University of Groningen

Bachelor Thesis

Message-Passing Concurrency: Links between Concurrent ML and the π-calculus

Luis David Reyes Vargas

supervised by

First supervisor: Dr. J.A Pérez Parra
Second supervisor: Prof. dr. G.R. Renardel de Lavalette

July 27, 2020

Abstract

Concurrent ML (CML) is a message-passing programming language sharing many similarities with the π-calculus. Its distinctive feature is the abstract *event* value, which allows programmers to define synchronization abstractions as first-class citizens. This abstraction brings the modular approach of functional programming into the realm of concurrency. It is a powerful tool to structure communication that could find many uses in the π-calculus.

In this paper, we encode the basic concurrency primitives available in CML into the π-calculus. We first codify the process of synchronization of *events* with session types. Then, using this framework, we translate the event-generating primitives. This equips the π-calculus with the mechanism to generate programs in CML style that we can verify.

Contents

1 Introduction

In today's era, communicating systems are a necessity in many of our society's activities. From bank transactions to online video games, the demand for their services is almost constant. The underlying mechanisms of these systems are expected to behave correctly. It would be unfortunate, for example, if a bank transaction gets lost due to an error in the software handling it.

Due to this demand, there is a prominent field of computer science dedicated to the study of *protocols*. These structural units dictate the kind of messages to be exchanged and in which sequence during some procedure using communication. Within this context, *session types* offer a framework to describe protocols formally. The session types theory's central concept is to *type* channels with an abstract description of its agreed use. This procedure is comparable to assigning data-types like *int* or *bool* to describe how a compiler should interpret data.

Session types were initially designed for the π-calculus, a mathematical model used to study processes whose interconnections can change as they interact [13, 15]. The language treats links as first-class citizens that can be passed around, much like any other object. Its model of computation is message-passing concurrency and produces channels and processes dynamically.

The first incentive of this project was the striking similarity between the programming language Concurrent ML [20] and the π-calculus. Concurrent ML (CML) is also a message-passing language, and its treatment of communication resources is the same as in the π-calculus. In effect, we can draw several intriguing parallels between the two languages. Driven by this similarity, we interest ourselves in studying CML's most notable feature.

In particular, this feature allows a very distinctive treatment of concurrency; with it, the programmer can take a compositional approach to concurrency inspired by functional programming. The fundamental instrument enabling this technique is CML's *event* value, which allows us to type and pass synchronization protocols as first-class values. Such events are the core of what is known as higher-order concurrency [19], which permits us to define complex inter-process communication protocols by composing simpler ones.

The connection between CML's approach to concurrency and the π-calculus has been studied by Chaudhuri [4], who produced a "distributed synchronization protocol" that models CML's selective communication in the π-calculus. Our main goal is to extend this work by studying the protocol with session types.

Our primary focus is obtaining a well-behaved reconstruction of the protocol devised by Chaudhuri with session types. The main result obtained in this paper is that it is possible to type it. Then, we leverage this reconstruction to explore the translation of CML's message-passing primitives into the π-calculus.

In Section 2, we present a preamble to the research: the π-calculus, session types, Concurrent ML's event mechanism, and Chaudhuri's protocol. Then, in Section 3, we produce a session type description of the protocol in question. Afterward, in Section 4, we combine our results to translate the CML primitives into the π-calculus. Finally, in Section 5, we present an overview of our findings, followed by two sections containing conclusions and ideas for future work.

$P ::=$		processes:
	$\mathbf{0}$	inaction
	$P \mid P$	parallel composition
	$!P$	replication
	$(\nu x x)P$	scope restriction
	if v then P else P	conditional
	$x(x).P$	input
	$\overline{x}{<}v{>}.P$	output
	$x \triangleleft l.P$	select
	$x \triangleright \{l_i : P_i\}_{i \in I}$	branch

$v ::=$		names:
	x	variable
	true | false	Boolean

Syntax of π-calculus

Figure 1: Syntax of π-calculus

2 Research Background

In this project, we use the π-calculus and type system introduced by Vasconcelos in “Fundamentals of Session Types” [25]. Any deviations will be mentioned explicitly.

2.1 The π-calculus

The π-calculus [13] is the language bridging the synchronization protocol of interest and session types. It is a process calculus, used to model the communication of concurrent processes. Its syntax appears in Figure 1. The base set of our language is comprised of *variables*, which we also refer to as *channel ends.*

Specifically, processes are constructed sequentially with the “.” operator. A terminated process reaches *inaction*, which is denoted by process **0**. The *parallel composition* rule models processes running in parallel and *input* and *output* model sending and receiving messages.

New communication channels (or *names*) are created using the *scope restriction* term. Vasconcelos uses the notion of a double restriction, which instantiates two channel ends. For example, the term $(\nu x^+ x^-)P$ creates two channel ends, x^+ and x^-, which are bound to the scope of P. These ends can then be used for internal communication in P, or be passed to other parallel components in the system to establish a link. Although the new ends can have any name, we stick to the convention of polarizing them with positive and negative superscripts, as we demonstrated. This nomenclature is used in [8] and makes it clear that these ends are part of the same channel.

Another relevant element of our repertoire is choice, provided by the terms *select* and *branch.* Branching allows a process to offer a finite set of alternatives, which clients can choose using selection. The lower case l is an indexed set of *labels.* If a selection is made on label l_j, the branching process handles the request by executing the process P_j.

Finally, our π-calculus language also has the conditional construction, which executes process P or Q in process (if v then P else Q) depending on the boolean value v.

The operational semantics for the π-calculus is found in "Fundamentals of Session Types" [26]. We specifically note scope extrusion, which allows freshly created channel ends to be extended from one process to another if composed in parallel (the contrary of scope intrusion, the implication in the opposite direction):

$$(\nu xy)P \mid Q \equiv (\nu xy)(P \mid Q)$$

where $\equiv$ is a symbol that identifies two agents that "intuitively represent the same thing" [15]. This relation is known as *structural congruence*. The other important relation is *reduction*, denoted by symbol $\longrightarrow$. The reduction semantics expose the computations that processes may perform. For example, $P \longrightarrow Q$ denotes that 'P can evolve to Q'.

Now that we can model processes, we can introduce our type system.

2.2 Session Types

Session types are a formalism used to structure the behavior of communicating processes with a type-based approach. It is protocol-centered; pre-established patterns of message exchange are formulated to dictate how processes should use channels during communication. Therefore, a process which is well-typed conforms to a protocol it is assigned.

For the development of the session types we are interested in, we adhere to the syntax of types in Figure 2 [26]. Within this context, a channel end a typed at lin!$(bool).end$ creates a session where a boolean value can be sent, and after that, no further communication is allowed. Furthermore, the *lin* qualifier denotes that it is linear, limiting it to appear only in one thread. We will be omitting this qualifier to lighten the syntax.

Then, we can naturally assume that the other channel end for a is typed as $?(bool).end$, which receives that boolean value and terminates communication. This concept, where both ends of a channel follow complementary protocols, is known as duality and is essential for communication safety. Indeed, duality ensures that the session types are being respected at both ends of a connection.

In this regard, the duality function is used to generate complements of session types. It is defined as

$$\overline{q?T.U} = q!T.U \qquad \overline{q!T.U} = q?T.U \qquad \overline{end} = end$$
$$\overline{\oplus\{l_i : T_i\}_{i\in I}} = \&\{l_i : T_i\}_{i\in I}$$

Due to duality, double restrictions are a requirement of our type system. This condition contrasts with the single restrictions used in Chaudhuri's work, where it is enough to describe a link with a single channel name. We adjust the protocol to this formality in Section 2.5 by replacing the defined channels by polarized channel ends. For example, a channel a defined by Chaudhuri becomes a^+, a^- in our version of the protocol.

$q ::=$	Qualifiers:
lin	linear
un	unrestricted
$p ::=$	Pretypes:
$?T.T$	receive
$!T.T$	send
$\oplus\{l_i : T_i\}_{i \in I}$	select
$\&\{l_i : T_i\}_{i \in I}$	branch
$T ::=$	Types:
$bool$	boolean
end	termination
ϕ	arbitrary boolean
$q\ p$	qualified pretype
$\Gamma ::=$	Contexts:
$\varnothing$	empty
$\Gamma, x : T$	assumption

Figure 2: Syntax of types

Example 1

Following our example of channel a, the session types $a^+ :!(bool).end$, $a^- :?(bool).end$ allow us to type a process such as

$$a^+\text{<true>}\ .\mathbf{0} \mid a^-(x).\mathbf{0}$$

because it respects the session protocols: a^+ is used exactly once to send a boolean value, and a^- is used exactly once to receive one boolean value.
Under this scheme, an example of an untypable process is

$$a^+(x).\mathbf{0} \mid a^-\text{<false>}.\mathbf{0}$$

because the protocol is not respected: the process attempts the interaction in reverse, using a^- for sending and a^+ for receiving, but this is prevented by the session types.

As opposed to linear channel ends, unrestricted ends are available to zero or more threads as a shared resource. These commonly serve well when modeling server-type behavior, which can expect communication from more than one client through a globally known channel. The un qualifier designates this.

Once we specify the protocols of a particular communication system, we can then verify that the processes involved behave in accordance to the protocols. This procedure is known as *process typing* and is the discipline that lets us formally trace the correct usage of channels in a concurrent environment.

In this regard, we type processes against a *typing context* Γ. This context gathers the type informa-

Context split, $\Gamma = \Gamma \circ \Gamma$

$$\varnothing = \varnothing \circ \varnothing \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathrm{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathrm{lin}\ p = (\Gamma_1, x : \mathrm{lin}\ p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathrm{lin}\ p = \Gamma_1 \circ (\Gamma_2, x : \mathrm{lin}\ p)}$$

Context Update, $\Gamma = \Gamma \circ \Gamma$

$$\frac{x : U \notin \Gamma}{\Gamma + x : T = \Gamma, x : T} \qquad \frac{\mathrm{un}(T)}{(\Gamma, x : T) + x : T = (\Gamma, x : T)}$$

Figure 3: Context operations

tion of variables in a process. We denote that a process P is *typable* (or *well-typed*) in a context Γ as $\Gamma \vdash P$. An assignment of the form $a : T$ is called a *judgement*, which assigns type T to variable a, and a context is an unordered collection of judgements. Thus, a process $a(\text{true}).\mathbf{0}$ is typable as $a : ?bool.end \vdash a(\text{true}).\mathbf{0}$.

Invariant 1: References to linear channel ends occur in exactly one thread [26].

A critical aspect of the theory of session types is maintaining linear channel ends in a single thread. To maintain linearity invariant, the context split operator in Figure 3 is essential. This rule implies that when we are type-checking processes that split into two separate sub-processes, we pass the unrestricted part of the context to both sub-processes. Conversely, the linear part is divided in two, and a unique part is given to each one of them. Hence, if a is linear, then the process
a<false> | a<true> is not typable, since a can only occur in one of these components by linearity.

On the other hand, *context update* is used for input and output, when new types are being bound to a context. It defines two rules when adding new variables into a context: if the variable is linear, then it should not exist in the context already; if the variable is unrestricted, then its type should remain unchanged.

With the notions of type duality and context operators, we are now in a position to utilize the language's typing rules. These rules are given in Figure 4 [26].

For a process to be typable with this type theory, we perform induction on the structure of context Γ. The base case is given by the typing rule [T-INACT]. This rule specifies that, upon inaction, a well-typed process will have only unrestricted session types in its context. A session type T is

- $\mathrm{un}(T)$ if and only if $T = bool$ or $T = \phi$ or $T = end$ or $T = \mathrm{un}\ p$
- $\mathrm{lin}(T)$ only if true

And given a qualifier q(lin or un):

- q(Γ) if and only if $(x : T) \in \Gamma$ implies $q(T)$

As an important note, we also extend our session types language with the derived *arbitrary boolean* type, denoted with symbol ϕ. In certain situations of this project, we need a mechanism to type channels that send empty messages to convey a simple signal. Instead of producing a new notion, *Typing rules for values,* $\Gamma \vdash v : T$

$$\frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathrm{true} : \mathit{bool}} \qquad \frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathrm{false} : \mathit{bool}} \qquad \frac{\mathrm{un}(\Gamma)}{\Gamma, x : T \vdash x : T}$$

[T-TRUE] [T-FALSE] [T-VAR]

Typing rules for processes, $\Gamma \vdash P$

$$\frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu x y) P}$$

[T-INACT] [T-PAR] [T-RES]

$$\frac{\Gamma_1 \vdash v : \mathit{bool} \qquad \Gamma_2 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \text{if } v \text{ then } P \text{ else } Q} \qquad \frac{\Gamma_1 \vdash x : q?T.U \qquad (\Gamma_2 + x : U), y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}$$

[T-IF] [T-IN]

$$\frac{\Gamma_1, x : q!T.U \qquad \Gamma_2 \vdash v : T \qquad \Gamma_3 + x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}{<}v{>}.P}$$

[T-OUT]

$$\frac{\Gamma_1 \vdash x : q\&\{l_i : T_i\}_{i \in I} \qquad \Gamma_2 + x : T_i \vdash P_i \qquad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i \in I}}$$

[T-BRANCH]

$$\frac{\Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I} \qquad \Gamma_2 + x : T_j \vdash P \qquad j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P}$$

[T-SEL]

Figure 4: Typing Rules

we add this type to our syntax to represent a boolean value that can be discarded from a context without consideration. For example, we have the well-formed process $p :!\phi.end \vdash p\text{<>}.\mathbf{0}$.

