

IPDL User Manual

Kristina Sojakova
Vrije Universiteit Amsterdam
k.sojakova@vu.nl

Mihai Codescu
Research Group of the IPDL Project - NLNet
mscodescu@gmail.com

Contents

1	Introduction	5
2	The IPDL Logic	7
2.1	Background: Simulation-Based Security	7
2.2	IPDL Syntax	7
2.3	IPDL Typing	8
2.4	Equational Logic of IPDL	8
2.4.1	Exact Equality in IPDL	9
2.4.2	Approximate Equality in IPDL	10
2.5	Hello World Example	11
3	The IPDL Prover	17
3.1	Maude	17
3.2	Syntax	18
3.3	Normal Forms	19
3.4	Families of Channels	21
3.5	Exact Equality Proofs	22
3.5.1	Prover State	24
3.6	Derived Rules	25
3.7	Operators on protocols	26
3.8	Proof Steps for Exact Equality	28
3.9	Approximate Proofs	49
3.9.1	Approximate Congruence	49
3.9.2	Approximate Equality	51
3.10	Working on Complex Proofs: Trying Steps	53
3.11	Reusing Proofs	55
3.12	Evaluating the Results: IPDL Commands	57
3.13	Modularity	58
3.14	Revisiting the Hello World Example	59
3.15	Case Study: Authenticated-To-Secure Channel	61
4	Conclusions	67

Chapter 1

Introduction

Imagine you and your friends want to calculate your average salary without any of you revealing your specific income to each other. In a traditional world, you would need a trusted third party – like an accountant – to take everyone’s numbers, do the math, and report the result. *Secure Multi-Party Computation (MPC)* is a subfield of cryptography that eliminates the need for that middleman. It allows a group of parties to jointly compute a function over their inputs while keeping those inputs private. At its core, MPC is about distributed trust. Instead of keeping data in one “vault”, the data is broken into encrypted fragments (often called “shares”) and distributed across multiple participants. No single participant can see the whole picture, but together they can perform mathematical operations on the pieces. If an MPC protocol is secure, leaking the shares of any one party does not compromise the secrets of the remaining parties. For example, layer 2 blockchains use MPC technology to split the private key of an MPC wallet among multiple parties for enhanced security.

However, the inherent complexity of MPC protocols introduces the risk that the protocol itself is insecure. State-of-the-art MPC protocols have been broken before: in 2023, Fireblocks discovered a vulnerability in the widely-used Gennaro and Goldfeder protocols GG18 and GG20 that allowed the attacker to extract the secret key in its entirety [7]. To increase the trustworthiness of the MPC protocols that our sensitive data rely on, we need machine-checked mathematical proofs that certify them as secure. The IPDL process calculus – short for Interactive Probabilistic Dependency Logic – was specifically designed to support formal verification of distributed message-passing protocols such as MPC.

A crucial part of a cryptographic proof of security is reasoning about the runtime of adversaries. To see why runtime analysis is important, consider the following example. The decisional Diffie-Hellman assumption on a cyclic group of order q with generator g says that if we uniformly (and independently) sample $a, b \in \mathbb{Z}_q$, the element g^{ab} will be indistinguishable from a randomly sampled element of G , even if the adversary knows the values g^a, g^b . That is, the protocol P that uniformly samples a, b and returns the triple (g^a, g^b, g^{ab}) is computationally indistinguishable from the protocol P' that uniformly samples a, b, c and returns the triple (g^a, g^b, g^c) .

Now assume Q is a protocol that uses the Diffie-Hellman triple produced by P to construct a one-time pad that delivers a single secret message. We parallel-compose P with Q to obtain the One-Time Pad protocol $P \parallel Q$. As part of showing that our protocol is secure, we need to show that the probability of an adversary A distinguishing $P \parallel Q$ from $P' \parallel Q$ (the distinguishing advantage of A) is bounded by the probability of another adversary B distinguishing P from P' . The adversary B is formed by absorbing the protocol Q into A . That is, the interaction of B with P (or P') mimics the interaction of the original adversary A with $P \parallel Q$ (or $P' \parallel Q$), by having B perform the computations originally performed by Q .

If the runtime of the reduced adversary B is large – for example if it is bounded by a polynomial that contains a factor of $O(z^{500})$, where z is the “size” of the absorbed protocol Q – then B might be able to distinguish P from P' with probability 1. Consequently, the bound on the distinguishing advantage of the original adversary A is also 1, and our security result for the protocol $P \parallel Q$ is meaningless.

The IPDL proof system formally reasons about the runtime of adversaries by connecting the IPDL process calculus to a low-level computational semantics in terms of Turing Machines [6]. The prover that we present in this manual implements IPDL in the term-rewriting system Maude [2]. It delivers formally verified bounds on the probability that an adversary with bounded computational resources breaks the protocol. These bounds are sound with respect to the aforementioned Turing Machine semantics. To enable large-scale security proofs for interactive message-passing protocols, the tool provides a domain-specific logic (DSL) as a layer of abstraction for reasoning on a high level. Manipulations such as inlining the computation from one channel to another or removing parts of the protocol that have become unused after simplification only require a few lines of code. As such proof steps typically form the bulk of a proof for an MPC protocol, the DSL significantly reduces proof effort. The proofs are fast: our mid-size case studies run in a few seconds and a large-sized case study of over 15k lines of code takes under five minutes.

IPDL comes with the following limitations. It only considers static communication topologies, where the set-up of communication channels among parties and with the adversary is known ahead of time. It currently only targets static corruption, where corrupt parties are known ahead of time (an adversary cannot dynamically corrupt a party during the run of the protocol). Furthermore, protocols exhibiting threshold behavior (such as consensus protocols) are currently out of scope for IPDL. IPDL also does not support rewinding (resetting part of the protocol to a previous state to compare different potential outcomes).

The homepage of the IPDL prover is <https://ipdl.cs.vu.nl/>. There one can find information on how to run the tool, a list of case studies and more references.

Roadmap. Some parts of the user manual cover more advanced topics and can be skipped on a first lecture. A beginner's path includes sections 2.1, 2.2, 2.4.1, 2.5, 3.2, 3.3, 3.8, and 3.14 (comparing with 3.12 for the DSL constructs used in the proof of the Hello World example).

Acknowledgements We thank Ionut Tutu for helping us integrate IPDL in SpeX and Ruben Rubio for providing a Maude bugfix. Ionut Tutu and Engin Volkan provided valuable feedback on this user manual. This work was supported by NGI Zero Core, a fund established by NLnet with financial support from the European Commission's Next Generation Internet program.

Chapter 2

The IPDL Logic

2.1 Background: Simulation-Based Security

Simulation-based security relates the behavior of a protocol to that of an idealization, where cryptographic mechanisms are replaced by trusted resources. In this sense, an idealization is a specification for the behavior that the real-world protocol aims to approximate with cryptographic methods. For example, a real-world protocol utilizes encryption to securely send a message from Alice to Bob over a public network. The idealization instead relies on a trusted third party that securely obtains the message from Alice and forwards it to Bob.

The simulation-based paradigm, as employed *e.g.*, in Universal Composability (UC) [1] and Constructive Cryptography [4] is powerful and unifies other security notions such as secrecy and integrity. A protocol is an interactive system consisting of *parties*, *e.g.*, Alice and Bob, and *functionalities*, *e.g.*, a public network that forwards messages from Alice to Bob, or a key generation mechanism that randomly generates a secret key and securely delivers it to Alice and Bob. In simulation-based security, we prove that a protocol is secure by showing that it is computationally indistinguishable from its idealization combined with a *simulator*. The simulator reproduces the behavior of the original protocol by interacting with the idealization, and computational indistinguishability means that any adversary with computationally bounded resources cannot tell whether it is interacting with the protocol itself or with its idealization via the simulator. In the IPDL prover, we prove a protocol secure by: 1) encoding the protocol, its idealization, and the simulator; and 2) using sound equational rewrites to show computational indistinguishability between the protocol and its idealization combined with the simulator. Currently, simulators are explicitly supplied by the user. In future work, we plan to extend the prover to automatically synthesize simulators.

2.2 IPDL Syntax

IPDL is a process calculus for distributed cryptographic protocols (*e.g.*, MPC) that enables one to prove simulation-based security results similar to those analyzed in UC [1]. IPDL *protocols* are composed of mutually interacting *reactions*, which are sequential monadic programs that probabilistically compute an *expression*. In the context of a protocol, a reaction operates on a unique *channel* and may read from other channels, thereby utilizing computations coming from other reactions. The syntax and judgements of IPDL are outlined in Figure 2.1, and Table 2.1, respectively, and are parameterized by a user-defined *signature*. A signature Σ specifies the base types and the built-in (probabilistic) functions:

Definition 1 (Signature). *A signature Σ consists of:*

- *type constants* $\mathbf{t}$,
- *function symbols* $\mathbf{f} : \sigma \rightarrow \tau$, and
- *distribution symbols* $\mathbf{d} : \sigma \rightarrow \tau$.

We have a minimal set of data types, consisting of the unit type $\mathbf{1}$, Booleans, products, and type constants $\mathbf{t}$ drawn from the signature Σ . Expressions are used for non-probabilistic computations, and are standard. The expression $\checkmark$ denotes the canonical inhabitant of the unit type and $\mathbf{app}_{\sigma \rightarrow \tau} \mathbf{f} \ e$ denotes the application of the function $\mathbf{f} : \sigma \rightarrow \tau$ declared in Σ to an expression e . All expressions in IPDL are interpreted as bitstrings of a fixed length given by their type, so we annotate the operations $\mathbf{fst}_{\tau \times \sigma}$ and $\mathbf{snd}_{\tau \times \sigma}$ with the type of the pair to determine how to split the pair into two. A variable context Γ is a list of variables annotated by types. A substitution $\theta : \Gamma_1 \rightarrow \Gamma_2$ between variable contexts maps each variable in Γ_2 to an expression in Γ_1 of the same type.

As reactions are monadic programs, we have the standard monadic operations of return $\mathbf{ret} \ e$ and bind $x : \sigma \leftarrow R; S$. The reaction $\mathbf{samp}_{\sigma \rightarrow \tau} \mathbf{d} \ e$ denotes the application of the probabilistic function $\mathbf{d} : \sigma \rightarrow \tau$ declared in Σ to an expression e . The reaction $\mathbf{read}(c : \tau)$ denotes the read of a value of type τ from the channel c . Finally, we have the conditional *if* e *then* R_1 *else* R_2 . At the protocol level, we have the trivial protocol $\mathbf{0}$, the single-channel protocol $\mathbf{o} := R$ that assigns a reaction R to the channel

o , the parallel composition $P \parallel Q$ of two protocols, and the spawning $\text{new } o : \tau$ in P of a new internal channel o of type τ for use in P . A channel context Δ is a list of channel names annotated by types. An embedding $\phi : \Delta_1 \rightarrow \Delta_2$ between channel contexts maps each channel name in Δ_2 to a unique channel name in Δ_1 of the same type. That is, an embedding $\phi : \Delta_1 \rightarrow \Delta_2$ is an injective type-preserving mapping from Δ_2 to Δ_1 .

For readability, we often omit the type of the bound variable in a reaction bind, and write $\text{read } c$ instead of $\text{read}(c : \tau)$.

Syntax of IPDL

Variables	x, y, z	
Channels	i, o, c	
Channel Sets	I, O	$::= \{c_1, \dots, c_n\}$
Type Contexts	Γ	$::= \cdot \mid \Gamma, x : \tau$
Channel Contexts	Δ	$::= \cdot \mid \Delta, c : \tau$
Data Types	τ, σ	$::= \mathbf{t} \mid 1 \mid \text{Bool} \mid \tau_1 \times \tau_2$
Expressions	e	$::= \text{var}(x : \tau) \mid \checkmark \mid \text{true} \mid \text{false} \mid \text{app}_{\sigma \rightarrow \tau} f e \mid (e_1, e_2) \mid \text{fst}_{\sigma \times \tau} e \mid \text{snd}_{\sigma \times \tau} e$
Reactions	R, S	$::= \text{ret } e \mid \text{samp}_{\sigma \rightarrow \tau} d e \mid \text{read}(c : \tau) \mid \text{if } e \text{ then } R_1 \text{ else } R_2 \mid x : \sigma \leftarrow R; S$
Protocols	P, Q	$::= 0 \mid o := R \mid P \parallel Q \mid \text{new } o : \tau \text{ in } P$
Substitutions	θ	$::= \Gamma_1 \rightarrow \Gamma_2$
Embeddings	ϕ	$::= \Delta_1 \rightarrow \Delta_2$

Figure 2.1: Syntax of IPDL.

Table 2.1: Summary of IPDL Typing and Equality Rules

Judgment Form	Description	Reference
$\Gamma \vdash e : \tau$	Expression Typing	Fig. 2.2
$\Delta; \Gamma \vdash R : I \rightarrow \tau$	Reaction Typing	Fig. 2.3
$\Delta \vdash P : I \rightarrow O$	Protocol Typing	Fig. 2.4
$\Gamma \vdash e_1 = e_2 : \tau$	Expression Equality	Fig. 2.5
$\Delta; \Gamma \vdash R_1 = R_2 : I \rightarrow \tau$	Reaction Equality	Fig. 2.6, 2.7
$\Delta \vdash P_1 = P_2 : I \rightarrow O$	Exact Protocol Equality	Fig. 2.8, 2.9
$\Delta \vdash P \cong Q : I \rightarrow O$ ctxt ψ for axiom $\# k$	Approximate Congruence	Fig. 2.10
$\Delta \vdash P \approx Q : I \rightarrow O$ count ξ ctxt ψ	Approximate Equality	Fig. 2.11

2.3 IPDL Typing

The typing $\Gamma \vdash e : \tau$ for expressions is standard, see Figure 2.2. Figure 2.3 shows the typing rules for reactions, where we implicitly assume that $I \subseteq \Delta$. Informally, $\Delta; \Gamma \vdash R : I \rightarrow \tau$ holds when R uses variables from Γ , reads from channels in I typed according to Δ , and returns a value of type τ . Figure 2.4 gives the typing rules for protocols, where we implicitly assume that $I, O \subseteq \Delta$. Informally, $\Delta \vdash P : I \rightarrow O$ holds when P uses inputs in I to assign reactions to the channels in O , all typed according to Δ . Channel assignment $o := R$ has the type $I \rightarrow \{o\}$ when R is well-typed with an empty variable context, making use of inputs from I as well as of o . We allow R to read from its own output o to express divergence: the protocol $o := \text{read } o$ cannot reduce, which is useful for (conditionally) deactivating certain outputs. The typing rule for parallel composition $P \parallel Q$ states that P may use the outputs of Q as inputs while defining its own outputs, and vice versa. Importantly, the typing rules ensure that the outputs of P and Q are disjoint so that each channel carries a unique reaction. Finally, the rule for channel generation allows a protocol to select a fresh channel name o , assign it a type τ , and use it for internal computation and communication. Protocol typing plays a crucial role for modeling security. Simulation-based security in IPDL is modeled by the existence of a *simulator* with an appropriate typing judgment, $\Delta \vdash \text{Sim} : I \rightarrow O$. Restricting the behavior of Sim to only use inputs along I is necessary to rule out trivial results (*e.g.*, Sim simply copies a secret from the specification).

2.4 Equational Logic of IPDL

The equational logic of IPDL comes in two flavors: an *exact* equational logic for proving perfect equality between protocols, and an *approximate* equational logic for proving computational indistinguishability. We now describe each in turn.

Typing for expressions: $\Gamma \vdash e : \tau$					
$\frac{x : \tau \in \Gamma}{\Gamma \vdash \text{var}(x : \tau) : \tau}$	$\frac{}{\Gamma \vdash \checkmark : 1}$	$\frac{}{\Gamma \vdash \text{true} : \text{Bool}}$	$\frac{}{\Gamma \vdash \text{false} : \text{Bool}}$	$\frac{f : \sigma \rightarrow \tau \in \Sigma \quad \Gamma \vdash e : \sigma}{\Gamma \vdash \text{app}_{\sigma \rightarrow \tau} f e : \tau}$	
$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$		$\frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \text{fst}_{\sigma \times \tau} e : \sigma}$		$\frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \text{snd}_{\sigma \times \tau} e : \tau}$	

Figure 2.2: Typing for IPDL expressions.

Typing for reactions: $\Delta; \Gamma \vdash R : I \rightarrow \tau$					
$\frac{\Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \text{ret } e : I \rightarrow \tau}$	$\frac{d : \sigma \twoheadrightarrow \tau \in \Sigma \quad \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \text{samp}_{\sigma \rightarrow \tau} d e : I \rightarrow \tau}$	$\frac{c : \tau \in \Delta \quad c \in I}{\Delta; \Gamma \vdash \text{read}(c : \tau) : I \rightarrow \tau}$			
$\frac{\Gamma \vdash e : \text{Bool} \quad \Delta; \Gamma \vdash R_1 : I \rightarrow \tau \quad \Delta; \Gamma \vdash R_2 : I \rightarrow \tau}{\Delta; \Gamma \vdash \text{if } e \text{ then } R_1 \text{ else } R_2 : I \rightarrow \tau}$		$\frac{\Delta; \Gamma \vdash R : I \rightarrow \sigma \quad \Delta; \Gamma, x : \sigma \vdash S : I \rightarrow \tau}{\Delta; \Gamma \vdash (x : \sigma \leftarrow R; S) : I \rightarrow \tau}$			

Figure 2.3: Typing for IPDL reactions.

2.4.1 Exact Equality in IPDL

The bulk of the reasoning in IPDL is done using exact equalities. At the expression level, we assume an ambient finite set of axioms of the form $\Gamma \vdash e_1 = e_2 : \tau$, where $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. The rules for expression equality are standard, see Figures 2.5. At the reaction level, we similarly assume an ambient finite set of axioms of the form $\Gamma \vdash R_1 = R_2 : \tau$, where $\cdot; \Gamma \vdash R_1 : \emptyset \rightarrow \tau$ and $\cdot; \Gamma \vdash R_2 : \emptyset \rightarrow \tau$. The absence of any input channels means that the reactions R1 and R2 are restricted to sampling from distributions and calling functions. We will therefore refer to axioms of this form as distribution-level axioms. The rules for reaction equality are shown in Figures 2.6 and 2.7, and implicitly assume that $I \subseteq \Delta$. The rules ensure that reactions form a *commutative monad*: we have

$$(x \leftarrow R_1; y \leftarrow R_2; S(x, y)) = (y \leftarrow R_2; x \leftarrow R_1; S(x, y))$$

whenever R_2 does not depend on x and R_1 does not depend on y . All expected equalities for commutative monads hold for reactions, including the usual monad laws and congruence of equality under monadic bind. The SAMP-PURE rule allows us to drop an unused sampling, and the READ-DET rule allows us to replace two reads from the same channel by a single one. The rules IF-LEFT, IF-RIGHT, and IF-EXT allow us to manipulate conditionals.

At the protocol level, we again assume an ambient finite set of axioms of the form $\Delta \vdash P_1 = P_2 : I \rightarrow O$, where $\Delta \vdash P_1 : I \rightarrow O$ and $\Delta \vdash P_2 : I \rightarrow O$. We use these axioms to specify user-defined functional assumptions, *e.g.*, the correctness of decryption. Exact protocol equalities allow us to reason about communication between subprotocols and functional correctness, and to simplify intermediate computations. Exact equality of protocols implies perfect computational indistinguishability against an arbitrary adversary. The rules for the exact equality of protocols are in Figures 2.8, 2.9, and implicitly assume that $I, O \subseteq \Delta$.

The COMP-NEW rule allows us to permute parallel composition and the creation of a new channel, and the same as *scope extrusion* in process calculi [5]. The ABSORB-LEFT rule allows us to discard a component in a parallel composition if it has no outputs; this allows us to eliminate internal channels once they are no longer used. The DIVERGE rule allows us to simplify diverging reactions: if a channel reads from itself and continues as an arbitrary reaction R , then we can safely discard R as we will never reach it in the first place. The three (un)folding rules FOLD-IF-LEFT, FOLD-IF-RIGHT, and FOLD-BIND allow us to simplify composite reactions by bringing their components into the protocol level as separate internal channels. The rule SUBSUME states that channel dependency is transitive: if we depend on o_1 , and o_1 in turn depends on o_0 , then we also depend on o_0 , and this dependency can be made explicit. The SUBST rule allows us to inline certain reactions into **read** commands. Inlining $o_1 := R_1$ into $o_2 := x \leftarrow \text{read } o_1; R_2$ is sound provided R_1 is *duplicable*: observing two independent results of evaluating R_1 is equivalent to observing the same result twice. This side condition is easily discharged whenever R_1 does not contain probabilistic sampling. Finally, the DROP rule allows dropping unused reads from channels in certain situations. Due to timing dependencies among channels, we only allow dropping reads from the channel $o_1 := R_1$ in the context of $o_2 := _ \leftarrow \text{read } o_1; R_2$ when we have that $(_ \leftarrow R_1; R_2) = R_2$. This side condition is met in particular whenever all reads present in R_1 are also present in R_2 .

Typing for protocols: $\Delta \vdash P : I \rightarrow O$		
$\frac{}{\Delta \vdash 0 : I \rightarrow \emptyset}$	$\frac{o : \tau \in \Delta \quad o \notin I \quad \Delta; \cdot \vdash R : I \cup \{o\} \rightarrow \tau}{\Delta \vdash (o := R) : I \rightarrow \{o\}}$	$\frac{\Delta \vdash P : I \cup O_2 \rightarrow O_1 \quad \Delta \vdash Q : I \cup O_1 \rightarrow O_2}{\Delta \vdash P \parallel Q : I \rightarrow O_1 \cup O_2}$
$\frac{\Delta, o : \tau \vdash P : I \rightarrow O \cup \{o\}}{\Delta \vdash (\text{new } o : \tau \text{ in } P) : I \rightarrow O}$		

Figure 2.4: Typing for IPDL protocols.

