to state machines.

- We adapt modular proof techniques from process algebra to connect simulation proofs of individual threads to those of multithreaded programs.

# Chapter 2

# Related Work

## 2.1 Formally Verified Cryptography

Given the challenge of designing cryptographic systems and their widespread deployment on the modern Internet, a variety of formal verification tools have been developed to provide a high level of assurance in their reported security properties. Broadly speaking, approaches to formally verified cryptography typically rely on one of two protocol models. On one hand, some tool authors choose to operate within the Dolev-Yao symbolic model, where cryptographic primitives are modeled as black-box function symbols and messages as terms supplied as arguments to these functions [3]. The symbolic model has the benefit of being conducive to proof autmotation, since one can exhaustively compute the set of messages that the adversary knows [4].

Though many tools have been written in this vein, including ProVerif [5], Tamarin [6], DeepSec [7], and others, security guarantees in the symbolic model are weaker than those granted in the computational model. In this model, cryptographic primitives are modeled as functions on bitstrings and the adversary is seen as a probabilistic polynomial time Turing machine. Security properties are stated and proven as propositions about probability distributions. One salient framework for cryptographic proof in ths vein is Universal Composability (UC), particularly due to its high expressive power and ability to compose proofs of protocols together in a modular way. Security proofs in UC operate by proving observational equivalence between the protocol in question and an idealization that serves as a trusted source of security. Though the computational model provides strong security guarantees, probabilistic reasoning in this setting has historically proven tricky to automate. Additionally, the semantics of distributed message-passing protocols complicate proofs of observational equivalence. In particular, concurrent message passing means one has to wrestle with nondeterminism and deal with multiple possible interleavings of messages. Capturing the interactivity in such protocols can involve a lot of proof effort since one must establish observational equivalence by writing out explicit bisimulations over protocol states [8].

Recent attempts have tried to bring together the best of both worlds. These include IPDL, a language for specifying distributed protocols in a message-passing style that comes coupled with an equational logic. By constraining the language to a subset of confluent protocols, IPDL is able to provide means to prove observational equivalences using simple equational reasoning rules, without needing to write out explicit bisimulations. Since the logic is proven to be sound with respect to the computational model, any proofs established within IPDL hold within the assumptions of the computational model.

## 2.2 Proof-generating Compilation

While formal verification of protocol specifications helps to bolster our confidence in their purported security properties, sticking to this domain brings us no closer to a world where vulnerability-free cryptographic code is the norm. In order to connect the lessons of protocol verification to real-world production code, efforts have been made to construct formally verified compilation schemes that generate code that is secure by construction.

To date, many such efforts are restricted to certain algorithms or subsets thereof. For instance, Fiat Cryptography introduces a formally verified compilation scheme for translating high-level Coq code of certain cryptographic primitives into C and x86 assembly [9]. Ikebuchi et. al produce verified nested state machine code common in protocols like TLS by compiling down from a high-level functional language, avoiding the laborious and often error-prone process of writing such code by hand [10].

Ikebuchi et. al's work makes use of a form of program derivation, where compilation is framed as a proof search for the existence of a program that satisfies a given spec. This is made possible by Coq's powerful Ltac tactic scripting language, which allows one to introduce existential variables to propositions and fill them in as additional constraints become known. As a result, Coq is able to guarantee correctness of the compiled program by constructing it in tandem with its proof.

More comprehensive projects include EverCrypt [11], a suite of verified and performant cryptographic functions exposed through an agile API. Through the use of generic types, EverCrypt allows clients to switch between multiple algorithms that provide the same functionality.

# Chapter 3

# Small-Step Operational Semantics for IPDL

As a first step, we showcase a small-step operational semantics for defining the execution of IPDL programs under the backdrop of an external environment that mediates the exchange of data between threads. But before doing so, we give a brief overview of IPDL, explaining its security guarantees and language features.

## 3.1 Background on IPDL

### 3.1.1 Security in IPDL

Following in the tradition of UC-style security proofs, cryptographic protocols in IPDL are modeled as distributed, message-passing systems comprising of communicating parties and functionalities, ideal services whose security is assumed, like an authenticated communication channel [12]. Additionally, protocols in UC interact with the external world through two interfaces: an environmental interface and an attacker interface [12]. The environmental interface specifies the general I/O mechanism that the parties use to communicate. On the other hand, the attacker interface details the capabilities of the attacker to undermine the protocol, like being able to eavesdrop on or tamper with in-flight messages.

Security properties in IPDL are proved by comparing protocols to idealizations wherein computations are performed by trusted functionalities. Compared to the attacker interface of a generic protocol, the idealization's attacker interface is extermely constrained, being able to only tell whether or not a computation has been carried to completion. A security proof in this domain typically aims to establish *observational equivalence* between a protocol $\pi$ and an idealization $\mathsf{Ideal}$. The $\mathsf{Ideal}$ protocol is typically connected to a simulator whose job it is to convert the attacker interface of $\mathsf{Ideal}$ to that of $\pi$. In doing so, it is shown that a polynomial-time attacker has no means of distinguishing between $\pi$ and $\mathsf{Ideal}$, and so no attack launched on $\pi$ is more powerful than one directed towards $\mathsf{Ideal}$.

Such proofs usually involve long chains of equivalence steps comprising of both exact and approximate congruences. The former concerns *semantic equivalences* between protocols, while the latter is typically applied to take advantage of an indistinguishability assumption.

Most non-trivial proof effort is confined to establishing exact equivalences, which commonly involves writing bisimulations. However, bisimulations are tedious to write and require a disproportionate amount of low-level reasoning, complicating proofs and distracting from high-level cryptographic details. By limiting the expresssiveness of the language, IPDL is able to admit general equational-reasoning principles for establishing semantic and approximate equivalences without the need for explicit hand-written bisimulations.