Furthermore, Vasconcelos' session types also include the $*$ qualifier to assign unrestricted and recursive behavior. For example, a channel end a typed as $a : *?(!bool)$ can receive an unbounded number of boolean-sending channels, and it can appear in more than one thread of execution.

The rationale behind typing processes is to gain from the properties offered by this theory in our communication systems. The first property is *communication safety*, which guarantees that only the expected information is being exchanged through the channels. The second property is *session privacy*, which guarantees that channels are only known to the communicating entities. The final valuable property is *session fidelity*, which ensures that channels follow their expected structure. The combined outcome is freeing communication from errors. With this scheme, we can study the processes involved in Chaudhuri's protocol to obtain these guarantees.

We will now present Concurrent ML's *event* value in the following section, which will lead us to understand Chaudhuri's work.

2.3 Concurrent ML Events and the Synchronization Primitives

The abstract value *event* sets up the foundation for Concurrent ML. It has the polymorphic type constructor

$$type : {}'a\ event$$

The concrete type τ *event* encapsulates the abstraction of a synchronization protocol, which, after synchronization, returns a value of type τ. To produce these events, we rely on the terms $recvEvt, sendEvt, wrap, choose$.

The *event* value works by making the whole communication process explicit to the programmer. In CML, the programmer must oversee the sending and receiving of messages, the operations to be done during synchronization (*Synchronous operations*), and finally, the act of carrying out synchronization itself. This technique allows us to reason about the structure of a protocol meticulously. The most famous result of this separation of concerns was to reconcile selective communication [9] and procedural abstraction of synchronization protocols, allowing their simultaneous use.

The class of functions that provide the underlying synchronous message passing capabilities of Concurrent ML are known as *base-event constructors*. The name is appropriate because they express the simplest possible communication protocols: those of sending and receiving. The terms *recvEvt* and *sendEvt* conform this category of functions. Their CML types are given by

- $val\ sendEvt :\ ({}'a\ chan * {}'a) -> unit\ event$
- $val\ recvEvt :\ {}'a\ chan -> {}'a\ event$

Where the type constructor $'a\ chan$ is for channels that can send values of type $'a$ and *unit* denotes the empty tuple.

On the other hand, we have event combinators, which enhance base-events and produce more complex protocols. These primitives are responsible for the compositional approach to concurrency available in this language. Particularly important instances are *choose* and *wrap* (although many others exist).

The *choose* combinator takes a list of events and returns an event that, upon synchronization, allows the non-deterministic choice of one of these events, aborting the others. It is typed as

- $val\ choose:\ 'a\ event\ list->'a\ event$

The *wrap* combinator takes in an event ev and a function f, and returns an event that, upon synchronization, applies the function f to the return value of synchronizing on ev. It is typed as

- $val\ wrap:\ ('a\ event * ('a->'b))->'b\ event$

Thus, event combinators transform synchronization protocols that are fed to them into more complex synchronization protocols. They are elegant, and permit describing communication with a bottom-up approach.

The final duty we must consider in formulating a synchronization protocol is synchronization itself. We demand it explicitly with the *sync* term, which outputs a value from an event computation.

- $sync: 'a\ event \rightarrow\ 'a$

Selective communication is realized by the function *select* in Concurrent ML. However, this function is already a composition of terms: $select = sync \circ choose$.

Example 2

In this example, we will give a demonstration on how events can be composed in Concurrent ML to describe complex protocols. The program in Figure 5 demonstrates an accumulator acting as a server on function *acc*. The *select* term offers different interactions based on contact with the channels $addCh, subCH, readCH$, and $stopCH$. We consider the event construction of communication on the channel $addCH$, which offers the possibility of increasing the value of the accumulator

1. First, we specify that the channel in question passes integers.

$$val\ addCh = channel():\ int\ chan$$

2. Then, we create an event that describes receiving a value on channel $addCH$.

$$recEvt\ addCh$$

3. Afterward, we enhance this event with *wrap* so that when we synchronize the resulting event, the accumulator is updated with the *input* value passed on the channel.

$$wrap(recvEvt\ addCH, fn\ input => loop(sum + input))$$

4. Finally, we explicitly synchronize the event with sync. This event can synchronize with the function *add* appearing in Figure 1, since it performs the complementary action $sync(sendEvt\ (addCh, x))$

$$sync(wrap(recvEvt\ addCH, fn\ input => loop(sum + input)))$$

The reader can now verify the construction of protocols for the other channels and how they can be combined with *select*.

```
fun acc sum = let
        val addCh = channel() : int chan
    ...
        fun loop sum =
            select [
                wrap(recvEvt addCh, fn input => loop (sum+input)),
                wrap(recvEvt subCh, fn input => loop (sum-input)),
                wrap(sendEvt (readCh,sum), fn () => loop (sum)),
                wrap(recvEvt stopCh, fn () => exit())
            ]
            ...
fun add ((ACC(addCh,_, _, _)), x) = sync(sendEvt(addCh,x))
```

Figure 5: Accumulator example

Notice as well in Figure 5 the use of selective communication with procedural abstraction. The channels do not need to be exposed to the client for the server to offer the selection; they remain concealed "with data and type abstraction" [18] so that only the methods and accumulator are known to the user. This technique prevents misuse of channels, promoting safety while enjoying the liveness of selective communication.

2.4 Chaudhuri's Distributed Synchronization Protocol

Based on the theory of events, Chaudhuri develops a model for the synchronization and generation of events built to emulate CML's selective communication using the π-calculus. Throughout the paper, we will refer to his model as the *synchronization protocol*. Informally, the logic of this protocol involves three different agents: *points*, *channels*, and *synchronizers*.

An important observation is that an event synchronizes on a *commit point*, which is the moment when a message is sent or received over a channel. Naturally, then, points are processes that carry input or output actions modeled after *recEvt* and *sendEvt*. In a selection, only one of the candidate points is selected among a group of candidates for synchronization.

Hence, points are the agents that seek to be committed. To govern this procedure, a point p is bound to a synchronizer s denoted $p \in dom(s)$. Points inside a select statement have a common synchronizer.

The synchronizer's job is to regulate *who* can synchronize. Points require permission from their respective synchronizers to carry out their actions. The action α of a point is subject to the synchronizer so that $s(p) = \alpha$ denotes the action of p if synchronized by s.

In a server/client view of the protocol, *points* are like clients who make requests to a server, which is the *synchronizer*. However, to reach this server, there is an intermediary, which is the *channel*. The *channel*, apart from being the medium of communication, arbitrates whether the points fulfill some requirements needed for a secure synchronization, thus protecting the synchronizers from unexpected activity.

With this plan, Chaudhuri produces a set of states (Figure 6) that agents follow when running a CML-style synchronization program. To formalize his conceptualization, Chaudhuri introduces a small source language to produce programs that exhibit the synchronization of CML's selective communication.

States of Points

$(p \to c)[\![s^-, i_c^-]\!] \triangleq (\nu\ cd_p^+, cd_p^-)\overline{i_c^-}{<}cd_p^-{>}.cd_p^+(d_p^+).\heartsuit_p[\![d_p^+, s^-, c]\!]$

$(q \to \overline{c})[\![s^-, o_c^-]\!] \triangleq (\nu\ cd_p^+, cd_p^-)\overline{o_c^-}{<}cd_p^-{>}.cd_p^+(d_p^+).\heartsuit_p[\![d_p^+, s^-, \overline{c}]\!]$

$\heartsuit_p[\![d_p^+, s^-, \alpha]\!] \triangleq \overline{s^-}{<}p^-, d_p^+{>}.p^+().\alpha$

States of Channels

$$\begin{aligned}\odot_c[\![i_c^+, o_c^+]\!] \triangleq\ & i_c^+(cd_p^-).o_c^+(cd_q^-).\\ & ((\nu\ d_p^+, d_p^-, d_q^+, d_q^-)\overline{cd_p^-}{<}d_p^+{>}.\overline{cd_q^-}{<}d_q^+{>}.\oplus_c[\![d_p^-, d_q^-]\!]\\ & \mid \odot_c[\![i_c^+, o_c^+]\!])\end{aligned}$$

$$\begin{aligned}\oplus_c[\![d_p^-, d_q^-]\!] \triangleq\ & d_p^- \triangleright \{\\ & \quad acceptP : d_p^-(conf_p^-, cxl_p^-).d_q^- \triangleright \{\\ & \quad\quad acceptQ^- : d_q^-(conf_q^-, cxl_q^-).\overline{conf_p^-}{<>}.\overline{conf_q^-}{<>}.\mathbf{0},\\ & \quad\quad declineQ : d_q^-().\overline{cxl_p^-}{<>}.\mathbf{0}\},\\ & \quad declineP : d_p^-().d_q^- \triangleright \{\\ & \quad\quad acceptQ : d_q^-(conf_q^-, cxl_q^-).\overline{cxl_q^-}{<>}.\mathbf{0},\\ & \quad\quad declineQ : d_q^-().\mathbf{0}\}\}\end{aligned}$$

States of Synchronizers

$\boxdot_s \triangleq s^+(p^-, d_p^+).(\checkmark_s[\![d_p^+, p^-]\!] \mid \boxtimes_s)$

$\boxtimes_s \triangleq s^+(p^-, d_p^+).(\times_s[\![d_p^+, p^-]\!] \mid \boxtimes_s)$

$$\begin{aligned}\checkmark_s[\![d_p^+, p^-]\!] \triangleq\ & (\nu\ conf_p^+, conf_p^-, cxl_p^+, cxl_p^-)d_p^+ \triangleleft accept.\\ & \overline{d_p^+}{<}conf_p^-, cxl_p^-{>}.(conf_p^+().happy[\![p^-]\!] \mid cxl_p^+().sad)\end{aligned}$$

$\times_s[\![d_p^+, p^-]\!] \triangleq d_p^+ \triangleleft reject.\overline{d_p^+}{<>}.\mathbf{0}$

$happy[\![p^-]\!] \triangleq \overline{p^-}{<>}.\mathbf{0}$

$sad \triangleq (\nu s^+, s^-, \overrightarrow{p_i^+}, \overrightarrow{p_i^-})(\boxdot_s \mid \prod_{i_{1..n}}(p_i \to \alpha_i))$ where $\mathrm{dom}(s) = \{p_i\}$ and $\forall i \in 1..n\ s(p_i) = \alpha_i$

Figure 6: Interpretation of states as processes

Let c ranges over channels. We use $\overrightarrow{\phi_l}$ to denote a sequence $\phi_1, \phi_2, \ldots, \phi_n$ where $l \in 1..n$. The language has the following syntax [4]:

- Actions α are of the form $\overline{c}$ or c (input or output on c). Informally, actions model communication events built with [sendEvt and recvEvt].
- Programs are of the form $S_1 \mid ... \mid S_m$ (parallel composition of $S_1, ..., S_m$), where each $S_k (k \in 1..m)$ is either an action α, or a selection of actions, $select(\overrightarrow{\alpha_i})$. Informally, a selection of actions models the synchronization of a choice of events, following the CML function *select*.

The language has one important reduction rule for selective communication:

$$\frac{c \in \overrightarrow{\alpha_i} \qquad \overline{c} \in \overrightarrow{\beta_i}}{select(\alpha_i) \ \mid \ select(\overrightarrow{\beta_i}) \longrightarrow c \mid \overline{c}} \text{ (SEL COMM)}$$

To compile a program running the protocol, we denote indexed parallel composition with $\prod$ so that a program $S_1 \mid S_2 ... \mid S_m$ can be represented as $\prod_{k \in \{1..m\}} S_k$. Assuming C is the set of CML channels in a program $\prod_{k \in \{1..m\}} S_k$, the synchronization protocol is compiled into the π-calculus as $(\nu_{c \in C} i_c^+, i_c^-, o_c^+, o_c^-)(\prod_{c \in C} \odot_c \mid \prod_{k \in \{1..m\}} \overset{\approx}{S})$, where

$$\overset{\approx}{S} = \begin{cases} \alpha & \text{if } S = \alpha \\ (\nu s^+, s^-, \overrightarrow{p_i^+}, \overrightarrow{p_i^-})(\boxdot_s \mid \prod_{i_{1..n}} (p_i \to \alpha_i)) & \text{if } S = select(\overrightarrow{\alpha_i}), i \in 1..n \text{ and } s,\ \overrightarrow{p_i} \text{ are fresh} \\ & \text{names.} \end{cases}$$

This compilation follows the processes found in Figure 6. Items inside $[\![\cdot]\!]$ represent variables that are in the internal state in Chaudhuri's notation. The reader should use this as a guideline to locate variables at this stage. When we introduce contexts for our processes, this notation will be obsolete as all variables available to a process will be included in their context.