Equality for expressions: $\Gamma \vdash e_1 = e_2 : \tau$			
$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e = e : \tau} \text{ REFL}$	$\frac{\Gamma \vdash e_1 = e_2 : \tau}{\Gamma \vdash e_2 = e_1 : \tau} \text{ SYM}$	$\frac{\Gamma \vdash e_1 = e_2 : \tau \quad \Gamma \vdash e_2 = e_3 : \tau}{\Gamma \vdash e_1 = e_3 : \tau} \text{ TRANS}$	$\frac{\Gamma \vdash e_1 = e_2 : \tau \text{ axiom}}{\Gamma \vdash e_1 = e_2 : \tau} \text{ AXIOM}$
$\frac{\theta : \Gamma_1 \rightarrow \Gamma_2 \quad \Gamma_2 \vdash e_1 = e_2 : \tau}{\Gamma_1 \vdash \theta^*(e_1) = \theta^*(e_2) : \tau} \text{ SUBST}$		$\frac{f : \sigma \rightarrow \tau \in \Sigma \quad \Gamma \vdash e = e' : \sigma}{\Gamma \vdash \mathbf{app}_{\sigma \rightarrow \tau} f e = \mathbf{app}_{\sigma \rightarrow \tau} f e' : \tau} \text{ APP-CONG}$	
$\frac{\Gamma \vdash e_1 = e'_1 : \tau_1 \quad \Gamma \vdash e_2 = e'_2 : \tau_2}{\Gamma \vdash (e_1, e_2) = (e'_1, e'_2) : \tau_1 \times \tau_2} \text{ PAIR-CONG}$		$\frac{\Gamma \vdash e = e' : \sigma \times \tau}{\Gamma \vdash \mathbf{fst}_{\sigma \times \tau} e = \mathbf{fst}_{\sigma \times \tau} e' : \sigma} \text{ FST-CONG}$	
$\frac{\Gamma \vdash e = e' : \sigma \times \tau}{\Gamma \vdash \mathbf{snd}_{\sigma \times \tau} e = \mathbf{snd}_{\sigma \times \tau} e' : \tau} \text{ SND-CONG}$		$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{fst}_{\tau_1 \times \tau_2} (e_1, e_2) = e_1 : \tau_1} \text{ FST-PAIR}$	
$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{snd}_{\tau_1 \times \tau_2} (e_1, e_2) = e_2 : \tau_2} \text{ SND-PAIR}$		$\frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash e = (\mathbf{fst}_{\sigma \times \tau} e, \mathbf{snd}_{\sigma \times \tau} e) : \sigma \times \tau} \text{ PAIR-EXT}$	
		$\frac{\Gamma \vdash e : 1}{\Gamma \vdash e = \checkmark : 1} \text{ UNIT-EXT}$	

Figure 2.5: Equality for IPDL expressions.

2.4.2 Approximate Equality in IPDL

The approximate equational logic of IPDL consists of two layers: *approximate congruence* and *approximate equality*. Informally, when proving that two protocols P, Q are computationally indistinguishable (written $P \approx Q$), we exhibit a sequence of intermediate protocols $P_0, \dots, P_n$ such that

$$P \equiv P_0 \approx \dots \approx P_n \equiv Q$$

and each step $P_i \approx P_{i+1}$ comes in one of two flavors. Firstly, it can be an exact equality $P_i = P_{i+1}$, for example if P_{i+1} simplifies P_i by inlining the computation on a channel into another channel. Secondly, it can be an application of a single indistinguishability assumption suitably nested in a program context. A typical proof thus starts from a protocol, applies exact rules to simplify and massage it into a form where an indistinguishability assumption applies, and repeats this process.

The structure of our judgements reflects this intuition. Approximate congruence in Figure 2.10 applies a program context to a single indistinguishability assumption, from now on referred to as an *approximate axiom*. We assume a set of n approximate axioms of the form $\Delta^k \vdash P^k \approx Q^k : I^k \rightarrow O^k$ for $1 \leq k \leq n$, where $\Delta^k \vdash P^k : I^k \rightarrow O^k$ and $\Delta^k \vdash Q^k : I^k \rightarrow O^k$. The approximate congruence judgement takes the form $\Delta \vdash P \cong Q : I \rightarrow O \text{ ctx } \psi$ for axiom $\# k$. Informally, it says that the protocol Q was obtained from the protocol P by applying the k -th approximate axiom nested in a context of size at most ψ . The ctx parameter tracks the increase in the adversary's resources incurred by the proof and is computed by the tool.

There are five canonical ways to use an approximate axiom, corresponding to the five rules in Figure 2.10. We can appeal to the axiom directly (rule AXIOM); nest it in a program context that introduces a new internal channel (rule CONG-NEW) or forms a parallel composition with another protocol (rule CONG-COMP); and we can pick different channel names (rule EMBD) or formally add an unused channel to the protocols' inputs (rule INPUT-UNUSED).

Approximate equality in Figure 2.11 chains together a sequence of exact equalities and approximate congruence steps. The judgement takes the form $\Delta \vdash P \approx Q : I \rightarrow O \text{ count } \xi \text{ ctx } \psi$. The parameter count counts the number of axiom invocations for each approximate axiom, and the parameter ctx tracks per axiom the maximum increase in the adversary's resources incurred by the axiom's invocations. Informally, ψ is the maximum size of the program context in which each axiom was applied, and is computed by the tool. Rules SYM and TRANS ensure that approximate equality is an equivalence relation. Rules EXACT and APPROX-CONG import an exact equality step and an approximate congruence step, respectively.

Equality for reactions: $\Delta; \Gamma \vdash R_1 = R_2 : I \rightarrow \tau$	
$\frac{\Delta; \Gamma \vdash R : I \rightarrow \tau}{\Delta; \Gamma \vdash R = R : I \rightarrow \tau} \text{REFL}$	$\frac{\Delta; \Gamma \vdash R_1 = R_2 : I \rightarrow \tau}{\Delta; \Gamma \vdash R_2 = R_1 : I \rightarrow \tau} \text{SYM}$
$\frac{\Delta; \Gamma \vdash R_1 = R_2 : I \rightarrow \tau \quad \Delta; \Gamma \vdash R_2 = R_3 : I \rightarrow \tau}{\Delta; \Gamma \vdash R_1 = R_3 : I \rightarrow \tau} \text{TRANS}$	$\frac{\Gamma \vdash R_1 = R_2 : \tau \text{ axiom}}{\cdot; \Gamma \vdash R_1 = R_2 : \emptyset \rightarrow \tau} \text{AXIOM}$
$\frac{i \notin I \quad \Delta; \Gamma \vdash R_1 = R_2 : I \rightarrow \tau}{\Delta; \Gamma \vdash R_1 = R_2 : I \cup \{i\} \rightarrow \tau} \text{INPUT-UNUSED}$	$\frac{\phi : \Delta_1 \rightarrow \Delta_2 \quad \Delta_2 \vdash R_1 = R_2 : I \rightarrow \tau}{\Delta_1 \vdash \phi^*(R_1) = \phi^*(R_2) : \phi^*(I) \rightarrow \tau} \text{EMBED}$
$\frac{\theta : \Gamma_1 \rightarrow \Gamma_2 \quad \Delta; \Gamma_2 \vdash R_1 = R_2 : I \rightarrow \tau}{\Delta; \Gamma_1 \vdash \theta^*(R_1) = \theta^*(R_2) : I \rightarrow \tau} \text{SUBST}$	$\frac{\Gamma \vdash e = e' : \tau}{\Delta; \Gamma \vdash \text{ret } e = \text{ret } e' : I \rightarrow \tau} \text{CONG-RET}$
$\frac{\mathbf{d} : \sigma \rightarrow \tau \in \Sigma \quad \Gamma \vdash e = e' : \sigma}{\Delta; \Gamma \vdash \text{samp}_{\sigma \rightarrow \tau} \mathbf{d} \ e = \text{samp}_{\sigma \rightarrow \tau} \mathbf{d} \ e' : I \rightarrow \tau} \text{CONG-SAMP}$	
$\frac{\Gamma \vdash e = e' : \text{Bool} \quad \Delta; \Gamma \vdash R_1 = R'_1 : I \rightarrow \tau \quad \Delta; \Gamma \vdash R_2 = R'_2 : I \rightarrow \tau}{\Delta; \Gamma \vdash (\text{if } e \text{ then } R_1 \text{ else } R_2) = (\text{if } e' \text{ then } R'_1 \text{ else } R'_2) : I \rightarrow \tau} \text{CONG-IF}$	
$\frac{\Delta; \Gamma \vdash R = R' : I \rightarrow \sigma \quad \Delta; \Gamma, x : \sigma \vdash S = S' : I \rightarrow \tau}{\Delta; \Gamma \vdash (x : \sigma \leftarrow R; S) = (x : \sigma \leftarrow R'; S') : I \rightarrow \tau} \text{CONG-BIND}$	

Figure 2.6: Equality for IPDL reactions. Additional rules are given in Figure 2.7.

2.5 Hello World Example

In our simplest example, the IPDL *Hello World* analogue, Alice receives a Boolean message, encodes it by xor-ing it with a randomly generated Boolean, and leaks the encoding to the adversary. We show that this is equal to leaking a randomly generated ciphertext.

Formally, our signature consists of two symbols: $\oplus : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$ for the Boolean sum, and $\text{flip} : 1 \rightarrow \text{Bool}$ for the uniform distribution on Booleans. We write $x \oplus y$ in place of $\oplus(x, y)$.

The Assumptions

Our single axiom states that flip is invariant under xor-ing with a fixed Boolean:

- $\cdot; x : \text{Bool} \vdash (y \leftarrow \text{samp flip}; \text{ret } x \oplus y) = \text{samp flip} : \emptyset \rightarrow \text{Bool}$

This is indeed the case if (and only if) flip is uniform.

The Ideal Functionality

Upon receiving the input message, the ideal functionality generates a random ciphertext on an internal channel Ctxt and leaks its value to the adversary:

- `new Ctxt : bool in`
- `Ctxt := m ← read In; samp flip ||`
- `LeakCtxtadvid := read Ctxt`

The Real Protocol

In the real protocol, Alice generates a random Boolean key on an internal channel Key , constructs the ciphertext by xor-ing the input message with the key, again in an internal channel and leaks the resulting ciphertext to the adversary:

- `new Key : bool in`
- `new Ctxt : bool in`
- `Key := samp flip ||`

Equality for reactions (cont.): $\Delta; \Gamma \vdash R_1 = R_2 : I \rightarrow \tau$

$$\begin{array}{c}
\frac{\Gamma \vdash e : \sigma \quad \Delta; \Gamma, x : \sigma \vdash R : I \rightarrow \tau}{\Delta; \Gamma \vdash (x : \sigma \leftarrow \text{ret } e; R) = R[x := e] : I \rightarrow \tau} \text{RET-BIND} \qquad \frac{\Delta; \Gamma \vdash R : I \rightarrow \tau}{\Delta; \Gamma \vdash (x : \tau \leftarrow R; \text{ret } x) = R : I \rightarrow \tau} \text{BIND-RET} \\
\\
\frac{\Delta; \Gamma \vdash R_1 : I \rightarrow \sigma_1 \quad \Delta; \Gamma, x_1 : \sigma_1 \vdash R_2 : I \rightarrow \sigma_2 \quad \Delta; \Gamma, x_2 : \sigma_2 \vdash S : I \rightarrow \tau}{\Delta; \Gamma \vdash (x_2 : \sigma_2 \leftarrow (x_1 : \sigma_1 \leftarrow R_1; R_2); S) = (x_1 : \sigma_1 \leftarrow R_1; x_2 : \sigma_2 \leftarrow R_2; S) : I \rightarrow \tau} \text{BIND-BIND} \\
\\
\frac{\Delta; \Gamma \vdash R_1 : I \rightarrow \sigma_1 \quad \Delta; \Gamma \vdash R_2 : I \rightarrow \sigma_2 \quad \Delta; \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash S : I \rightarrow \tau}{\Delta; \Gamma \vdash (x_1 : \sigma_1 \leftarrow R_1; x_2 : \sigma_2 \leftarrow R_2; S) = (x_2 : \sigma_2 \leftarrow R_2; x_1 : \sigma_1 \leftarrow R_1; S) : I \rightarrow \tau} \text{EXCH} \\
\\
\frac{d : \rho \Rightarrow \sigma \in \Sigma \quad \Gamma \vdash e : \rho \quad \Delta; \Gamma \vdash R : I \rightarrow \tau}{\Delta; \Gamma \vdash (x : \sigma \leftarrow \text{samp}_{\rho \rightarrow \sigma} d e; R) = R : I \rightarrow \tau} \text{SAMP-PURE} \\
\\
\frac{i : \sigma \in \Delta \quad i \in I \quad \Delta; \Gamma, x : \sigma, y : \sigma \vdash R : I \rightarrow \tau}{\Delta; \Gamma \vdash (x : \sigma \leftarrow \text{read } i; y : \sigma \leftarrow \text{read } i; R) = (x : \sigma \leftarrow \text{read } i; R[y := x]) : I \rightarrow \tau} \text{READ-DET} \\
\\
\frac{\Delta; \Gamma \vdash R_1 : I \rightarrow \tau \quad \Delta; \Gamma \vdash R_2 : I \rightarrow \tau}{\Delta; \Gamma \vdash (\text{if true then } R_1 \text{ else } R_2) = R_1 : I \rightarrow \tau} \text{IF-LEFT} \qquad \frac{\Delta; \Gamma \vdash R_1 : I \rightarrow \tau \quad \Delta; \Gamma \vdash R_2 : I \rightarrow \tau}{\Delta; \Gamma \vdash (\text{if false then } R_1 \text{ else } R_2) = R_2 : I \rightarrow \tau} \text{IF-RIGHT} \\
\\
\frac{\Delta; \Gamma, x : \text{Bool} \vdash R : I \rightarrow \tau \quad \Gamma \vdash e : \text{Bool}}{\Delta; \Gamma \vdash (\text{if } e \text{ then } R[x := \text{true}] \text{ else } R[x := \text{false}]) = R[x := e] : I \rightarrow \tau} \text{IF-EXT}
\end{array}$$

Figure 2.7: Equality for IPDL reactions.

- $\text{Ctxt} := m \leftarrow \text{read In}; k \leftarrow \text{read Key}; \text{ret } m \oplus k \parallel$
- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := \text{read Ctxt}$

The Simulator

The simulator mediates between the two leakage channels $\text{LeakCtxt}_{\text{adv}}^{\text{id}}$ and $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}}$ by forwarding the former to the latter:

- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := \text{read LeakCtxt}_{\text{adv}}^{\text{id}}$

Real = Ideal + Sim

On the left-hand side of the above equality we have the real protocol. On the right-hand side, we have the composition of the ideal functionality with the simulator, followed by the hiding of the channel $\text{LeakCtxt}_{\text{adv}}^{\text{id}}$:

- $\text{new LeakCtxt}_{\text{adv}}^{\text{id}} : \text{bool in}$
- $\text{ideal} \parallel \text{sim}$

We call this protocol $\text{ideal} + \text{sim}$.

The protocols real and $\text{ideal} + \text{sim}$ now have identical inputs (the channel In) as well as outputs (the channel $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}}$).

We now simplify both protocols so that they have the same internal structure.

On the left-hand side we do the following steps.

1. We use the NEW-EXCH rule to get
 - $\text{new Ctxt} : \text{bool in}$
 - $\text{new Key} : \text{bool in}$
 - $\text{Key} := \text{samp flip} \parallel$
 - $\text{Ctxt} := m \leftarrow \text{read In}; k \leftarrow \text{read Key}; \text{ret } m \oplus k \parallel$
 - $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := \text{read Ctxt}$
2. We want to modify the protocol inside the first **new**, so we must use the CONG-NEW rule;
3. We want to get $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}}$ out of the inner **new** to obtain the protocol P:

Exact equality for protocols: $\Delta \vdash P_1 = P_2 : I \rightarrow O$

$$\begin{array}{c}
\frac{\Delta \vdash P : I \rightarrow O}{\Delta \vdash P = P : I \rightarrow O} \text{REFL} \quad \frac{\Delta \vdash P_1 = P_2 : I \rightarrow O}{\Delta \vdash P_2 = P_1 : I \rightarrow O} \text{SYM} \quad \frac{\Delta \vdash P_1 = P_2 : I \rightarrow O \quad \Delta \vdash P_2 = P_3 : I \rightarrow O}{\Delta \vdash P_1 = P_3 : I \rightarrow O} \text{TRANS} \\
\\
\frac{\Delta \vdash P_1 = P_2 : I \rightarrow O \text{ axiom}}{\Delta \vdash P_1 = P_2 : I \rightarrow O} \text{AXIOM} \quad \frac{i \notin I \cup O \quad \Delta \vdash P_1 = P_2 : I \rightarrow O}{\Delta \vdash P_1 = P_2 : I \cup \{i\} \rightarrow O} \text{INPUT-UNUSED} \\
\\
\frac{\phi : \Delta_1 \rightarrow \Delta_2 \quad \Delta_2 \vdash P_1 = P_2 : I \rightarrow O}{\Delta_1 \vdash \phi^*(P_1) = \phi^*(P_2) : \phi^*(I) \rightarrow \phi^*(O)} \text{EMBED} \quad \frac{o \notin I \quad o : \tau \in \Delta \quad \Delta; \cdot \vdash R = R' : I \cup \{o\} \rightarrow \tau}{\Delta \vdash (o := R) = (o := R') : I \rightarrow \{o\}} \text{CONG-REACT} \\
\\
\frac{\Delta \vdash P = P' : I \cup O_2 \rightarrow O_1 \quad \Delta \vdash Q : I \cup O_1 \rightarrow O_2}{\Delta \vdash P \parallel Q = P' \parallel Q : I \rightarrow O_1 \cup O_2} \text{CONG-COMP-LEFT} \\
\\
\frac{\Delta, o : \tau \vdash P = P' : I \rightarrow O \cup \{o\}}{\Delta \vdash (\text{new } o : \tau \text{ in } P) = (\text{new } o : \tau \text{ in } P') : I \rightarrow O} \text{CONG-NEW} \\
\\
\frac{\Delta \vdash P_1 : I \cup O_2 \rightarrow O_1 \quad \Delta \vdash P_2 : I \cup O_1 \rightarrow O_2}{\Delta \vdash P_1 \parallel P_2 = P_2 \parallel P_1 : I \rightarrow O_1 \cup O_2} \text{COMP-COMM} \\
\\
\frac{\Delta \vdash P_1 : I \cup O_2 \cup O_3 \rightarrow O_1 \quad \Delta \vdash P_2 : I \cup O_1 \cup O_3 \rightarrow O_2 \quad \Delta \vdash P_3 : I \cup O_1 \cup O_2 \rightarrow O_3}{\Delta \vdash (P_1 \parallel P_2) \parallel P_3 = P_1 \parallel (P_2 \parallel P_3) : I \rightarrow O_1 \cup O_2 \cup O_3} \text{COMP-ASSOC} \\
\\
\frac{\Delta, o_1 : \tau_1, o_2 : \tau_2 \vdash P : I \rightarrow O \cup \{o_1, o_2\}}{\Delta \vdash (\text{new } o_1 : \tau_1 \text{ in new } o_2 : \tau_2 \text{ in } P) = (\text{new } o_2 : \tau_2 \text{ in new } o_1 : \tau_1 \text{ in } P) : I \rightarrow O} \text{NEW-EXCH} \\
\\
\frac{\Delta \vdash P : I \cup O_2 \rightarrow O_1 \quad \Delta, o : \tau \vdash Q : I \cup O_1 \rightarrow O_2 \cup \{o\}}{\Delta \vdash P \parallel (\text{new } o : \tau \text{ in } Q) = (\text{new } o : \tau \text{ in } (P \parallel Q)) : I \rightarrow O_1 \cup O_2} \text{COMP-NEW} \\
\\
\frac{\Delta \vdash P : I \rightarrow O \quad \Delta \vdash Q : I \cup O \rightarrow \emptyset}{\Delta \vdash P \parallel Q = P : I \rightarrow O} \text{ABSORB-LEFT}
\end{array}$$

Figure 2.8: Exact equality for IPDL protocols. Additional rules are given in Figure 2.9.

- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := \text{read Ctxt} \parallel$
- $\text{new Key} : \text{bool in}$
- $\text{Key} := \text{samp flip} \parallel$
- $\text{Ctxt} := m \leftarrow \text{read In}; k \leftarrow \text{read Key}; \text{ret } m \oplus k$

We notice that by applying the rule COMP-NEW to P we get to our current protocol, so we can apply the SYM rule as follows: $\text{SYM}[P_2 = P] (\text{COMP-NEW})$. Here we use the notation $[P_2 = P]$ to state that the protocol P2 in the rule SYM is our protocol P;

4. We can now fold Key. Since P is a parallel composition, we must use the CONG-COMP rule, as follows: $\text{CONG-COMP}[Q = \text{LeakCtxt}_{\text{adv}}^{\text{Alice}} ::= \text{read Ctxt}] (\text{FOLD-BIND})$;
5. The result of doing

```

CONG-NEW(
  SYM[P2 = P] (COMP-NEW) ;
  CONG-COMP[Q = LeakCtxtAliceadv ::= read Ctxt] (FOLD-BIND)
)

```

where we used the semi-colon to denote the applications of the TRANS rule is

- $\text{new Ctxt} : \text{bool in}$
- $\text{Ctxt} := m \leftarrow \text{read In}; k \leftarrow \text{samp flip}; \text{ret } m \oplus k \parallel$
- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := \text{read Ctxt}$

6. We can fold the internal channel Ctxt, using FOLD-BIND, to obtain

Exact equality for protocols (cont.): $\Delta \vdash P = Q : I \rightarrow O$

$$\begin{array}{c}
\frac{o \notin I \quad o : \tau \in \Delta \quad \Delta; \cdot \vdash R : I \cup \{o\} \rightarrow \tau}{\Delta \vdash (o := x \leftarrow \text{read } o; R) = (o := \text{read } o) : I \rightarrow \{o\}} \text{ DIVERGE} \\
\\
\frac{o \notin I \quad o : \tau \in \Delta \quad \Delta; \cdot \vdash R : I \cup \{o\} \rightarrow \text{Bool} \quad \Delta; \cdot \vdash S_1 : I \cup \{o\} \rightarrow \tau \quad \Delta; \cdot \vdash S_2 : I \cup \{o\} \rightarrow \tau}{\Delta \vdash (\text{new } l : \tau \text{ in } o := x \leftarrow R; \text{ if } x \text{ then } \text{read } l \text{ else } S_2 \parallel l := S_1) = (o := x \leftarrow R; \text{ if } x \text{ then } S_1 \text{ else } S_2) : I \rightarrow \{o\}} \text{ FOLD-IF-LEFT} \\
\\
\frac{o \notin I \quad o : \tau \in \Delta \quad \Delta; \cdot \vdash R : I \cup \{o\} \rightarrow \text{Bool} \quad \Delta; \cdot \vdash S_1 : I \cup \{o\} \rightarrow \tau \quad \Delta; \cdot \vdash S_2 : I \cup \{o\} \rightarrow \tau}{\Delta \vdash (\text{new } r : \tau \text{ in } o := x \leftarrow R; \text{ if } x \text{ then } S_1 \text{ else } \text{read } r \parallel r := S_2) = (o := x \leftarrow R; \text{ if } x \text{ then } S_1 \text{ else } S_2) : I \rightarrow \{o\}} \text{ FOLD-IF-RIGHT} \\
\\
\frac{o \notin I \quad o : \tau \in \Delta \quad \Delta; \cdot \vdash R : I \cup \{o\} \rightarrow \sigma \quad \Delta; x : \sigma \vdash S : I \cup \{o\} \rightarrow \tau}{\Delta \vdash (\text{new } c : \sigma \text{ in } o := x \leftarrow \text{read } c; S \parallel c := R) = (o := x \leftarrow R; S) : I \rightarrow \{o\}} \text{ FOLD-BIND} \\
\\
\frac{o_1 \neq o_2 \quad o_1, o_2 \notin I \quad o_1 : \tau_1, o_2 : \tau_2 \in \Delta \quad \Delta; \cdot \vdash R_1 : I \cup \{o_1, o_2\} \rightarrow \tau_1 \quad \Delta; x_1 : \tau_1 \vdash R_2 : I \cup \{o_1, o_2\} \rightarrow \tau_2}{\Delta; \cdot \vdash (x_1 \leftarrow R_1; x'_1 \leftarrow R_1; \text{ret } (x_1, x'_1)) = (x_1 \leftarrow R_1; \text{ret } (x_1, x_1)) : I \cup \{o_1, o_2\} \rightarrow \tau_1 \times \tau_1} \\
\frac{}{\Delta \vdash (o_1 := R_1 \parallel o_2 := x_1 \leftarrow \text{read } o_1; R_2) = (o_1 := R_1 \parallel o_2 := x_1 \leftarrow R_1; R_2) : I \rightarrow \{o_1, o_2\}} \text{ SUBST} \\
\\
\frac{o_1 \neq o_2 \quad o_1, o_2 \notin I \quad o_1 : \tau_1, o_2 : \tau_2 \in \Delta \quad \Delta; \cdot \vdash R_1 : I \cup \{o_1, o_2\} \rightarrow \tau_1 \quad \Delta; \cdot \vdash R_2 : I \cup \{o_1, o_2\} \rightarrow \tau_2 \quad \Delta; \cdot \vdash (x_1 \leftarrow R_1; R_2) = R_2 : I \cup \{o_1, o_2\} \rightarrow \tau_2}{\Delta \vdash (o_1 := R_1 \parallel o_2 := x_1 \leftarrow \text{read } o_1; R_2) = (o_1 := R_1 \parallel o_2 := R_2) : I \rightarrow \{o_1, o_2\}} \text{ DROP}
\end{array}$$

Figure 2.9: Additional rules for exact equality of IPDL protocols. Distinguishing changes of equalities are highlighted in red.

- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := m \leftarrow \text{read } \text{In}; k \leftarrow \text{samp flip}; \text{ret } m \oplus k$

7. We notice that we can apply the assumption. Since it is a reaction-level assumption, we will need to use the CONG-REACT rule that allows us to modify the reaction assigned to the channel $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}}$.
8. Since $m \leftarrow \text{read } \text{In}$ appears in front of the left-hand side of our assumption, we need to use the CONG-BIND rule. This rule takes two arguments, one for rewriting the reaction appearing before the binder, and one for rewriting the reaction appearing after the binder. We do not want to modify the first reaction, so we will mark this by **idle**. The second reaction will be modified by the AXIOM rule.
9. The entire step is then

```

CONG-REACT(
  CONG-BIND(
    idle,
    AXIOM ) )

```

To sum up, the entire proof rewriting the protocol **real** is

```

NEW-EXCH ;
CONG-NEW(
  SYM[P2 = P](COMP-NEW) ;
  CONG-COMP[Q = LeakCtxtadvAlice := read Ctxt] (
    FOLD-BIND
  )
)
FOLD-BIND ;
CONG-REACT(
  CONG-BIND(
    idle,
    AXIOM
  )
)

```

The result is

- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := m \leftarrow \text{read } \text{In}; \text{samp flip}$

We can now proceed to rewrite the protocol **ideal** + **sim**:

Approximate congruence: $\Delta \vdash P \cong Q : I \rightarrow O$ ctxt ψ for axiom # k	
$\frac{}{\Delta^k \vdash P^k \cong Q^k : I^k \rightarrow O^k \text{ ctxt } 0 \text{ for axiom \# } k} \text{AXIOM}$	
$\frac{c \notin I \cup O \quad \Delta \vdash P \cong Q : I \rightarrow O \text{ ctxt } \psi \text{ for axiom \# } k}{\Delta \vdash P \cong Q : I \cup \{c\} \rightarrow O \text{ ctxt } \psi \text{ for axiom \# } k} \text{INPUT-UNUSED}$	
$\frac{\phi : \Delta_1 \rightarrow \Delta_2 \quad \Delta_2 \vdash P \cong Q : I \rightarrow O \text{ ctxt } \psi \text{ for axiom \# } k}{\Delta_1 \vdash \phi^*(P) \cong \phi^*(Q) : \phi^*(I) \rightarrow \phi^*(O) \text{ ctxt } \psi \text{ for axiom \# } k} \text{EMBED}$	
$\frac{\Delta \vdash P \cong P' : I \cup O_2 \rightarrow O_1 \text{ ctxt } \psi \text{ for axiom \# } k \quad \Delta \vdash Q : I \cup O_1 \rightarrow O_2}{\Delta \vdash P \parallel Q \cong P' \parallel Q : I \rightarrow O_1 \cup O_2 \text{ ctxt } (\psi + \ Q\ + 3) \text{ for axiom \# } k} \text{CONG-COMP}$	
$\frac{\Delta, o : \tau \vdash P \cong P' : I \rightarrow O \cup \{o\} \text{ ctxt } \psi \text{ for axiom \# } k}{\Delta \vdash (\text{new } o : \tau \text{ in } P) \cong (\text{new } o : \tau \text{ in } P') : I \rightarrow O \text{ ctxt } \psi \text{ for axiom \# } k} \text{CONG-NEW}$	

Figure 2.10: Approximate congruence of protocols.

Approximate equality: $\Delta \vdash P \approx Q : I \rightarrow O$ count ξ ctxt ψ	
$\frac{\Delta \vdash P = Q : I \rightarrow O}{\Delta \vdash P \approx Q : I \rightarrow O \text{ count } (i \mapsto 0) \text{ ctxt } (i \mapsto 0)} \text{EXACT}$	
$\frac{\Delta \vdash P_1 \approx P_2 : I \rightarrow O \text{ count } \xi \text{ ctxt } \psi}{\Delta \vdash P_2 \approx P_1 : I \rightarrow O \text{ count } \xi \text{ ctxt } \psi} \text{SYM}$	
$\frac{\Delta \vdash P \cong Q : I \rightarrow O \text{ ctxt } \psi \text{ for axiom \# } k}{\Delta \vdash P \approx Q : I \rightarrow O \text{ count } (k \mapsto 1, i \neq k \mapsto 0) \text{ ctxt } (k \mapsto \psi, i \neq k \mapsto 0)} \text{APPROX-CONG}$	
$\frac{\Delta \vdash P_1 \approx P_2 : I \rightarrow O \text{ count } \xi_1 \text{ ctxt } \psi_1 \quad \Delta \vdash P_2 \approx P_3 : I \rightarrow O \text{ count } \xi_2 \text{ ctxt } \psi_2}{\Delta \vdash P_1 \approx P_3 : I \rightarrow O \text{ count } (i \mapsto \xi_1(i) + \xi_2(i)) \text{ ctxt } (i \mapsto \max(\psi_1(i), \psi_2(i)))} \text{TRANS}$	

Figure 2.11: Approximate equality of protocols.

- `new LeakCtxtadvid : bool in`
- `sim || ideal`

or, by replacing the protocols with their definitions:

- `new LeakCtxtadvid : bool in`
- `LeakCtxtadvAlice := read LeakCtxtadvid ||`
- `new Ctxt : bool in`
- `Ctxt := m ← read In; samp flip ||`
- `LeakCtxtadvid := read Ctxt`

as follows.

1. We notice that we could fold both `Ctxt` and `LeakCtxtadvid`. The proof is simpler if we first fold `Ctxt`. We work inside a `new` and a composition, so we must apply the corresponding congruence rules:

```

CONG-NEW (
  CONG-COMP [Q = LeakCtxtadvAlice ::= read LeakCtxtadvid] (
    FOLD-BIND
  )
)

```

to obtain

- `new LeakCtxtadvid : bool in`

- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := \text{read } \text{LeakCtxt}_{\text{adv}}^{\text{id}} \parallel$
- $\text{LeakCtxt}_{\text{adv}}^{\text{id}} := m \leftarrow \text{read In}; \text{samp flip}$

2. We can now fold $\text{LeakCtxt}_{\text{adv}}^{\text{id}}$ to obtain

- $\text{LeakCtxt}_{\text{adv}}^{\text{Alice}} := m \leftarrow \text{read In}; \text{samp flip}$

To sum up, the result of rewriting $\text{ideal} + \text{sim}$ by

```

CONG-NEW(
  CONG-COMP[Q = LeakCtxtAliceadv := read LeakCtxtidadv] (
    FOLD-BIND
  )
);
FOLD-BIND

```

is now the protocol that we obtained by rewriting real . We can then combine the two proofs using the SYM rule:

```

NEW-EXCH ;
CONG-NEW(
  SYM[P2 = P] (COMP-NEW) ;
  CONG-COMP[Q = LeakCtxtAliceadv := read Ctxt] (
    FOLD-BIND
  )
);
FOLD-BIND ;
CONG-REACT(
  CONG-BIND(
    idle,
    AXIOM
  )
);
SYM[P2 = ideal + sim] (
  CONG-NEW(
    CONG-COMP[Q = LeakCtxtAliceadv := read LeakCtxtidadv] (
      FOLD-BIND
    )
  );
  FOLD-BIND
)

```

and thus we have rewritten real to $\text{ideal} + \text{sim}$, proving their equivalence.

While the proof is correct, it has some inconveniences. We must explicitly apply the congruence and exchange rules. This quickly clutters the proof for a medium size protocol, making it unreadable. The IPDL proof language will address these limitations.

Chapter 3

The IPDL Prover

In this section we describe the IPDL prover. We start by giving a general introduction to Maude, its implementation language, such that we are later able to explain some language design decisions. In particular, Maude provides a *strategy language* which allows us to control how the IPDL rules are applied to a protocol. We then present how the IPDL syntax is supported, with declarations of types, function and distribution symbols, embeddings, assumptions, and protocols. We discuss two notational conventions that are introduced: *normal forms* of reactions and protocols, allowing us to simplify the way the IPDL equality rules are applied, and *families of channels*, providing a compact notation for related channels. We then introduce the IPDL proof language, as a domain-specific language (DSL) that translates into Maude strategy calls. Some of the strategies have been defined once and for all. Others are defined on the fly – those going beyond the capabilities of the Maude strategy language, which does not let a strategy take another strategy as an argument. The DSL is tailored to efficiency, hiding the application of structural IPDL equality rules (congruence and exchange ones) from the user. The DSL covers both the exact and the approximate equality calculus. It supports subproofs and proof attempts. Besides the proof language, there are commands for inspecting the results and basic support for modularity. We finish the section by comparing the proof of the Hello World example in the DSL with its rules-only counterpart and discussing a more complex case study in more detail.

3.1 Maude

Maude [2] is a high-level declarative language and a high-performance logical framework supporting both equational and rewriting logic computation for a wide range of applications. Maude features several kinds of modules:

- *functional modules*, which are theories (with an initial model semantics) in membership equational logic that allow definitions of data types and operations on them, via multiple sorts, subsort relations between them, equations between terms, and assertions of membership of a term to a sort,
- *system modules*, which are theories in rewriting logic that extend functional modules with definitions of rewrite rules, representing transitions between states, and
- *strategy modules*, which control the way the rewriting rules are applied, by means of strategy combinators, such as concatenation, iterations and others.

We now present the features of the Maude language that we make use of in formalizing IPDL. Maude functional modules are introduced with the syntax `fmod NAME is ... endfm`. In a functional module we can declare sorts, using the keyword `sort`, state that two sorts are in the subsort relation, written `subsort s1 < s2`, declare operations on the sorts, using `op f : s1 ... sk -> s` for an operation `f` with argument sorts `s1 ... sk` and result sort `s`. Moreover, operations may have attributes, written in square brackets after their declarations, like `comm` for commutativity or `assoc` for associativity. In Maude, terms are rewritten to a normal form modulo the declared attributes and the equations of defined operations. More precisely, equations are used as equational rules: instances of the left-hand side pattern that match subterms of a term are replaced with the corresponding instances of the right-hand side. The process is called term rewriting and the result of simplifying a term by complete application of equational rules is called its normal form. We can indicate which operations will appear in these ground forms by adding the attribute `ctor` to them. An operation that is not a constructor of a sort is typically regarded as defined. Equations are introduced by the syntax `eq t1 = t2`, where `t1` and `t2` are terms of sorts related via subsorting. We can assert sort membership using the syntax `t : s` where `t` is a term and `s` is a sort. Conditional equations are written `ceq t = t' if C1 ^ ... ^ Cn` where `Ci` is either an equation or a membership. We may declare variables using the keyword `var`. Functional modules are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness, see details in [2]. The semantics of functional models is given in terms of the initial model whose elements are ground equivalence classes of terms modulo equations.

Rewriting logic extends equational logic by introducing the notion of rewrites corresponding to transitions between states. Unlike equations, rewrites are not symmetric. Maude system modules are introduced with the syntax `mod NAME is ... endm`.

Rules are declared with the syntax `rl [label] : t1 => t2`. Conditional rules are written with the keyword `cr1 [label] : t => t' if C1 ∧ ... ∧ Cn` and their conditions C_i may be equations, memberships or rewrites. Rewrites are not expected to be terminating, confluent or deterministic. Rewrites denote transitions between the elements of the initial model of the functional part of a system module.

Maude strategy modules are introduced with the syntax `smod NAME is ... endsm`. In addition to declarations allowed in system modules, we can have strategy declarations and definitions. Strategies are generally built out of rule labels by repeated applications of combinators. The main strategy combinators are `;` for concatenation of strategy expressions, `|` for alternative, `*` for iteration of an expression zero or more times, `idle` for the strategy giving as result its argument, `fail` for the strategy that gives no result, `s1 ? s2 : s3` for the strategy that attempts to run the strategy `s1` then, if the run is successful, it runs `s2`, otherwise it runs `s3`. Several other derived constructions are also supported, *e.g.*, `try s` for `s ? idle : idle` and `s1 or-else s2` for `s1 ? idle : s2`. The match and rewrite operator `matchrew` restricts the application of a strategy to a specific subterm of the subject term, see details in [2]. Strategies are declared as `strat NAME : s1 ... sk @ s .`, where `s1 ... sk` are the sorts of the arguments of the strategy and `s` is the subject sort to which the strategy is applied. The syntax for definitions is `sd NAME(v1, ..., vk) := Exp .` where v_i are variables of sort s_i and `Exp` is a strategy expression.

Maude supports module imports, using the keyword `protecting`, which means that no new elements of an imported sorts may be added and no identification between elements of an imported sorts via equations are allowed. Two more importation modes are supported, but we do not make use of them.

Maude provides several predefined data types. We will use Booleans, natural numbers, lists, sets and maps.

3.2 Syntax

We have the predefined data types `unit` and `bool` and we can declare new types using the keyword `type`. We write the binary product on types as `unit * msg` and we use parentheses to disambiguate, *e.g.* `(unit * msg) * msg`.

Example 1: Declaring Types

We introduce types for messages, keys and crypted texts by writing

```
type msg .
type key .
type ctxt .
```

Function and distribution symbols are defined by writing `function NAME : T1 -> T2 .` and `distribution NAME : T1 -> T2 .`, where T_1, T_2 are types.

Example 2: Declaring Functions and Distributions

We introduce a logical xor operation and the uniform distribution on booleans:

```
function xor : bool * bool -> bool .

distribution flip : unit -> bool .
```

IPDL expressions include variable names, which are just identifiers, and the literals `True`, `False` and `()`. Function application is written `f(e)` where `f` is an identifier standing for the name of the function symbol and `e` is an expression. Moreover we have constructors for pairs, written `(e1, e2)` and projections on first and second component of a pair, written `fst e` and `snd e` respectively.

Channel contexts declare the inputs and outputs for a protocol, using the syntax `channel context NAME = input C1 : T1 ; ... ; output Cn : Tn .` where C_i are channel names and T_i are types.

Example 3: Declaring Channel Contexts

The channel context from the Hello World example in Sec. 2.5 is

```
channel context delta =
input In : bool ;
input X : bool ;
output LeakCtxtAliceAdv : bool .
```

IPDL embeddings are declared as `embedding NAME from delta1 to delta2 = (C1 |-> D1) ;; ... (Cn |-> Dn) .` where `delta1`, `delta2` are channel contexts, `Ci` are channels from `delta1` and `Di` are channels from `delta2`.

Example 4: Declaring Embeddings

Recall that embeddings must preserve typing. The channels come from their corresponding channel contexts:

```
embedding phi from deltaSRW to deltaNMK =
  OTMsg-0[s, r, w] |-> OTMsg-0[n, m, k] ;
  OTMsg-1[s, r, w] |-> OTMsg-1[n, m, k] ;
  OTMsg-2[s, r, w] |-> OTMsg-2[n, m, k] ;
  OTMsg-3[s, r, w] |-> OTMsg-3[n, m, k] ;

  OTChc-0[s, r, w] |-> OTChc-0[n, m, k] ;
  OTChc-1[s, r, w] |-> OTChc-1[n, m, k] .
```

Reactions are written as follows. If `e` is an expression, `return e` is a reaction. If `d` is an identifier, standing for the name of a distribution symbol and `e` is an expression, `samp d(e)` is a reaction. If `c` is a channel name, `read c` is a reaction. Moreover, we write `if e then R1 else R2` for branching, if `e` is an expression and `R1`, `R2` are reactions, and `x : T <- R1 ; R2` for binding, when `x` is an identifier, `T` is a type and `R1`, `R2` are reactions. We don't declare reactions independently, but they will appear as parts of protocols.

Protocols also follow the grammar for protocols, using the following constructors. For the empty protocol we write `emptyProtocol`. If `c` is a channel name and `R` is a reaction, `c ::= R` is a protocol. If `P1`, `P2` are protocols, so is `P1 || P2`. Finally, `new c : T in P` is a protocol, if `c` is a channel name, `T` is a type and `P` is a protocol.

We declare protocols by writing `protocol NAME = P` . where `P` is a protocol. The tool checks that `P` is well-typed and returns an error if it's not, also pointing to the source of the error inside the protocol. The name of the protocol must be new.

Example 5: Declaring Protocols

The real protocol from the Hello World example in Sec. 2.5 is

```
protocol real =
  new Ctxt : bool in
  new Key : bool in
  ((Key ::= samp flip (()) )
  ||
  (Ctxt ::= m : bool <- read In ;
   k : bool <- read Key ;
   return xor((m, k)) )
  ||
  (LeakCtxtAliceAdv ::= c : bool <- read Ctxt ; return c) ) .
```

We can use local declarations of protocols using the syntax

```
protocol NAME = P
  where NAME1 = P1 and ... and NAMEk = Pk
```

and `NAME1`, ..., `NAME_k` can be used in `P` wherever a protocol is expected. For an example, see the protocol `real` in Sec. 3.15.