### 3.1.2 Language Features

Having covered security in IPDL, we now segue into a slightly informal exposition of the core language features. At its most basic level, the IPDL language is a process calculus consisting of two layers: protocols and reactions. Protocols describe systems of concurrent, interactive channels that are assigned to probabilistic, monadic programs known as reactions. Reactions are capable of reading from channels, sampling from distributions, and signaling when a value should be broadcast to other channels. Below, we show a simplified version of the IPDL syntax more closely mirroring the shallow embedding used in the Coq library.

| | | | | |
|---|---|---|---|---|
| Types | $\tau$ | $::=$ | $\tau$ | types |
| Values | $v$ | $::=$ | $v$ | values (of type $\tau$) |
| Expressions | $e$ | $::=$ | $e$ | expressions |
| Distributions | $d$ | $::=$ | $d$ | distributions |
| Channels | $c$ | $::=$ | $c$ | channels |
| Reactions | $R, X$ | $::=$ | $\mathsf{ret}(e)$ | return |
| | | $\mid$ | $\mathsf{samp}(d)$ | sampling from distribution |
| | | $\mid$ | $\mathsf{read}\ c$ | channel read |
| | | $\mid$ | $x\colon \tau \leftarrow R; X$ | reaction bind |
| Protocols | $P, Q$ | $::=$ | $\mathsf{Zero}$ | inert protocol |
| | | $\mid$ | $c ::= R$ | channel assignment |
| | | $\mid$ | $P \Vert Q$ | parallel composition |
| | | $\mid$ | $\mathsf{new}\ c\colon \tau\ \mathsf{in}\ P$ | local channel generation |

Protocols are built from three main constructors: $c ::= R$ assigns to channel $c$ a reaction $R$, $P_1||P_2$ allows protocols $P_1$ and $P_2$ to engage in concurrent interaction, $\mathsf{new}\ c : \tau\ \mathsf{in}\ P$ generates a new local channel for private use in $P$. $\mathsf{Zero}$ stands for the inert protocol, which is primarily used as the identity in parallel composition. Letting $\mathsf{unit}$ stand for the unit type, it can be derived from the $\mathsf{new}$ constructor: $\mathsf{new}\ c\colon \mathsf{unit}\ \mathsf{in}\ c ::= \mathsf{ret}(())$

The basic I/O for IPDL protocols occurs inside reactions: $\mathsf{read}\ c$ retrieves a value that was broadcast from channel $c$ while $c ::= \mathsf{ret}(v)$ broadcasts value $v$ to any protocol that wants to read from $c$. The reaction bind construct $x\colon \tau \leftarrow R; X$ enables reaction composition, threading the value $x$ returned by $R$ into reaction $X$.

To formalize the features we have discussed so far, we present a fragment of the operational semantics for IPDL below. Protocols possess several stepping relations. $P \xrightarrow{o:=v} Q$ indicates that the reaction assigned to $o$ has terminated, broadcasting value $v$ and yielding protocol $Q$. In addition, Gancher et al. [8] define both a small-step internal stepping relation

$P \to \eta$ and a big-step relation $P \Downarrow \eta$ for relating protocols to distributions over protocols. For the sake of brevity, we omit these relations and focus our attention on the small-step output relation, where the syntax has been extended slightly to allow reactions and protocols to hold intermediate values:

$$\frac{}{o ::= \mathsf{val}(v) \xrightarrow{o:=v} o ::= v} \qquad \frac{P \xrightarrow{o:=v} P'}{P \| Q \xrightarrow{o:=v} P' \| Q[\mathsf{read}\ o := \mathsf{val}(v)]}$$

$$\frac{Q \xrightarrow{o:=v} Q'}{P \| Q \xrightarrow{o:=v} P[\mathsf{read}\ o := \mathsf{val}(v)] \| Q'} \qquad \frac{P \xrightarrow{o:=v} Q \quad o \neq c}{\mathsf{new}\ c\colon \tau\ \mathsf{in}\ P \xrightarrow{o:=v} \mathsf{new}\ c\colon \tau\ \mathsf{in}\ Q}$$

Assuming we already have big-step semantics defined for expressions and distributions, the behavior of reactions is specified by a stepping relation from reactions to distributions over reactions:

$$\frac{e \Downarrow v}{\mathsf{ret}(e) \to \mathsf{unit}(\mathsf{val}(v))} \qquad \frac{}{x\colon \tau \leftarrow \mathsf{val}(v); S \to \mathsf{unit}(S[x := v])}$$

$$\frac{d \Downarrow \sum_i c_i \mathsf{unit}(v_i)}{\mathsf{samp}(d) \to \sum_i c_i \mathsf{unit}(\mathsf{val}(v_i))} \qquad \frac{R \to \sum_i c_i \mathsf{unit}(R_i)}{x : \tau \leftarrow R; X \to \sum_i c_i \mathsf{unit}(x : \tau \leftarrow R_i; X)}$$

where $\mathsf{unit}(v)$ stands for a distribution where $\Pr[X = v] = 1$ and $\Pr[X = a] = 0$ for all $a \neq v$.

IPDL's type system guarantees that, for well-typed IPDL protocols, channel reads are always in scope and that all channels are assigned unique reactions. For a finer-grained discussion of the IPDL typing system, we refer the reader to Gancher et. al's paper [8].

It is worth noting that channels in IPDL may emit outputs in any order. Usually, this feature poses problems for cryptographic reasoning, since nondeterminism may leak crucial details about the execution of the protocol that an attacker could then use to undermine security. However, since reads are blocking in IPDL, a well-typed protocol eventually converges to the same state regardless of channel output order. This notion is formalized through IPDL's confluence theorem:

**Theorem** (Confluence Theorem)**.** If protocol $P$ is well-typed, then:

- If $P \xrightarrow{o:=v} Q$ and $P \xrightarrow{o:=v'} Q'$, then $v = v'$ and $Q = Q'$.
- If $P \xrightarrow{o_1:=v_1} Q_1$ and $P \xrightarrow{o_2:=v_2} Q_2$ with $o_1 \neq o_2$, then there exists $Q$ such that $Q_1 \xrightarrow{o_2:=v_2} Q$ and $Q_2 \xrightarrow{o_1:=v_1} Q$.
- If $P \xrightarrow{o:=v} Q$ and $P \to \eta$, then there exists $\eta'$ such that $\eta \xrightarrow{o:=v} \eta'$ and $Q \to \eta'$.
- If $P \to \eta_1$ and $P \to \eta_2$, then either $\eta_1 = \eta_2$, or there exists $\eta$ such that $\eta_1 \to \eta$ and $\eta_2 \to \eta$.

## 3.2 Operational Semantics

In this section, we will present an alternate small-step operational semantics for IPDL for specifying protocol interaction with an external environment with persistent state, modeled as a dictionary data structure mapping channel names to the data they have outputted so far. In this way, we aim to develop a metaphor for how IPDL protocols could be executed in a low-level environment.