Terminology: When talking about the synchronization protocol, we will use the word *state* to signify its corresponding π-calculus process.

An interesting design detail about this protocol is that it works as a state machine with an underlying π-calculus mechanism. Under certain conditions, the processes 'transition' to new processes. Indeed, Figure 6 states the correspondence between the machine's states and their interpretation as π-calculus processes. Since it might help the reader understand how processes interact in the protocol, we provide the operational semantics for the state machine interpretation in Appendix C [4]. However, our primary interest are the π-calculus processes themselves and verifying that they are typable.

In the next section, we will use our session type system to study the protocol, focusing on devising session types for the relevant channels and then verifying that the processes are typable.

3 Session Types for the Protocol

In this section, our first step will be to describe the current communication model using session types. We first focus on the communication channels appearing in Figure 6. Namely: $cd_p, d_p, i_c, o_c, p, conf_p$, cxl_p and s. Once we develop session protocols for these, we will proceed to type-check processes.

3.1 Channel Types

In Figure 7, we propose session protocols for the mentioned channels. These protocols precisely describe the communication over the channels exhibited in Figure 6, since we begin with the assumption that the synchronization protocol is typable. We will discover that this is not the case in Section 3.2.

$P \triangleq ?\phi.end$	[point name type]
$CONF \triangleq !\phi.end$	[confirm type]
$CXL \triangleq !\phi.end$	[cancel type]
$D \triangleq \&\{accept : ?CONF.?CXL.end,$ $reject : ?\phi.end\}$	[decision type]
$S \triangleq *(?P.?D)$	[synchronizer type]
$C \triangleq ?(D.end).end$	[candidate type]
$I \triangleq *?((?D.end).end)$	[input on c type]
$O \triangleq *?((?D.end).end)$	[output on c type]
$CH \triangleq *?(T)$	[CML chan type]

Figure 7: Proposed Protocol Types

Specifically, the correspondence of session types to channel ends is as follows: P types p^+, $CONF$ types $conf^-$, CXL types cxl^-, D types d_p^+, S types s^+, C types cd_p^+, I types i_c^+ and O types o_c^+, where c is an arbitrary channel and p an arbitrary point. The opposite channel-ends are typed in accordance with type duality. For example, $\overline{P}$ types p^-.

Types S, I, and O have been marked with the $*$ qualifier, meaning channels following these protocols should be available to zero or more processes for use multiple times. Intuitively, the channel ends typed at I and O receive requests from processes that want to do input or output on a channel c, so they must remain available for multiple processes. A similar argument holds for S: a channel end following S is a synchronizer channel that must be available to several points.

Example 3

To demonstrate how we derived the session types in Figure 7, we consider the following process:

$\boxdot_s \triangleq s^+(p^-, d_p^+).(\checkmark_s[\![d_p^+, p^-]\!] \mid \boxtimes_s)$

We first notice that s^+ is used to send p^- and d_p^+. We then create session types S, P and D for each channel-end, respectively. Given our description, we have

$S \triangleq \star(?P.?D)$

So that s^+ can send p^- and d_p^+ sequentially. Furthermore, we add the $\star$ qualifier to S since s^+ is used recursively in the protocol. Now, we want to define the behavior for P and D. In the case of P, we pay attention to the following process.

$happy[\![p^-]\!] \triangleq \overline{p^-}{<>}.\mathbf{0}$

We see that p^- is used to send an empty message, so we define P as

$P \triangleq !\phi.end$

We continue this recursive procedure for all available channel-ends.

Compare our session types to the channel types used by Chaudhuri in his implementation of the protocol in Concurrent Haskell (where MVar is a cell comparable to a channel in the π-calculus, and the *Maybe* keyword is used to the effect of branching in session types) [17, 10]. These appear in Figure 8, showing a similar recursive structure.

```
type I = MVar Cd
type O = MVar Cd
type Cd = MVar D
type S = MVar (P, D)
type D = MVar (Maybe (Conf, CXL))
type Conf = MVar ()
type Cxl = MVar ()
type P = MVar ()
```

Figure 8: Protocol's Channels in Concurrent Haskell

We also introduce the dummy type CH for communication realized in the α action stage. This is an unrestricted type that is polymorphic since it is, in essence, a normal CML channel. However, since it is a case of compile-time polymorphism, it does not need to change during run-time. We leave this unknown as is for now since Chaudhuri does not explicitly consider it. However, in Section 4, we resolve its type as we study how the protocol splits into our Concurrent ML primitives.

With these session types, we have a type system that is sufficient to describe the communication in the synchronization protocol. We can now investigate whether the interactions realized in it are indeed well-formed.

3.2 Typing the Processes of a Protocol Run

With our type system, we can now investigate the typability of the synchronization protocol. Our approach will be to build typing contexts for the states appearing in Figure 6. Subsequently, we try to prove whether these contexts can type their respective states using the rules in Figure 4. By the end of this subsection, we discover that our type assignments cannot produce contexts that type every process in the protocol.

Ill-typed processes will be marked in blue. Changes to the protocol will ensue from these problematic processes. Although our primary concern is the application of session type theory, we also provide a small informal description of each state's task during execution. In some cases, these descriptions help to make typability problems clear.

3.2.1 States of a point

Points start off interacting with channel agents to indicate that they want to communicate. When a channel agent c is instantiated, two channels are made available for requests: i_c^- and o_c^-, corresponding to input and output.

A point p expecting input on channel c becomes process $(p \to c)$, which signals on i_c^-, while a point q wanting to send information on c becomes process $(q \to \overline{c})$, which signals on o_c^-.

In both instances, the points create a pair of channel ends cd_p^+ and cd_p^- (or cd_q^+ and cd_q^-). The signal sent on the request channel shares the end cd_p^- to the channel agent. The point then expects to receive a key d_p^+ on the end cd_p^+. Once this key is received, the point transitions to state $\heartsuit_p$. This key is used to communicate with its synchronizer, and is only received once the channel agent matches two complementary points $(p \to c)$ and $(q \to \overline{c})$, each with different synchronizers. That is, if $p \in \mathrm{dom}(s), q \in \mathrm{dom}(s'), s \neq s' \implies s \cap s' = \varnothing$.

We can now proceed with the derivations.

[$(p \to c)$: input request]

This process describes a point (labeled p) that wants to read on a channel c.

Context: $s^- : \overline{S}, p^+ : P, p^- : \overline{P}, i_c^- : \overline{I}, c : CH$

Type derivation:

$$\dfrac{\dfrac{\dfrac{s^- : \overline{S}, p^+ : P, p^- : \overline{P}, d_p^+ : \overline{D}, cd_p^+ : end, i_c^- : \overline{I}, c : CH \vdash \heartsuit_p}{s^- : \overline{S}, p^+ : P, p^- : \overline{P}, cd_p^+ : C, i_c^- : \overline{I}, c : CH \vdash cd_p^+(d_p^+).\heartsuit_p}\text{ T-IN}}{s^- : \overline{S}, p^+ : P, p^- : \overline{P}, i_c^- : \overline{I}, cd_p^+ : C, cd_p^- : \overline{C}, c : CH \vdash \overline{i_c^-}\text{<}cd_p^-\text{>}.cd_p^+(d_p^+).\heartsuit_p}\text{ T-OUT}}{s^- : \overline{S}, p^+ : P, p^- : \overline{P}, i_c^- : \overline{I}, c : CH \vdash (p \to c)}\text{ T-RES}$$

[$(q \to \overline{c})$: output request]

This process describes a point that wants to write on c.

Context: $s^- : \overline{S}, q^+ : P, q^- : \overline{P}, o_c^- : \overline{O}, c : \overline{CH}$

Type derivation:

$$\dfrac{\dfrac{\dfrac{s^- : \overline{S}, q^+ : P, q^- : \overline{P}, d_q^+ : \overline{D}, cd_q^+ : end, o_c^- : \overline{O}, c : \overline{CH} \vdash \heartsuit_q}{s^- : \overline{S}, q^+ : P, q^- : \overline{P}, cd_q^+ : CAND, o_c^- : \overline{O}, c : \overline{CH} \vdash cd_p^+(d_p^+).\heartsuit_q}\ \text{T-IN}}{s^- : \overline{S}, q^+ : P, q^- : \overline{P}, o_c^- : \overline{O}, cd_q^+ : CAND, cd_q^- : \overline{CAND}, c : \overline{CH} \vdash \overline{o_c}{<}cd_q^-{>}.cd_q^+(d_q^+).\heartsuit_q}\ \text{T-OUT}}{s^- : \overline{S}, q^+ : P, q^- : \overline{P}, o_c^- : \overline{O}, c : \overline{CH} \vdash (q \to \overline{c})}\ \text{T-RES}$$

[$\heartsuit_p$: release action α]

In this state, the connection between the point and its synchronizer is established, meaning it has already been *matched* with another point. A point identifies itself with its name and key d_p to the synchronizer. If the synchronizer replies back on the name p, the point releases its action α so that it can execute it. The typing context for this process is dependent on whether it is coming from $(p \to c)$ or $(q \to \overline{c})$. Both derivations are similar; we just switch session types I for O. Since they are unrestricted, they do not prevent us from applying rule [T-INACT]. Therefore we only demonstrate one of them.

Context $\heartsuit_p$:
$s^- : \overline{S}, p^+ : P, p^- : \overline{P}, d_p^+ : \overline{D}, cd_p^+ : end, i_c^- : \overline{I}, c : CH \vdash \heartsuit_p$
Context $\heartsuit_q$:
$s^- : \overline{S}, q^+ : P, q^- : \overline{P}, d_q^+ : \overline{D}, cd_q^+ : end, o_c^- : \overline{O}, c : \overline{CH} \vdash \heartsuit_q$

Type derivation for $\heartsuit_p$ coming from $(p \to c)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{s^- : \overline{S}, p^+ : end, cd_p^+ : end, i_c^- : \overline{I}, c : CH \vdash \mathbf{0}}\ \text{T-INACT}}{s^- : \overline{S}, p^+ : end, cd_p^+ : end, i_c^- : \overline{I}, c : CH \vdash \alpha}\ \text{T-IN}}{s^- : \overline{S}, p^+ : P, cd_p^+ : end, i_c^- : \overline{I}, c : CH \vdash p^+().\alpha}\ \text{T-IN}}{s^- : \overline{S}, p^+ : P, p^- : \overline{P}, d_p^+ : \overline{D}, cd_p^+ : end, i_c^- : \overline{I}, c : CH \vdash \heartsuit_p}\ \text{T-OUT} \times 2$$

We kept Chaudhuri's tuple passing in the process $\overline{s}{<}p, d_p{>}$. This is simply syntactic sugar for $\overline{s}{<}p{>}.\overline{s}{<}d_p{>}$, a sequential sending of the messages, so we apply the [T-OUT] rule twice. We have reached our base case [T-INACT] in the type derivations of our point processes, so our types hold up.

3.2.2 States of a channel

The following states are meant for our channel agents. They monitor channel usage and control that communication is sound, taking different courses of action depending on the communication's current conditions.

Specifically, the protocol selects a course of action with a non-deterministic choice in the state $\oplus_c$. As originally depicted by Chaudhuri, it would have been impossible to type this state with session types. We show this in Figure 9.

The issue with this process is how choice is conducted. It breaks **Invariant 1** from Section 2.2. Notably, when d_p^- receives linear channels $conf_p^-$ and cxl_p^- in line 2, it splits into two parallel processes that receive these channels in lines 3 and 4. Of course, this implies that there are linear channels

$$
\begin{array}{ll}
\oplus_c[\![d_p^-, d_q^-]\!] \triangleq & \\
\quad (d_p^-(conf_p^-, cxl_p^-). & \text{[line 2]}\\
\qquad (d_q^-(conf_q^-, cxl_q^-).conf_p^-\texttt{<>}.conf_q^-\texttt{<>}.\mathbf{0} & \text{[line 3]}\\
\qquad \mid d_q^-().cxl_p^-\texttt{<>}.\mathbf{0}) & \text{[line 4]}\\
\quad \mid d_p^-(). & \\
\qquad (d_q^-(conf_q^-, cxl_q^-).cxl_p^-\texttt{<>}.\mathbf{0} & \\
\qquad \mid d_q^-().\mathbf{0})) &
\end{array}
$$

Figure 9: Original $\oplus_c$ with untypable non-determinism

appearing in more than one thread of execution. Therefore, we must find an alternative encoding that respects the invariant.