Parameters, typically counting parties, sessions, number of messages, are declared by writing `parameter NAME : nat .`, where `NAME` must be new.

We can record global hypotheses by writing `hypothesis BT` . where `BT` is a boolean term. Typical examples include that a party is honest or semi-honest, written `hypothesis isHonest(N + 1) .`, where `isHonest` is an underspecified predicate and `N` is a parameter.

3.3 Normal Forms

We work with protocols that start with a list of declarations of internal channels, using `new`, followed by a parallel composition of channel assignments. The reactions in these assignments can be transformed into a list of binds of the form `x : T <- read c`, called bind-read reactions, followed by a reaction without binds. The list of binds can be regarded as commutative, as two

reactions with the same list of binds in different order are equivalent due to the reaction equivalence rule EXCH. Similarly, different order of declarations of internal channels gives equivalent protocols, by using the protocol equivalence rule NEW-EXCH. When writing equivalence proofs, we do not want to make the use of these rules explicit. Instead, we want to be able to apply the rules as though we could freely consider a certain declaration of an internal channel or a certain bind read reaction as the first.

Therefore, we introduce normal forms of reactions and protocols. For reactions, normal forms $\text{nf}(L, R)$ consist of a commutative list L of bind-read reactions and a bind-free reaction R .

Example 6: Normal Form of Reactions

The normal form of

```
d : bool <- read ce ;
m0 : bool <- read in0 ;
m1 : bool <- read in1 ;
k0 : bool <- read key0 ;
k1 : bool <- read key1 ;
if d then return k0 else return k1
```

is

```
nf(
  (d : bool <- read ce)
  (m0 : bool <- read in0)
  (m1 : bool <- read in1)
  (k0 : bool <- read key0)
  (k1 : bool <- read key1),
  if d then return k0 else return k1
)
```

During equivalence proofs, we may obtain in a normal form $\text{nf}(L, R)$ either arbitrary binds in L (*e.g.*, by substituting a read from a channel with the reaction assigned to that channel) or reactions R that are not bind-free. This will be represented as a pre-normal-form, written $\text{preNF}(L, R)$, which is a normal form without restrictions on the occurring reactions. If L contains a bind that is not a read bind, we will write it as $x_1 : T_1 \sim R_1$. The general strategy will be to transform pre-normal-forms $\text{preNF}(L, R)$ to normal forms using the following steps:

- if $x_1 : T_1 \sim R_1$ is in L and R_1 is of the form $\text{nf}(L_2, R_2)$, move the inner binds from L_2 at the level of L , and simplify the reaction of x_1 to R_2 .
- if $x_1 : T_1 \sim R_1$ is in L and R_1 is bind-free, rewrite the entire reaction as $\text{preNF}(L', x_1 : T_1 \sim R_1 ; R)$, where L' is obtained by removing $x_1 : T_1 \sim R_1$ from L .
- apply reaction-level axioms to R to bring it in the form $L' ; R'$, where L' is a list of bind reads and R' is bind-free, then move L' at the outer level of L .

At the level of protocols, normal forms $\text{newNF}(L, P)$ consist of a commutative list L of declarations of internal channels and a protocol P that does not start with internal channel declarations.

The normal form of

```
new ce : bool in new key0 : bool in new key1 : bool in new flip : bool in new choice : bool in (
  (ce ::= f : bool <- read flip ;
   c : bool <- read choice ;
   if f then (if c then return False else return True)
   else (if c then return True else return False)
)
|| (msgenc0 ::= d : bool <- read ce ; m0 : bool <- read in0 ;
   m1 : bool <- read in1 ; k0 : bool <- read key0 ;
   k1 : bool <- read key1 ;
   if d then return k0 else return k1)
|| (key0 ::= samp genKey (())) || (key1 ::= samp genKey (()))
|| (flip ::= samp flip (())) || (choice ::= return False) )
```

is

```
newNF( (ce : bool) (key0 : bool) (key1 : bool) (flip : bool) (choice : bool),
  (ce ::= f : bool <- read flip ;
    c : bool <- read choice ;
    if f then (if c then return False else return True)
    else (if c then return True else return False)
  )
|| (msgenc0 ::= d : bool <- read ce ; m0 : bool <- read in0 ;
    m1 : bool <- read in1 ; k0 : bool <- read key0 ;
    k1 : bool <- read key1 ;
    if d then return k0 else return k1)
|| (key0 ::= samp genKey (())) || (key1 ::= samp genKey (()))
|| (flip ::= samp flip (())) || (choice ::= return False) )
```

The user does not have to write reactions or protocols in their normal form. The tool transforms reactions to their normal forms when doing the analysis of protocol declarations, and protocols to their normal forms when they occur inside a proof.

These notations allow us to manage rewriting easily using the Maude strategy language for each IPDL rule. The main idea is to define strategies using the following pattern. We first check if the current protocol starts with declarations of hidden channels. If it does we apply the congruence rule for normal forms for protocols and recursively call the strategy. If it does not, we use the congruence rule for parallel composition and apply the IPDL rule only to the protocol fragment involved. This happens internally and the user does not have to specify the calls of the congruence rules.

3.4 Families of Channels

Families of channels provide a convenient abbreviation for the composition of semantically related channels $P[0] \dots P[n]$, where the value of n is typically not known. The semantical relation translates in the protocols being assigned similar reactions. We illustrate the syntax with the help of an example:

```
family SumCommit[bound (n + 2)]
  indices: i bounds: bound (n + 2) ::=
  (
    (when (i = 0) --> return False) ;;
    (when (i = (n + 2)) -->
      x : bool <- read SumCommit[n + 1] ;
      f : bool <- read LastCommit) ;
      return xor((x, f))
    )
    ;;
    (otherwise -->
      x : bool <- read SumCommit[i - 1] ;
      c : bool <- read Commit[i - 1] ;
      return xor((x, c))
    )
  )
```

Here i is an index variable ranging between 0 and $n + 1$. The reaction assigned to the channel `SumCommit[i]` is given with alternatives. We will refer to them as branches or cases. We allow the bound to be a natural number, an identifier denoting a natural number or an expression involving natural numbers and identifiers. The conditions occurring in the alternatives are of sort `BoolTerm`, and can be comparisons between expressions, written $e1 = e2$, $e1 < e2$, $e1 \leq e2$, user-defined predicates ($p(e)$) where p is the name of the predicate and e is an expression, negation of a `BoolTerm`, written `not t`, conjunction and disjunction of `BoolTerms`, written $t1 \text{ and } t2$ and $t1 \text{ or } t2$ respectively.

We allow families of channels with two or three indices as well. We write

```
family F[bound m bound n]
  indices: i, j
  bounds: bound m bound n ::= ...
```

for a family indexed by i ranging from 0 to $m - 1$ and by j ranging from 0 to $n - 1$. For families with more than one index we may have two more kinds of bounds. The first one is *dependent bounds*, written `dependentBound I`, where I is the name of a function taking naturals to naturals. The meaning of

```

family F[bound m dependentBound I]
  indices: i, j
  bounds: bound m dependentBound I ::= ...

```

is then that the family groups the channels $F[0, 0] \dots F[0, I(0)] \dots F[m - 1, 0] \dots F[m - 1, I(m - 1)]$. Dependent bounds are only allowed for the last index of a family. The second of the new kinds of bounds is *fixed bounds*, written `fixedBound n`, with the meaning that the corresponding index always has the value n . These can occur on every position in a list of bounds. The meaning of

```

family F[bound m fixedBound 0]
  indices: i, j
  bounds: bound m fixedBound 0 ::= ...

```

is then that the family groups the channels $F[0, 0] \dots F[m - 1, 0]$.

We can also have constructions of the form `family F[bounds] indices: idxs bounds: bds ::= P`, where P is a protocol, or branching with protocols on each case. We call these declarations groups, but there is no distinguished notation in the syntax so far.

In this new setting, we may introduce a channel directly or via a family of channels. The rules in our DSL may operate on a channel or an entire family at once. We refer to these by writing either `chn C` where C is a channel name that may include indices, e.g. $C[i, j, k]$ or `fam F[bounds]`, e.g. `fam SendOutShare[bound N + 2 bound N + 2 bound K]`. The reason why we must include the bounds with the family names is that we are allowed to *split* a family of channels: we can write a family of protocols $F[\text{bound } (N + 2) \text{ bound } (N + 2)]$ as the composition of $F[\text{bound } (N + 1) \text{ bound } (N + 2)]$ and $F[\text{fixedBound } (N + 1) \text{ bound } (N + 2)]$. In the presence of splitting, the family names are no longer unique, and we must be able to properly identify the family that a rule works with.

In a channel context we will write `input F[bounds] : T` to record that the family $F[\text{bounds}]$ is an input family of channels of type T . In a normal form for protocols, an internal family of channels `family F[bds] indices: idx bounds: bds ::= ...` appears as $\{F[bds] \text{ idx} : T\}$.

3.5 Exact Equality Proofs

In this section we present briefly how the IPDL rules are implemented. This is not essential to the user, but it helps build an understanding of what the current state of the prover is and how the rules operate on it.

Expression equality is implemented with the help of expression configurations, for efficiency reasons. An expression configuration is a tuple of the form `expConfig(Sigma, Gamma, T, e)`, where Σ is a signature, Γ is a type context, T is a type and e is an expression. Recall that the equality judgement for expressions is of the form $\Gamma \vdash e_1 = e_2 : T$. We will represent it with the help of Maude rewrite rules by writing

```

expConfig(Sigma, Gamma, T, e1)
=>
expConfig(Sigma, Gamma, T, e2)

```

Example 7: Congruence Rule for Application

We write

```

crl [app-cong] :
  expConfig(Sigma (f : T1 ~> T2), Gamma, T2, ap f M1)
=>
  expConfig(Sigma (f : T1 ~> T2), Gamma, T2, ap f M2)
if
  expConfig(Sigma (f : T1 ~> T2), Gamma, T1, M1)
=>
  expConfig(Sigma (f : T1 ~> T2), Gamma, T1, M2) .

```

At the reaction level, exact equality is given with judgements of the form $\Delta; \Gamma \vdash R_1 = R_2 : I \rightarrow \tau$. Let us consider the following example:

```
(A :: bool) (B :: bool); empty ⊢
x : bool <- return True ; if x then read A else read B = read A : {A, B} → bool.
The proof of this is obtained by applying the TRANS rule to
(A :: bool) (B :: bool); empty ⊢
x : bool <- return True ; if x then read A else read B =
if True then read A else read B : {A, B} → bool
```

that we prove by RET-BIND and

```
(A :: bool) (B :: bool); empty ⊢ if True then read A else read B = read A : {A, B} → bool
```

that we prove by IF-LEFT.

From a practical point of view, it is inconvenient to write this proof in this way, because we have to make explicit all intermediate steps, and this is tedious and error-prone. Instead, we will work with a transition system. Its states are *configurations* containing the context, i.e. the current signature Σ , Δ , Γ , I , T , and the current reaction: $\text{rConfig}(\Sigma, \Delta, \Gamma, R, I, T)$. The transitions in the system are determined by rewrite rules, which are obtained by orienting the axioms of the exact equality calculus from left to right. Since we can apply the SYM axiom, the choice of direction is not important.

Example 8: Reaction Equality

The IF-LEFT axiom becomes

```
cr1 [if-left] :
  rConfig(Sigma, Delta, Gamma, if True then R1 else R2, I, A, T)
=>
  rConfig(Sigma, Delta, Gamma, R1, I, A, T)
if typeOf(Sigma, Delta, Gamma, I, A, R1) == T
/\ typeOf(Sigma, Delta, Gamma, I, A, R2) == T .
```

We also employ the Maude strategy language to conveniently write application of the TRANS axiom as rule composition, denoted $;$. The proof in the example above becomes

```
srew
rConfig(emptySig, (A :: bool) (B :: bool), emptyTypeCtx,
  x : bool <- return True ; if x then read A else read B,
  (A, B), bool)
using ret-bind ; if-left .
```

and Maude returns the following result

```
Solution 1
rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
result ReactionConfig:
rConfig(emptySig, (A :: bool) (B :: bool), emptyTypeCtx,
  read A, (A , B), bool)
```

If the condition of a rule is a rewrite, we will need to explicitly provide a sub-proof for that step as well. For example, the rule CONG-BIND is

```
cr1 [cong-bind] :
  rConfig(Sigma, Delta, Gamma, x : T1 <- R1 ; R2, I , A, T2)
=>
  rConfig(Sigma, Delta, Gamma, x : T1 <- R3 ; R4, I, A, T2)
if rConfig(Sigma, Delta, Gamma, R1, I, A, T1)
=>
  rConfig(Sigma, Delta, Gamma, R3, I, A, T1)
/\ rConfig(Sigma, Delta, Gamma (x : T1), R2, I, A, T2)
=>
  rConfig(Sigma, Delta, Gamma (x : T1), R4, I, A, T2) .
```

and we can apply it to rewrite the reaction $x : \text{bool} \leftarrow \text{if True then read A else read B ; return x}$ to $x : \text{bool} \leftarrow \text{read A ; return x}$ by writing `cong-bind{if-left, idle}`.

The IF-EXT axiom has the particularity that it establishes an equality between reactions where a variable has been substituted with a term. Maude cannot apply this rule, because it cannot match that e occurs in R and replace this occurrence with true or false. For this reason, we have written it as a rule that replaces the occurrences of a variable q in R with true and false, respectively:

```
rl [if-intro-ext] :
  rConfig(Sigma, Delta, Gamma (q : bool), R, I, A, T)
=>
  rConfig(Sigma, Delta, Gamma (q : bool),
    if q then (R[q / True]) else (R[q / False]), I, A, T)
.
```

We have also introduced an alpha-renaming rule for convenience, as we can also derive it from the exact equality axioms. The reaction equality calculus uses the expression equality calculus, as illustrated in the rule below:

```
cr1 [cong-ret] :
  rConfig(Sigma, Delta, Gamma, return M1, I, A, T)
=>
  rConfig(Sigma, Delta, Gamma, return M2, I, A, T)
if expConfig(Sigma, Gamma, T, M1)
=>
  expConfig(Sigma, Gamma, T, M2) .
```

The same principle is applied for exact equality of protocols. This time we rewrite *protocol configurations* of the form $pConfig(Sigma, Delta, P, I, O)$ where $Sigma$ is a signature, $Delta$ is a channel context, P is a protocol, I is the set of inputs of P and O is the set of outputs of P . The rules of exact equality for protocols may make use of exact equality of reactions. For example, the CONG-REACT rule is

```
cr1 [CONG-REACT] :
  pConfig(Sigma, Delta (c :: T), c ::= R, I, c)
=>
  pConfig(Sigma, Delta (c :: T), c ::= R', I, c)
if rConfig(Sigma, Delta (c :: T), emptyTypeCtx, R, (I, c), T)
=>
  rConfig(Sigma, Delta (c :: T), emptyTypeCtx, R', (I, c), T)
/\ not c in I .
```

All these rewrites take place internally. The application of rules is hidden to the user and is guided by the IPDL DSL for proofs.

3.5.1 Prover State

The prover maintains a state consisting of

- a stack of protocol configurations. We work on the top of the stack, and we allow to temporarily push a new configuration on the stack to try proofs, as we will explain later;
- all declared channel contexts, identified by name;
- all declared protocols, identified by name;
- all declared embeddings, identified by name;
- an internal Maude module. This allows us to generate Maude code from definitions and declarations in the DSL. Here is where we store assumptions and generate on the fly strategies for some DSL constructs;
- the names of all assumptions, to avoid duplicates;
- the names of all parameters;

- a configuration - we have a DSL construct allowing us to save the configuration on top of the stack and reuse it.

3.6 Derived Rules

As introducing normal forms and families of channels impacts on the syntax, we introduce new rules that operate effectively with them. Moreover, we introduce several rules that do not relate to normal forms or families of channels, but are convenient to have in practice, as they provide abbreviations for longer proofs. It is important to notice that both normal forms and families of channels are just notational variants of protocols that can already be written with the original IPDL syntax. All derived rules can be proven sound using only the core IPDL rules or rules that we have already proven to be sound.

We start with the derived rules over the core calculus without normal forms or families. These allow us to do transformations at the level of reactions and are of the following kind:

- if we have the same reaction on the two branches of an `if` reaction, we can drop the branching

```
cr1 [same-reaction-if] :
  rConfig(Sigma, Delta, Gamma, if M then R else R, I, A, T)
=>
  rConfig(Sigma, Delta, Gamma, R, I, A, T)
if typeOf(Sigma, Gamma, M) == bool .
```

- take `if` branching out of a binding and its converse, possibly involving multiple conditionals, as long as the conditions of the `ifs` match. We show only one of these rules here:

```
r1 [if-over-bind-same] :
  rConfig(Sigma, Delta, Gamma,
    x : T1 <- if M then R1 else R2 ;
    if M then R3 else R4, I, A, T)
=>
  rConfig(Sigma, Delta, Gamma,
    if M then (x : T1 <- R1 ; R3)
      else (x : T1 <- R2 ; R4), I, A, T) .
```

- bring `if` branching in front of sampling:

```
cr1 [samp-over-if] :
  rConfig(Sigma (d : T2 ~>> T1), Delta, Gamma,
    x : T1 <- samp (d < e >) ;
    if M then R1 else R2, I, A, T)
=>
  rConfig(Sigma (d : T2 ~>> T1), Delta, Gamma,
    if M then x : T1 <- samp (d < e >) ;
      R1
    else x : T1 <- samp (d < e >) ;
      R2, I, A, T)
if typeOf(Sigma, Gamma, M) == bool .
```

- do an alpha renaming of variables.

The rules for normal forms are counterparts of the IPDL core rules. We show here only the substitution rule for normal forms:

```

cr1 [SUBST-nf] :
  pConfig(Sigma, Delta,
    (cn1 ::= R1) ||
    (cn2 ::= nf( (x1 : T1 <- read cn1) BRL , R2) ),
    I, (chn cn1, chn cn2), A)
=>
  pConfig(Sigma, Delta,
    (cn1 ::= R1) ||
    (cn2 ::= preNF((x1 : T1 <~ R1) BRL , R2)),
    I, (chn cn1, chn cn2), A)
if isSampFree(R1) /\ typeOf(Sigma, Delta, emptyTypeContext, (I, chn cn1, chn cn2), A, R1) == T1 .

```

The rule states that if we read in the channel `cn2` from a channel `cn1`, we can replace that read with the reaction `R1` assigned to `cn1`, if `R1` does not include any samplings. The result is then a pre-normal form.

The rules for families of channels are of two kinds. On one side, we have counterparts of the IPDL core rules. We show here only the substitution rule for families:

```

cr1 [subst-families-gen] :
  pConfig(Sigma, Delta,
    (family (F2[blist2]) nlist2 blist2 ::= R2)
    ||
    (family (F1[blist1]) nlist1 blist1 ::=
      nf((x : T <- read (F2[tlist]) ) BRL,
        R1
      )
    ),
    I, (fam (F1[blist1]), fam (F2[blist2])), A)
  )
=>
  pConfig(Sigma, Delta,
    (family (F2[blist2]) nlist2 blist2 ::= R2)
    ||
    (family (F1[blist1]) nlist1 blist1 ::=
      preNF((x : T <~ R2[nlist / tlist]) BRL,
        R1
      )
    ),
    I, (fam (F1[blist1]), fam (F2[blist2])), A)
  )
if isSampFree(R2) /\
  typeOf(Sigma, Delta, emptyTypeContext, (fam (F1[blist1]), fam (F2[blist2])), I), A, R2) == T .

```

The rule states that if we read in the family `F2` from a family `F1`, we can replace that read with the reaction obtained by substituting the indices of the family `F2` with the terms occurring in the read in the reaction `R2` assigned to `F1`, if `R2` does not include any samplings. The result is then a pre-normal form.

On the other hand, we have rules that allow us to operate with the new structure imposed by families, e.g. a congruence rule for families, a congruence rule for branching in families, a rule for accessing the protocol assigned to a group in order to rewrite another protocol composed to the group, a similar rule for using the protocol on the branch of a group, when we can prove the condition of a branch, rules for proofs by case distinction on the branches of a family, rules for rewriting a family by inductive reasoning. We refrain from introducing these rules in full detail here. Instead we will present examples of how the rules apply using their corresponding DSL construct.

3.7 Operators on protocols

Listing an entire large protocol again for a small modification is tedious. Instead, we introduce a number of operators on protocols, all giving back protocols. We can use them wherever a protocol is expected in the DSL. Below we illustrate the

syntax by giving an example. In each case we show the protocol before applying the operator, how to write the operator application and the result.