To model a form of persistent memory, we use an off-the-shelf finite-map implementation to represent the dictionary data structure. As shown in the code excerpt below, our dictionary maps strings to a dynamic data type, implemented in Coq as a record type with two fields: a bitstring and an integer indicating its width. We choose this representation for several reasons. For one, this grants IPDL programs the ability to exchange cryptographic data of varying lengths, so that keys, ciphertexts, passwords, etc. do not all have to be confined to the same length if they need not be. Additionally, we aimed to enhance compatibility with the match expression in Gallina functions. Since, in general, parameter inference in dependent pattern matching is undecidable [13], [14], [15], we need to restrict values to a type for which there is a decidable procedure for equality checking, in order to more smoothly operate on dynamic values using match expressions.

```
Definition map (ran : Type) := fmap string ran.

(* Dynamic data type *)
Record dyn := {
    width : nat ;
    value : width.-bv
}.

(* Dictionary type definition *)
Definition c_state := map (option dyn).
```
Listing 3.1: Dictionary data structure

Now we transition to the rules of our operational semantics. As a note regarding notation, we use $m$ \$? $k$ to mean a lookup operation in dictionary $m$ for key $k$, and $m$ \$+ $(k, v)$ to stand for adding key-value pair $(k, v)$ to dictionary $m$. Additionally, in order to facilitate the construction of a labeled transition system later, we assign each rule in the operational semantics exactly one label. A `Send ch v` indicates that a protocol has broadcast value `v` along channel `ch`, whereas `Receive ch v` signals a read of value `v` from channel `ch`. `LocalGen ch` is emitted when a protocol has generated a new local channel and assigns it the name `ch`. When the state is neither modified nor read from, we generate a `Silent` label.

The semantics for reactions consists of just three rules. We step directly from `Read ch` to `Ret v` if channel `ch` is mapped to a value `v` in the channel state. To handle sampling, we nondeterministically select a value `v` from the distribution being sampled from and step to `Ret v`. Note that this is only a approximation of probabilistic choice since values can be chosen in a way that is inconsistent with the distribution e.g choosing zero probability values from the sample space. Lastly, we allow binds to step if the initial reaction can step to Ret v, in which case, we thread v through a continuation holding the next reaction.

For protocols, we start with the rule that handles local channel generation, LEvalNew. In order to successfully step, we need to ensure that the new channel name generated is not contained in the set of keys associated with the starting channel state. If this check succeeds, then we update the state with the new channel, mapping it to a None value to indicate that the channel is yet to be filled. This measure also prevents naming conflicts between protocols set in parallel composition with one another.

When we encounter channel assignment, we defer to the labeled stepping relation for reactions to check whether the assigned reaction can step. If it can, then so does the protocol. However, when the assigned reaction is a ret, signaling that the channel is ready to broadcast a value, we update the state with the corresponding value and terminate computation by stepping to Zero.

When either side of a parallel composition is Zero, we garbage collect the Zero and step to the other side.

```
Inductive leval_ipdl : c_state → c_state → @ipdl str → label → @ipdl str → Prop :=
| LEvalNew :
   forall {w : nat}, forall prog, forall (state: c_state), forall ch,
    ∼ InMap state ch →
    leval_ipdl state (state $+ (ch, None)) (New (w.-bv) prog) (LocalGen ch) (prog ch)

| LEvalRxn:
  forall {w: nat}, forall (state : c_state), forall o,
  forall (r1 r2: @rxn str (w.-bv)), forall label,
    leval_rxn state r1 label r2 →
    leval_ipdl state state (o ::= r1) label (o ::= r2)

| LEvalTerminateOnReturn :
  forall {w : nat}, forall (state : c_state), forall o, forall (v : (w.-bv)),
    leval_ipdl state (state $+ (o, Some {| width := w; value := v |})) (o ::= Ret v)
    (Send o ({| width := w; value := v |})) Zero

| LEvalGarbageCollectOnLeft :
   forall (state : c_state), forall p2,
    leval_ipdl state state (Zero ||| p2) Silent p2

| LEvalGarbageCollectOnRight :
   forall (state : c_state ), forall p1,
    leval_ipdl state state (p1 ||| Zero) Silent p1

| LEvalParsLeft :
  forall (state state' : c_state ), forall p1 p1' p2 label,
    leval_ipdl state state' p1 label p1' →
    disjoint_supp p1 p2 →
    leval_ipdl state state' (p1 ||| p2) label (p1' ||| p2)

| LEvalParsRight :
  forall (state state' : c_state ), forall p2 p2' p1 label,
    leval_ipdl state state' p2 label p2' →
    disjoint_supp p2 p1 →
    leval_ipdl state state' (p1 ||| p2) label (p1 ||| p2').
```
Listing 3.2: Operational semantics for protocols

```
Inductive leval_rxn : forall {w : nat}, c_state → @rxn str (w.-bv) → label →
@rxn str (w.-bv) → Prop :=
| LEvalRead :
    forall {w : nat} (state : c_state) (ch : str (w.-bv)) (v : w.-bv),
    state $? ch = Some (Some {| width := w;  value := v |}) →
    leval_rxn state (Read ch) (Receive ch {| width := w; value := v |}) (Ret v)
| LEvalSamp :
    forall {w : nat} (state : c_state) (dist : Dist (w.-bv)),
    forall v, In v (interpDist dist) →
    leval_rxn state (Samp dist) Silent (Ret v)
| LEvalBind :
    forall {w w': nat}, forall (state : c_state), forall init,
    forall (r : w.-bv → rxn (w'.-bv)), forall label, forall v,
    leval_rxn state init label (Ret v) →
    leval_rxn state (Bind (init) r) label (r v).
```
Listing 3.3: Operational semantics for reactions

Finally, `LEvalParsLeft` and `LEvalParsRight` allow us to lift a step to parallel composition. Crucial to these rules is a notion of a protocol's *support* (defined below), encoding the set of nonlocal channels in current use by a protocol. Importantly, we are only permitted to do the lifting if the supports of both sides of the parallel composition are disjoint. This restriction helps to bring our semantics in line with the constraints of IPDL's type system, which prohibits well-typed protocols from assigning more than one reaction to a channel. Moreover, support disjointness aids in simplifying proofs by ruling out cases in which protocols compete to output along the same channels.

```
Inductive supp_ipdl: @ipdl str → name_set → Prop :=
| SuppIPDLZero :
  supp_ipdl Zero [::]
| SuppIPDLOut :
  forall t o, forall (rxn : @rxn str t),
  supp_ipdl (o ::= rxn) [:: o]
| SuppIPDLPars :
  forall pr1 pr2 chs1 chs2,
  supp_ipdl pr1 chs1 →
  supp_ipdl pr2 chs2 →
  supp_ipdl (Par pr1 pr2) (chs1 ++ chs2)
| SuppIPDLNew :
  forall t,
  forall (new_ch : str t → ipdl),
  forall chs ch, supp_ipdl (new_ch ch) chs →
            supp_ipdl (New t new_ch) chs.

Definition disjoint_supp (pr1 pr2 : @ipdl str) : Prop :=
  forall lbls1 lbls2,
    supp_ipdl pr1 lbls1 →
    supp_ipdl pr2 lbls2 →
    (forall name, (In name lbls1 → ~ In name lbls2 ) ∧ (In name lbls2 → ~ In name lbls1)).
```
Listing 3.4: Support definition

