MEng Individual Project

Imperial College London

Department of Computing

Multiparty Session-Based Distributed Programming in Scala

Author:
Jia Qing Lim

Supervisor:
Prof. Nobuko Yoshida

Second Marker:
Dr. Iain Phillips

June 10, 2021

Abstract

Scala is a functional programming language with a strong type system and offers a number of libraries that support parallelism through abstractions - making it a favoured language choice for development in distributed system. It gained reputation in the industry over the recent years as it compiles on the well-established JVM, and is the 2nd most popular JVM-based language as of 2021. However, the language does not provide much support against concurrency errors such as deadlocks and race condition.

Effpi is a theoretically established tool-kit embedded in Scala, it provides expressive types that specify the behaviour of concurrent agents, and domain-specific languages(DSLs) for implementing message-passing programs, in a manner similar to the mainstream tool-kits such as Akka and Erlang. On the other hand, Multiparty Session Types (MPST) is a typing discipline for concurrent processes that communicate via message passing. By generating Effpi type accordance to MPST, we can ensure communication safety and protocol conformance in the Scala program. Yet, doing this manually is deemed to be extremely inefficient and error-prone due to the huge amount of communication states and complexity of channels assignment.

In order to solve this issue, we define a mapping from MPST to Effpi types and develop a toolkit based on it. The toolkit takes a global communication protocol as an input and generate a Scala program, consisting of the corresponding local Effpi types and default functions for each participant. With this toolkit, the programmer can easily extend the functionality of the program without worrying about the underlying communication safety. We then extend Effpi type to include error handling to tackle cases where one or more participants may crash suring communication.

We then evaluate the expressiveness and metrics of our tool by applying it to a few protocols from existing example. We then show hour the performance of our tool change in accordance to the complexity and size of the input protocol. Finally, we demonstrate the use case on two real life examples and compare the efficiency with and without using the tool.

Acknowledgements

I would like to thank Prof. Nobuko Yoshida and Dr Adam Barwell for providing me with their invaluable advice and insight throughout the project. I would also like to thank Prof. Alceste Scalas and Fangyi Zhou for helping me with my doubts in Effpi. I extend my gratitude to the members of the Dotty community which have been extremely reponsive to my questions.

I would like to thank all the friends I have made throughout my time at Imperial. In particular, Ye Li and Helen, I am forever grateful for all the support you have provided me over the past four years.

Finally, I would like to thank my family, including Zeyi and Hedwig, for their unconditional love and support throughout my life, and encouraging me to pursue a degree abroad.

Contents

Chapter 1

Introduction

1.1 Motivation

Distributed and concurrent programming has become more popular in recent years, ranging from multiple threads and processes inside the CPU to communications in the client-server model. Multiple tasks running at the same time improves computation model efficiency and scalability, but it also introduces concurrency issues.

The ATM Bank architecture shown in Figure 1 is an example of client server model. The communication starts with the ATM sending the BankServer the user's bank account number, the user can then choose through the ATM, either to *Withdraw* some money or *CheckBalance* of their account and the ATM notifies the BankServer about the choice made. If the user chooses to CheckBalance of their account, the BankServer will respond with the user's remaining balance without further checks. However, if the user chooses to withdraw certain amount, the BankServer will check if the user's balance is sufficient and respond with Confirm or Reject. Even for this simple example, there are few concurrency/ communication issues we need to consider:

- **Deadlocks -** how can we prevent both sides from waiting for each other's response?
- **Communication Mismatch -** how can we make sure that the program executes in accordance to the specification? What if ATM sends accountNum in the form of integer but BankServer is expecting a string?