Change a channel Changing the value assigned to a channel or family:

```
C ::= i : bool <- read In; return i || D ::= return False

change D with return True in P

C ::= i : bool <- read In; return i || D ::= return True
```

Add internal Adding a new internal channel or family:

```
C ::= i : bool <- read In; return i || D ::= return False

extend P with internal channel X typed: bool assigned: return True

newNF( (X : bool),
C ::= i : bool <- read In; return i || D ::= return False || X ::= return True
)
```

and

```
C ::= i : bool <- read In; return i || D ::= return False

extend P with internal family F
  indices: i bounds: bound N typed: bool assigned: return True

newNF( {F[bound N] i : bool},
C ::= i : bool <- read In; return i || D ::= return False ||
family F[bound N] indices: i bounds: bound N ::= return True
)
```

Grouping Grouping a list of families:

```
family F1[bound N] indices: i bounds: bound N ::= R1 ||
... ||
family Fk[bound N] indices: i bounds: bound N ::= Rk ||
Q

create group G indices: i bounds: bound N
  from fam F1[bound N], ..., fam Fk[bound N]

family G[bound N] indices: i bounds: bound N ::= (
family F1[bound N] indices: i bounds: bound N ::= R1 ||
... ||
family Fk[bound N] indices: i bounds: bound N ::= Rk ) ||
Q
```

Change on a branch Modify a family/channel only on a branch of a group:

```

family G[bound N] indices: i bounds: bound N ::=
  when isHonest(i) -->
    C ::= read in || D ::= c : bool <- read c ; return neg c
  when isSemiHonest(i) -->
    C ::= read in || D ::= d : bool <- read d ; return d

on branch when isSemiHonest(i) of group G[bound N]
  change chn C with return False

family G[bound N] indices: i bounds: bound N ::=
  when isHonest(i) -->
    C ::= read in || D ::= c : bool <- read c ; return neg c
  when isSemiHonest(i) -->
    C ::= return False || D ::= d : bool <- read d ; return d

```

Refactor structure Group a protocol fragment into an internal normal form:

```

newNF( (C1 : bool) (C2 : bool) ,
  C1 ::= return True || C2 ::= return True || C3 ::= return False
)

gather C1, C3 from P hiding C1

newNF((C2 : bool),
  C3 ::= return False ||
  newNF((C1 : bool), C1 ::= return True || C2 ::= return True)
)

```

Apply an embedding apply an embedding to a protocol: `embed(P, phi)`.

3.8 Proof Steps for Exact Equality

In this section we start introducing the DSL for proving. We present the language constructs with the help of examples. Each example shows a protocol, a proof step that we apply to it and the resulting protocol. Recall that we have several layers of exact equality rules: the core calculus, the derived rules over it, the rules for normal forms and the rules for families of channels. The latter two include counterparts of the rules of the proof calculus. The aim of the DSL is to hide as much as possible of this complexity from the user, who should ideally write only “substitute family X into family Y” and the identification of the appropriate rule to be used happens internally. This works only to some extent because sometimes more parameters must be specified. To ease presentation we have chosen to group the proof steps in our DSL by their shared core rule, e.g. all different form of substitution.

The rule REFL is not explicitly applied, as it is implied by the properties of the rewrite relation in Maude. The rules CONG-COMM and CONG-ASSOC are also not applied explicitly, as it sufficed to specify the parallel composition in Maude as a commutative and associative operation.

The rules INPUT-UNUSED and CONG-NEW are applied inside the strategy definitions, and their usage is not visible to the user.

IDLE

We have two rules that leave the current protocol unchanged, `idle` and `todo`. Using the first one denotes that we do not want to do a proof at all and the second is used as a temporary placeholder.

For an example of using `idle`, see Ex. 30. For an example of using `todo`, see Ex. 57.

AXIOM

Axioms are written using assumptions. We have assumptions at various levels:

- expression-level assumptions,
- reaction-level assumptions,
- protocol-level assumptions.

Expression-level Assumptions. Expression-level assumptions are written

```
expression-assumption FLAG NAME using VARS : TCTXT |=
  E1 = E2 .
```

where **FLAG** is either `%manual` or `%automatic` and impacts the way the axiom is added in the generated Maude module - either as an equation, and then Maude applies it automatically, or as a rewrite rule, and we must apply it ourselves, **NAME** is a new identifier giving the name of the assumption, **VARS** is a list of names of variables appearing in the expressions **E1**, **E2**, as we need to distinguish between the variables and the constants appearing in the axiom and we do this by listing the variables in the **using** clause and **TCTXT** is a type context, written as a list of typed variables **x** : **T**.

Example 9: Assumption Applied Automatically

The following assumption states that inverses cancel on the right:

```
expression-assumption %automatic mulInv using x y : (x : msg) (y : msg) |=
  mul( mul((x, y)), inv(y)) = x .
```

Maude will apply this equation automatically whenever it encounters a term matching the left-hand side of it.

Example 10: Assumption Applied Manually

The following assumption states that consecutive exponents commute. If we were to add it as an equation, we would lose termination, as Maude can apply it infinitely many times. Instead we add it as a rewrite:

```
expression-assumption %manual commute using k l : (k : key) (l : key) |=
  exp( exp( (g ()), l) , k)
  =
  exp( exp( (g ()), k) , l)
  .
```

The DSL syntax for making use of an expression-level assumption that has the flag set to `%manual` is

```
use expression assumption NAME on X at i
```

where **X** is the family or the channel where we want to apply the axiom and **i** allows us to select the *i*-th occurrence of a term matching the left-hand side of the assumption in that family or channel.

Example 11: Use an Axiom

The `neg-over-xor` expression assumption states that negation commutes with xor:

```
ShareSum[N + 1, B] ::=
  x : bool <- read Share[N + 1, wire0(B)] ;
  xs : bool <- read ShareSum[N, B] ;
  return xor((xs, neg(x)))

expression-assumption %manual neg-over-xor using x, y : (x : bool) (y : bool) |=
  xor( (x, neg(y)) ) = neg( xor((x, y)) ) .

use expression assumption neg-over-xor on chn ShareSum[N + 1, B] at 1

ShareSum[N + 1, B] ::=
  x : bool <- read Share[N + 1, wire0(B)] ;
  xs : bool <- read ShareSum[N, B] ;
  return neg xor((xs, x))
```

Reaction-level Assumptions Reaction-level assumptions are written

```
reaction-assumption NAME using VARS :
  CCTXT ; TCTXT |=
  R1 = R2 .
```

where **NAME** is a new identifier giving the name of the assumption, **VARs** is a list of names of variables appearing in the reactions **R1**, **R2**, **CCTXT** is a channel context written as a list of typed channels of the form (**chn** **C** :: **T**) and a list of typed families of the form (**fam** **F**[**bounds**] :: **T**), **TCTXT** is a type context, written as a list of typed variables **x** :: **T**.

The DSL syntax for making use of a reaction assumption is

```
use assumption NAME on X
```

where **X** is the family or the channel where we want to apply the axiom.

Example 12: Reaction Assumption

The following assumption states that sampling a random secret key **k** from **unif_key** and returning g^k is the same as sampling from **unif_msg** directly. The assumption requires no channel and type contexts and fixes no inputs:

```
reaction-assumption sampExp using k :
  emptyCtxt ; emptyTtxt |=
  k : key <- samp unif_key(); return exp((g()), k))
  =
  samp unif_msg()
```

We can then apply this assumption:

```
SharedKey ::=
  l : key <- samp unif_key();
  return exp((g()), l))

use assumption sampExp on chn SharedKey

SharedKey ::= samp unif_msg()
```

Since **k** appears in the **using** clause of the assumption, the name of the variable in **SharedKey** does not have to be **k**.

Protocol-level Assumptions Protocol-level assumptions are written

```
protocol-assumption NAME using VARS :
  CCTXT ; INS ; ASSUMS |=
  P1 = P2 .
```

where **NAME** is a new identifier giving the name of the assumption, **VARs** is a list of names of variables appearing in the protocols **P1**, **P2**, **CCTXT** is a channel context written as a list of typed channels of the form (**chn** **C** :: **T**) and a list of typed families of the form (**fam** **F**[**bounds**] :: **T**), **ASSUMS** is a list of boolean terms allowing to require further hypotheses to hold and **INS** lists the channels that we are allowed to read from. Note that **VARs** contains now both variable and channel names, as we also want to be able to distinguish the possible interpretations of the protocol **C** ::= **read** **D** – is the channel **C** required from a channel named **D** or is the channel **C** required to read from an arbitrary channel.

Example 13: Encryption Schema Correctness

The correctness assumption on an encryption schema states that encoding and decoding a single message with the same key yields the original message:

```
protocol-assumption enc-dec-correctness using m k c keyN encN decN inN :
  (chn keyN :: key) (chn encN :: ctxt) (chn decN :: msg) (chn inN :: msg) ; chn inN ; no assumptions
  |=
    ( keyN ::= samp gen_key (()) )
    ||
    ( encN ::= m : msg <- read inN ; k : key <- read keyN ; samp enc((m, k)) )
    ||
    ( decN ::= c : ctxt <- read encN ; k : key <- read keyN ; return dec((c, k)) )
  =
    ( keyN ::= samp gen_key (()) )
    ||
    ( encN ::= m : msg <- read inN ; k : key <- read keyN ; samp enc((m, k)) )
    ||
    ( decN ::= in : msg <- read inN ; return in )
  .
```

Note that all variables and channels appearing in the axiom also appear in the **vars** clause, so any choice of names for them would match; we do not require e.g. to read from a channel named `inN`. We explicitly state that `inN` must be an input channel.

We can apply this assumption by writing

```
( Key ::= samp gen_key (()) ) ||
( Enc ::= m : msg <- read In ; k : key <- read Key ; samp enc((m, k)) ) ||
( Dec ::= c : ctxt <- read Enc ; k : key <- read Key ; return dec((c, k)) )

use assumption enc-dec-correctness on chn Key, chn Dec, chn Enc

( Key ::= samp gen_key (()) ) ||
( Enc ::= m : msg <- read In ; k : key <- read Key ; samp enc((m, k)) ) ||
( Dec ::= in : msg <- read In ; return in )
```

HYPOTHESIS

Recall that we maintain the declared hypotheses in the current configuration. Here we also store facts that are temporarily true, e.g. the index of a family is smaller than the corresponding bound or the condition of a branch is true during a proof by case distinction.

We write `use hypothesis for x in X` to replace all occurrences of a variable `x` in the channel or family `X` with the expression `t` if the current set of hypotheses contains `x = t`.

Example 14: Use Hypotheses

Assume we do a case distinction proof and work on the case `B = 0`, which is added to the hypotheses:

```
Sqr[B] ::= (when B = 0 --> c : bool <- read ColSum[B, B] ; return c) ;;
          (otherwise --> s : bool <- read Sqr[B] ; return s)

Sqr[B] ::= c : bool <- read ColSum[B, B] ; return c

use hypothesis for B in chn Sqr[B]

Sqr[B] ::= c : bool <- read ColSum[0, 0] ; return c
```

CONG-REACT

The CONG-REACT rule is most commonly used in strategies, so its application is not visible to the user. We have introduced a rule that allows us to rewrite $C ::= \text{nf}(\dots, R1)$ to $C ::= \text{nf}(\dots, R2)$ if we can rewrite $R1$ to $R2$ using the reaction equality rules. A similar rule works for pre-normal forms, and both rules have counterparts for families instead of channels.

We write **on reaction of X (reactionProof)** where X is the family or the channel where we want to apply the rule and **reactionProof** is a proof using reaction equality rules.

We only needed this rule for applications of the IF-EXT rule so far, so the proof language for reactions is restricted to IF-EXT and transitivity rule, as we show in the example below.

Example 15: Applying if-ext

In the following fragment, we have **xor** and ****** for exclusive or and conjunction, respectively.

```
family RcvdBit[bound N + 2 bound N + 2 bound K]
indices: n, m, k bounds: bound N + 2 bound N + 2 bound K ::=
  nf(
    (b : bool <- read SendBit[m, n, k])
    (xm : bool <- read Share[m, wire0(k)])
    (xn : bool <- read Share[n, wire0(k)])
    (ym : bool <- read Share[m, wire1(k)])
    (yn : bool <- read Share[n, wire1(k)]),
    return xor (
      ( xor ( ( **((xm, yn)) ,
                **((xn, ym)) )
        ),
      b)
    )
  )

on reaction
of fam RcvdBit[bound N + 2 bound N + 2 bound K] (
  if-ext on yn then if-ext on xn
)

family RcvdBit[bound N + 2 bound N + 2 bound K]
indices: n, m, k bounds: bound N + 2 bound N + 2 bound K ::=
  nf(
    (b : bool <- read SendBit[m, n, k])
    (xm : bool <- read Share[m, wire0(k)])
    (xn : bool <- read Share[n, wire0(k)])
    (ym : bool <- read Share[m, wire1(k)])
    (yn : bool <- read Share[n, wire1(k)]),
    if xn
    then
      if yn
      then xor( ( xor ((xm, ym)) , b) )
      else xor ((ym, b))
    else
      if yn
      then xor ((xm, b))
      else b
  )
```

Moreover, we have a language construct for the application of the UNIT-EXT rule. We write **unit-ext on X at i** where i is a natural number to replace the i -th occurrence of a variable of type **unit** with **()**.

Example 16: Applying unit-ext

```

ShareSum-OK[M, B] ::=
  w : unit <- read Wire-OK[B] ; return w

unit-ext on chn ShareSum-OK[M, B] at 1

ShareSum-OK[M, B] ::=
  w : unit <- read Wire-OK[B] ; return ()

```

CONG-COMP

The congruence rule for parallel composition is mostly used inside strategies so we don't typically need to make its usage explicit. We write `focus X1, ... Xn ( proof )` to run `proof` on the channels and families `X1, ... Xn` from the current protocol.

Example 17: Applying congruence rule for parallel composition

```

C1 ::= return True || C2 ::= c3 : bool <- read C3 ; return c3 || C3 ::= in : bool <- read In ; return in
focus chn C2, chn C3 (subst chn C3 into chn C2)

C1 ::= return True || C2 ::= in : bool <- read In ; return in || C3 ::= in : bool <- read In ; return in

```

TRANS

We write `proofStep1 then proofStep2`.

We encountered a performance issue when parsing multi-line text and we have noticed it is more convenient to write

```

proofStep1
proofStep2

```

especially for large proofs. However, we cannot do this inside subproofs, meaning the following proof is **not valid**:

```

sym from P1 (
  subst chn C3 into chn C2
  subst chn C2 into chn C1
)

```

Example 18: Chaining substitutions

```

C1 ::= c2 : bool <- read C2 ; return c2 || C2 ::= c3 : bool <- read C3 ; return c3 || C3 ::= in : bool
  <- read In ; return in

subst chn C3 into chn C2 then
subst chn C2 into chn C1

C1 ::= in : bool <- read In ; return in || C2 ::= in : bool <- read In ; return in || C3 ::= in : bool
  <- read In ; return in

```

SYM

We have two variants: `sym from P` (proof) and `sym from P over Delta` (proof) where the second is needed for splitting.

Example 19: Applying sym

Here we also illustrate how the operators on protocols work:

```

Key-Alice ::=
  ob : msg <- read OkPublicKey-BobAdvNet ;
  ka : key <- read SecretKey-Alice ;
  kb : key <- read SecretKey-Bob ;
  return exp((exp((g(()), ka)), kb))
||
SharedKey ::=
  ka : key <- read SecretKey-Alice ;
  kb : key <- read SecretKey-Bob ;
  return exp((exp((g(()), ka)), kb))

sym from change chn Key-Alice with
ob : msg <- read OkPublicKey-BobAdvNet ;
sk : msg <- read SharedKey ;
return sk
in currentProtocol(
  subst chn SharedKey into chn Key-Alice
)

Key-Alice ::=
  ob : msg <- read OkPublicKey-BobAdvNet ;
  sk : msg <- read SharedKey ;
  return sk
|| SharedKey ::= ... # unchanged

```

SUBST

Here we have many variations, but we used the Maude strategy language to write strategies that combine many of them. We make use of the following constructs:

- `subst X into Y` where `X`, `Y` can be channels or families.

Example 20: Simple substitution

```

Key-Alice ::=
  ob : msg <- read OkPublicKey-BobAdvNet ;
  sk : msg <- read SharedKey ;
  return sk
||
SharedKey ::=
  ka : key <- read SecretKey-Alice ;
  kb : key <- read SecretKey-Bob ;
  return exp((exp((g(()), ka)), kb))

  subst chn SharedKey into chn Key-Alice

Key-Alice ::=
  ob : msg <- read OkPublicKey-BobAdvNet ;
  ka : key <- read SecretKey-Alice ;
  kb : key <- read SecretKey-Bob ;
  return exp((exp((g(()), ka)), kb))
|| SharedKey ::= ... # unchanged

```

- **subst** X **into** Y **at** c which allows us to choose which of the several occurrences of **read** X in Y we use for a substitution by specifying the variable c appearing in the $c \leftarrow \text{read } X$;

Example 21: Specifying where substitution takes place

The channel `ShareSum[N + 1, B]` reads from two channels from the family `Wire`. We want to control the names after doing both substitutions, so we should not let Maude decide which substitution to make first. Instead we specify that first we want to substitute the read assigned to x :

```
ShareSum[N + 1, B] ::=
  x : bool <- read Wire[wire0(B)] ;
  y : bool <- read Wire[wire1(B)] ;
  return xor((x, y))
||
family Wire[bound B] indices: k bounds: bound B ::=
  s : bool <- read ShareSum[N + 1, k] ; return s

subst fam Wire[bound B] into chn ShareSum[N + 1, B] at x

ShareSum[N + 1, B] ::=
  s : bool <- read ShareSum[N + 1, wire0(B)] ;
  y : bool <- read Wire[wire1(B)] ;
  return xor((s, y))
|| family Wire[bound B] ... # unchanged
```

- substituting from a branch of a family or a channel by specifying the condition of the branch with **subst branch** COND **of** X **into** Y and **subst branch** COND **of** X **into** Y **at** c where COND is the condition of the branch, X , Y are channels and/or families and c allows us to identify which **read** should be substituted.

Example 22: Substitution from a branch

Notice that $N + 1$ cannot be equal to 0:

```
ShareSum[N + 1, B] ::=
  c : bool <- read ColSum[N + 1, N + 1] ;
  return c
||
family ColSum[bound (N + 2) bound (N + 2)]
  indices: i, j bounds: bound (N + 2) bound (N + 2) ::=
    (when (0 = i) -->
      c : bool <- read Col[0, j] ; return c
    )
  ;;
  (when (not 0 = i) -->
    bi : bool <- read Col[i, j] ;
    bs : bool <- read ColSum[i - 1, j] ;
    return xor((bs, bi))
  )

subst branch when (not (i = 0))
of fam ColSum[bound (N + 2) bound (N + 2)]
into chn ShareSum[N + 1, B]

ShareSum[N + 1, B] ::=
  bi : bool <- read Col[N + 1, N + 1] ;
  bs : bool <- read ColSum[N, N + 1] ;
  return xor((bs, bi))
|| family ColSum... # unchanged
```

- **reverse subst** X **in** Y **using** Z applies an older strategy for doing a reverse substitution. It can be replaced with an equivalent construction involving symmetry and substitution.

- **subst or diverge** X **into** Y **var** c performs a substitution of X into Y handling in the same time the case when X diverges by introducing a corresponding case when Y diverges and c specifies the name of the new variable.

Example 23: Substitute and diverge

```

(family OTOut[bound N + 2 bound N + 2 bound K]
  indices: n, m, k
  bounds: bound N + 2 bound N + 2 bound K ::=
  m0 : bool <- read OTMsg-0[n, m, k] ;
  m1 : bool <- read OTMsg-1[n, m, k] ;
  m2 : bool <- read OTMsg-2[n, m, k] ;
  m3 : bool <- read OTMsg-3[n, m, k] ;
  c0 : bool <- read OTChc-0[n, m, k] ;
  c1 : bool <- read OTChc-1[n, m, k] ;
  if c0
  then if c1 then return m3 else return m2
  else if c1 then return m1 else return m0
)
||
( family OTMsg-0[bound N + 2 bound N + 2 bound K ]
  indices: i, j, k
  bounds: bound N + 2 bound N + 2 bound K ::=
  (when (i < j) -->
    b : bool <- read SendBit[i, j, k] ;
    x : bool <- read Share[i, wire0(k)] ;
    y : bool <- read Share[i, wire1(k)] ;
    return b
  )
  ;;
  (when (j <= i) -->
    om : bool <- read OTMsg-0[i, j, k];
    return om
  )
)

subst or diverge fam OTMsg-0[bound N + 2 bound N + 2 bound K]
into fam OTOut[bound N + 2 bound N + 2 bound K] var z

family OTOut[bound N + 2 bound N + 2 bound K]
indices: n, m, k
bounds: bound N + 2 bound N + 2 bound K ::=
(when (n < m) -->
  b : bool <- read SendBit[i, j, k] ;
  x : bool <- read Share[n, wire0(k)] ;
  y : bool <- read Share[n, wire1(k)] ;
  m1 : bool <- read OTMsg-1[n, m, k] ;
  m2 : bool <- read OTMsg-2[n, m, k] ;
  m3 : bool <- read OTMsg-3[n, m, k] ;
  c0 : bool <- read OTChc-0[n, m, k] ;
  c1 : bool <- read OTChc-1[n, m, k] ;
  if c0
  then if c1 then return m3 else return m2
  else if c1 then return m1 else return b
)
;;
(when (m <= n) -->
  z : bool <- read OTOut[m, n, k];
  return z
)

```

DROP

We have a rule that allows us to remove gratuitous dependencies between channels. We write `drop read X from Y` and `drop read branch COND of X from Y` to remove the dependency on `X` from `Y` or respectively from the branch with condition `COND` of `Y`.

Example 24: Drop read

```
SharedKey ::= samp unif_msg (()) ||
Out ::= m : msg <- read In ;
      oka : msg <- read OkKey-AliceAdvId ;
      okb : msg <- read OkKey-BobAdvId ;
      okc : msg <- read OkCtxtAdvNet ;
      sk : msg <- read SharedKey ;
      return m

drop read chn SharedKey from chn

Out ::= m : msg <- read In ;
      oka : msg <- read OkKey-AliceAdvId ;
      okb : msg <- read OkKey-BobAdvId ;
      okc : msg <- read OkCtxtAdvNet ;
      return m
|| SharedKey ::= samp unif_msg (())
```

ABSORB

The absorb rule allows us to remove an internal channel if the rest of the protocol does not read from it. We write `absorb X` and `group-absorb group F[bounds]` which performs multiple absorbs of several channels and families grouped in `F`.