`SuppIPDLNew` highlights another simplification we made to make proofs more tractable. Instead of adding the freshly generated local channel name, we discard it and simply add all output channels that exist independent of the choice of local channel name. Unfortunately, this feature precludes any form of reasoning about local channels generated by `New` but, in exchange, allows us to exploit convenient properties about our semantics that would have been infeasible to prove otherwise. One such example is the invariance of support disjointness. Let $\mathsf{Disj}$ denote a binary relation over IPDL protocols expressing the fact that two protocols have disjoint supports. Then, we have the following result:

**Theorem 1.** If $\mathsf{Disj}\ p_1\ p_2$ and $p_1 \to p_1'$ then $\mathsf{Disj}\ p_1'\ p_2$.

*Proof.* Induct on the proof tree of the small-step relation and appeal to the observation that no rule introduces new channel names to the support of a protocol or renames existing ones. Therefore, any name that exists in the support of $p_1'$ must also be inside the support of $p_1$. □

If we had included local channel names in the support, this property would be unprovable since a New constructor is permitted to add arbitrary channel names from outside of channel state, which could include those in the support of $p_2$.

# Chapter 4

# State Machine Representation

In this section, we detail our state machine representation.

## 4.1 State machine definition

In our implementation, a state machine is comprised of a step function and a state type. We let the state type be implicitly parameterized throughout this work. The objective of the step function is to signal for a potential I/O operation and output the next state given the current one. This is achieved by forcing step functions to emit a label indicating either a read from or a write to a channel, and a continuation that, when evaluated, reveals the next state.

```
Inductive transition_state {state : Type} : Type :=
    | GoNext : state → transition_state
    | Terminate : state → transition_state.

(* name is an alias for Coq's string type. *)
Inductive step_op : Type :=
    | SilentOp : step_op
    | ReadOp : name → step_op
    | WriteOp : name → dyn → step_op.

Definition step_range (state : Type) : Type :=
option (step_op * (dyn → @transition_state state)).

Definition stepT {state : Type} : Type := state → step_range state.
```

Listing 4.1: State machine definitions

We introduce a wrapper type over the state type to draw a distinction between passing and terminating states.

The `step_op` inductive type provides a means to interface with channel state without requiring an explicit channel state argument to the step function. If a step function outputs `ReadOp ch`, then we need only perform a lookup on the channel state for `ch` and thread the resulting data through the continuation. Likewise, if `WriteOp ch d` is emitted instead, we should update the channel state so that channel `ch` is associated with data `d`. As the name suggests, `SilentOp` indicates no such I/O operation should be performed, leaving the channel state untouched.

We chose this structure for a number of reasons. By guarding the next state behind a continuation, we create an explicit syntactic marker for channel reads. Had we instead opted for a representation whereby channel state was passed into the step function, and updated alongside program state, we would have no way to detect channel read events. Requiring I/O labels that parallel those used in our labeled IPDL semantics allows us to meaningfully bridge the two representations by reasoning about their mutual interactions with channel state, as we will see in the next chapter. Finally, wrapping the output in an optional type lets one differentiate between stalling and termination.

### 4.1.1 Low-level channel state

In order to capture the form of sequential memory access that is common in low-level settings, we introduce an alternative channel state representation built on top of Coq's native list type. With this representation, channel state is instantiated as an associative array, where each element is a key-value pair bundling a channel's name with its respective contents. To access and update channels, we use a fixpoint function to scan through the list in sequential order to find a match with the input key. Furthermore, we introduce functions for collecting channel names and converting to the finite-map representation.

```
Definition channel_pair : Type := name * (option dyn).

Definition channel_array : Type := list (channel_pair).

Fixpoint lookup_assoc {B : Type} (array : list (string * B)) (key : string)
: option B :=
  match array with
  | [::] ⇒ None
  | (a, b) :: tail ⇒ if String.eqb a key then Some b else (lookup_assoc tail key)
  end.

Fixpoint update_assoc {B : Type} (array : list (string * B)) (key : string) (v : B) :
list (string * B) :=
  match array with
  | [::] ⇒ [::]
  | (a, b) :: tl ⇒
  if String.eqb a key then (key, v) :: tl else (a, b) :: update_assoc tl key v
  end.

Fixpoint keys_of  (array : channel_array) : name_set :=
  match array with
  | [::] ⇒ [::]
  | (s, Some _) :: tail ⇒ s :: (keys_of tail)
  | (s, None) :: tail ⇒ s :: keys_of tail
  end.
Fixpoint to_cstate (ch_arr : channel_array) : c_state :=
  match ch_arr with
  | [:: ] ⇒ empty_cstate
  | (str, None) :: tl ⇒ to_cstate tl $+ (str, None)
  | (str, Some v) :: tl ⇒ to_cstate tl $+ (str, Some v)
  end.
```

Listing 4.2: Associative array implementation

## 4.2 Modeling concurrency

With what has been presented so far, we can model all I/O functionality that occurs inside IPDL reactions. However, to bring the state machine representation closer to parity with IPDL programs we must have an analogy to parallel composition. In what follows, we provide a means to compose state machines together and have them engage in concurrent interaction.