The majority of concurrent programming languages have little or no support for statically detecting concurrency issues. Some tools have been developed to do this, such as RacerX[7] and RELAY[8] for Java, but not all concurrency issues are detected and these tools don't scale well in accordance to program size[9]. Multiparty session types(MPST)[6] provides an alternative solution by

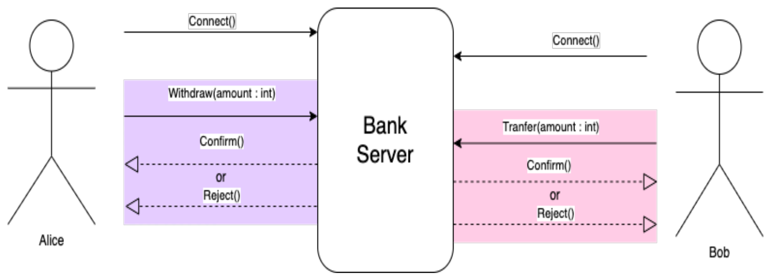


Figure 1: Message-Passing Architecture of ATM Bank Service

reasoning about the communication between multiple processes .It formalises the different communication types in the form of a protocol such that it ensures freedom from concurrency issues.

To ease development in concurrent system, Actor models(e.g. Erlang[2], Akka[3]) and channel-based programming languages(e.g. Go, Scala) have been developed. They adopt units of concurrent execution (actor and channels respectively) which share information through sending and receiving of messages. This allows a simpler reasoning with respective to shared memory concurrency hence being widely used by companies. For example, Erlang is used by Whatsapp for it's messaging server[33], Akka is used by Intel for it's streaming engine[34], Go is used by Uber in most of it's services[32]. However, they don't ensure that the program communicates in accordance to a given communication specification/protocol.

Effpi[25], a functional language built on π-calculus[26], is introduced to tackle this issue. It can specify and verify the intended behaviour of channel-based programming using types. It also provides a simplified actor-based API and supports communication through channels, in a manner similar to the well-established, popular Akka and Erlang.

Scala is a programming language which has become widely used in industry. It's extensive built-in concurrent library and strong type system make it a popular choice for development in distributed system [10]. For example, it is used by Twitter for their backend servers[4] and by Netflix for their streaming server[5]. Despite this, Scala offers little support against concurrency bugs such as deadlocks and communication mismatch we showed above. Since Effpi has been implemented in Dotty as an internal embedded domain-specific language (EDSL), it is a compatible type with Scala and can be used to check the behaviour of a Scala program. By generating Effpi types for a Scala program in accordance to it's session types, we can ensure that the program is free from concurrency issues if it type checked.

Unfortunately, manually generating Effpi types from session types is deemed as highly inefficient and error-prone. This is due to the huge amount of complex channel generations and the combination of various message types needed to be taken into consideration. Much more time has to be invested into implementing the type and debugging rather than writing the function bodies where the main functionality of the program resides. Prior knowledge in Effpi is required for the programmer as well.

1.2 Objectives

The aim of this project is to provide an end-to-end solution for statically verifying and generating implementation of a protocol for Scala. We define a mapping from various communication types in session type to their respective Effpi type. We then implement a code generating tool that generates the local Effpi types and Scala functions for each participant of the protocol. The programmers can use the tools to generate a default implementation for the protocol and guarantee the implementation is free from concurrency issues, hence no runtime checks is required. No prior knowledge of Effpi is required for the programmer and the functionality of the implementation can be easily extended.

1.3 Contributions and Project Structure

The result of this project is an end-to-end solution for generating an Effpi type and a default implementation in Scala for each participant in the protocol. The program generated communicate in accordance to the protocol and is free from deadlocks and communication mismatch.

In Chapter 2, we introduce the typing discipline of session types and Effpi types. Then we present some of the key types and structures supported by the Dotty compiler. We also describe some existing approaches for generating API implementation from session types.

In Chapter 3, we define a mapping from local session types to Effpi types such that the Effpi type describe the same communication as of local session type. We also define a method to assign and match channels for each communication between the participants from their local session types to ensure each message is sent to the correct participant.