For this purpose, we make use of our language's selection and branching constructs, and indeed this is enough to express Chaudhuri's concept. We modified this state into a branched choice that conforms to our theory. It maintains linearity in our session type model while still permitting a flexible selection of behaviors for d_p and d_q. We preserve the fact that only one branch of execution is chosen; the rest are aborted.

Before proceeding, it is worth to note that this change does not affect the non-determinism of CML selective communication. Our modification concerns an inner mechanism to check that synchronizers can cooperate to perform synchronization; it does not limit synchronization itself.

The π-calculus employed by Chaudhuri [13] in his work did not contain branched choices. Nonetheless, it is a suitable tool for our needs.

[$\odot_c$: Open Channel]

This process is an intermediary that connects two points p and q that wish to communicate over the same channel c. If it receives signals on both i_c^+ and o_c^+, it creates the decision fresh names $d_p^+, d_p^-, d_q^+, d_q^-$. The positive ends are the keys for the points to communicate with their respective synchronizers. This process refreshes itself recursively for further communication over c.

Context: $i_c^+ : I, o_c^+ : O$

Type derivation

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{cd_p^- : \overline{end}, cd_q^- : \overline{end}, d_p^- : D, d_q^- : D \vdash \oplus_c}{cd_p^- : \overline{end}, cd_q^- : \overline{CAND}, d_p^- : D, d_q^+ : \overline{D}, d_q^- : D \vdash \overline{cd_q^-}<d_q^+>.\oplus_c}\ \text{T-OUT}}{cd_p^- : \overline{CAND}, cd_q^- : \overline{CAND}, d_p^+ : \overline{D}, d_p^- : D, d_q^+ : \overline{D}, d_q^- : D \vdash \overline{cd_p^-}<d_p^+>.\overline{cd_q^-}<d_q^+>.\oplus_c}\ \text{T-OUT}}{cd_p^- : \overline{CAND}, cd_q^- : \overline{CAND}, d_p^+ : \overline{D}, d_p^- : D \vdash (\nu\ d_q^+, d_q^-)\overline{cd_p^-}<d_p^+>.\overline{cd_q^-}<d_q^+>.\oplus_c}\ \text{T-RES}}{cd_p^- : \overline{CAND}, cd_q^- : \overline{CAND}, \vdash (\nu\ d_p^+, d_p^-, d_q^+, d_q^-)\overline{cd_p^-}<d_p^+>.\overline{cd_q^-}<d_q^+>.\oplus_c}\ \text{T-RES} \qquad i_c^+ : I, o_c^+ : O \vdash \odot_c}{i_c^+ : I, o_c^+ : O, cd_p^- : \overline{CAND}, cd_q^- : \overline{CAND} \vdash (\nu\ d_p^+, d_p^-, d_q^+, d_q^-)\overline{cd_p^-}<d_p^+>.\overline{cd_q^-}<d_q^+>.\oplus_c \mid \odot_c}\ \text{T-PAR}}{i_c^+ : I, o_c^+ : O, cd_p^- : \overline{CAND} \vdash o_c^+(cd_q^-).((\nu\ d_p^+, d_p^-, d_q^+, d_q^-)\overline{cd_p^-}<d_p^+>.\overline{cd_q^-}<d_q^+>.\oplus_c \mid \odot_c)}\ \text{T-IN}}{i_c^+ : I, o_c^+ : O \vdash \odot_c}\ \text{T-IN}
$$

[$\oplus_c$: Announced channel]

The $\oplus_c$ state awaits an answer on d_p^- and d_q^- from the synchronizers of the matching points. Since these synchronizers do not have state in common, $\oplus_c$ coordinates their actions depending on the responses received on these channel ends. There are four possible procedures:

- Reduction $acceptP \rightarrow acceptQ$ signals back to both synchronizers that synchronization can be continued. This is done through $conf_p^-$ and $conf_q^-$.
- Reduction $declineP \rightarrow acceptQ$ signals back to the synchronizer of q that p cannot synchronize. This is done through cxl_q^-.
- Reduction $acceptP \rightarrow declineQ$ signals back to the synchronize of p that q cannot synchronize through cxl_p^-.
- Reduction $declineP \rightarrow declineQ$ terminates this process without further action since no synchronization can occur on both ends.

The channels $conf_p^-$ and cxl_p^- for a point p are used to communicate a Boolean response. Writing on cxl_p^- represents a negative response, while writing on $conf_p^-$ represents a positive point. This type of logic first appears in [13].

Evidenced by the many branches of execution, the type derivation for this process is very large. Therefore we split the tree upon the four main choices. For a process x, A_x and R_x will abbreviate the accept and declining reductions respectively.

Context: $cd_p^- : end, cd_q^- : end, d_p^- : D, d_q^- : D$

Type derivation p accepted and q accepted

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end,\\ conf_p^- : end, cxl_p^- : CXL, conf_q^- : end,\\ cxl_q^- : CXL \vdash \mathbf{0}\end{array}}\ \text{ILL PROCESS}}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end,\\ conf_p^- : CONF, cxl_p^- : CXL, conf_q^- : CONF, cxl_q : CXL \vdash\\ \overline{conf_p}\texttt{<>}\overline{conf_q}\texttt{<>}.\mathbf{0}\end{array}}\ \text{T-OUT } \times 2}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : ?CONF.?CXL.end,\\ conf_p^- : CONF, cxl_p^- : CXL \vdash\\ d_q(conf_q^-, cxl_q^-).\overline{conf_p}\texttt{<>}\overline{conf_q}\texttt{<>}.\mathbf{0} \qquad R_q\end{array}}\ \text{T-IN } \times 2}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : D, conf_p^- : CONF, cxl_p^- : CXL \vdash\\ d_q \triangleright \{...\}\end{array}}\ \text{T-BRANCH}}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : ?CONF.?CXL.end, d_q^- : D \vdash\\ d_p(conf_p^-, cxl_p^-).d_q \triangleright \{...\} \qquad R_p\end{array}}\ \text{T-IN } \times 2}{cd_p^- : end, cd_q^- : end, d_p^- : D, d_q^- : D \vdash \oplus_c}\ \text{T-BRANCH}
$$

We cannot type the process because the context is not unrestricted when reaching inaction. The derivation shows that we cannot reach the base case [T-INACT] with the current session types. We marked the place where this necessary rule should occur as 'ILL PROCESS'.

This issue arises because of the way the channel agent signals back to the synchronizer in our protocol: in any possible reduction, the reply uses either channel $conf^-$ or cxl^-, even though we create both linear channels. This situation leads to either the session type *CONF* or *CXL* not being consumed after the agent generates the reply.

While this form of Boolean decision over two different channels is known in the literature, it does not favor our goals because it does not lead to consuming all the session types in a context.

3.2.3 States of a synchronizer

The following states enclose the behavior of the synchronizer agents. A synchronizer naturally starts *open* for synchronization. Once one of the points in its domain synchronizes, it remains *closed* for further synchronization requests.

[$\boxdot_s$: open synchronizer]

$\boxdot_s \triangleq s^+(p^-, d_p^+).(\checkmark_s \mid \boxtimes_s)$

This state represents an open synchronizer. It listens on its channel s^+ and offers the possibility of synchronization. Once the open synchronizer is used, it is 'consumed' and becomes closed $\boxtimes_s$ for the remaining interactions, unless rebooted.

Context: $s^+ : S$

Type derivation

$$\dfrac{\dfrac{d_p^+ : \overline{D}, p^- : \overline{P} \vdash \checkmark_s \qquad s^+ : S \vdash \boxtimes_s}{s^+ : S, d_p^+ : \overline{D}, p^- : \overline{P} \vdash \checkmark_s \mid \boxtimes_s}\,\text{T-PAR}}{s^+ : S \vdash \boxdot_s}\,\text{T-IN} \times 2$$

[$\boxtimes_s$:closed synchronizer]

$\boxtimes_s \triangleq s^+(p^-, d_p^+).(\times_s \mid \boxtimes_s)$

This state represents a closed synchronizer. It listens on its channel s^+ for requesting processes and rejects their requests since the synchronizer is no longer offering synchronization. It replenishes itself recursively for further point requests.

Context: $s^+ : S$

Type derivation

$$\dfrac{\dfrac{d_p^+ : \overline{D}, p^- : \overline{P} \vdash \times_s \qquad s^+ : S \vdash \boxtimes_s}{s^+ : S, d_p^+ : \overline{D}, p^- : \overline{P} \vdash \times_s \mid \boxtimes_s}\,\text{T-PAR}}{s^+ : S \vdash \boxtimes_s}\,\text{T-IN} \times 2$$

[$\checkmark_s$: synchronizable point]

This process signals back to the channel that the synchronizer for point p is available for synchronization. It awaits a reply from $\oplus_c$, which investigates whether the point q is also available for synchronization. If this is the case, we reach state $happy_s$; otherwise, we get to state sad_s.

Context: $d_p^+ : \overline{D}, p^- : \overline{P} \vdash \checkmark_s$

Type derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} d_p^+ : end, p^- : \overline{P}, \\ conf_p^+ : end \vdash happy \end{array}}{\begin{array}{c} d_p^+ : end, p^- : \overline{P}, conf_p^+ : \overline{CONF} \vdash \\ conf_p^+().happy \end{array}}\,\text{T-IN} \quad \dfrac{cxl_p^+ : end \vdash sad}{\begin{array}{c} cxl_p^+ : \overline{CXL} \vdash \\ cxl_p^+().sad \end{array}}\,\text{T-IN}}{\begin{array}{c} d_p^+ : end, p^- : \overline{P}, conf_p^+ : \overline{CONF}, cxl_p^+ : \overline{CXL} \vdash \\ (conf_p^+().happy \mid cxl_p^+().sad) \end{array}}\,\text{T-PAR}}{\begin{array}{c} d_p^+ : !CONF.!CXL.end, p^- : \overline{P}, conf_p^+ : \overline{CONF}, cxl_p^+ : \overline{CXL}, conf_p^- : CONF \\ , cxl_p^- : CXL \vdash \overline{d_p^+}{<}conf_p^-, cxl_p^-{>}.(conf_p^+().happy \mid cxl_p^+().sad) \end{array}}\,\text{T-OUT} \times 2}{\begin{array}{c} d_p^+ : \overline{D}, p^- : \overline{P}, conf_p^+ : \overline{CONF}, cxl_p^+ : \overline{CXL}, conf_p^- : CONF, cxl_p^- : CXL \vdash \\ d_p^+ \lhd accept.\overline{d_p^+}{<}conf_p^-, cxl_p^-{>}.(conf_p^+().happy \mid cxl_p^+().sad) \end{array}}\,\text{T-SEL}}{d_p^+ : \overline{D}, p^- : \overline{P} \vdash \checkmark_s}\,\text{T-RES}$$

[$\times_s$ unsynchronizable point]

This process tells the channel that the synchronizer is closed and cannot synchronize the point.

Context: $d_p^+ : \overline{D}, p^- : \overline{P}$

Type derivation

$$\dfrac{\dfrac{\dfrac{}{d_p^+ : end, p^- : \overline{P} \vdash \mathbf{0}}\ \text{ILL PROCESS}}{d_p^+ :!\phi.end, p^- : \overline{P} \vdash \overline{d_p}\texttt{<>}.\mathbf{0}}\ \text{T-OUT}}{d_p^+ : \overline{D}, p^- : \overline{P} \vdash \times_s}\ \text{T-SEL}$$

This process fails to use channel p when terminating, so we cannot conclude that $d_p^+ : \overline{D}, p^- : \overline{P} \vdash \times_s$. Again, we mark the location in the derivation where we expect to use rule [T-INACT] with 'ILL PROCESS' to denote that the derivation was not successful.

[happy: synchronized]

This state approves the synchronization of a point, so it signals over p^- that its request was accepted. When the point receives this signal, it can go on and perform its action.

Context: $d_p^+ : end, p^- : \overline{P}, conf_p^+ : end \vdash happy$

Type derivation

$$\dfrac{\dfrac{}{d_p^+ : end, p^- : end, conf_p^+ : end \vdash \mathbf{0}}\ \text{T-INACT}}{d_p^+ : end, p^- : \overline{P}, conf_p^+ : end \vdash happy}\ \text{T-OUT}$$

[sad : reboot]

Process: $sad \triangleq (\nu s^+, s^-, \overrightarrow{p_i^+}, \overrightarrow{p_i^-})(\boxdot_s \mid \prod_{i_{1..n}}(p_i \to \alpha_i))$ where $\mathrm{dom}(s) = \{p_i\}$ and $\forall i \in 1..n\ s(p_i) = \alpha_i$

A synchronizer willing to synchronize transitions to this state if the channel decides that synchronization is no longer authorized. Specifically, when the synchronizer of the other matched point does not approve the synchronization, the channel sends a signal to cancel the synchronization. In this case, the synchronizer terminates the procedure. What it subsequently does is to reboot the synchronizer and the points in its domain to attempt everything again.