### 4.2.1 Multi-threaded state representation

We represent multi-threaded state programs with a product type, parameterized by two state types. Each "thread" of the computation is associated with a bit indicating whether it has reached termination. Additionally, we define a suite of helper functions for inspecting and updating the contents of a given thread.

```
Inductive threadProd {A B : Type} : Type :=
| Prod : A → B → threadProd.

Notation  "{ tr1 , tr2 }" := (Prod tr1 tr2) (at level 70).

Definition bit := bool.

Definition get_thread {A B : Type} (prod : @threadProd A B) (b : bit) : A + B :=
  match prod with
    | { x , y } ⇒ if b then inr y else inl x
  end.

Definition left_thread {A B : Type} (prod : @threadProd A B) : A :=
  match prod with
    | { x, y } ⇒ x
  end.

Definition right_thread {A B : Type} (prod : @threadProd A B) : B :=
  match prod with
  | { x, y } ⇒ y
  end.
```

```
Definition update_thread {A B : Type}
(prod : @threadProd A B) (val : A + B)  : threadProd :=
  match prod with
  | { l1, l2 } ⇒ match val with
                 | inl a ⇒ { a, l2 }
                 | inr b ⇒ { l1, b }
                 end
  end.

Definition threadState {state state' : Type} : Type :=
@threadProd (state * bit) (state' * bit).

Definition left_st {A B : Type} (threads : @threadState A B) : A :=
fst (left_thread threads).

Definition right_st {A B : Type} (threads : @threadState A B) : B :=
fst (right_thread threads).

Definition left_term {A B : Type} (threads : @threadState A B) : bool :=
snd (left_thread threads).

Definition right_term {A B : Type} (threads : @threadState A B) : bool :=
snd (right_thread threads).

Definition get_term {A B : Type} (threads : @threadState A B) : bit → bool :=
(fun i ⇒ match get_thread threads i with
    | inl (_, term) ⇒ term
    | inr (_, term) ⇒ term
    end).

Definition get_st {A B : Type} (threads : @threadState A B) : bit → A + B :=
(fun i ⇒ match get_thread threads i with
    | inl (a, _) ⇒ inl a
    | inr (b, _) ⇒ inr b
    end).
```

Listing 4.3: Multi-threaded state representation

### 4.2.2 Multi-threaded step function

Now that we have the state representation underway, we transition to our multi-threaded step function implementation. We assume the existence of a scheduler that outputs a bit representing which thread is ready to step. Coupled with a `step_table`, associating threads with their respective step functions, the `multi_step` function runs the appropriate step function on the thread selected by the scheduler. If it results in a legitimate step (which it should if the scheduler is operating correctly), we feedforward the I/O operation and a modified continuation leading to the next state, being careful to update the terminataion flags if the step results in a termination state. If both threads have terminated, then, accordingly, we mark termination of the entire state machine.

```
Definition schedulerT {state state' : Type} : Type :=  @threadState state state' → option bit.

Definition step_table  {state state' : Type} := @threadProd (@stepT state) (@stepT state').

Definition multi_step {state state': Type} (scheduler : @schedulerT state state')
(step_table : @step_table state state') (threads : @threadState state state')
: (step_range (@threadState state state')) :=
  let branch := scheduler threads in
      match branch with
      | Some i ⇒ match (get_thread step_table i)  with
                 | inl step ⇒ let a := left_st threads in
                              match step a with
                              | Some (op, cont) ⇒
                                Some (op, fun val ⇒ match cont val with
                                | GoNext st ⇒
                                    GoNext (update_thread threads (inl (st, false)))
                                | Terminate st ⇒
                                    let threads' := update_thread threads (inl (st, true)) in
                                                    if right_term threads then
                                                    Terminate threads' else GoNext threads'
                                end)
                              | None ⇒ None
                              end
                 | inr step' ⇒ let b := right_st threads in
                               match step' b with
                               | Some (op, cont) ⇒
                                 Some (op, fun val ⇒ match cont val with
                                    | GoNext st ⇒
                                      GoNext (update_thread threads (inr (st, false)))
                                    | Terminate st ⇒
                                      let threads' := update_thread threads (inr (st, true)) in
                                      if left_term threads then
                                      Terminate threads' else GoNext threads'
                                     end)
                               | None ⇒ None
                               end
                 end
         | None ⇒ None
  end.
```

Listing 4.4: Multi-threaded step function

# Chapter 5

# Refinement and Simulation

Now that we have presented our state-machine representation, we move to developing the formal machinery to connect state-machine programs to IDPL. To achieve this, we adapt proof techniques from process algebra to equate the behavior of state-machine code to that of IPDL. Namely, we will be appealing to a notion of *refinement* to show that when a state machine is able to make a step, we can make a matching step in IPDL that exhibits the same effect on the external environment. Furthermore, we state and prove an algebraic property of refinement, promoting a modular proof approach for concurrency by which we can break down a proof of a multithreaded program by proving a refinement property for each component thread.

## 5.1 Labeled transition system

To begin, we showcase a labeled transition system for state machines. Each rule of the `lstep_state` predicate dictates how a state machine should interact with channel state according to the output of the step function.

`StateSilent` marks a silent transition from one state to another when the step function outputs `SilentOp`, leaving the channel state untouched. Since no channel reads are required here, we enforce that the continuation is a constant function by quantifying over all possible inputs.

On a channel send, which is triggered when a step function outputs `WriteOp`, we check that the write destination is actually contained within the channel state before updating it. Once again, there are no channel reads occuring here, so the continuation ought to be constant.

Finally, when the step function signals a read, we ensure that the channel we want to read from is contained in the channel state. If so, we take the value paired with the channel name and thread it through the continuation to obtain the next state.

```
Inductive lstep_state {state : Type} : @stepT state → channel_array → state →
label → channel_array → @transition_state state → Prop :=
| StateSilent :
    forall (step : @stepT state), forall cont ch st st',
      step st = Some (SilentOp, cont)  →
      (* Quantification over all channel_state is necessary here to ensure that
      output is independent of the input *)
      (forall v, cont v = st') →
      lstep_state step ch st Silent ch st'
| StateSend :
  forall (step : @stepT state), forall write_to val cont ch ch' st st',
    step st = Some (WriteOp write_to val, cont) →
    In (keys_of ch) write_to →
    (forall v, cont v = st') →
    ch' = update_assoc ch write_to (Some val) →
    lstep_state step ch st (Send write_to val) ch' st'
| StateReceive :
  forall (step : @stepT state), forall read_from ch cont st st' val,
    step st = Some (ReadOp read_from, cont) →
    In (keys_of ch) read_from →
    lookup_assoc ch read_from = Some (Some val) →
    cont val = st' →
    lstep_state step ch st (Receive read_from val) ch st'.
```

Listing 5.1: State machine transition system

We derive a labeled transition system for IPDL from our small-step relation.

```
Definition lstep_ipdl (c1 : c_state) (p1 : @ipdl_bits str) (l :label)
(c2 :c_state) (p2 : @ipdl_bits str) :  Prop :=
leval_ipdl c1 c2 (bits_to_ipdl p1) l (bits_to_ipdl p2).
```

Listing 5.2: IPDL transition system

## 5.2 Simulation

To connect the behavior of state machines to that of IPDL programs, we introduce a notion of *simulation*. Intuitively, we say that a protocol simulates a state machine if, for every step that the state machine can make, we can make a matching step in the IPDL transition system that yields the same eventual channel state. Before getting into the formal definition, we take a brief pause to establish notation and terminology. For the rest of this document, we identify state machines as pairs of instances of an arbitrary type and a step function over that type $(st, \mathsf{step})$, where the term "state" is used to refer to the former. We overload the labeled transition arrow $\xrightarrow{l}$ for both the state-machine and IPDL transition systems, letting context resolve the question of which is being denoted. With that clarified, we present the following definition of simulation:

**Definition 1.** A ternary relation $R$ between channel state, state machines, and IPDL protocols is said to be a simulation when

- **Silent, nonterminating steps match up**: if $R\ ch\ (st, \mathsf{step})\ pr$ and $(ch, st) \to (ch', \mathsf{GoNext}\ st')$ then there exists protocol $pr'$ such that $(ch, pr) \to^* (ch', pr')$ and $R\ ch'\ (st', \mathsf{step})\ pr'$.