In Chapter 4, we discuss the design and implementation of our tool. We describe how codes are generated for both Effpi types and function bodies. Some design features and functionalities have been implemented to assist the developers in modifying the generated code. We also mention the contributions we have made to the Dotty, we spotted some issues in the compiler and raised it to the community.

In Chapter 5, we optimise our program by reducing the number of actual channels and channel instances generated. We define a method to identify the communications between the participants the may share the same channel instance such that the actual channel assigned to it can be reuse.

In Chapter 6, we extend Effpi and our mapping from local session type for error handling, taking into consideration cases where messages are not received and an alternative protocol is used. We explain how our tool is extended to generate program with error handling.

In Chapter 7, we showcase the expressiveness of our tool by running a benchmark on multiple examples from the session type literature. We then evaluate our work through case studies, showing the program generated and how it can be modified without affecting the communication between participants. We also provide a comparison in the time taken to generate the Effpi type and functions for the program, with and without using our tool.

Finally, we conclude in Chapter 8 by summarising our work and outlining possible future improvements to our implementation.

We integrated nuscr[11] and Effpi[12] into our tool[13]. nuScr is an existing OCaml implementation of the framework to generate local session types from a global protocol.

Chapter 2

Background

In this chapter, we introduce and discuss the structures and functionalities of Multiparty Session Types(Section 2.1) and Effpi (Section 2.2), present key constructs of Dotty(Section 2.3) and discuss related works of API development/ Effpi implementations (Section 2.4)

2.1 Multiparty Session Types

Multiparty session type extended binary session type theory to the multiparty level which prevents the occurrence of concurrency bugs such as deadlocks and communication errors[17]. It is expressed in the form of global session type which describe the communication between each participant from a global perspective. Global session types are then projected onto a local session type for each participant, which is used to validate an application through type-checking from a local point of view. We can ensure that each participant communicates with each other in accordance to the global session type if they communicate according to their local session type respectively. The syntax of global session types, local session types, participants of global session types and the mapping from global session types to local session types are defined in Definition 2.1.1, 2.1.2, 2.1.3 and 2.1.4 respectively.

Definition 2.1.1 (Sorts and Global Session Types)**.** *Sorts, ranged over S, is defined as:*

$$S ::= \ \texttt{int} \ | \ \texttt{nat} \ | \ \texttt{bool} \ | \ \texttt{string} \ | \ \ldots$$

Global Session Types, ranged over by G, are defined as:

$$\begin{array}{rll} G ::= & \texttt{end} & \textit{Termination} \\ & | \ t & \textit{Type Variable} \\ & | \ \mu \mathbf{t}.G & \textit{Recursive Type} \\ & | \ \mathsf{p} \rightarrow \mathsf{q} \, \{l_i(S_i) : G_i\}_{i \in I} & \textit{Message/ Branching} \end{array}$$

where:

- *p and q are the participants/parties and* $p \neq q$
- $I \neq \emptyset$ *and* $\forall i, j \in I\{i \neq j \implies l_i \neq l_j\}$
- *Recursion is guarded and global types are closed unless specified:* t *only bounded by* $\mu\mathbf{t}$.

Definition 2.1.2 (Local Session Types)**.** *Grammar for Local Session Types, ranged over by* T*:*

$$\begin{array}{rlr} T ::= & \texttt{end} & \textit{Termination} \\ | & t & \textit{Type Variable} \\ | & \mu\mathbf{t}.T & \textit{Recursive Type} \\ | & \oplus_{i\in I}\mathsf{q}!l_i(S_i).T_i & \textit{Message Sending/ Selection} \\ | & \&_{i\in I}\mathsf{p}?l_i(S_i).T_i & \textit{Message Receiving/ Branching} \end{array}$$

where:

- *p and q are the participants/parties*
- $\forall i, j \in I\{i \neq j \implies l_i \neq l_j\}$
- *Recursion is guarded and session types are closed unless specified*