Context: $cxl_p^+ : end \vdash sad$

The incoming type context $cxl_p^+ : end$ is a closed linear channel. It is well-typed in this context because it carries out no new communication; it just reboots the synchronizer. The underlying construction, is built upon the processes we have been studying (specifically, $\boxdot_s$, $(p \to c)$ and $(q \to \overline{c})$), so the protocol is invoked again.

Now, we have studied all the states of a run of the synchronization protocol. Nevertheless, this section demonstrates that our current assignment of types to processes makes it impossible to type some specific instances. In the next section, we update our session types to resolve this issue.

3.3 Revised Session Types

We uncovered some untypable processes while analyzing the protocol. As it is, our protocol cannot gain the guarantees the type theory has to offer. Hence, the objective is to improve the session types we developed in section 3.1 to fix the assignments of types to protocols. In particular, we pay attention to states $\oplus_c$ and $\times_s$ and how they violate session types laws.

The violations in these states stem from reaching process inaction with unused linear channels. This occurrence goes against the guarantee that session channels follow their expected structure: *session fidelity*. Chaudhuri's protocol relies on a garbage collector available in Concurrent Haskell that deals with unused resources. We do not count on such luxury in our session type model.

In process $\oplus_c$, we encountered the choice of following either the CONF or CXL session to execute certain behavior, which leaves the other unconsumed. We can simplify this decision's logic into a boolean choice: CONF merely transmits a positive signal to the synchronizer, while CXL transmits a negative one. This can easily be encoded with the boolean passing capabilities of our session types language. Therefore, we condense these sessions into the new session type SIG.

$$SIG \triangleq !bool.end$$

The session type SIG encompasses our old types CONF and CXL in a single boolean type. This transition changes our choice from 'what channel to signal on,' to 'what boolean to send'. The content is the same, but the session type will always be consumed. We reflect on this change in our decision type since it was dependent on the offending protocols.

$$D \triangleq \&\{accept : ?SIG.end,$$

$$reject : ?\phi.end\}$$

We also modify our point name type P so that we can type the state $\times_s$. Given point p and synchronizer s, where $p \in \text{dom}(s)$, the state $\times_s$ is invoked when the synchronization of point p is rejected. The current issue is that this decision is never communicated to the point.

In Chaudhuri's model, the matched point $\heartsuit_p$ only reacts when allowed to synchronize. More specifically, when it receives a message on channel p^+ from process $\checkmark_s$. However, when the synchronizer is closed, and synchronization is not possible, rather than aborting the point by sending a message from state $\times_s$, the state terminates prematurely and lets the garbage collector handle the point. Consequently, the linear channel p^+ is never used in this instance. To resolve this, we transform protocol P into yet another boolean-passing session.

$$P \triangleq ?bool.end$$

Our new session types are given in Figure 10. With a slight change in our synchronization protocol, we will demonstrate that we can use new type assignments to maintain session-fidelity and thus type all of the processes involved in a protocol run.

$P \triangleq ?bool.end$	[point name type]
$SIG \triangleq !bool.end$	[signal type]
$D \triangleq \&\{accept : ?SIG.end,$	
$\quad reject : ?\phi.end\}$	[decision type]
$S \triangleq *(?P.?D)$	[synchronizer type]
$C \triangleq ?(D.end).end$	[candidate type]
$I \triangleq *?((?D.end).end)$	[input on c type]
$O \triangleq *?((?D.end).end)$	[output on c type]
$CH \triangleq *?(T)$	[CML chan type]

Figure 10: New Protocol Session Types

3.4 The Revised Synchronization Protocol

With the new session type system from Section 3.3, we produce one final revised synchronization protocol in Figure 11. The states are now presented with the contexts in which they are typable. Most states remain the same. For the sake of completeness, we will type-check the states with a dramatic change.

[$\heartsuit_p$: release action]

We accommodate $\heartsuit_p$ to our new session type for P. This brings us to a decision on the value v received over p^+. We now model this state as

$\heartsuit_p \triangleq \overline{s}\text{<}p^-, d_p\text{>}p^+(v).(\text{if } v \text{ then } \alpha \text{ else } \mathbf{0})$

In particular, if v is true, then the point is synchronized, else it is aborted. The following type derivation is for $\heartsuit_p$ when coming from an input request.

Context: $s : \overline{S}, p^+ : P, p^- : \overline{P}, d_p^+ : \overline{D}, cd_p^+ : end, i_c : end \vdash \heartsuit_p$

Type derivation for $\heartsuit_p$ coming from $(p \to c)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c}s^- : \overline{S}, p^+ : end,\\ cd_p^+ : end, i_c : \overline{I}, c : CH \vdash \mathbf{0}\end{array}}\;\text{T-INACT}}{\begin{array}{c}s^- : \overline{S}, p^+ : end, cd_p^+ : end,\\ i_c : \overline{I}, c : CH \vdash \alpha\end{array}}\;\text{T-IN} \qquad \dfrac{}{\begin{array}{c}s^- : \overline{S}, p^+ : end, cd_p^+ : end,\\ i_c : \overline{I}, c : CH \vdash \mathbf{0}\end{array}}\;\text{T-INACT}}{s^- : \overline{S}, p^+ : end, cd_p^+ : end, i_c : \overline{I}, v : bool, c : CH \vdash (\text{if } v \text{ then } \alpha \text{ else } \mathbf{0})}\;\text{T-IF}}{s^- : \overline{S}, p^+ : P, cd_p^+ : end, i_c : \overline{I}, c : CH \vdash p^+(v).(\text{if } v \text{ then } \alpha \text{ else } \mathbf{0})}\;\text{T-IN}}{s^- : \overline{S}, p^+ : P, p^- : \overline{P}, d_p^+ : \overline{D}, cd_p^+ : end, i_c : \overline{I}, c : CH \vdash \heartsuit_p}\;\text{T-OUT} \times 2$$

[$\oplus_c$: Announced channel]

We now show the revised $\oplus_c$ is typable using the new session type SIG. We again consider the reduction $acceptP \to acceptQ$ from Section 3.2.2; the remaining cases can be found in Appendix B and shows that all of the possible execution branches are well-typed in the current context.

States of Points

$(p \to c) \triangleq (\nu\ cd_p^+, cd_p^-)\overline{i_c^-}{<}cd_p^-{>}.cd_p^+(d_p^+).\heartsuit_p$
Typable as $p^+ : P, p^- : \overline{P}, s^- : \overline{S}, i_c^- : \overline{I}, c : CH \vdash (p \to c)$

$(q \to \overline{c}) \triangleq (\nu\ cd_p^+, cd_p^-)\overline{o_c^-}{<}cd_p^-{>}.cd_p^+(d_p^+).\heartsuit_p$
Typable as $p^+ : P, p^- : \overline{P}, s^- : \overline{S}, o_c^- : \overline{O}, c : CH \vdash (q \to \overline{c})$

$\heartsuit_p \triangleq \overline{s^-}{<}p^-, d_p^+{>}.p^+(v).(\text{if } v \text{ then } \alpha \text{ else } \mathbf{0})$
Typable as $^-s : \overline{S}, p^+ : P, p^- : \overline{P}, d_p^+ : \overline{D}, cd_p^+ : end, i_c^- : end \vdash \heartsuit_p$
or $s^- : \overline{S}, q^+ : P, q^- : \overline{P}, d_p^+ : \overline{D}, cd_p^+ : end, o_c^- : end \vdash \heartsuit_q$

States of Channels

$\odot_c \triangleq i_c^+(cd_p^-).o_c^+(cd_q^-).$
$\quad((\nu\ d_p^+, d_p^-, d_q^+, d_q^-)\overline{cd_p^-}{<}d_p^+{>}.\overline{cd_q^-}{<}d_q^+{>}.\oplus_c$
$\quad\mid \odot_c)$

Typable as $i_c^+ : I, o_c^+ : O \vdash \odot_c$

$\oplus_c \triangleq d_p^- \triangleright \{$
$\quad acceptP : d_p^-(sig_p^-).d_q^- \triangleright \{$
$\qquad acceptQ^- : d_q^-(sig_q^-).\overline{sig_p^-}{<}\text{true}{>}.\overline{sig_q^-}{<}\text{true}{>}.\mathbf{0},$
$\qquad declineQ : d_q^-().\overline{sig_p^-}{<}\text{false}{>}.\mathbf{0}\},$
$\quad declineP : d_p^-().d_q^- \triangleright \{$
$\qquad acceptQ : d_q^-(sig_q^-).\overline{sig_q^-}{<}\text{false}{>}.\mathbf{0},$
$\qquad declineQ : d_q^-().\mathbf{0}\}\}$

Typable as $cd_p^- : end, cd_q^- : end, d_p^- : D, d_q^- : D \vdash \oplus_c$

States of Synchronizers

$\boxdot_s \triangleq s^+(p^-, d_p^+).(\checkmark_s \mid \boxtimes_s)$
Typable as $s^+ : S \vdash \boxdot_s$

$\boxtimes_s \triangleq s^+(p^-, d_p^+).(\times_s \mid \boxtimes_s)$
Typable as $s^+ : S \vdash \boxtimes_s$

$\checkmark_s \triangleq (\nu\ sig_p^+, sig_p^-)d_p^+ \lhd accept.$
$\qquad\qquad \overline{d_p^+}{<}sig_p^-{>}.sig_p^+(v).(\text{if } v \text{ then } happy \text{ else } sad)$

Typable as $d_p^+ : \overline{D}, p^- : \overline{P} \vdash \checkmark_s$

$\times_s \triangleq d_p^+ \lhd reject.\overline{d_p^+}{<>}.\overline{p^-}{<}\text{false}{>}.\mathbf{0}$

Typable as $d_p^+ : \overline{D}, p^- : \overline{P} \vdash \times_s$
$happy \triangleq \overline{p^-}{<}\text{true}{>}.\mathbf{0}$

Typable as $d_p : end, p^- : \overline{P}, sig_p^+ : end \vdash happy$
$sad \triangleq \overline{p^-}{<}\text{false}{>}.(\nu s^+, s^-, \overrightarrow{p_i^+}, \overrightarrow{p_i^-})(\boxdot_s \mid \prod_{i_{1..n}}(p_i \to \alpha_i))$ where $\text{dom}(s) = \{p_i\}$ and $\forall i \in 1..n\ s(p_i) = \alpha_i$
Typable as $d_p : end, p : \overline{P}, sig_p^+ : end \vdash sad$

Figure 11: The Revised Protocol

Context: $cd_p : \overline{CAND}, cd_q : \overline{CAND}, d_p^- : D, d_q^- : D \vdash \oplus_c$

Type derivation p accepted and q accepted

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end, \\ sig_p^- : end, sig_q^- : end \vdash \mathbf{0} \end{array}}\ \text{T-INACT}}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- :?end, \\ sig_p^- : SIG, sig_q^- : SIG \vdash \\ \overline{sig_p^-}\text{<true>}\overline{sig_q^-}\text{<true>}.\mathbf{0} \end{array}}\ \text{T-OUT} \times 2}{\begin{array}{c} cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- :?SIG.end, sig_p^- : SIG \vdash \\ d_q^-(sig_q^-).\overline{sig_p^-}\text{<true>}\overline{sig_q}\text{<true>}.\mathbf{0} \end{array}}\ \text{T-IN} \quad R_q}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : D, sig_p^- : SIG \vdash d_q^- \triangleright \{...\}}\ \text{T-BRANCH}}{cd_p^- : end, cd_q^- : end, d_p^- :?SIG.end, d_q^- : D \vdash d_p^-(sig_p).d_q^- \triangleright \{...\}}\ \text{T-IN} \quad R_p}{cd_p^- : end, cd_q^- : end, d_p^- : D, d_q^- : D \vdash \oplus_c}\ \text{T-BRANCH}$$

[$\checkmark_s$: synchronizable point]

We modify this process to follow the new SIG type. We introduce a conditional construct for this reason.

Context: $d_p^+ : \overline{D}, p^- : \overline{P} \vdash \checkmark_s$

Type derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{d_p^+ : end, p^- : \overline{P}, sig_p^+ : end \vdash happy \qquad d_p^+ : end, p^- : \overline{P}, sig_p^+ : end \vdash sad}{d_p^+ : end, p^- : \overline{P}, sig_p^+ : end, v : bool \vdash (\text{if } v \text{ then } happy \text{ else } sad)}\ \text{T-IF}}{d_p^+ : end, p^- : \overline{P}, sig_p^+ : \overline{SIG} \vdash sig_p^+(v).(\text{if } v \text{ then } happy \text{ else } sad)}\ \text{T-IN}}{\begin{array}{c} d_p^+ :!SIG.end, p^- : \overline{P}, sig_p^+ : \overline{SIG}, sig_p^- : SIG \vdash \\ \overline{d_p^+}\text{<}sig_p^-\text{>}.sig_p^+(v).(\text{if } v \text{ then } happy \text{ else } sad) \end{array}}\ \text{T-OUT}}{\begin{array}{c} d_p^+ : \overline{D}, p^- : \overline{P}, sig_p^+ : \overline{SIG}, sig_p^- : SIG \vdash \\ d_p^+ \triangleleft accept.\overline{d_p^+}\text{<}sig_p^-\text{>}.sig_p^+ v).(\text{if } v \text{ then } happy \text{ else } sad) \end{array}}\ \text{T-SEL}}{d_p^+ : \overline{D}, p^- : \overline{P} \vdash \checkmark_s}\ \text{T-RES}$$

[$\times_s$ unsynchronizable point]

We modify this process to follow the new P type. When a synchronizer cannot synchronize further, it signals back false on p to abort the point.