- **Nonsilent, nonterminating steps match up**: if $R\ ch\ (st, \mathsf{step})\ pr$ and $(ch, st) \xrightarrow{l} (ch', \mathsf{GoNext}\ st')$ for nonsilent $l$, there exists channel state $ch''$ and protocols $pr', pr''$ such that $(ch, pr) \to^* (ch'', pr'')$, $(ch'', pr'') \xrightarrow{l} (ch', pr')$, and $R\ ch'\ (st', \mathsf{step})\ pr'$.

- **Terminating steps match up**: if $R\ ch\ (st, \mathsf{step})\ pr$ and $(ch, st) \xrightarrow{l} (ch', \mathsf{Terminate}\ st')$ then there exists channel state $ch''$ and protocol $pr''$ such that $(ch, pr) \to^* (ch'', pr'')$, $(ch'', pr'') \xrightarrow{l} (ch', \mathsf{Zero})$, and $R\ ch'\ (st', \mathsf{step})\ \mathsf{Zero}$.

where $\to^*$ denotes the transitive-reflexive closure over the stepping relation.

Furthermore, such a relation gives rise to a pleasant property for comparing the two representations: if $R$ is a valid simulation relation and $R\ ch\ (st, \mathsf{step})\ pr$, then $(st, \mathsf{step})$ cannot exhibit any more behaviors than $pr$! In other words, if a simulation exists, we can be confident that every sequence of moves that a state machine may make can also be performed by IPDL. This notion is formalized through trace inclusion, and we prove it here. Let $\mathsf{GenState}\ (st, \mathsf{step})\ tr$ and $\mathsf{GenIPDL}\ pr\ tr$ be trace-generation predicates for state machines and protocols respectively (defined below), expressing the fact that, by collecting all nonsilent labels, we can generate a trace $tr$. Then,

**Theorem 2** (Trace inclusion)**.** Suppose $R$ is a valid simulation relation. If $\mathsf{GenState}\ (st, \mathsf{step})\ tr$ and $R\ ch\ (st, \mathsf{step})\ pr$, then $\mathsf{GenIPDL}\ pr\ tr$.

*Proof.* Induct on the proof tree of $\mathsf{GenState}$. The base case is when the trace is empty, in which case we can just apply the `GenIPDLNothing` constructor. In the inductive cases, we have $\mathsf{GenState}\ (st, \mathsf{step})\ tr$, a stepping relation $(ch, st) \xrightarrow{l} (ch', st')$, and need to show that $\mathsf{GenIPDL}\ pr\ (l ::\ tr)$. We proceed by selecting the appropriate arm of the simulation assumption that corresponds to the new labeled step and use that to tease out the matching sequence of IPDL steps that generates $l$. Then, we apply the inductive hypothesis. $\square$

```
Inductive generate_ipdl : c_state → @ipdl_bits str → list label → Prop :=
| GenIPDLNothing :
    forall ch prog,
    generate_ipdl ch prog [:: ]
| GenIPDLSilent :
    forall ch ch’ prog prog’ tr,
    lstep_ipdl ch prog Silent ch’ prog’ →
    generate_ipdl ch’ prog’ tr →
    generate_ipdl ch prog tr
| GenIPDLSend :
    forall ch ch’ prog prog’ data tr c,
    lstep_ipdl ch prog (Send c data) ch’ prog’ →
    generate_ipdl ch’ prog’ tr →
    generate_ipdl ch prog ((Send c data) :: tr)
| GenIPDLReceive :
    forall ch ch’ prog prog’ data tr c,
    lstep_ipdl ch prog (Receive c data) ch’ prog’ →
    generate_ipdl ch’ prog’ tr →
    generate_ipdl ch prog ((Receive c data) :: tr)
| GenIPDLLocal :
    forall ch ch' prog prog'  tr c,
    lstep_ipdl ch prog (LocalGen c) ch' prog' →
    generate_ipdl ch' prog' tr →
    generate_ipdl ch prog ((LocalGen c) :: tr).

Inductive generate_state {state : Type} : @stepT state →
channel_array → state → list label → Prop :=
| GenStNothing :
    forall step ch st,
    generate_state step ch st [:: ]
| GenStSilent :
    forall step ch ch' st st' tr,
    lstep_state step ch st Silent ch' (GoNext st') →
    generate_state step ch' st' tr →
    generate_state step ch st tr
| GenStSend :
    forall step ch ch' st st' data tr c,
    lstep_state step ch st (Send c data) ch' (GoNext st') →
    generate_state step ch' st' tr →
    generate_state step ch st ((Send c data) :: tr)
| GenStReceive :
    forall step ch ch' st st' data tr c,
    lstep_state step ch st (Receive c data) ch' (GoNext st') →
    generate_state step ch' st' tr →
    generate_state step ch st ((Receive c data) :: tr)
| GenStTerminateLabel :
    forall step ch ch' st l st',
    l <> Silent →
    lstep_state step ch st l ch' (Terminate st') →
    generate_state step ch st [:: l]
| GenStTerminateSilent :
    forall step ch ch' st st',
    lstep_state step ch st Silent ch' (Terminate st') →
    generate_state step ch st [:: ].
```

Listing 5.3: Trace-generation predicates

## 5.3 Refinement

We define refinement $(ch, (st, \mathsf{step})) \leqslant pr$ as there existing a simulation $R$ such that $R$ $ch$ $(st, \mathsf{step})$ $pr$. With this definition in hand, we proceed to state and prove an algebraic property of refinement, namely that, under certain conditions, refinement is a congruence for parallel composition.