Example 25: Absorb

Assuming we start with a protocol where the family `Recv` is **internal** and the only read from it is in the family `Out`, then we can substitute `Recv` into `Out` and absorb `Recv`:

```
newfamily Recv[bound n] indices: i bounds: bound n : ctxt in
family Out[bound n] indices: i bounds: bound n ::=
  c : ctxt <- read Recv[i] ;
  k : key <- read Key ;
  return dec((c, k ))
||
family Recv[bound n] indices: i bounds: bound n ::=
  okCtxt : unit <- read OkCtxtAdvNet[i] ;
  send : ctxt <- read Send[i] ;
  return send
)

subst fam Recv[bound n]
into fam Out[bound n] then
absorb fam Recv[bound n]

family Out[bound n] indices: i bounds: bound n ::=
  okCtxt : unit <- read OkCtxtAdvNet[i] ;
  send : ctxt <- read Send[i] ;
  k : key <- read Key ;
  return dec((send, k ))
```

`Recv` is removed both from the main protocol and from the list of hidden channels maintained by the normal form.

FOLD

We use the construct `fold X into Y`.

Example 26: Folding an internal channel

```

new D : bool in (
  ( D ::= samp flip (()) )
  ||
  ( C ::= x : bool <- read In ; y : bool <- read D ; return y )
)

fold chn D into chn C

C ::= x : bool <- read In ; samp flip (())

```

SPLITTING

Splitting was introduced in Sec. 3.4. The core cases are

- `split fam F[bounds] on first index`;
- `split fam F[bounds] on second index`;
- `unsplit fam F[bounds]` unplits a family with one index, and thus it suffices to specify the family, as the channel's name can be computed;
- `unsplit fam F1[bounds1] and fam F2[bounds2]` unplits a family with more than one index, and here we must specify both arguments to be able to control the case of multiple splits on different indices for the same family.

Example 27: Splitting

```

family F[bound N + 1 bound K] indices: i, j
bounds: bound N + 1 bound K ::= R

split fam F[bound N + 1 bound K] on first index

(family F[bound N bound K] indices: i, j
 bounds: bound N bound K ::= R)
||
(family F[fixedBound N bound K] indices: i, j
 bounds: fixedBound N bound K ::= R[i / N])

```

We write `unsplit fam F[bound N bound K] and fam F[fixedBound N bound K]` to undo this step.

Then we handle the cases of splitting inside a group or on branches:

- `split fam F[bounds1] on first index inside group G[bounds2]`;
- `split fam F[bounds1] inside group G[bounds2]`;
- `unsplit fam F1[bounds1] and fam F2[bounds2] inside group G[bounds3]`;
- `on all branches of fam G[bounds1] split fam F[bounds2] on first index`

Example 28: Splitting

```

family G... ::=
when bt1 --> (P1 ||
family F[bound N + 1 bound K] indices: i, j
bounds: bound N + 1 bound K := R1)
;; ... ;;
when btn --> (Pn ||
family F[bound N + 1 bound K] indices: i, j
bounds: bound N + 1 bound K := Rn)

on all branches of fam G...
split fam F[bound N + 1 bound K] on first index

family G... ::=
when bt1 --> (P1 ||
family F[bound N bound K] indices: i, j
bounds: bound N bound K := R1 ||
family F[fixedBound N bound K] indices: i, j
bounds: fixedBound N bound K := R1[i / N])
;; ... ;;
when btn --> (Pn ||
family F[bound N bound K] indices: i, j
bounds: bound N bound K := Rn ||
family F[fixedBound N bound K] indices: i, j
bounds: fixedBound N bound K := Rn[i / N])

```

- on all branches of fam G[bounds1] split fam F[bounds2] on second index.

DIVERGE

Assume a channel/family reads from a divergent channel/family. Then it is divergent itself. We write **diverge on X with Y** **var v**. The result is that X will be assigned **v <- read X ; return v** where v is the variable name given in the proof step.

Example 29: Diverging

```

C ::= d : bool <- read D ; ... ; return exp
||
D ::= d : bool <- read D ; return d

diverge on chn C with chn D var c

C ::= c : bool <- read C ; return c
||
D ::= d : bool <- read D ; return d

```

We write **sym var x terms t1 ... tn diverge X into Y var y** to replace the current value of X with a read from Y, provided we can get back the original value with a subst or diverge proof. This DSL construct is an abbreviation and can be avoided.

CASE DISTINCTION

We write **case distinction on X (PROOFS)**, where X is a channel or a family. For each branch we must have a proof in the list of PROOFS, written **case: (PROOF)**.

Example 30: Case distinction proof

We leave the first branch of `SumCommit` unchanged using `idle` and on the second branch we rename two variables:

```
family SumCommit[bound N + 3] indices: j bounds: bound N + 3 ::=
  when 0 = j --> return False ;;
  otherwise -->
    c : bool <- read Commit[j - 1] ;
    x : bool <- read SumCommit[j - 1] ;
    return xor((x ,c))

case distinction on family SumCommit[bound N + 3] (
  case: (
    idle
  )
  case: (
    rename c to cj, x to xj in fam SumCommit[bound N + 3]
  )
)

family SumCommit[bound N + 3] indices: j bounds: bound N + 3 ::=
  when 0 = j --> return False ;;
  otherwise -->
    cj : bool <- read Commit[j - 1] ;
    xj : bool <- read SumCommit[j - 1] ;
    return xor((xj ,cj))
```

We can also rewrite only on a branch by writing `on branch COND of X (PROOF)`, and a case distinction is equivalent to rewriting separately on each of the branches.

Example 31: Proof on a branch

```
(C ::= x : bool <- read X ; return x) ||
(family F[bound N] indices: i bounds: bound N ::=
  ( (when isHonest(i) --> c : bool <- read C ; i : bool <- read In[i] ; return xor((c, i)))
    ;;
    (otherwise --> f : bool <- read F[i] ; return f)
  )
)

on branch isHonest(i) of family F[bound N](
  subst chn C into fam F[bound N]
)

(C ::= x : bool <- read X ; return x) ||
(family F[bound N] indices: i bounds: bound N ::=
  ( (when isHonest(i) --> x : bool <- read X ; i : bool <- read In[i] ; return xor((x, i)))
    ;;
    (otherwise --> f : bool <- read F[i] ; return f)
  )
)
```

We can rewrite using a branch of a group or family, if its condition holds, using `using branch COND of X (proof)`. This typically happens in a case distinction proof, when substituting from another family or channel with branching. We can diverge using the branch of a family, using `diverge on X with branch COND of X var v` where `v` is the name we choose for the variable that we diverge on.

Example 32: Using a branch

We typically employ this rule when doing a proof by case distinction and we want to use the corresponding case of a group or family:

```
family G[bound N] indices: i bounds: bound N ::=
when isHonest(i) -->
  (family C[bound N] indices: i bounds: bound N ::= return True
  ||
  family D[bound N] indices: i bounds: bound N ::= return True
  )
;;
when isSemiHonest(i) -->
  (family C[bound N] indices: i bounds: bound N ::= return False
  ||
  family D[bound N] indices: i bounds: bound N ::= return False
  )

family F[bound N] indices: i bounds: bound N ::=
when isHonest(i) --> c : bool <- read C[i] ; return c
when isSemiHonest(i) --> f : bool <- read F[i] ; return f

on branch isHonest(i) of family F[bound N] (
  using branch when isHonest(i) of group G[bound N] (
    subst fam C[bound N] into fam F[bound N]
  )
)

family F[bound N] indices: i bounds: bound N ::=
when isHonest(i) --> return True
when isSemiHonest(i) --> f : bool <- read F[i] ; return f
```

ADD INTERNALS

We may introduce a new internal channel or family in a sound way, as they are not used anywhere in the protocol and the ABSORB rule allows us to remove them.

We write

```
add internal channel C typed: T assigned: R

add internal family F indices: IDXS bounds: BDS typed: T assigned: R
```

where C , F are names, T is a type, R is a reaction, $IDXS$ is a list of indices, BDS is a list of bounds of same length as the list of indices. With the same construct we can also assign a protocol to a family or a definition with branching for both channels and protocols.

Example 33: Adding an internal family

We can add a new family as below:

```
newNF(..., P)
add internal family Dec
indices: i bounds: bound n typed: msg
assigned:
  k : key <- read Key ;
  send : ctxt <- read Send [i] ;
  return dec ( (send, k) )
newNF({Dec [bound n] i : ctxt} ..., P || family Dec ...)
```

INDUCTION

The induction rules allow us to rewrite the reaction or the protocol assigned to a family. A proof by induction is done as follows. Assume we start with $P \parallel \text{family } F[\text{bound } N] \text{ indices: } i \text{ bounds: bound } N ::= R1$ and we want to prove by induction on i that we can rewrite to $P \parallel \text{family } F[\text{bound } N] \text{ indices: } i \text{ bounds: bound } N ::= R2$. We record the inductive hypothesis that the property holds up to an arbitrarily chosen $x < N - 1$: $\text{family } F[\text{bound } x] \text{ indices: } i \text{ bounds: bound } x ::= R2$ and we must show in the inductive step that $P1 \parallel \text{family } F[\text{bound } x] \text{ indices: } i \text{ bounds: bound } x ::= R2 \parallel F[x] ::= R1[i/x]$ rewrites to $P1 \parallel \text{family } F[\text{bound } x] \text{ indices: } i \text{ bounds: bound } x ::= R2 \parallel F[x] ::= R2[i/x]$ where P gets divided as $P1 \parallel P2$. We must separate P in two parts, otherwise the protocol $P \parallel \text{family } F[\text{bound } x] \text{ indices: } i \text{ bounds: bound } x ::= R2 \parallel F[x] ::= R1[i/x]$ may have fewer channels than necessary to type check. Consider the case when P contains $\text{family } G[\text{bound } N] \text{ indices: } i \text{ bounds: bound } N ::= \text{read } F[i]$. Then for $x < i < N$ the channels $G[i]$ must read from a channel that is not present in the protocol and therefore typechecking fails.

We write this proof step as

```
in the presence of X1, ..., Xn
rewrite family F[bound N]
to R2
by induction on i
bound x ( PROOF )
```

where $X1, \dots, Xn$ are the families and channels that we select from the current protocol as members of $P1$.

We also have a shorthand notation for rewriting the entire protocol, not just a selection from it, by omitting the **in the presence of** part, and then $P1 = P$.

Similar rules are defined to cover the cases when the family F is assigned a protocol or its definition contains branching.

We can also do induction on families with more than one index. We can select any position for induction, as long as its corresponding bound is not fixed or dependent.

In the example below, assume the following definitions:

```
fam ShareSum[bound N + 2 bound K] ...
||
family ShareSum-OK[bound (N + 1) bound K] indices: m, k bounds: bound (N + 1) bound K ::=
  (when m = 0 --> s : unit <- read ShareOK[0, k] ; return () ) ;;
  (when not m = 0 -->
    ss : unit <- read ShareSum-OK[m - 1, k] ;
    s : unit <- read ShareOK[m, k] ;
    return () )
||
family SharesOK[bound K] indices: k bounds: bound K ::= (
  family ShareOK[bound (N + 2) bound K]...
  ||
  ...
)
```

and assume we want to obtain

```
fam ShareSum[bound N + 2 bound K] ...
||
family ShareSum-OK[bound (N + 1) bound K] indices: m, k bounds: bound (N + 1) bound K ::=
  s : bool <- read ShareSum[m, k] ; return ()
||
family SharesOK[bound K] indices: k bounds: bound K ::= (
  family ShareOK[bound (N + 2) bound K]...
  ||
  ...
)
```

Example 34: Inductive proof

We write

```

in the presence of fam ShareSum[bound N + 2 bound K], fam SharesOK[bound K]
rewrite family ShareSum-OK[bound (N + 1) bound K]
to s : bool <- read ShareSum[m, k] ; return ()
by induction on m bound M
(
  case distinction on
  family ShareSum-OK[fixedBound M bound K]
  (
    case: (
      using group SharesOK[bound K]
      (subst fam ShareOK[bound (N + 2) bound K]
        into fam ShareSum-OK[fixedBound M bound K] then
        call symShareSumOKO
      )
    )
    case: ( using group SharesOK[bound K]
      (subst fam ShareOK[bound (N + 2) bound K]
        into fam ShareSum-OK[fixedBound M bound K] then
        subst fam ShareSum-OK[bound M bound K]
        into fam ShareSum-OK[fixedBound M bound K] then
        call symShareSumOKM
      )
    )
  ) then
  merge cases for fam ShareSum-OK[fixedBound M bound K]
)

```

The proof illustrates a typical pattern: the inductive proof is done by case distinction and we obtain the same result on each branch so we can remove branching. Note that on the second branch we make use of the induction hypothesis when we substitute `fam ShareSum-OK[bound M bound K]` into `fam ShareSum-OK[fixedBound M bound K]`.

Syntactic transformations

Under this heading we group a number of strategies that change only the shape of a protocol. The rules that apply are derivable from the core rules.

- **turn neg into otherwise for** `X` if `X` has two branches and the condition of the second one is the negation of the condition of the first, we can write **otherwise** instead.
- **turn otherwise into neg for** `X` is the reverse transformation. Both operations are needed when we obtain on one side a negation e.g. via proving and on the other side the **otherwise** clause e.g. by definition and we must make the notation uniform.

Example 35: Turn Otherwise to Negation

```

family F[bound N] indices: i bounds: bound N ::=
  when i = 0 --> return True ;;
  otherwise --> return False

turn otherwise into neg for fam F[bound N]

family F[bound N] indices: i bounds: bound N ::=
  when i = 0 --> return True ;;
  when not (i = 0) --> return False

```

- **restructure** `P` is used to check that the current protocol has the same normal form as `P` and then rewrites to it, usually for proofs by symmetry or checking that a proof ended.

Example 36: Restructuring

Below we have the current protocol before restructuring, as obtained via proofs. Note that all internal channels are in front.

```
newNF(< Ctxt : bool > < LeakCtxtIdAdv : bool > ,
  (Ctxt ::= nf(m : bool <- read In, samp flip ()))
)||
(LeakCtxtAliceAdv ::= nf(c : bool <- read LeakCtxtIdAdv , return c )
)||
LeakCtxtIdAdv ::= nf(c : bool <- read Ctxt, return c ))

restructure idealPlusSim

newNF(< LeakCtxtIdAdv : bool > ,
  (LeakCtxtAliceAdv ::= nf(c : bool <- read LeakCtxtIdAdv , return c )
)||
newNF(< Ctxt : bool > ,
  (Ctxt ::= nf(m : bool <- read In, samp flip ()))
)||
LeakCtxtIdAdv ::= nf(c : bool <- read Ctxt, return c )))
```

After restructuring we obtain the protocol `idealPlusSim` as it was originally declared.

- **bring declarations of hidden channels in front** brings a protocol obtained via application of assumptions to its normal form. An example can be obtained by reading the previous example from bottom to the top.
- **fix index for X** if the bound corresponding to the index j of the family X is **fixedBound** N , replace j with N in the body of X .

Example 37: Fix an index

```
family RcvdBit[bound N + 1 fixedBound N + 1 bound K]
  indices: n, m, k bounds: bound N + 1 fixedBound N + 1 bound K ::=
    s : bool <- read SendBit[n, m, k]; return s

fix index for fam RcvdBit[bound N + 1 fixedBound N + 1 bound K ]

family RcvdBit[bound N + 1 fixedBound N + 1 bound K ]
  indices: n, m, k bounds: bound N + 1 fixedBound N + 1 bound K ::=
    s : bool <- read SendBit[n, N + 1, k]; return s
```

- **merge cases for X** if we have the same reaction or protocol on each branch, we can remove the branching. This situation is typical for a proof by case distinction.
- **merge case $i = nt$ with otherwise for X** if the branch with condition $i = nt$ has the same reaction as the otherwise branch, we can remove the $i = nt$ branch.
- **discard branch COND of F** if we can prove the negated **COND**, we can remove the corresponding branch and, if we get a unique branch as a result, branching can be omitted. This may happen when we do splitting of families and on a branch we test for equality on the index that we have splitted on.

Example 38: Removing superfluous branch

Here we first split a family on the first index, then we notice the condition of a branch in the definition of the family cannot be true.

```
family F[bound N + 2 bound N + 2 bound K]
  indices: n, m, k bounds: bound N + 2 bound N + 2 bound K ::=
    (when n = N + 2 --> ...)
  ;;
  ...

split fam F[bound N + 2 bound N + 2 bound K ] on first index

(family F[bound N + 1 bound N + 2 bound K ]
  indices: n, m, k bounds: bound N + 1 bound N + 2 bound K ::=
    (when n = N + 2 --> ...)
  ;;
  ...
) || (family F[fixedBound N + 1 bound N + 2 bound K] ... )

discard branch n = N + 2 of fam F[bound N + 1 bound N + 2 bound K ]

(family F[bound N + 1 bound N + 2 bound K ]
  indices: n, m, k bounds: bound N + 1 bound N + 2 bound K ::=
    ...
) || (family F[fixedBound N + 1 bound N + 2 bound K] ... )
```

We can now discard the branch and only the other cases are kept.

- `group X1, ... Xn in G indices: idxs bounds: bds` creates the group `G` containing `X1, ... Xn` from the current protocol.

Example 39: Creating a group

```
(family X1[bds] indices: idxs bounds: bds := R1) || ... (family Xk[bds] indices: idxs bounds: bds
  := Rk) || P

group X1, ..., Xk in G indices: idxs bounds: bds

P || family G[bds] indices: idxs bounds: bds := (
  (family X1[bds] indices: idxs bounds: bds := R1) || ... (family Xk[bds] indices: idxs bounds: bds
    := Rk) )
```

- `ungroup G` discards a group, keeping its members in the current protocol.
- `using group G (proof)` use the protocol assigned to a group without removing the group. An equivalent sequence would be to `ungroup`, do the proof, then `group` again.

Example 40: Using a group

```
P1 || (family G ... := P)

using group G... (proof)

P2 || (family G ... := P)
```

In this case `proof` rewrites `P || P1` to `P || P2`, so `P` must stay unchanged.

- `compose X with Y in group` introduces a new group, named `X and Y`, containing `X` and `Y`.

Example 41: Grouping two families

```

(family Dec[bound n] indices: i bounds: bound n ::=
  e : ctxt <- read Enc[i] ; k : key <- read Key ; return dec ((e, k))
) ||
(family Enc[bound n] indices: i bounds: bound n ::=
  in : msg <- read In[i] ; k : key <- read Key ; samp enc((in, k))
)

compose fam Dec[bound n] with fam Enc[bound n] in group

family Dec[bound n] and Enc[bound n] indices: i bounds: bound n ::=
(family Dec[bound n] indices: i bounds: bound n ::=
  e : ctxt <- read Enc[i] ; k : key <- read Key ; return dec ((e, k))
) ||
(family Enc[bound n] indices: i bounds: bound n ::=
  in : msg <- read In[i] ; k : key <- read Key ; samp enc((in, k))
)

```

- **decompose** X **with** Y is the reverse operation.
- **combine** G removes a group with branching by pushing the branching on the elements of the group.

Example 42: Combining a group

```

family G[bounds] ... =
  when bt1 -> (F1 ::= R11) || ... || (Fk ::= R1k)
  ;;
  ...
  ;;
  when btN -> (F1 ::= RN1) || ... || (Fk ::= RNk)

combine G

(F1 ::=
when bt1 --> R11
;; ... ;;
when btN --> RN1) || ... ||
(Fk ::=
when bt1 --> R1k
;; ... ;;
when btN --> RNk)

```

- **combine** X **into** families similar to the previous case, but turns the channels $C[x, y, z]$ occurring on the branches of G into families.
- **rename** x_1 **to** y_1 , ... x_n **to** y_n in X does an alpha-renaming.

Example 43: Renaming variables

```

C ::= d : bool <- read In ; return d

rename d to x in chn C

C ::= x : bool <- read In ; return x

```

- **rename** indices **to** y_1, y_2, y_3 in X does an alpha-renaming of the indices of X .

Example 44: Renaming indices

```

family F[bound N bound N] indices: i, j bounds: bound N bound N ::=
  in <- read In[i, j] ; return in

rename indices to x, y in fam F[bound N bound N]

family F[bound N bound N] indices: x, y bounds: bound N bound N ::=
  in <- read In[x, y] ; return in

```

- **rename variables** x_1 to y_1 , ... x_n to y_n in X does an alpha-renaming of the indices x_1 , ... x_n of X and also modifies the normal form of the current protocol – X must be an internal family.

Example 45: Renaming indices of an internal family

```

newNF( DECLS { F[bound N] i : bool},
P || family F[bound N] indices: i bounds: bound N ::= R )

rename variables i to k in fam F[bound N]

newNF( DECLS { F[bound N] k : bool},
P || family F[bound N] indices: k bounds: bound N ::= R[i / k] )

```

- **remove only branch of** X if we have just one branch, we can omit branching.
- **change name** X to Y renames a group, a family or a channel.
- **remove** X from group G remove a family/channel from every case of a group, keeping it in the protocol.

Example 46: Removing a protocol from the branches of a group, with branching

```

P ||
family G... ::=
  when bt1 --> (P1 || X ::= R1)
  ;; ... ;;
  when btk --> (Pk || X ::= Rk)

remove X from group G

P || X ::=
  when bt1 --> R1
  ;; ... ;;
  when btk --> Rk
||
family G... ::=
  when bt1 --> P1
  ;; ... ;;
  when btk --> Pk

```

- **remove and merge** X from group G remove a family/channel from every case of a group, checking that its value is the same, so no branching is introduced.