Context: $d_p^+ : \overline{D}, p^- : \overline{P} \vdash \times_s$
Type derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{d_p^+ : end, p^- : end \vdash \mathbf{0}}\ \text{T-INACT}}{d_p^+ : end, p^- : \overline{P} \vdash p^-\text{<false>}.\mathbf{0}}\ \text{T-OUT}}{d_p^+ :!\phi.end, p^- : \overline{P} \vdash \overline{d_p^+}\text{<>}.p^-\text{<false>}.\mathbf{0}}\ \text{T-OUT}}{d_p^+ : \overline{D}, p^- : \overline{P} \vdash \times_s}\ \text{T-SEL}$$

[happy: synchronized]

We modify this process to follow the new P type. It signals back true on p to synchronize the requesting point.

Context: $d_p^+ : end, p : \overline{P}^-, sig_p^+ : end \vdash happy$

Type derivation

$$\dfrac{\dfrac{}{d_p^+ : end, p^- : end, sig_p^+ : end \vdash \mathbf{0}}\text{ T-INACT}}{d_p^+ : end, p^- : \overline{P}, sig_p^+ : end \vdash happy}\text{ T-OUT}$$

The consequence of these changes is a fully typable version of the synchronization protocol's states, as appearing in Figure 11. To conclude this chapter, we will zoom back out to the compilation of a program into the π-calculus demonstrated in Section 2.5. This time, the compilation will be typed.

3.5 Typed Program Compilation

A typed compilation following the source language of Section 2.5 is given by

$(\nu_{c\in C} i_c^+ : I, o_c^+ : O, c^+ : CH, i_c^- : \overline{I}, o_c^- : \overline{O}, c^- : \overline{CH})(\prod_{c\in C} \odot_c \mid \prod_{k\in\{1..m\}} \tilde{\tilde{S}})$, where

$$\tilde{\tilde{S}} = \begin{cases} \alpha & \text{if } S = \alpha \\ (\nu s^+ : S, s^- : \overline{S}, \overrightarrow{p_i}^+ : P, \overrightarrow{p_i}^- : \overline{P})(\boxdot_s \mid \prod_{i_{1..n}} (p_i \to a_i)) & \text{if } S = select(\overrightarrow{\alpha_i}), i \in 1..n \text{ and} \\ & s,\ \overrightarrow{p_i} \text{ arefresh names.} \end{cases}$$

New Notation: We introduce *explicit annotations*, written as $(\nu x : T)$, to associate a session type with a variable at creation.

We now employ the explicit annotations introduced by Vasconcelos in his "Algorithmic type checking" [26]. With our typing rules, session types are introduced into the context of a process using the rule [T-RES] from Figure 4. It is worthy of attention that this rule does not impose any type on a channel at creation; we can give it any type at that moment. Therefore, we use these annotations to delineate the intended session types we wish to bind to a channel.

With this new compilation scheme, we can now give an example run of a typed program using the protocol.

Compilation Example

Consider the program

$$S \triangleq select(x, \overline{y}) \mid select(\overline{x}, z)$$

Via the (SEL COMM) rule from Section 2.5, we derive

$$select(x, \overline{y}) \mid select(\overline{x}, z) \to x \mid \overline{x}$$

Which is, ultimately, the synchronization we seek with our compiled program.

We compile the original program to a typed π-calculus process.

$$\begin{aligned}
&\textcircled{1}\ (\nu C)(\odot_x \mid \odot_y \mid \odot_z \mid \\
&\qquad\qquad \textcircled{2}\ ((\nu s_1^+ : S, p_1^+ : P, p_2^+ : P, s_1^- : \overline{S}, p_1^- : \overline{P}, p_2^- : \overline{P})(\boxdot_{s_1} \mid (p_1 \to x) \mid (p_2 \to \overline{y})) \mid \\
&\qquad\qquad \textcircled{3}\ ((\nu s_2^+ : S, p_3^+ : P, p_4^+ : P, s_2^- : \overline{S}, p_3^- : \overline{P}, p_4^- : \overline{P})(\boxdot_{s_2} \mid (p_3 \to \overline{x}) \mid (p_4 \to z)))
\end{aligned}$$

where

$$\begin{aligned}
(\nu C) \equiv (\nu & i_x^+ : I, o_x^+ : O, x^+ : CH, i_x^- : \overline{I}, o_x^- : \overline{O}, x^- : \overline{CH}, \\
& i_y^+ : I, o_y^+ : O, y^+ : CH, i_y^- : \overline{I}, o_y^- : \overline{O}, y^- : \overline{CH}, \\
& i_z^+ : I, o_z^+ : O, z^+ : CH, i_z^- : \overline{I}, o_z^- : \overline{O}, z^- : \overline{CH})
\end{aligned}$$

This process contains several sub-processes that we need to examine:

- channel agents: $\odot_x, \odot_y, \odot_z$;
- point agents: $(p_1 \to x), (p_2 \to \overline{y}), (p_3 \to \overline{x}), (p_4 \to z)$
- synchronizers: $\boxdot_{s_1}, \boxdot_{s_2}$, where $\{p_1, p_2\} \in \text{dom}(s_1)$ and $\{p_3, p_4\} \in \text{dom}(s_2)$.

With this breakdown, we now recognize that our task is to assign the right context to every state. Hence, our goal is to provide the appropriate context to every state, as exhibited in Figure 11. Again, we refer to the rules in Figure 4 to perform this.

We use [T-RES] to introduce session types into a context. Important to note is the location of the restrictions, which effectively set the scope of channel ends. By design, the names $s_1^+, s_1^-, p_1^+, p_1^-, p_2^+$ and p_2^- in $\textcircled{2}$ are internal to the process
$(\boxdot_s \mid (p_1 \to x) \mid (p_2 \to \overline{y}))$, while $s_2^+, s_2^-, p_3^+, p_3^-, p_4^+$ and p_4^- in $\textcircled{3}$ are internal to process
$(\boxdot_{s_2} \mid (p_3 \to \overline{x}) \mid (p_4 \to z))$.

Firstly, this scoping implies that the synchronizers are mutually exclusive, such that $\text{dom}(s_1) \cap \text{dom}(s_2) = \varnothing$. This property makes sure that every point has a single synchronizer. The other advantage is that these channel ends cannot be used for external communication, which is desirable. The channels in the group C are, on the other hand, 'declared globally' with respect to the program and are available for everyone.

Accordingly, we use [T-PAR] to provide a context for every state that we listed. Via our context split operation we maintain the invariant that linear ends occur in a single thread. In the end, we reduce the program to the following typed sub-processes:

1. $i_x^+ : I, o_x^+ : O \vdash \odot_x$
2. $i_y^+ : I, o_y^+ : O \vdash \odot_y$
3. $i_z^+ : I, o_z^+ : O \vdash \odot_z$
4. $p_1^+ : P, p_1^- : \overline{P}, s^- : \overline{S}, i_x^- : \overline{I}, x^+ : CH \vdash (p \to x)$
5. $p_2^+ : P, p_2^- : \overline{P}, s^- : \overline{S}, i_c^- : \overline{I}, y^- : \overline{CH} \vdash (p \to \overline{y})$
6. $p_3^+ : P, p_3^- : \overline{P}, s^- : \overline{S}, i_x^- : \overline{I}, x^- : \overline{CH} \vdash (p \to \overline{x})$
7. $p_4^+ : P, p_4^- : \overline{P}, s^- : \overline{S}, i_c^- : \overline{I}, z^+ : CH \vdash (p \to z)$
8. $s_1^+ : S \vdash \boxdot_{s_1}$

9. $s_2^+ : S \vdash \boxdot_{s_2}$

The reader can now verify that these typed processes concur with the formulas provided in Figure 11. This means that the compilation is well-typed and the synchronization protocol can proceed.

Indeed, by the transitions provided in Appendix C, The processes $(p \to \overline{y})$ and $(p \to z)$ are aborted and $(p \to x)$ and $(p \to \overline{x})$ are synchronized so that x and $\overline{x}$ become available for communication. Note that, in practice, this can occur over a number of reboots. The resulting process is $x \mid \overline{x}$ as expected. The proof that assures the progress of any run of the protocol is found in [3].

4 Translating the CML Synchronization Primitives

In the previous sections, we have studied and modified CML's synchronization protocol under our session type theory. We then obtained a version that guarantees type-safety and communication-safety. Consequently, we have achieved a verified method of applying selective communication in the style of CML in the π-calculus.

Concurrent ML's selective communication is an intriguing tool. Although this mechanism is not new to the π-calculus or process calculi in general [9], in CML, it is possible to use it in conjunction with abstract synchronization protocols and use it to insert(or remove) execution branches altogether at run time [16]. We discussed that this combination grants liveness without compromising safety in communication systems.

Acknowledging this is an advantage in describing complex systems, our focus now shifts towards the generation of synchronization protocols through event construction. More importantly, we wish to explore the compositional style of concurrent programming in CML and export it to the π-calculus. Accordingly, we will translate the CML primitives introduced in Section 2.3 into the π-calculus.

We begin this section by refining the unknown type CH to match the *chan* type constructor of Concurrent ML. After this, we provide a grammar that exposes the mechanism for event generation. We conclude by researching the individual Concurrent ML types and providing possible translations for them.

Channel Creation

A new channel in CML is created, just like in the π-calculus, dynamically, with the term *channel*(), typed at:

$$val\ channel() = unit \to {}'a\ chan$$

A channel typed at τ *chan* can pass values of type *tau* indefinitely, and is restricted to its scope. This is, notably, as a typed π-calculus channel. Therefore, we define interactions over this channel with a recursive, unrestricted channel. We use the following session type constructor:

$${}'a\ CH \triangleq \star?({}'a) \qquad \text{[channel type]}$$

This session type constructor will help us later translate the *channel* primitive, which will assume the form of a channel agent.

The Event Type

Event values are mainly produced by the Concurrent ML primitives discussed in Section 2, although others exist. Once more, they are typed as:

- $val\ sendEvt:\ ('a\ chan, 'a)->unit\ event$
- $val\ recvEvt:\ 'a\ chan->'a\ event$
- $val\ choose:\ 'a\ event\ list->'a\ event$
- $val\ wrap:\ ('a\ event * ('a->'b))->'b\ event$
- $val\ guard:\ (\text{unit} \to 'a\ event)->'a\ event$

We also added the additional *guard* term. This term is another event-combinator and is, in some respects, the opposite of wrap; it associates a pre-synchronization action to an event. Although less crucial to our investigation, we include it because our translation will also yield an encoding for this term into the π-calculus.

The CML *event* is used to type objects that perform a synchronous computation. The reader might have already made the comparison between the event value to Haskell Monads, as also studied in [7]. While their mechanisms are not the same, they both type objects that correspond to a computation that yields a value. In the case of Concurrent ML, it is the mechanism that allows us to type and compose synchronization protocols.

In Section 2.3, we saw an example of how to build such a synchronization protocol. We first produce an event that yields basic communication, such as sending or receiving. Then, we use event-combinators to augment our protocols, and finally, we perform event synchronization. We will now formalize event construction with a recursive grammar.

Let c range over channels, f over functions, and v range over values. We produce event values with the grammar presented in Figure 12.

$event ::=$		
	$\mid sendEvt(c)$	send event
	$\mid recvEvt(c, v)$	receive event
	$\mid choose([event])$	choice event
	$\mid wrap(event, f)$	wrap event
	$\mid guard(f)$	guard event

Figure 12: Event construction grammar

The base cases are *sendEvt* and *recvEvt*: they are the simplest events (guard could be considered one as well, but we do not as a simplification). We now also extend the types available in our session types model to the basic types naturally found in programming languages, shown in Figure 13.

With Figure 12, we get a notion of how event construction works in CML. The next step is to integrate event construction with the logic of event synchronization that we developed from Chaudhuri's work. We inquire over the two most important aspects that we must consolidate: what are the base-case events in the model we studied; and how do we perform event synchronization.

Firstly, as mentioned in Section 3, the point agents in our protocol model basic communication: they are processes wanting to send or receive data through a channel c. The points $(p \to c)$ and $(q \to \overline{c})$ correspond to *sendEvt* and *recvEvt*, fulfilling the role. As modeled by Chaudhuri, these *are* the base-case events in question. Therefore, all event construction in our translation should be built upon points.