Before we do so, we will focus on some of the simplifying assumptions we made to make proving such a property more tractable. For one, we require *channel independence* between $ch$ and $ch'$, meaning that no channel name in $ch$ can be found in $ch'$ and vice versa. Furthermore, $ch$ and $ch'$ must have unique channel names each. Imposing these conditions ensures that the concatenation of $ch$ and $ch'$ yields no duplicate channel names and the order of the concatenation does not affect the outcome of lookup and update operations.

We must also assume that any step functions that comprise the multithreaded state machine emit *proper operations*, I/O actions that respect the boundaries of channel state. Such a step function always requests to read from and write to a channel name that exists within the channel state. Without this condition, we would have no way of determining the location of a read or write in a composite state machine where the channel state is formed by concatenation.

Finally, if $(ch, S) \leqslant pr$ and $(ch', S') \leqslant pr'$, we must also ensure that the supports of $pr$ and $pr'$ are disjoint. In addition to preventing potential violations of IPDL's write-once semantics for channels, this condition rules out cases in which threads try to compete to write to the same channel.

**Theorem 3** (Parallel composition refinement)**.** Let $S = (st, \mathsf{step})$, $S' = (st', \mathsf{step}')$, and $S \oplus S'$ be the multithreaded state machine constructed by composing $S$ and $S'$ as detailed in the last chapter. If $(ch, S) \leqslant pr$ and $(ch', S') \leqslant pr'$, then $(ch \texttt{ ++ } ch', S \oplus S') \leqslant pr \| pr'$ provided that

- $ch$ and $ch'$ are independent.
- $\mathsf{step}$ and $\mathsf{step}'$ emit proper operations with respect to $ch$ and $ch'$ respectively.
- $pr$ and $pr'$ have disjoint supports.

*Proof.* According to the refinement assumptions, there exist simulations $R$ and $R'$ such that $R\ ch\ S\ pr$ and $R'\ ch'\ S'\ pr'$. We construct an inductive predicate $R \parallel R'$ (defined below as `R_pars` in Coq) that we claim serves as a valid simulation relation.

```
Inductive R_pars {state state': Type} {step : @stepT state} {step' : @stepT state'}
{R: channel_array → state → @ipdl_bits str → Prop}
{R': channel_array → state' → @ipdl_bits str → Prop} :
channel_array → (@threadState state state') → @ipdl_bits str → Prop :=
| RParsInit :
  forall ch ch' st st' pr pr' b b',
    R ch st pr  →
    R' ch' st' pr' →
    channel_independence ch ch' →
    disjoint_supp (bits_to_ipdl pr) (bits_to_ipdl pr') →
    proper_op step ch →
    proper_op step' ch' →
    ch_unique ch →
    ch_unique ch' →
    (b = true → pr = ZeroBits) →
    (b' = true → pr' = ZeroBits) →
    R_pars (ch ++ ch') ({(st , b) ,(st' , b')}) (ParBits pr pr')
```

```
| RParsTerm :
  forall ch ch' st st',
    R ch st ZeroBits →
    R' ch' st' ZeroBits →
    channel_independence ch ch' →
    ch_unique ch →
    ch_unique ch' →
    R_pars (ch ++ ch') ({(st, true), (st', true)}) ZeroBits.
```

Listing 5.4: Explicit simulation relation

In order to establish this claim, we need to show that each arm of the simulation definition is satisfied by $R \parallel R'$, yielding four proof obligations:

1. For all state machines $S_0, S_1, S_2$, channel states $ch_0, ch_1, ch_2$, and IPDL programs $pr_0, pr_1$,
   if $(R \parallel R')$ $(ch_0$ ++ $ch_1)$ $(S_0 \oplus S_1)$ $(pr_0 \| pr_1)$ and a silent step is possible $(ch_0$ ++ $ch_1, S_0 \oplus S_1) \to (ch_2, \mathsf{GoNext}\ S_2)$ then there must exist an IPDL program $pr_2$ that we can reach from $p_0 \| p_1$, $(ch_0$ ++ $ch_1, pr_0 \| pr_1) \to^* (ch_2, pr_2)$ and for which $(R \parallel R')$ $ch_2$ $S_2$ $pr_2$ holds.

2. For all state machines $S_0, S_1, S_2$, channel states $ch_0, ch_1, ch_2$, and IPDL programs $pr_0, pr_1$,
   if $(R \parallel R')$ $(ch_0$ ++ $ch_1)$ $(S_0 \oplus S_1)$ $(pr_0 \| pr_1)$ and we make a a nonsilent step, $(ch_0$ ++ $ch_1, S_0 \oplus S_1) \xrightarrow{l} (ch_2, \mathsf{GoNext}\ S_2)$, there must exist intermediate states $ch_2', pr_2'$, and final state $pr_2$ such that $(ch_0$ ++ $ch_1, pr_0 \| pr_1) \to^* (ch_2', pr_2')$, $(ch_2', pr_2') \xrightarrow{l} (ch_2, pr_2)$, and $(R \parallel R')$ $ch_2$ $S_2$ $pr_2$ holds.

3. For all state machines $S_0, S_1, S_2$, channel states $ch_0, ch_1, ch_2$, and IPDL programs $pr_0, pr_1$,
   if $(R \parallel R')$ $(ch_0$ ++ $ch_1)$ $(S_0 \oplus S_1)$ $(pr_0 \| pr_1)$ and we make a terminating step, $(ch_0$ ++ $ch_1, S_0 \oplus S_1) \xrightarrow{l} (ch_2, \mathsf{Terminate}\ S_2)$, there must exist intermediate $ch_2', pr_2'$ such that $(ch_0$ ++ $ch_1, pr_0 \| pr_1) \to^* (ch_2', pr_2')$, $(ch_2', pr_2') \xrightarrow{l} (ch_2, \mathsf{Zero})$, and $(R \parallel R')$ $ch_2$ $S_2$ $\mathsf{Zero}$ holds.

4. $(R \parallel R')$ $(ch$ ++ $ch')$ $(S \oplus S')$ $(pr \| pr')$

For (1)-(3), we proceed by unfolding the definition of the `multi_step` step function covered in the previous chapter and perform a case split over the termination flags of each thread and then the output of the scheduler. After pruning out degenerate cases, we identify the thread that made a step. Since only one thread may step at a given time, either $S_0$ or $S_1$ may step but not both. Without loss of generality, assume that $S_0$ stepped to $S_0'$, resulting in $S_2 = S_0' \oplus S_1$. The rest of the proof diverges depending on which obligation we aim to prove.

If we are proving (1), we use the silent-step assumption in conjunction with $(ch, S) \leqslant pr$ to deduce that there is a $pr_0'$ such that $(ch_0, pr_0) \to^* (ch_0, pr_0')$. The $pr_2$ that proves the statement would then be $pr_0' \| pr_1$.