Definition 2.1.3 (Participants of Global Session Types)**.** *We define the set of participants of a global session type* G*,* $pt\{G\}$ *by structural induction on* G*, such that:*

$$\begin{array}{rcl} pt\{\texttt{end}\} & = & \emptyset \\ pt\{t\} & = & \emptyset \\ pt\{\mu\mathbf{t}.G\} & = & pt\{G\} \\ pt\{\mathsf{p} \to \mathsf{q}\ \{l_i(S_i) : G_i\}_{i\in I}\} & = & \{\mathsf{p}, \mathsf{q}\} \cup \bigcup_{i\in I} pt\{G_i\} \end{array}$$

Definition 2.1.4 (Mapping from Global to Local Session Type With Full Merging)**.** *We define a projection of a global type onto participant* r*, with full merging operator* $\sqcap$*, defined by recursion on* G*:*

$$\begin{array}{rcl} \texttt{end} \upharpoonright \mathsf{r} & = & \texttt{end} \\ t \upharpoonright \mathsf{r} & = & t \\ (\mu\mathbf{t}.G) \upharpoonright \mathsf{r} & = & \begin{cases} \mu\mathbf{t}.G \upharpoonright \mathsf{r} & \mathsf{r} \in pt\{G\} \\ \texttt{end} & \mathsf{r} \notin pt\{G\} \end{cases} \end{array}$$

$$\mathsf{p} \to \mathsf{q}\,\{l_i(S_i) : G_i\}_{i\in I} \upharpoonright \mathsf{r} \quad = \quad \begin{cases} \oplus_{i\in I}\mathsf{q}!l_i(S_i).G_i \upharpoonright \mathsf{r} & \mathsf{r} = \mathsf{p} \\ \&_{i\in I}\mathsf{p}?l_i(S_i).G_i \upharpoonright \mathsf{r} & \mathsf{r} = \mathsf{q} \\ \sqcap_{i\in I}(G_i \upharpoonright \mathsf{r}) & \mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\} \end{cases}$$

$$undefined\ otherwise$$

The merging operator, $\sqcap$ *is a partial operation over session types, defined as:*

$$T_1 \sqcap T_2 \quad = \quad \begin{cases} T_1 & T_1 = T_2 \\ T_3 & \exists I, J : \begin{cases} T_1 = \&_{i\in I}\mathsf{p'}?l_i(S_i).T_i & \wedge \\ T_2 = \&_{j\in J}\mathsf{p'}?l_j(S_j).T_j & \wedge \\ T_3 = \&_{k\in I\cup J}\mathsf{p'}?l_k(S_k).T_k \end{cases} \\ undefined\ \ otherwise \end{cases}$$

The merging operators only allows us to combine different external choices if and only if common labels have identical sorts and continuations:

$$\begin{aligned} T_1 &= \&_{i\in I}\mathsf{p'}?l_i(S_i).T_i \\ T_2 &= \&_{j\in J}\mathsf{p''}?l_j(S_j).T_j \\ T_1 \sqcap T_2\ \ defined \quad &\iff \quad \mathsf{p'} = \mathsf{p''} \wedge \forall i \in I, j \in J(l_i = l_j \implies S_i = S_j \wedge T_i = T_j) \end{aligned}$$

In full merging, if G starts with a communication between **p** and **q**, when we project G to a third participant **r**, we allow **r** to receive different message types from a same participant **p'** in different branches of a global type.

2.2 Effpi

Concurrent and distributed programming is very complex and prone to error. Over the last decades, mainstream programming languages did not adopt channel-based models but supported concurrency through multi-threading and external libraries. In recent years, channel-based languages and toolkits have been introduced to mainstream programming, such as:

Akka a library that provides an Actor Class that has a built in receive function (that can be further implemented) to receive messages and an ActorRef Class to send messages to a specific Actor.[1]

Go Language Multiple processes can be executed concurrently and communicate with each other through specific channels.[14]

Even though they simulate message passing in concurrent/ distribute programming, they don't ensure that the behaviour of the program follows certain specifications, which will cause communication issues such as communication mismatches and deadlocks. Huge efforts have been dedicated by researchers to understand the behaviour of channel-based communications, e.g. Hoare's

CSP[20], Milner's CCS[21] and π-calculus[22]. Some verification methods for Go have also been developed: a static analyser that generates a global communication graph from local session types[23], a verification framework that infers a communication model from a program's behaviour[24].