$T ::=$

char	character
int	integer
bool	boolean
string	string
'a chan	channel
'a event	event
end	end
unit	unit
q p	qualified pretype

Figure 13: Extended types

The other essential ingredient of synchronization, as conceptualized by Chaudhuri, is the synchronizer agent. A point needs to communicate with a synchronizer to synchronize, with no exceptions. Hence, the point must establish a connection with the synchronizer by receiving a channel end s^- from it.

Assembling this information, and knowing that Figure 12 gives us an idea on how to combine events, we create the notion of an *event generator*.

Definition: An event generator $E«x»$ is an object that

1. is synchronized on a point agent;
2. can be enhanced by means of *choose* and *wrap*;
3. x is an internal parameter expecting a variable s typed as $\overline{S}$ which is given by a synchronizer;
4. when s is received, we produce the concrete event $E«s»$;
5. The channel s is recursively passed, until it arrives to a base case: $(p \to c)«x»$ or $(q \to \overline{c})«x»$.

To treat items 3, 4, and 5, we must find a way to perform a name substitution to form an event. We will do so using the notation for *abstractions* and *applications* loosely based on [1].

Briefly explained, an abstraction $(\lambda x.P«x»)$ is a process P, which takes in a parameter x. On the other hand, an application is a process $A\ u$ that substitutes a name on abstraction A. The reduction for this relationship is given by

$$(\lambda x.P«x»)\ u \to P«u» \qquad \text{[App]}$$

The intended purpose of this construct is to be “syntactic sugar" for scope extrusion. We use it only to substitute names inside a process, unlike in the exhibited paper. Using applications and abstractions, we want to explicitly expose the fact that an event generator *requires* a connection with a synchronizer to become an event.

Thus, our primary goal in event synchronization is to connect points to their synchronizers. Following our model from the previous sections, we want to make a point receive the channel end $s^- : \overline{S}$ that officially opens communication with a synchronizer agent.

Next, we will translate the *sync* term, followed by the *channel* term and then the event-producing terms. We now work with event generators and the assumption that c is a typed CML channel. We also type the protocol channels again as such: $i_c^+ : I, o_c^+ : O, cd_{p_k} : C, p_k^+ : P, d_{p_k} : D, sig_{p_k}^+ : \overline{SIG}, c^+ : {}'a\ CH$, where $'a \in T$ and $k \in \{1..n\}$. We device the translating function $[\![\cdot]\!]$ from CML events to π-calculus processes.

We will also need to use function application à la lambda calculus to translate *wrap*. These applications are possible to encode in the π-calculus as shown by Milner in [11, 12]. Since a concept of abstractions and applications already exists in our language, we denote function application as $M \leftarrow N$ to avoid confusion. It should be read as 'function M is applied to parameter N'.

For any states mentioned in the translations, please refer to Figure 11.

4.1 π-calculus Synchronization

$[\![sync(E«x» : {}'a\ event)]\!] \triangleq (\nu s^+ : S, s^- : \overline{S})\boxdot_s \mid ([\![E«x»]\!]\ s^-)$ where s^+, s^- are unique.

sync creates an instance of a synchronizer, spawning $\boxdot_s$ and passing the name s^- onto the event E.

4.2 π-calculus Channel Creation and Event Production

i $[\![channel()]\!] \triangleq (\nu i_c^+ : I, i_c^- : \overline{I}, o_c^+ : I, o_c^- : \overline{O}, c^+ : CH, c^- : \overline{CH})\odot_c$

The term *channel* creates an instance of $\odot_c$ together with the pair i_c and o_c. We instantiate a channel c as well for communication after synchronization.

ii $[\![sendEvt(c : {}'a\ chan, z : {}'a)]\!] \triangleq \lambda s.(\nu \overrightarrow{p}^+ : P, \overrightarrow{p}^- : \overline{P})((p \to c)«x»\ s)$ where $\alpha = c{<}z{>}$ and $s, \overrightarrow{p}$ are fresh.

The term *sendEvt* instantiates a point in state $(p \to c)$, with a synchronizer s and a fresh name p. It takes in a channel c and a value z to send.

iii $[\![recvEvt(c : {}'a\ chan)]\!] \triangleq \lambda s.(\nu \overrightarrow{q}^+ : P, \overrightarrow{q}^- : \overline{P})((q \to \overline{c})«x»\ s)$ where $\alpha = c(y)$ and $y : {}'a$ and $s, \overrightarrow{q}$ are fresh.

The term *recvEvt* instantiates a point in state $(q \to \overline{c})$, with a synchronizer s and a fresh name q. It takes in a channel c.

iv $[\![choose(E_{i\in 1...n}«x» : {}'a\ event\ list)]\!] \triangleq \lambda s.(\prod_{i\in 1...n}[\![E_i«x»]\!]\ s)$

The term choose takes as input a list of events and runs them in parallel. The synchronizer s is passed to each event in the list.

v $[\![wrap(E«x» : {}'a\ event, f : ({}'a{-}{>}{}'b))]\!] \triangleq \lambda s.(f \leftarrow ([\![E«x»]\!]\ s))$

This term enhances our event by adding a function call on the return value of synchronizing on E. Specifically, the created process receives a synchronizer s, which it passes onto the event E. If E synchronizes, then function f is applied to the post-synchronization value.

vi $[\![guard(f : (unit{-}{>}{}'a\ event))]\!] \triangleq \lambda s.(f«x»\ s)$

This term takes in a function f, which runs a computation with no input that produces an event as output. Then, synchronizer s is passed onto that event for synchronization.

Notice how the event-production terms are translated into abstractions that contain an event generator, and that the channel ends received through applications are recursively passed to the event generators inside of its structure. Once *sendEvt* or *recvEvt* is reached, the event is ready for synchronization.

Results (i, ii, iii, iv) are unsurprising: They are merely a restatement of a program's compilation.

To illustrate, (iv) is just the parallel composition of instances of (ii, iii, iv,v, vi) to the effect of assigning them the same single synchronizer. In other words, the expression *sync*(*choose*[...]) creates a unifying process that receives a single channel end s to supervise its synchronization.

Equivalence 1: $sync(wrap(ev, f)) \equiv f(sync(ev))$

On the other hand, the term (v) is more subtle and not covered by our studied protocol. Reppy, the creator of Concurrent ML, exposes Equivalence 1 in [20]. Intuitively, this signifies that the mechanism of *wrap* applies function f to return value of event ev after synchronization. Seeminlgly, there is a specific order of evaluation when using *wrap* that we want to simulate.

To express this order, Milner's encoding of the call-by-value lambda-calculus can mimic different forms of evaluation. Given an application $(M \leftarrow N)$, M can run first, or N can run first, or they can both run in parallel [6]. The order of evaluation is vital to our translation to agree with Equivalence 1. At synchronization of (v), we want process $(E«x» \ s)$ to compute its return value first so that it can be passed onto function f as a parameter and agree with the equivalence mentioned before. The cases that are compatible with our explanation are where N runs first, or both run in parallel.

Although we do not explicitly provide the encoding of lambda-calculus to π-calculus, we rely on this mechanism. A full description of this encoding is in [11, 12].

4.3 Other CML Primitives

Many other primitives exist in CML, but Reppy claims that *sendEvt*, *recvEvt*, *choose* and *wrap* are the core primitives for CML's distinctiveness as a modular message-passing language. Of course, there are additional terms for other kinds of behavior.

The *guard* term is used to express a computation that must be carried out before synchronization. It is often used to allocate resources like reply channels to a synchronization protocol or doing some pre-computation before event evaluation. We give its translation in element (vi).

Related to *wrap* are the combinators *wrapAbort* and its modern successor *withNack*. These perform actions on points that have been aborted. Notably, *withNack* provides a mechanism for informing servers that a client has aborted a transaction [20]. However, the implementation of *wrapAbort* requires enriching the definition of our event type constructor, whereas *withNack* requires using negative acknowledgments. In this respect, they are beyond the scope of this investigation.

Some other base-event constructors showcase CML synchronization, although not for message-passing per se. The simplest example is the *never* constant, with type

$$val\ never : {}'a\ event$$

which is never available for synchronization. It is the base-case of *choose*(since it is equivalent to $choose([\,])$). The encoding for this primitive results from (iv).

On the other hand, *alwaysEvt* takes in a value and produces an event that always returns this value upon synchronization. It has type

$$val\ alwaysEvt : {}'a\ event$$

Nonetheless, we do not concern ourselves a lot with these primitives because they are not directly related to CML's message-passing capabilities; they are enhancements for specific situations.

4.4 Example Program using the Primitives in the π-calculus

Threads in Concurrent ML are created with *spawn*. However, this is naturally modelled by parallel composition in the π-calculus, so we use this operation instead. A minimal (yet synchronizable) code fragment from Appendix A can be written as such with our translation:

$$sync(wrap(recvEvt(addCh), f)) \mid sync(sendEvt(addCh, 2))$$

where $f := \backslash v(v + sum)$, a function that adds its argument to an internal sum. We assume channel $addCh : int\ CH$ has been created using $[\![channel()]\!]$.

This is equivalent to

$(\nu i^{+}_{addCh} : I, i^{-}_{addCh} : \overline{I}, o^{+}_{addCh} : I, o^{-}_{addCH} : \overline{O}, addCh^{+} : int\ CH, addCh^{-} : \overline{int\ CH})\odot_{addCh}$
$\mid (\nu s^{+}_{p}, s^{-}_{p})\boxdot_{s_p} \mid f \leftarrow ((\nu p^{+}, p^{-})\lambda s((p \rightarrow addCh)«x»\ s))$
$\mid (\nu s^{+}_{q}, s^{-}_{q})\boxdot_{s_q} \mid (\nu q^{+}, q^{-}).\lambda s((q \rightarrow \overline{addCh})«x»\ s)$

Which by [App] reduces to

$(\nu i^{+}_{addCh} : I, i^{-}_{addCh} : \overline{I}, o^{+}_{addCh} : I, o^{-}_{addCH} : \overline{O}, addCh^{+} : int\ CH, addCh^{-} : \overline{int\ CH})\odot_{addCh}$
$\mid (\nu s^{+}_{p}, s^{-}_{p})\boxdot_{s_p} \mid f \leftarrow ((\nu p^{+}, p^{-})(p \rightarrow addCh)«s^{-}_{p}»$
$\mid (\nu s^{+}_{q}, s^{-}_{q})\boxdot_{s_q} \mid (\nu q^{+}, q^{-}).(q \rightarrow \overline{addCh})«s^{-}_{q}»$

Synchronization can occur with the given configuration using the transitions in Appendix C. We remember that although we treat it as a state machine, the underlying mechanism is provided by the π-calculus.

In particular, synchronization is possible because the synchronizers are open and only have a point each in their domain, bound to complementary actions. We perform the following reduction operations using the semantics of Appendix C.

1. Grouping elements

$$(p \rightarrow addCH) \mid (q \rightarrow \overline{addCh}) \mid \odot_{addCH} \longrightarrow \heartsuit_p \mid \heartsuit_q \mid \oplus_{addCH} \mid \odot_{addCH}$$

2. Since $p \in \mathrm{dom}(s_p)$

$$\heartsuit_p \mid \boxdot_{s_p} \longrightarrow \checkmark_{s_p}(p) \mid \boxtimes_{s_p}$$

3. Since $q \in \mathrm{dom}(s_q)$

$$\heartsuit_q \mid \boxdot_{s_q} \longrightarrow \checkmark_{s_q}(q) \mid \boxtimes_{s_q}$$

4. by steps 1, 2 and 3

$$\checkmark_{s_p}(p) \mid \checkmark_{s_q}(q) \mid \oplus_{addCh} \longrightarrow happy_{s_p}(p) \mid happy_{s_q}(q)$$

5. Since $s_p(p) = addCH(x)$

$$happy_{s_p}(p) \longrightarrow addCh(x)$$

6. Since $s_q(q) = \overline{addCh}$<2>

$$happy_{s_q}(q) \longrightarrow \overline{addCh}\text{<2>}$$

7. by piping the value of synchronization yielded by step 5 into the respective function f and putting it together with step 6. This is the expected synchronization of the program.

$$f \leftarrow (addCh(x)) \mid \overline{addCh}\text{<2>}$$

8. Then, the processes can communicate as such

$$f \leftarrow (addCh(x)) \mid \overline{addCh}\text{<2>} \longrightarrow f \leftarrow 2 \;\mid\; \mathbf{0}$$

Internally, f and its input $addCh(x)$ live in different processes in Milner's encoding of the lambda-calculus in the π-calculus. Therefore communication over the channel *addCh* can occur, and the passed value can then be given to function f. After this, the left-hand process can carry out its internal computation with the received input.

5 Results

We have managed to represent Chaudhuri CML's synchronization protocol with session types. We embraced a slight modification of the protocol to replace non-determinism with branched execution. Nevertheless, we explained that this is merely a low-level change that does not affect the non-deterministic behavior of the application of selective communication in our programs. With this fine-tuning, we managed to successfully capture the typing of the processes involved with the theory of session types, guaranteeing type safety.