(2)-(3) proceed similarly. Like (1), we use the refinement assumption with the stepping assumption to tease out the matching step in IPDL and locate the appropriate $ch'_2, pr'_2$, and $pr_2$ that emerges from $pr_0$.

To prove (4), we simply apply the `RParsInit` constructor and use the assumptions laid out in the theorem statement.

To demonstrate that $(R \parallel R')$ is still satisfied after each step in (1)-(3), we make use of the convenient property that all the requisite preconditions are actually invariants with respect to the state-machine transition system. Channel independence is preserved after a step since state machines are not allowed to remove bindings from the channel state or rename existing channel names. For similar reasons, `ch_unique` and `proper_op` are both invariants. As we established in chapter 3, `disjoint_supp` is also invariant over the IPDL transition system.

□

In some stages of the proof, we need to be able to lift single-threaded steps into environments where the channel state has been expanded via concatenation. To do so, we make liberal use of the following lemmas:

**Lemma 1** (Lifting in reactions)**.** Let $r$ be an IPDL reaction. If $(ch, r) \xrightarrow{l} (ch, r')$ then, for all $ch'$, $(ch \texttt{ ++ } ch', r) \xrightarrow{l} (ch \texttt{ ++ } ch', r')$.

*Proof.* Induct on the proof tree of the small-step reaction relation. The only I/O that occurs at the reaction level are read operations, and no lookup outcome is affected by the concatenation due to the fact that the lookup operation performs a linear scan of the channel state. □

**Lemma 2** (Lifting in protocols)**.** Let $pr$ be an IPDL protocol. If $(ch, pr) \xrightarrow{l} (ch', pr')$ and $l$ does not correspond to local channel generation, then, for all $ch'$, $(ch \texttt{ ++ } ch', pr) \xrightarrow{l} (ch \texttt{ ++ } ch', pr')$.

*Proof.* Induct on the proof tree of the small-step protocol relation. We can resolve most cases through direct application of a rule of the small-step relation or by appealing to the inductive hypothesis. In the case of channel assignment $(o := r)$, we employ Lemma 1. □

# Chapter 6

# Future Work

In this thesis, we have developed robust conceptual tools to verify the correctness of a low-level implementation of a cryptographic protocol. However, more work must be done to account for the full set of features IPDL provides and to extract the most practical benefit from verified state-machine code.

## 6.1 Local channel generation

When introducing the main constructs of the IPDL language in chapter 3, we defined the $\mathsf{new}\ c : \tau\ \mathsf{in}\ P$ constructor as one that allocates a fresh channel $c$ for private use in $P$. However, our IPDL semantics and state-machine representation does not adequately implement access privileges so that one thread cannot read from another thread's local channel. The channel state makes no distinction betweeng global and local channels so, from any given thread's point of view, a local channel belonging to another thread appears the same as any other. This stems from the fact that there is no sense of "where a channel was defined" built into channel state. To address this issue, one would need to rework the channel state and operational semantics to track channel scope and ensure that all reads in reactions are in scope.

## 6.2 Proof-generating compilation

To spare code authors from the labor of writing state-machine implementations by hand, one could extend our work to automatically derive such code via compilation. Additionally, one may not even need to certify the implementation separately. In the past, Coq's tactic engine has been used successfully to implement *proof-generating compilation* by framing compilation as a proof search for equivalence between source and target programs [10]. Ideally, without much extension, one could use the notion of simulation we have presented in this thesis as an equivalence relation and use that as a basis for compiling IPDL programs to low-level state machine code.

## 6.3 Lowering to C

Finally, as a way to broaden the impact from the work presented in this thesis, one could consider ways to connect the state-machine representation to a popular low-level programming language like C. Naturally, this would involve determining an appropriate memory layout scheme for channel state and rephrasing the various components of the step function as idiomatic C constructs.

# References

[1] "SSL/TLS MITM vulnerability (CVE-2014-0224)," 2014. URL: https://openssl-library.org/news/secadv/20140605.txt (visited on 08/13/2024).

[2] A. Langley. "Early ChangeCipherSpec Attack (05 Jun 2014)," ImperialViolet. (2014), URL: https://www.imperialviolet.org/2014/06/05/earlyccs.html (visited on 08/13/2024).

[3] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983. DOI: 10.1109/TIT.1983.1056650.

[4] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *Principles of Security and Trust*, P. Degano and J. D. Guttman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–29, ISBN: 978-3-642-28641-4.

[5] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Found. Trends Priv. Secur.*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016, ISSN: 2474-1558. DOI: 10.1561/3300000004. URL: https://doi.org/10.1561/3300000004.

[6] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701, ISBN: 978-3-642-39799-8.

[7] V. Cheval, S. Kremer, and I. Rakotonirina, "Deepsec: Deciding equivalence properties in security protocols theory and practice," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 529–546. DOI: 10.1109/SP.2018.00033.

[8] J. Gancher, K. Sojakova, X. Fan, E. Shi, and G. Morrisett, "A core calculus for equational proofs of cryptographic protocols," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, Jan. 2023. DOI: 10.1145/3571223. URL: https://doi.org/10.1145/3571223.

[9] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1202–1219. DOI: 10.1109/SP.2019.00005.

[10] M. Ikebuchi, A. Erbsen, and A. Chlipala, "Certifying derivation of state machines from coroutines," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022. DOI: 10.1145/3498685. URL: https://doi.org/10.1145/3498685.

[11] J. Protzenko, B. Parno, A. Fromherz, *et al.*, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 983–1002. DOI: 10.1109/SP40000.2020.00114.

[12] R. Canetti, *Universally composable security: A new paradigm for cryptographic protocols*, Cryptology ePrint Archive, Paper 2000/067, https://eprint.iacr.org/2000/067, 2000. URL: https://eprint.iacr.org/2000/067.

[13] G. P. Huet, "The undecidability of unification in third order logic," *Information and Control*, vol. 22, no. 3, pp. 257–267, 1973, ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(73)90301-X. URL: https://www.sciencedirect.com/science/article/pii/S001999587390301X.

[14] C. L. Lucchesi, "The undecidability of the unification problem for third order languages," *Report CSRR*, vol. 2059, pp. 129–198, 1972.

[15] W. D. Goldfarb, "The undecidability of the second-order unification problem," *Theoretical Computer Science*, vol. 13, no. 2, pp. 225–230, 1981, ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(81)90040-2. URL: https://www.sciencedirect.com/science/article/pii/0304397581900402.