The idea is to develop a type that can work on all channel-based languages which ensures that, if a program type-checks and compiles, it will run as the behaviour specified. A concurrent functional language, **Effpi** $\lambda^{\pi}_{\leq}$[25] is formalised from behavioural type (π-calculus) and dependent function types (Dotty) such that it verifies the safety and liveness of the program while accurately validates the channel passings and high order interactions.

2.2.1 Effpi Calculus

The syntax of Effpi Calculus is defined as Figure 3 below. We can observe that it extended and modified the variant introduced by Yoshida in [26].
Effpi's operational semantics model the interactions between channel-based processes. We highlight some added/ modified calculus alongside their reduction rules below.

- **Termination** has the syntax **end** and represent inactivity
- **Channel Creation** is a modified syntax from restriction (ν a) with the same functionality. It creates a fresh channel instance from $\mathbb{C}$ (all elements are generated during runtime and cannot be determined by programmers):

$$\frac{\texttt{a}\ \mathit{fresh}}{\mathbf{chan}() \implies \texttt{a}}\ \mathbb{R}\text{-}\mathbf{chan}()$$

- **Conditionals** is represented by the syntax `if` t `then` t_1 `else` t_2. The term will reduce to t_1 or t_2 based on the value of t:

$$\frac{}{\mathbf{if\ tt\ then}\ t_1\ \mathbf{else}\ t_2 \implies t_1}\ \mathbb{R}\text{-}\mathbf{if\text{-}tt}$$

$$\frac{}{\mathbf{if\ ff\ then}\ t_1\ \mathbf{else}\ t_2 \implies t_2}\ \mathbb{R}\text{-}\mathbf{if\text{-}ff}$$

- **Function/ Function Application**: Functions are represented in the form $\lambda\mathtt{x}.t$ where it takes a parameter $\mathtt{x}$ and has a body of term t. We can write it as $\lambda\_.t$ if $\mathtt{x}$ is not a free variable of t ($\mathtt{x} \notin t$) based on definition of free variable in [26]. Function Application takes a value and pass into the function as a parameter, the reduction is shown as below:

$$\frac{}{(\lambda\mathtt{x}.t)v \implies t\{v/\mathtt{x}\}}\ \mathbb{R}-\lambda$$

- **Sending/ Receiving Messages**: $\mathtt{send}(t, t', t'')$ sends a message t' through channel t and continue as term t''. $\mathtt{recv}(t, t')$ receives a message from

$$\frac{v \notin \mathbb{B}}{\neg v \implies \textbf{err}} \qquad \frac{u \notin \{\lambda \mathtt{x}.t \mid \mathtt{x} \in \mathbb{X}, t \in \mathbb{T}\}}{u\ v \implies \textbf{err}} \qquad \frac{v \notin \mathbb{B}}{\mathtt{if}\ v\ \mathtt{then}\ t'\ \mathtt{else}\ t'' \implies \textbf{err}}$$

$$\frac{t \in \mathbb{V}}{t \parallel t' \implies \textbf{err}} \qquad \frac{u \notin \mathbb{C}}{\mathtt{recv}(u, v) \implies \textbf{err}} \qquad \frac{u \notin \mathbb{C}}{\mathtt{send}(u, v_1, v_2) \implies \textbf{err}}$$

Figure 2: Error rules in Effpi semantics

channel t and continue as term t'. `send` and `recv` can interact if they are running in parallel and are using the same channel:

$$\frac{}{\mathtt{send}(\mathtt{a}, u, v_1) \parallel \mathtt{recv}(\mathtt{a}, v_2) \implies v_1\ ()\parallel v_2\ u}\ \mathbb{R} - C_{OMM}$$