We thus enabled a typed compilation of a program using selective communication in CML style. Then, we explored an example program compiled from our source language and proved that we could indeed derive a state where all processes involved had the right channel types.

The typed compilation then paved our way into translating the CML primitives. As a preliminary, we adjusted the CML channel types to fit the logic of the synchronization protocol and explored event construction in Section 4. The grammar we created enabled us to split the protocol into the CML base-event constructors. The encoding for the *choose* combinator and the *sync* term became apparent almost immediately due to the nature of the source language introduced in Section 2.5.

By further extending our π-calculus to include abstractions and applications, and relying on the expressiveness of the π-calculus to encode the lambda calculus, we also managed to translate the *wrap* and *guard* terms into the π-calculus.

We also mentioned some other primitives that offer superior functionality in specific cases of communication. However, we focused on translating the terms that cover the main tools for message-passing concurrency in Concurrent ML.

Interestingly, other embodiments of session types offer more sophisticated results from typing processes than the one we used, such as [2], which offers deadlock freedom. Given more time, it would be convenient to explore an encoding with such a session type theory.

6 Conclusion

Concurrent ML was one of the early examples that message-passing concurrency could work in a high-level language [14]. Such a model is today represented by the likes of Go [5] and Elixir [24]. As such, there seems to be a tendency towards high-level message-passing languages for communication purposes.

A possible explanation is that this paradigm tends to be more reliable. On the one hand, it is well known that high-level languages abstract computer resources for the programmer, which leads to fewer bugs. Cumbersome and error-prone tasks, such as memory management, are done automatically without the programmer's knowledge.

Moreover, synchronous message passing offers further ease, particularly in reasoning about concurrency. The process of a rendezvous is a distinct procedure. A sending process performs blocked waiting; its state when sending a message and when its message is received is the same. Therefore, synchronous systems rarely suffer from an overflow of buffered messages. Instead, the primary source of failure is deadlock, which is detectable [23]. Thus, the process of debugging in these languages is more straightforward.

These features make of Concurrent ML a suitable language of study in the π-calculus with session types. Both systems embody similar practices [21]: synchronous message passing, dynamic process allocation, and typed channels. Therefore, the investigation of Concurrent ML events is a more transparent endeavor.

This project's goal was to use session types to abstract the synchronization protocol's functionality to gain a deeper understanding of it. Concurrent ML events are a useful subject because of the insight they give in structuring correct communication protocols with concurrency. Distinctly, we saw how the event abstraction permits selective communication with abstract protocols, empowering the programmer to work modularly with concurrency. Our research entails that similar flexibility exists in session type theory.

With the session type encoding of the protocol and the π-calculus translation of the CML primitives, we gain another tool to reason about concurrency in the π-calculus. Namely, it is easier to move from this abstract implementation into concrete examples in other programming languages. Furthermore, the session types oversee the safety of the protocol. We managed to achieve our main objective, but there is still significant room for development from this initial idea.

7 Future Work

Chaudhuri provides a correctness proof for the synchronization protocol, where he observes the progress of states occurring in a program [3]. With our session type encoding, it could be possible to simplify it by looking at the preservation of types in an arbitrary program, which guarantees a well-typed process. Indeed, looking at the preservation of types would be similar to analyzing state transitions. Along the same lines, we could use a richer session type theory (as the one alluded to in the Results section) to bring forth stronger properties of the protocol.

Another possible result, as suggested by Chaudhuri, is that "it may be possible to show alternate encodings of the process-passing primitives of Hopi in pi-like languages, via an intermediate encoding with CML-style primitives" [4]. The Higher-order π-calculus (Hopi) is an extension of the π-calculus that allows the transmission of processes through channels. Historically, several encodings have been produced to express higher-order constructs with the usual, first-order π-calculus [22, 15].

Chaudhuri suggests that it may be possible to describe another encoding using CML's approach, most likely by exploiting the event mechanism. As mentioned before, in Concurrent ML, synchronization protocols are first-class citizens expressed by the *event* construct, which means that we can pass them around. If we find the relationship between *event*-passing and process-passing, we can expose this new encoding. However, we have yet to explore this attractive proposition further.

Finally, Concurrent ML's real value comes from embracing composition in communication protocols akin to how Haskell treats functions. The impressive result of CML's selective communication is building protocols on the fly at run time with the system of events. Indeed, whole branches of concurrent computations are inserted or aborted at run time. With our encoding, it would be interesting to study a possible property of composition in session types.

A Accumulator in CML

```
(*
File: accumulator.sml
Author: Luis Reyes
*)

signature ACC =
sig
   type 'a acc
   val acc : int  -> int acc
   val add : 'a acc * int -> unit
   val sub : 'a acc * int -> unit
   val get : 'a acc -> int
   val stop: 'a acc -> unit
end;

structure Acc :> ACC =
struct
    open CML

    datatype 'a acc = ACC of int chan * int chan * int chan * unit chan

    fun acc sum = let
        val addCh = channel() : int chan
        val subCh = channel() : int chan
        val readCh = channel() : int chan
        val stopCh = channel() : unit chan
        fun loop sum =
            select [
                wrap(recvEvt addCh, fn input => loop (sum+input)),
                wrap(recvEvt subCh, fn input => loop (sum-input)),
                wrap(sendEvt (readCh,sum), fn () => loop (sum)),
                wrap(recvEvt stopCh, fn () => exit())
            ]
    in
        spawn(fn() => loop sum);
        ACC(addCh, subCh, readCh, stopCh)
    end

    fun get (ACC(_,_, readCh, _)) = sync(recvEvt(readCh))
    fun add ((ACC(addCh,_, _, _)), x) = sync(sendEvt(addCh,x))
    fun sub ((ACC(_,subCh,_, _)), x) = sync(sendEvt(subCh,x))
    fun stop ((ACC(_,_,_, stopCh))) = sync(sendEvt(stopCh,()))
end
```

Listing 1: Accum.sml

B Other Type Derivations $\oplus_c$

Let R be the decline reduction and A the accept reduction.

Context: $\Gamma \triangleq cd_p : \overline{CAND}, cd_q : \overline{CAND}, d_p^- : D, d_q^- : D \vdash \oplus_c$

Type derivation p accepted and q rejected

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end, sig_p^- : end \vdash \mathbf{0}}\;\text{T-INACT}}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end, sig_p^- : SIG \vdash sig_p^-\text{<false>}.\mathbf{0}}\;\text{T-OUT}}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- :?\phi, sig_p^- : SIG \vdash d_q().sig_p^-\text{<false>}.\mathbf{0}}\;\text{T-IN} \quad A_q}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : D, sig_p^- : SIG \vdash \quad d_q^- \triangleright \{...\}}\;\text{T-BRANCH}}{cd_p^- : end, cd_q^- : end, d_p^- :?SIG.end, d_q^- : D \vdash d_p(sig_p^-).d_q \triangleright \{...\}}\;\text{T-IN} \quad R_p}{cd_p^- : end, cd_q^- : end, d_p^- : D, d_q^- : D \vdash \oplus_c}\;\text{T-BRANCH}$$

Type derivation p rejected and q accepted

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end, sig_q^- : end \vdash \mathbf{0}}\;\text{T-INACT}}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end, sig_q^- : SIG \vdash sig_q^-\text{<false>}.\mathbf{0}}\;\text{T-OUT}}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- :?SIG.end \vdash d_q^-(sig_q^-).sig_q^-\text{<false>}.\mathbf{0}}\;\text{T-IN} \quad R_q}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : D \vdash d_q^- \triangleright \{...\}}\;\text{T-BRANCH}}{cd_p^- : end, cd_q^- : end, d_p^- :?\phi, d_q^- : D \vdash d_p^-().d_q \triangleright \{...\}}\;\text{T-IN} \quad A_p}{cd_p^- : end, cd_q^- : end, d_p^- : D, d_q^- : D \vdash \oplus_c}\;\text{T-BRANCH}$$

Type derivation p rejected and q rejected

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : end \vdash \mathbf{0}}\;\text{T-INACT}}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- :?\phi \vdash d_q^-().\mathbf{0}}\;\text{T-IN} \quad A_q}{cd_p^- : end, cd_q^- : end, d_p^- : end, d_q^- : D \vdash d_q^- \triangleright \{...\}}\;\text{T-BRANCH}}{cd_p^- : end, cd_q^- : end, d_p^- :?\phi, d_q^- : D \vdash d_p^-().d_q^- \triangleright \{...\}}\;\text{T-IN} \quad A_p}{cd_p^- : end, cd_q^- : end, d_p^- : D, d_q^- : D \vdash \oplus_c}\;\text{T-BRANCH}$$

C Operational Semantics for the Protocol as a State Machine

For the given semantics [4], parallel composition, scope restriction, and inaction are denoted just like in the π-calculus from Section 2. Furthermore, let σ range over states of the machine, p over points, c over channels, and s over synchronizers.

Operational Semantics $\sigma \to \sigma$

(i) $(p \to c) \mid (q \to \overline{c}) \mid \oplus_c \longrightarrow \heartsuit_p \mid \heartsuit_q \mid \oplus_c(p, q) \mid \odot_c$

(ii)
$$\frac{p \in \mathrm{dom}(s)}{\heartsuit_p \mid \boxdot_s \longrightarrow \checkmark_s(p) \mid \boxtimes_s}$$

(iii)
$$\frac{p \in \mathrm{dom}(s)}{\heartsuit_p \mid \boxtimes_s \longrightarrow \times_s(p) \mid \boxtimes_s}$$

(iv) $\checkmark_s(p) \mid \checkmark_{s'}(q) \mid \oplus_c(p, q) \longrightarrow \mathit{happy}_s(p) \mid \mathit{happy}_{s'}(q)$

(v) $\checkmark_s(p) \mid \times_{s'}(q) \mid \oplus_c(p, q) \longrightarrow \mathit{sad}_s$

(vi) $\times_s(p) \mid \checkmark_{s'}(q) \mid \oplus_c(p, q) \longrightarrow \mathit{sad}_{s'}$

(vii) $\times_s(p) \mid \times_{s'}(q) \mid \oplus_c(p, q) \longrightarrow \mathbf{0}$

(viii)
$$\frac{s(p) = \alpha}{\mathit{happy}_s(p) \longrightarrow \alpha}$$

(ix) $\mathit{sad}_s \triangleq (\nu \overrightarrow{p_i^+}, \overrightarrow{p_i^-})(\boxdot_s \mid s)$ where $\mathrm{dom}(s) = \{\overrightarrow{p_i^+}\} \cup \{\overrightarrow{p_i^-}\}$

References

[1] Alen Arslanagic, Jorge A. Pérez, and Erik Voogd. Minimal session types (extended version). *ArXiv*, abs/1906.03836, 2019.

[2] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Domain-aware session types. In *CONCUR*, 2019.

[3] Avik Chaudhuri. Event synchronization by lightweight message passing. 06 2008.

[4] Avik Chaudhuri. A concurrent ml library in concurrent haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, page 269–280, New York, NY, USA, 2009. Association for Computing Machinery.

[5] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

[6] Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. Eager functions as processes. pages 364–373, 07 2018.

[7] Matthew Fluet. A monadic account of first-class synchronous events. 2006.

[8] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42:191–225, 11 2005.

[9] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[10] S. Marlow. *Parallel and Concurrent Programming in Haskell*. Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming. O'Reilly, 2013.

[11] Robin Milner. Functions as processes. In Michael S. Paterson, editor, *Automata, Languages and Programming*, pages 167–180, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[12] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, June 1992.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

[14] Prakash Panangaden and John Reppy. *The Essence of Concurrent ML*, pages 5–29. Springer New York, New York, NY, 1997.

[15] Joachim Parrow. An introduction to the pi-calculus. 2001.

[16] Frederic Peschanski. Parallel computing with the pi-calculus. pages 45–54, 01 2011.

[17] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 295–308, New York, NY, USA, 1996. Association for Computing Machinery.

[18] John Reppy, Claudio Russo, and Yingqi Xiao. Parallel concurrent ml. volume 44, pages 257–268, 08 2009.

[19] John H. Reppy. Higher-order concurrency. In *Higher-order Concurrency*, 1992.

[20] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, USA, 1st edition, 2007.

[21] George Russell. Events in haskell, and how to implement them. volume 36, pages 157–168, 10 2001.

[22] Davide Sangiorgi. From π-calculus to higher-order π-calculus — and back. In M. C. Gaudel and J. P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, pages 151–166, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[23] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th edition, 2014.

[24] Dave Thomas. *Programming Elixir: Functional | Concurrent | Pragmatic | Fun*. Pragmatic Bookshelf, 1st edition, 2014.

[25] Vasco Vasconcelos. Sessions, from types to programming languages. *Bulletin of the European Association for Theoretical Computer Science EATCS*, 103, 01 2011.

[26] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52 – 70, 2012.