Example 47: Removing a protocol from the branches of a group, without branching

```

P ||
family G... ::=
  when bt1 --> (P1 || X ::= R)
  ;; ... ;;
  when btk --> (Pk || X ::= R)

remove and merge X from group G

P || X ::= R ||
family G... ::=
  when bt1 --> P1
  ;; ... ;;
  when btk --> Pk

```

- **move X on each branch of Y** adds the channel or family X on every branch of the group Y.

Example 48: Moving a protocol on the branches of a group

```

P || X ::= R ||
family G... ::=
  when bt1 --> P1
  ;; ... ;;
  when btk --> Pk

move X on each branch of G

P ||
family G... ::=
  when bt1 --> (P1 || X ::= R)
  ;; ... ;;
  when btk --> (Pk || X ::= R)

```

- **extract X1, ... Xn from group G1 into group G2** removes X1, ... Xn of G1 and adds them to a newly created group named G2.

Example 49: Extracting a subgroup

```

family G1... ::=
(family X1[bds] indices: idxs bounds: bds ::= R1) || ... (family Xk[bds] indices: idxs bounds: bds
  ::= Rk) || P

extract X1, ..., Xk from group G1 into group G2

(family G2... ::=
(family X1[bds] indices: idxs bounds: bds ::= R1) || ... (family Xk[bds] indices: idxs bounds: bds
  ::= Rk)
) ||
(family G1... ::= P)

```

- **merge case for chn X and chn Y into group G** creates a new group from two channels.

Example 50: Merging two channels with same cases

```

(cn1 ::=
  when bt1 --> R1
  ;; ... ;;
  when btk --> Rk
) ||
(cn2 ::=
  when bt1 --> Q1
  ;; ... ;;
  when btk --> Qk
)

merge case for chn cn1 and chn cn2 into group

family CGroup[bound 1] indices: i bounds: bound 1 ::=
  when bt1 --> cn1 ::= R1 || cn2 ::= Q1
  ;; ... ;;
  when btk --> cn1 ::= Rk || cn2 ::= Qk

```

Saving Current State

We can save the current configuration using `save current configuration`. It is stored in the dedicated slot of the prover state. We can then make use of its constituents using the keywords `savedProtocol` and `savedDelta`. The other elements of the current configuration will never be used, so we did not introduced syntax for accessing them.

3.9 Approximate Proofs

We have upgraded configurations from protocol configurations to approximate equality configuration which in addition also keep track of count and context. The prover state will now maintain a stack of approximate equality configurations instead of one of protocol configurations. Since we can extract a protocol configuration from an approximate equality configurations by omitting the count and the context, all our previous statements about the state remain valid.

3.9.1 Approximate Congruence

The congruence rules for normal forms of protocols, INPUT-UNUSED and for parallel composition are only called by the Maude strategies and are not visible to the user.

AXIOM

Approximate assumptions are written

```

approx-assumption NAME using VARS : CCTXT ; INS ; ASSUMS |=
  P1 = P2 .

```

where `NAME` is a new identifier giving the name of the assumption, `VARS` is a list of names of variables appearing in the protocols `P1`, `P2`, `CCTXT` is a channel context, `ASSUMS` is a list of boolean terms allowing to require further hypotheses to hold and `INS` lists the channels that we are allowed to read from. Note that `VARS` contains now both variable and channel names, as we want to be able to distinguish the possible interpretations of `C ::= read D`: `D` is the channel `C` required from a channel named `D` or is the channel `C` required to read from an arbitrary channel.

To apply an axiom, we write `use approx assumption NAME on X1, ..., Xn` or `use approx assumption NAME` if the application context is clear.

Example 51: Decisional Diffie-Hellman

We write the decisional Diffie-Hellman (DDH) cryptographic assumption: as long as the secret keys k, l are generated uniformly, even if the adversary knows the values g^k and g^l , they will be unable to distinguish $(g^k)^l$ from a uniformly generated element of G :

```

approx-assumption ddh using ka kb :
  (chn ddh :: msg * (msg * msg)) ; no inputs ; no assumptions |=
  ddh ::= ka : key <- samp unif_key() ;
         kb : key <- samp unif_key() ;
         return ( exp((g()), ka),
                  ( exp((g()), kb),
                    exp( (exp((g()), ka)), kb) )
                )
  =
  ddh ::= ka : key <- samp unif_key() ;
         kb : key <- samp unif_key() ;
         kr : key <- samp unif_key() ;
         return ( exp((g()), ka),
                  (exp((g()), kb),
                    exp((g()), kr))
                )
  .

```

We apply the assumption by writing `use approx assumption ddh on chn DDH3` where DDH3 is a channel assigned a reaction matching the left-hand side of the axiom.

EMBED

We write `approx embed P along phi with inputs I` (proof) where `phi` is the name of a declared embedding.

Example 52: Approximate embedding

The axiom HH2HH states that the protocols 10utOf40TReal-Honest-Honest and 10utOf40TIdeal-Honest-Honest are approximately equivalent. Then so are their embeddings along `phi`:

```

approx embed 10utOf40TReal-Honest-Honest along phi
  with inputs ( chn OTMsg-0[s, r, w] ,
               chn OTMsg-1[s, r, w] ,
               chn OTMsg-2[s, r, w] ,
               chn OTMsg-3[s, r, w] ,
               chn OTChc-0[s, r, w] ,
               chn OTChc-1[s, r, w] ) (
    use approx axiom HH2HH
  )

```

CASE DISTINCTION

We needed the approximate counterpart of the exact case distinction rule.

Example 53: Approximate embedding

Assuming the following definition:

```
10OutOf40TReal[n, m, k] ::=
  ( when ( isHonest(n) and isHonest(m) ) -->
    embed(10OutOf40TReal-Honest-Honest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isHonest(m) ) -->
    embed(10OutOf40TReal-SemiHonest-Honest, phi)
  ) ;;
  ( when ( isHonest(n) and isSemiHonest(m) ) -->
    embed(10OutOf40TReal-Honest-SemiHonest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isSemiHonest(m) ) -->
    embed(10OutOf40TReal-SemiHonest-SemiHonest, phi)
  )
```

we can do a proof by approximate case distinction where on the first case we repeat the proof by embedding from the previous example, and the other three cases are similar:

```
approx case distinction on chn 10OutOf40TReal[m, n, k] (
  case: (
    approx embed 10OutOf40TReal-Honest-Honest along phi
    with inputs ( chn OTMsg-0[s, r, w] , chn OTMsg-1[s, r, w] , chn OTMsg-2[s, r, w] ,
                  chn OTMsg-3[s, r, w] , chn OTChc-0[s, r, w] , chn OTChc-1[s, r, w] ) (
      use approx axiom HH2HH
    )
  )
  ...
)
```

The result is

```
10OutOf40TReal[n, m, k] ::=
  ( when ( isHonest(n) and isHonest(m) ) -->
    embed(10OutOf40TReal-Honest-Honest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isHonest(m) ) -->
    embed(10OutOf40TReal-SemiHonest-Honest, phi)
  ) ;;
  ( when ( isHonest(n) and isSemiHonest(m) ) -->
    embed(10OutOf40TReal-Honest-SemiHonest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isSemiHonest(m) ) -->
    embed(10OutOf40TReal-SemiHonest-SemiHonest, phi)
  )
```

3.9.2 Approximate Equality**EXACT**

We included exact proof steps as particular case of approximate ones and thus no special syntax is needed.

TRANS

We write `proof1 then-approx proof2`. The difference to `then` is that now the count and the context are updated according to the TRANS rule from the approximate equality calculus. An example is presented in Ex. 54.

SYM

We write `sym-approx from P (proof)`.

Example 54: Approximate symmetry proof

In the proof below we show a typical usage of `sym-approx`: since its corresponding subproof is approximate, we cannot use the `sym` DSL construct. In the subproof we first do a strict proof step then an approximate one, using the `AXIOM` approximate congruence rule, so we must use `then-approx`:

```
sym-approx from int3(
  restructure int3 then-approx
  use approx assumption cm
)
```

GENERALIZATION

We added a rule that allows us to do an approximate proof for a channel and then generalize it to a family of channels with the same definition.

We write `approx generalize for X (proof)`.

Example 55: Generalization

Assume we have a family instead of a channel in the Ex. 53. We can generalize the previous proof:

```
family 1OutOf40TReal[bound N + 2 bound N + 2 bound K]
  indices: n, m, k bounds: bound N + 2 bound N + 2 bound K
  ::=
  ( when ( isHonest(n) and isHonest(m) ) -->
    embed(1OutOf40TReal-Honest-Honest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isHonest(m) ) -->
    embed(1OutOf40TReal-SemiHonest-Honest, phi)
  ) ;;
  ( when ( isHonest(n) and isSemiHonest(m) ) -->
    embed(1OutOf40TReal-Honest-SemiHonest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isSemiHonest(m) ) -->
    embed(1OutOf40TReal-SemiHonest-SemiHonest, phi)
  )
approx generalize for fam 1OutOf40TReal[bound N + 2 bound N + 2 bound K] (
  approx case distinction on chn 1OutOf40TReal[m, n, k] (
    ...
  )
)
family 1OutOf40TReal[bound N + 2 bound N + 2 bound K]
  indices: n, m, k bounds: bound N + 2 bound N + 2 bound K
  ::=
  ( when ( isHonest(n) and isHonest(m) ) -->
    embed(1OutOf40TReal-Honest-Honest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isHonest(m) ) -->
    embed(1OutOf40TReal-SemiHonest-Honest, phi)
  ) ;;
  ( when ( isHonest(n) and isSemiHonest(m) ) -->
    embed(1OutOf40TReal-Honest-SemiHonest, phi)
  ) ;;
  ( when ( isSemiHonest(n) and isSemiHonest(m) ) -->
    embed(1OutOf40TReal-SemiHonest-SemiHonest, phi)
  )
```

The proof step in `the presence of X1, ... Xn rewrite family F to R by generalization of NAME` allows us to repeat an approximate proof that works on a channel for a family of channels.

Example 56: Generalization

Assume we have shown using the subproof `rewriteDDH2` that the channel `DDH2` rewrites as below:

```
(PublicKey ::= kb : key <- samp unif_key(); return exp((g()), kb)) ||
(DDH2 ::= pb : msg <- read PublicKey ;
  ka : key <- samp unif_key();
  return ( exp((g()), ka),
           exp((pb, ka)) ) )

call-approx rewriteDDH2

(DDH2 ::= kr : key <- samp unif_key();
  ka : key <- samp unif_key();
  return ( exp((g()), ka),
           exp((g()), kr)) ) || PublicKey ::= ... # unchanged
```

We can repeat the argument `q` times:

```
family DDH2[bound q] indices: i bounds: bound q ::=
  pb : msg <- read PublicKey ;
  ka : key <- samp unif_key();
  return ( exp((g()), ka),
           exp((pb, ka))
          ) || PublicKey ::= ... # unchanged

in the presence of chn PublicKey
rewrite family DDH2[bound q] to
  ka : key <- samp unif_key();
  kr : key <- samp unif_key();
  return(exp((g()), ka), exp((g()), kr))
by generalization of rewriteDDH2

family DDH2[bound q] indices: i bounds: bound q ::=
  kr : key <- samp unif_key();
  ka : key <- samp unif_key();
  return ( exp((g()), ka),
           exp((g()), kr)) || PublicKey ::= ... # unchanged
```

3.10 Working on Complex Proofs: Trying Steps

For proofs by symmetry, by case analysis and by induction, we introduced helper rules that allow us to work on the subproof(s) of the rule. The main idea is that the tool generates a new configuration corresponding to the subproof. This configuration is pushed in the configurations stack of the prover state. The user is able to inspect it and alter it via DSL proof steps until the expected result is reached. Once the subproof that we have been working on is finished, we can replace the proof attempt with a complete subproof.

SYM

In symmetry proofs we know the protocol that we are trying to obtain. If splitting of families of channels is involved, the channel context also changes.

We have language constructs for both situations:

```
try sym from P : proof

try sym from P over delta : proof .
```

Example 57: Starting a proof by symmetry

Assume the current protocol is P and we want to rewrite it to Q by symmetry.

```
P
try sym from Q : todo
Q
```

This pushes a protocol configuration containing Q in the stack of the prover state. We can then replace `todo` with proof steps and proceed with the subproof until we reach P , which concludes the proof by symmetry. The successful proof attempt

```
try sym from Q : step_1 then ... then step_n
```

can be discarded and replaced with

```
sym from Q (step_1 then ... then step_n)
```

CASE ANALYSIS

In proofs by case analysis we want to be able to work locally on each branch.

Example 58: Starting a proof by case distinction

Assume the current protocol is

```
P || family F[bound N] indices: i bounds: bound N ::= when bt1 --> R1 ;; when bt2 --> R2
```

and we want to rewrite it to

```
P || family F[bound N] indices: i bounds: bound N ::= when bt1 --> R3 ;; when bt2 --> R4
```

We start working on the branch corresponding to `bt1`

```
try on branch bt1 of fam F[bound N] : todo
```

This pushes in the stack of the prover state a protocol configuration containing

```
P || family F[bound N] indices: i bounds: bound N ::= R1
```

together with the hypothesis `bt1`. We can then replace `todo` with proof steps and proceed until we reach

```
P || family F[bound N] indices: i bounds: bound N ::= R3
```

which concludes the proof of this case. The successful proof attempt

```
try on branch bt1 of fam F[bound N] : step_1 then ... then step_n
```

can be replaced with

```
on branch bt1 of fam F[bound N] (step_1 then ... then step_n)
```

and proceed in a similar way for the other branch to finish the proof by case distinction.

INDUCTION

Induction proofs are the most complicated, because their subproofs require working on a configuration that is more complex to build. Assume we want to prove by induction that $Q = P1 \parallel P \parallel \text{family } F[\text{bound } N] \text{ indices: } i \text{ bounds: bound } N ::= R1$ rewrites to $Q = P1 \parallel P \parallel \text{family } F[\text{bound } N] \text{ indices: } i \text{ bounds: bound } N ::= R2$ where we already divided the rest of the protocol into a part that will stay outside the induction, $P1$ and a part that will be used during the induction subproof, P . If our current configuration was $\text{pConfig}(\text{Sigma}, \text{Delta}, Q, I, O, A)$, the configuration needed for the subproof is $\text{pConfig}(\text{Sigma}, \text{DeltaInd}, \text{PInd}, \text{IInd}, \text{OInd}, (A, x < n))$ where

- DeltaInd is obtained by removing the entry of $F[\text{bound } N]$ and adding entries for $F[\text{bound } x]$ and $F[x]$ with the same type as $F[\text{bound } N]$;
- $\text{PInd} = P \parallel \text{family } F[\text{bound } x] \text{ indices: } i \text{ bounds: bound } x ::= R2 \parallel F[x] ::= R1 [i / x]$;
- IInd is obtained by adding the channels and families of $P1$ to I ;
- OInd is the set of outputs of the protocol PInd , which can be computed;
- we record that the induction variable must be within bounds.

The syntax for the proof step for attempting induction proofs is

```
try in the presence of X1, ... Xn
  rewrite X to R by induction on i bound B : proof
```

We use the same syntax for the cases when the family F is assigned a protocol or it has a definition involving branching.

Example 59: Trying induction proofs

In the situation above, assume the families and channels $X1, \dots, Xn$ are all the elements of P and we start with

```
try in the presence of X1, ... Xn
  rewrite F[bound N] to R2 by induction on i bound x : todo
```

The current protocol is then PInd . We replace `todo` with proof steps until our current protocol becomes $P \parallel \text{family } F[\text{bound } x] \text{ indices: } i \text{ bounds: bound } x ::= R2 \parallel F[x] ::= R2 [i / x]$, so our proof by induction is successful. If the proof attempt was

```
try in the presence of X1, ... Xn
  rewrite family F[bound N] to R2 by induction on i bound x : step_1 then ... then step_k
```

we can replace it with

```
in the presence of X1, ... Xn
  rewrite F[bound N] to R2 by induction on i bound x ( step_1 then ... then step_k )
```

3.11 Reusing Proofs

We can save proofs for multiple uses. We write

```
strict subproof NAME : PROOF .
approx subproof NAME : PROOF .
```

We can then call a subproof either directly in the main proof or inside a DSL rule. The syntax is

```
call NAME
```

```
call-approx NAME
```

Example 60: Subproofs

This proof fragment shows strategy calls both at top level (`addInitOK`) and on the branches of a case distinction proof (`symSharesInput`, `symSharesNot`, `symSharesXor`, `symSharesAnd`, omitted here):

```
strict subproof addInitOK =
group
  fam InShareOK[bound N + 2 bound N + 2 dependentBound I],
  fam InShare$-OK[bound N + 2 bound N + 2 dependentBound I],
  fam InShare$Sum-OK[bound N + 1 bound N + 2 dependentBound I]
in InitOK
indices: m, n, i
bounds: bound N + 2 bound N + 2 dependentBound I .

case distinction on group Shares[fixedBound B] (
  case: (
    subst fam InShareOK[bound (N + 2) bound (N + 2) dependentBound I]
    into chn Share[N + 1, B] then
    call symSharesInput
  )
  case: (
    subst fam ShareOK[fixedBound (N + 1) bound K]
    into chn Share[N + 1, B] then
    call symSharesNot
  )
  case: (
    subst fam ShareOK[fixedBound (N + 1) bound K]
    into chn Share[N + 1, B] then
    rename w to w0 in chn Share[N + 1, B] then
    subst fam ShareOK[fixedBound (N + 1) bound K]
    into chn Share[N + 1, B] then
    rename w to w1 in chn Share[N + 1, B] then
    call symSharesXor
  )
  case: (
    subst fam CtrbSum-OK[bound (N + 2) bound (N + 2) bound K]
    into chn Share[N + 1, B] then
    call symSharesAnd
  )
) then
call addInitOK
```

Subproofs are always executed by performing their steps. This has the drawback that we cannot record the fact that we have already rewritten a protocol into the other and if we must do that again, we repeat the proof. We have introduced lemmas for this.

We have the following syntax for lemma declarations:

```
lemma NAME = P1 => P2 : PROOF .
```

where P_1 , P_2 are protocols (or protocol names) and `PROOF` is an IPDL proof that P_1 rewrites to P_2 .

When encountering a lemma declaration, the tool introduces the following in the Maude module stored in the environment:

- a Maude strategy for applying the proof, in the same way as for IPDL subproofs;
- a Maude rewrite rule that directly rewrites P_1 to P_2
- a strategy for applying this rewrite rule under the IPDL congruence rules for new declarations and parallel composition.

This allows us to handle two execution modes for lemmas, one that runs its proof and the other that check that the proof has already been executed and does not repeat the proof if this is the case. We have implemented this by further extending the Maude module stored in the environment with:

- a strategy with the same name as the lemma, performing no action (i.e., it is defined as `idle`), to be used as a flag and which is added to the module as soon as we called the strategy that applies the proof of the lemma;
- a top-level strategy for executing the lemma, that checks whether the strategy introduced at the previous step is present in the module. If this is the case, we can call the strategy that directly rewrites P_1 to P_2 . If this is not the case, we

call the strategy that applies the proof of the lemma, and this also adds the flag strategy, for further applications of the lemma.

Example 61: Lemmas

The following simple example shows that if we take a protocol and do a substitution of a channel into another one, we get the same result as doing the same operation except this time we have executed the lemma already and we don't need to run it again, so we can opt to trust it and do the rewrite directly.

```
protocol P1 =
  (o1 ::= return True)
  ||
  (o2 ::= i : bool <- read in ; return i)
  ||
  (o3 ::= o2 : bool <- read o2 ; return o2)
.

protocol P2 =
  (o1 ::= return True)
  ||
  (o2 ::= i : bool <- read in ; return i)
  ||
  (o3 ::= i : bool <- read in ; return i)
.

lemma L = P1 => P2 :
  subst chn o2 into chn o3
.

start with P1 over delta

use lemma L run then
sym from P1 (
  use lemma L trust
)

check-proof P1
```

A subproof is equivalent to a lemma executed in run mode. We can use a subproof when we do not want to explicitly list the protocols that the subproof rewrites. The drawback of using a subproof over a lemma is that the former cannot be executed in trust mode.

3.12 Evaluating the Results: IPDL Commands

IPDL commands allow us to inspect the results of declarations and/or proof steps.

show-types Shows all declared types.

show-ops Shows all declared function symbols with their typing.

show-dists Shows all declared distribution symbols with their typing.

show-channels Shows all channels and families declarations from the current channel context, with their typing.

show-parameters Shows all declared parameters, with their typing.

show-hypotheses Shows all declared hypotheses.

show-protocols Shows the names of all declared protocols.

show-all Shows all declared types, function and distribution symbols and protocols.

current-protocol Show the protocol from the current state. This allows us to see the result of applying proof steps.

show-protocol NAME Shows the protocol named **NAME**.

size NAME Shows the size of the protocol named **NAME**.

check-proof NAME Checks whether the protocol from the current state is equal with the protocol named **NAME**. If it is, the result is **Done**, otherwise it is **Ongoing**. The command is used to check whether a proof is finished with the expected result.

show-bounds Shows the count and context computed during the proof.