- **Binding** is represented as `let` $x = t$ `in` t', it replaces all free occurrences of x in t' with t. It can be used to simulate recursion by having x as a free and guarded variable in t and t'.
- **Parallel Composition** follows the syntax and semantics as of [26]
- **Error Detection** Effpi calculus provides us with a functionality to detect programming error by including an **err** syntax such that we can verify our process t is safe iff $\forall t' : t \implies {}^* t', t' \neq \textbf{err}$. We check the existent of errors based on following rules: (1) Using a non function for function application. (2)Receiving/ Sending message through a term that is not a channel.(3) Negating non boolean values. (4) Using non boolean values as condition. (5)Having a value in parallel composition. These rules are formalised in Figure 2.

2.2.2 Effpi Type System

We now go through the type system of Effpi, which is defined in Figure 4

Although type system in Effpi is the reminiscent of the simple λ-calculus, a lot of new types have been added. For every newly added type/ calculus, we will go through the type assignment (assign a process to a type under certain environment), type validation and subtyping. For convention purpose, there are some notation that we use: (1) $\vdash \Gamma$ `env` indicates Γ is a valid typing environment. (2) $\Gamma \vdash T$ `type` indicates T is a valid type in Γ. (3)$\Gamma \vdash T\ \pi -$ `type` indicates T is a valid process type in Γ.(4) $\Gamma \vdash T \leq U$ implies T is a subtype of U in Γ, if $\Gamma \vdash T, U^*$-`type`.

- **Process** is the top type among π-type, see $\leq$-PROC.

$$\frac{}{\Gamma \vdash T \leq \textbf{proc}}\ \leq\text{-PROC}$$

- **Union Type** follows the introduction of conditional process such that we allow it to reduce to processes of different types t-**if**. If two types(process types resp.) are valid in a given typing environment, Γ, the union of both types(process types resp.) is also valid type(process type resp.) in Γ as

$\mathbb{B} ::=$	$\{\texttt{tt}\ ,\ \texttt{ff}\}$	Booleans
$\mathbb{C} ::=$	$\{\texttt{a}\ ,\ \texttt{b}\ ,\ \texttt{c}\ ,\ \ldots\}$	Channel Instances
$\mathbb{X} ::=$	$\{\texttt{x}\ ,\ \texttt{y}\ ,\ \texttt{z}\ ,\ \ldots\}$	Variables
$\mathbb{V} \ni u, v, \ldots ::=$		Values
	$\mathbb{B} \mid \mathbb{C}$	Booleans/ Channel Instances
$\mid$	$\lambda\texttt{x}.t$	Function
$\mid$	$()$	Unit
$\mid$	`err`	Error
$\mathbb{P} \ni p, q, \ldots ::=$		Processes
	`end`	Termination
$\mid$	$\texttt{send}(t, t', t'')$	Message Sending
$\mid$	$\texttt{recv}(t, t')$	Message Receiving
$\mid$	$t \parallel t'$	Parallel Composition
$\mathbb{T} \ni t, t', \ldots ::=$		Terms
	$\mathbb{X} \mid \mathbb{V} \mid \mathbb{P}$	Variables/ Values/ Processes
$\mid$	$\neg t$	Negation
$\mid$	$\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	Conditional
$\mid$	$\texttt{let}\ x = t\ \texttt{in}\ t'$	Binding
$\mid$	$t\ t'$	Function Application
$\mid$	`chan()`	Channel Creation