Example 62: Showing the bounds

Below we see how bounds would typically look like: the name of the axiom, the count, which can be either a natural number or an expression on natural numbers and the context which can depend on the sizes of the involved types. In this example we work on a family of channels with three indices and bounded by the parameters **N** and **K**. The tool shows the resulting polynomial in descending order according to the degree of the terms.

```
indistinguishability assumption HH2HH :
count: |{(n, m, k) s.t. n < N + 2, m < N + 2, k < K,
  when isHonest(n) and isHonest(m)}|
context:
  N * K * max(N * 434 + 990, N * 434 + 1011,
    N * 607 + 1329, N * 434 + 983) +
  N * N * K *
    max(|10utOf40TReal-Honest-Honest|, 543) +
  N * N * K * 218 +
  N * K *
    max(|10utOf40TReal-Honest-Honest|, 543) * 4 +
  N * K * 932 +
  K * max(N * 434 + 990, N * 434 + 1011,
    N * 607 + 1329, N * 434 + 983) * 2 +
  N * N * maxValue(I, N + 2) * 300 +
  N * N * 68 + K * 992 + N * 352 +
  K * max(|10utOf40TReal-Honest-Honest|, 543) * 3 +
  (K - 1) * max(|10utOf40TReal-Honest-Honest|, 543) +
  N * maxValue(I, N + 2) * 1169 +
  maxValue(I, N + 2) * 1138 + 509
```

disable-typechecking To speed up proofs, disables typechecking in the analysis of protocol declarations. This is the default mode.

enable-typechecking Enables typechecking in the analysis of protocol declarations. The recommended workflow is to start with typechecking enabled to ensure that the protocols are correct, and then disable typechecking until the next protocol declaration. When all protocols have been typechecked once, we can work with typechecking disabled.

3.13 Modularity

While IPDL does not have a module system, we can include files containing declarations and/or proof fragments, using **include FILENAME**. The convention is to use the extension **.ipdl** for a main file, **.imod** for a declarations file and **.proof** for a proof file.

We can use **include** after any input that has been completely processed. For example, this means that we cannot write

```
sym from P over delta (include F)
```

In such a case, we can do the entire symmetry proof inside **F** and replace the line above with **include F**. Similar restrictions apply to all other IPDL proof steps with subproofs.

Example 63: Modularity

The modular version of the GMW-N case study is below. Typically modularity increases performance.

```
include gmwn-defs.imod
include realGMWN.imod
include restr.imod
include pInt1.imod
include idealPlusSimGMWN.imod

include gmwn-subs.imod

start with real over deltaReal

include 0gmwn.proof

include 1gmwn.proof

include 2gmwn.proof

include 3gmwn.proof

include 4gmwn.proof

include 5gmwn.proof

check-proof idealPlusSim
show-bounds
```

3.14 Revisiting the Hello World Example

In Sec. 2.5 we have introduced a simple example of a proof using IPDL rules. We argued that the use of structural rules like congruence and exchange clutter the proof, making it difficult to write and hard to read. We now present the same proof in the IPDL DSL.

We have to define the signature. Our example works with booleans, so we will not introduce new datatypes. We must introduce a function symbol for $\oplus$ and a distribution symbol for flip:

```
function xor : bool * bool -> bool .

distribution flip : unit -> bool .
```

We then write the channel context:

```
channel context delta =
  input In : bool ;
  input X : bool ;
  output LeakCtxtAliceAdv : bool
.
```

and the assumption as a reaction axiom:

```
reaction-assumption flipInvariantXorFixed using x y :
  emptyCCtxt ; x : bool ; nil |=
    y : bool <- samp flip (()) ; return xor( (x, y) )
  =
  samp flip (()) .
```

We then write the protocol **real**:

```

protocol real =
  new Ctxt : bool in
  new Key : bool in
  ((Key ::= samp flip (()) )
  ||
  (Ctxt ::= m : bool <- read In ;
   k : bool <- read Key ;
   return xor(m, k)) )
  ||
  (LeakCtxtAliceAdv ::= c : bool <- read Ctxt ; return c) ) .

```

and the protocol `idealPlusSim`:

```

protocol idealPlusSim =
  new LeakCtxtIdAdv : bool in
  (
    (LeakCtxtAliceAdv ::= c : bool <- read LeakCtxtIdAdv ; return c)
  ||
  (new Ctxt : bool
   in
    ((Ctxt ::= m : bool <- read In ; samp flip (()) )
    ||
    (LeakCtxtIdAdv ::= c : bool <- read Ctxt ; return c ) )
   )
  )
  .

```

The tool transforms the protocols to their normal forms.

We come to doing the proof. We will show it in parallel to the old rules-only proof, highlighting equivalent steps in the two proofs with the same color:

<pre> start with real over delta fold chn Key into chn Ctxt fold chn Ctxt into chn LeakCtxtAliceAdv use assumption flipInvariantXorFixed on chn LeakCtxtAliceAdv sym from idealPlusSim (fold chn LeakCtxtIdAdv into chn LeakCtxtAliceAdv then fold chn Ctxt into chn LeakCtxtAliceAdv) restructure idealPlusSim check-proof idealPlusSim </pre>	<pre> NEW-EXCH ; CONG-NEW(SYM[P2 = P](COMP-NEW) ; CONG-COMP[Q = LeakCtxtAliceAdv ::= read Ctxt] (FOLD-BIND)); FOLD-BIND ; CONG-REACT(CONG-BIND(idle, AXIOM)); SYM[P2 = ideal + sim] (CONG-NEW(CONG-COMP[Q = LeakCtxtAliceAdv ::= read LeakCtxtIdAdv] (FOLD-BIND)) ; FOLD-BIND) </pre>
---	--

Here we start from the real protocol and apply the FOLD-BIND rule twice then use the assumption. We end up with the same protocol that we get from folding twice in the `idealPlusSim` protocol, so we can combine the two proofs using SYM. We then restructure which transforms the current protocol to `idealPlusSim`, as they have the same normal form. This step is specific to the DSL, as the rules-only proof does not use normal forms. Finally, we can check that the current protocol is `idealPlusSim` and the proof ends.

Even on a proof of this short size, it becomes obvious that the DSL provides a more compact notation by allowing us to omit the application of exchange and congruence rules.

3.15 Case Study: Authenticated-To-Secure Channel

We now revisit the running example of [3]. Alice wants to securely communicate n messages to Bob using an authenticated channel which leaks all messages to the adversary. To do so, we will assume they share a pre-shared key, which enables them to encrypt and decrypt all messages. We show how to encode this protocol and its proof in our DSL. First, we declare the number of sessions as a parameter n to our case study, with the intended interpretation that n is a function of the security parameter λ :

```
parameter n : nat .
```

The Assumptions

We use a version of the IND-CPA assumption, which states that encoding n context-chosen messages with the same secret key is indistinguishable from encrypting zeros. For simplicity, we assume a type of messages with constant length, so that the constant `zeros` need not depend on the length. We express the assumption as an *axiom* about the equality between two protocols:

```
approx-assumption cpa using m k :
(fam Enc[bound n] :: ctxt) (fam In[bound n] :: msg);
fam In[bound n] ; no assumptions |=
new Key : key in
(
  (Key ::= samp gen_key (()))
  ||
  (family Enc[bound n] indices: i bounds: bound n ::=
    m : msg <- read In[i] ;
    k : key <- read Key ;
    samp enc((m, k))
  )
)
=
new Key : key in
(
  (Key ::= samp gen_key (()))
  ||
  (family Enc[bound n] indices: i bounds: bound n ::=
    m : msg <- read In[i] ;
    k : key <- read Key ;
    samp enc(( zeros(), k))
  )
)
.
```

The CPA axiom is parameterized by two *families* of channels: `In[bound n]`, for n input channels carrying messages, and `Enc[bound n]`, for n output channels carrying ciphertexts. The left side of the CPA assumption samples a key on channel `Key` and, for each i , encrypts `In[i]` under the key. We do this by declaring a *family* of channels — one for each i less than the parameter n — which reads from `In[i]`, reads from `Key`, and samples from the distribution of probabilistic encryptions under that key and message. We declare the `Key` channel as *internal* using `new` so that the outside context cannot read it. The right side of the CPA assumption is similar, but encrypts `zeros` (defined to be a constant) rather than the message. Importantly, the family `Enc[bound n]` on the right hand side still reads from `In[i]`, since the two protocols must have the same (logical) *timing* behaviors between the channels.

In addition to the CPA assumption, we also have the assumption about the encryption scheme's *correctness*: encrypting and decrypting must return the same message. We encode this in our DSL as an assumption similar to CPA, but since we assume that the encryption scheme is *perfectly* correct, with zero probability of error, we use the declaration `protocol-assumption` rather than `approx-assumption`:

```
protocol-assumption enc-dec-correctness
using m k c keyN encN decN inN :
(chN keyN :: key) (chN encN :: ctxt)
(chN decN :: msg) (chN inN :: msg) ;
chN inN ; no assumptions
|=
(keyN ::= samp gen_key (()) )
||
(encN ::= m : msg <- read inN ;
  k : key <- read keyN ;
  samp enc((m, k))
)
)
```

```

||
(decN ::= c : ctxt <- read encN ;
  k : key <- read keyN ;
  return dec((c, k))
)
=
(keyN ::= samp gen_key (()) )
||
(encN ::= m : msg <- read inN ;
  k : key <- read keyN ;
  samp enc((m, k))
)
||
(decN ::= in : msg <- read inN ;
  return in
)
.

```

The Protocol

The Authenticated-To-Secure Channel protocol **Real** now takes the following form:

```

protocol real =
  new Key : key in
  newfamily Recv[bound n]
    indices: i bounds: bound n : ctxt in
  newfamily Send[bound n]
    indices: i bounds: bound n : ctxt in (
  (Keygen || Alice || Channel || Bob)
  where Keygen =
    ( Key ::= samp gen_key (()) )
  and Alice =
    (family Send[bound n]
      indices: i bounds: bound n ::=
        in : msg <- read In[i] ;
        k : key <- read Key ;
        samp enc ( (in, k) )
    )
  and Channel =
    (family LeakCtxtNetAdv[bound n]
      indices: i bounds: bound n ::=
        send : ctxt <- read Send[i] ; return send
    )
  ||
  (family Recv[bound n]
    indices: i bounds: bound n ::=
      okCtxt : unit <- read OkCtxtAdvNet[i] ;
      send : ctxt <- read Send[i] ;
      return send
  )
  and Bob =
    (family Out[bound n]
      indices: i bounds: bound n ::=
        c : ctxt <- read Recv[i] ;
        k : key <- read Key ;
        return dec((c, k))
    )
  )
.

```

The body of the protocol is a parallel composition of the key generating functionality **Keygen**, the two parties **Alice** and **Bob**, and the authenticated channel functionality **Channel**. Alice encrypts each input with the shared key stored on the internal channel **Key**, samples a ciphertext from the resulting distribution, and sends the result to the authenticated channel functionality on the channel **Send[i]**. Bob reads the ciphertext forwarded to him from the authenticated channel functionality on the channel **Recv[i]**, decrypts it with the shared key, and outputs the plaintext on the channel **Out[i]**. The channels **Send[i]** and **Recv[i]** are connected by the **Channel** functionality, which allows the adversary to read/schedule messages via **Leak[i]** and **Ok[i]**.

Proving Protocols Secure Our protocol is named **Real** because we will compare it to an *ideal* version, where Alice and Bob communicate directly through a secure channel, without the need for encryption. As in UC, we do this by proving that the **Real** protocol is indistinguishable from the **Ideal** protocol composed with a *simulator* **Sim** that can emulate the **Leak**[i] messages without knowledge of the secret messages followed by making the communication between these two protocols internal:

$$\text{Real} = \text{Ideal} + \text{Sim}.$$

The key advantage of IPDL [3] is that it enables proofs through *equational reasoning* principles. Using basic identities of protocols (*e.g.*, inlining definitions of channels into other channels) and assumptions (*e.g.*, CPA), one proves a protocol secure by progressively rewriting the real protocol into its idealization (plus the simulator).

We illustrate some key steps of the proof. The ideal functionality for our secure channel example has two output channels per session *i*: the adversarial output channel **LeakMsgRcvdIdAdv**[i] reads the message from **In**[i] and lets the adversary know that a message has been received – by passing a term of the **unit** type – but divulges nothing about the value of the message. The channel **Out**[i] first waits to receive a confirmation on the adversarial input channel **OkMsgAdvId**[i] that gives the green light to the functionality to process the message. It subsequently reads the message from **In**[i] on behalf of Alice, and outputs it on behalf of Bob:

```
Ideal =
(
  (family LeakMsgRcvdIdAdv[bound n]
    indices: i bounds: bound n ::=
      in : msg <- read In[i] ; return ()
  )
  ||
  (family Out[bound n]
    indices: i bounds: bound n ::=
      okMsg : unit <- read OkMsgAdvId[i] ;
      in : msg <- read In[i] ;
      return in
  )
)
```

The simulator turns the adversarial interface of the real protocol into the adversarial interface of the ideal functionality, thereby converting any adversary for the real protocol into an adversary for the functionality. In our example, the channels **LeakMsgRcvdIdAdv**[i] and **OkCtxtAdvNet**[i] are the inputs to the simulator, while the channels **LeakCtxtNetAdv**[i] and **OkMsgAdvId**[i] are the outputs.

Hence, upon receiving the information from the ideal functionality that a message has been received, the simulator must conjure up a ciphertext to leak to the adversary. This is accomplished by randomly generating a secret key and encrypting the chosen message **zeros** in each session. Upon receiving the approval from the adversary for the generated ciphertext, the simulator gives the approval to the ideal functionality to output the message:

```
Sim = ( new Key : key in
  (
    ( Key ::= samp gen_key (()) )
    ||
    (family LeakCtxtNetAdv[bound n]
      indices: i bounds: bound n ::=
        x : unit <- read LeakMsgRcvdIdAdv[i] ;
        k : key <- read Key ;
        samp enc ( (zeros ()), k )
    )
    ||
    (family OkMsgAdvId[bound n]
      indices: i bounds: bound n ::=
        okCtxt : unit <- read OkCtxtAdvNet[i] ;
        return okCtxt
    )
  )
)
```

The protocol **idealPlusSim** is written as

```
protocol idealPlusSim =
newfamily LeakMsgRcvdIdAdv[bound n]
  indices: i bounds: bound n : unit in
newfamily OkMsgAdvId[bound n]
  indices: i bounds: bound n : unit in
ideal || sim
where ideal = ... and sim = ...
```

As the channel `OkMsgAdvId[i]` carries a deterministic computation, we want to inline the computation into `Out[i]` to yield the following:

```
family Out[bound n]
  indices: i bounds: bound n ::=
  okCtxt : unit <- read OkCtxtAdvNet[i];
  m : msg <- read In[i];
  return m
```

This form of substitution is justified by the rule SUBST from Figure 2.9, which says that if the computation R_1 assigned to a channel o_1 is deterministic, then we may replace every occurrence of `read  $o_1$` by R_1 . However, for the rule SUBST to apply, we must first massage the protocol into a form where `OkMsgAdvId[i]` appears immediately next to `Out[i]`. In the implementation of [3], this required tedious manual transformations that, *e.g.*, permute channels inside a parallel composition or move channels in and out of scope of a new channel declaration (lines 87–104 of `MultiChan.v`¹). In our tool, all the necessary massaging is performed automatically, and we can simply write:

```
subst fam OkMsgAdvId[bound n]
  into fam Out[bound n] then
subst fam LeakMsgRcvdIdAdv[bound n]
  into fam LeakCtxtNetAdv[bound n] then
absorb fam LeakMsgRcvdIdAdv[bound n] then
absorb fam OkMsgAdvId[bound n]
```

A crucial step in simplifying the real protocol is to conceptually separate the encryption and decryption actions from the message-passing by introducing new internal channels `Enc[i]` and `Dec[i]` along with their definitions:

```
add internal family Enc
  indices: i bounds: bound n typed: ctxt
  assigned:
    in : msg <- read In[i] ;
    k : key <- read Key ;
    samp enc ( (in, k) )
```

We can now modify the channels `Send[i]` and `Out[i]` to read from `Enc[i]` and `Dec[i]` directly:

```
sym from change fam Send[bound n] with
  e : ctxt <- read Enc[i] ; return e
  in currentProtocol(
    subst fam Enc[bound n] into fam Send[bound n]
  ) then
sym from change fam Out[bound n] with
  okCtxt : unit <- read OkCtxtAdvNet[i] ;
  d : msg <- read Dec[i] ;
  return d
  in currentProtocol(
    subst fam Dec[bound n] into fam Out[bound n]
  )
```

We can now invoke the correctness assumption in each individual session to cancel the effect of encryption followed by decryption. In [3], the generalization to n sessions was stated as a separate lemma with a nontrivial manual proof (lines 67 – 103 in `CPA.v`²). In our code, the corresponding proof looks like this:

```
in the presence of chn Key
rewrite group Dec[bound n] and Enc[bound n]
to
  (family Dec[bound n]
    indices: i bounds: bound n
    ::= in : msg <- read In[i] ; return in
  )
||
  (family Enc[bound n]
    indices: i bounds: bound n
    ::= in : msg <- read In[i] ;
      k : key <- read Key ;
      samp enc ( (in, k) )
  )
by induction on i bound x (
  use assumption enc-dec-correctness
  on chn Key, chn Dec[x], chn Enc[x]
```

¹<https://github.com/ipdl/ipdl/blob/main/theories/protocols/Chan/MultiChan.v>

²<https://github.com/ipdl/ipdl/blob/main/theories/protocols/Chan/CPA.v>

)

The above code snippet proves by induction on $i < n$ that if the channels `Dec(i)` with $i < x$ rewrite from the original formulation that performs the decryption to the new formulation that simply reads off the message `In[i]`, so does the channel `Dec[x]`.

On the other hand, our CPA assumption is applied just once across all sessions upon encountering the following protocol snippet:

```
new Key : key in
  ((Key ::= samp gen_key()) ||
   (family Enc[bound n]
    indices: i bounds: bound n ::=
      m : msg <- read In[i];
      k : key <- read Key;
      samp enc((m, k))))
```

We invoke the approximate CPA assumption as shown below:

```
use approx assumption cpa
```

This yields the following protocol snippet:

```
new Key : key in
  ((Key ::= samp gen_key()) ||
   (family Enc[bound n]
    indices: i bounds: bound n ::=
      m : msg <- read In[i];
      k : key <- read Key;
      samp enc((zeros()), k))))
```

A final step in the proof *folds* the internal channels `Enc[i]` and `Dec[i]` that we introduced earlier into the rest of the protocol:

```
fold fam Enc[bound n] into fam LeakCtxtNetAdv[bound n] then
fold fam Dec[bound n] into fam Out[bound n]
```

This is justified by the rule FOLD-BIND which states that if we only read from channel c once, we can soundly replace `read c` by the computation assigned to c , even if this computation is probabilistic.

Concrete Security Bounds Such a proof could be carried out in the IPDL logic alone. However, that proof would only guarantee asymptotic security with respect to the custom symbolic bounds defined in [3]. We also aim for *concrete* security with respect to a low-level Turing Machine semantics: a precise probabilistic bound for the difference in the probability that an adversary can distinguish `Real` from `Ideal + Sim`.

After encoding the proof in the DSL, our tool computes the following bounds:

```
indistinguishability assumption cpa :
count: 1
context: n * | msg | * 6 + n * | ctxt | * 3 + n * 96 + 12
```

Here `count` denotes the number of times the CPA assumption was applied, and `context` bounds the maximal size of the program context in which it was applied. The expressions `|msg|` and `|ctxt|` denote the concrete length of bitstrings needed to represent the two types.

Chapter 4

Conclusions

We have introduced the IPDL prover, an implementation of the IPDL process calculus for formal verification of distributed message-passing protocols. We have discussed some design decisions in the implementation and we have introduced a DSL for proofs in the IPDL prover, tailored towards performance and improved proof readability. We have compared a simple proof in the IPDL process calculus with its DSL counterpart and we have also presented a larger case study of an approximate equivalence proof.

Directions of future work include several syntactic enhancements. Some are a matter of further simplifying the notation, e.g. we allow to write `f(x, y)` instead of `f((x, y))` or `samp flip` instead of `samp flip()`. When possible (i.e. there is no splitting involved), we may also write e.g. `subst F into G` instead of `subst fam F[bound N] into fam G[bound N]`. As we already mentioned, we have families that are assigned protocols, possibly with branching, and it may be useful to introduce a distinguished notation for them by writing e.g. `group G[i < n] ::= P` instead of `family G[i < n] ::= P`. Finally, we plan to implement variables using de Bruijn indices such that alpha renaming is no longer needed in the language.

A more complex enterprise would be to complement the IPDL prover with an AI agent that assists the user in writing equivalence proofs. We expect that some of the repetitive parts of a proof can be automatically provided by such an AI agent.

The logical next step after reading the user manual is to write more equivalence proofs using the tool. We have a number of case studies available at https://github.com/kristinas/IPDL-Maude/blob/main/IPDL-AS/doc/case_studies.pdf. It would be an interesting exercise for a user to choose one of them and to attempt to write their equivalent in the DSL, comparing their results with the DSL formalization already available at the IPDL home page.

Bibliography

- [1] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067, 2000. <https://ia.cr/2000/067>.
- [2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
- [3] Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. A Core Calculus for Equational Proofs of Cryptographic Protocols. *Proc. ACM Program. Lang.*, 7(POPL):866–892, 2023.
- [4] Ueli Maurer. Constructive Cryptography – A New Paradigm for Security Definitions and Proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications*, pages 33–56, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [5] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.
- [6] Kristina Sojakova, Mihai Codescu, and Joshua Gancher. Concrete security bounds for simulation-based proofs of multi-party computation protocols, 2025.
- [7] Dmytro Tymokhanov and Omer Shlomovits. Alpha-rays: Key extraction attacks on threshold ECDSA implementations. Cryptology ePrint Archive, Paper 2021/1621, 2021.