Figure 3: Syntax of Effpi $\lambda^{\pi}_{\leq}$ terms

$S, T, U, \ldots ::=$	bool	Boolean
	$\mid\ ()$	Unit
	$\mid\ \top \mid \bot$	Top/Bottom
	$\mid\ T \vee U$	Union
	$\mid\ \mu\underline{x}.T$	Recursion
	$\mid\ \prod(\underline{x} : U)T$	Function
	$\mid\ \underline{x}$	Variable
	$\mid\ c^{io}[T] \mid c^{i}[T] \mid c^{o}[T]$	Channel
	$\mid\ \mu\underline{x}.T$	Equi-Recursive
	$\mid$ **process**	Process
	$\mid$ **nil**	Terminated Process
	$\mid\ \mathbf{o}[S, T, U] \mid \mathbf{i}[S, T]$	Output/Input
	$\mid\ \mathbf{p}[T, U]$	Parallel Composition

Figure 4: Types of Effpi $\lambda^{\pi}_{\leq}$

shown in $T - \vee$ and $\pi - \vee$. Subtyping rules are straightforward as presented in $\leq$-$\vee$L and $\leq$-$\vee$R.

$$\frac{\Gamma \vdash T \vee U^* \ \mathtt{type} \quad \Gamma \vdash t : \mathit{bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : U}{\Gamma \vdash \textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \ : \ T \ \vee U}\ t\text{-}\textbf{if}$$

$$\frac{\Gamma \vdash T\ \mathtt{type} \quad \Gamma \vdash U\ \mathtt{type}}{\Gamma \vdash T \vee U\ \mathtt{type}}\ T\text{-}\vee \qquad \frac{\Gamma \vdash T\ \pi\text{-}\mathtt{type} \quad \Gamma \vdash U\ \pi\text{-}\mathtt{type}}{\Gamma \vdash T \vee U\ \pi\text{-}\mathtt{type}}\ \pi\text{-}\vee$$

$$\frac{\Gamma \vdash T \leq S \quad \Gamma \vdash U \leq S}{\Gamma \vdash T \vee U \leq S}\ \leq\text{-}\vee\mathrm{L} \qquad \frac{\Gamma \vdash S \leq T}{\Gamma \vdash S \leq T \vee U}\ \leq\text{-}\vee\mathrm{R}$$

- **Function Type** is applied to functions and identifies the type of continuation term, assuming an input parameter of certain type t-λ. We allow the continuation term to be π-typed T-$\prod$ or typed Tπ-$\prod$. If a function type is a subtype of another function type, they must share the input parameter of the same type $\leq$-$\prod$. In other hand, function type is also essential when assigning type for function application t-APP. Note that we can replace our parameter type with it's subtype.

$$\frac{\Gamma, x : T \vdash U\ \mathtt{type}}{\Gamma \vdash \prod(\underline{x} : T)U\ \mathtt{type}}\ T\text{-}\prod \qquad \frac{\Gamma, x : T \vdash U\ \pi\text{-}\mathtt{type}}{\Gamma \vdash \prod(\underline{x} : T)U\ \mathtt{type}}\ \mathrm{T}\pi\text{-}\prod$$

$$\frac{\Gamma, x : T \vdash U \leq U'}{\Gamma \vdash \prod(\underline{x} : T)U \leq \prod(\underline{x} : T)U'}\ \leq\text{-}\prod \qquad \frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x^U.t : \prod(\underline{x} : U)T}\ t\text{-}\lambda$$

$$\frac{\Gamma \vdash t_1 : \prod(\underline{x} : U)T \quad \Gamma \vdash t_2 : U' \quad \Gamma \vdash U' \leq U}{\Gamma \vdash t_1 t_2 : T\{U'/\underline{x}\}}\ t\text{-}\mathtt{APP}$$

- **Channel Type** covers three different types: input channel, output channel and input output channel, it also specifies the message type used. Channel creation/ channel instances can be assigned an input output channel type through rules t-c, t-**chan** and T-c. The channel type can be later assigned to solely input/ output channel type based on rule $\leq$-c and the general subtyping rule. Subtyping is covariant for inputs and contravariant for outputs [27] such that higher type is more restricted.

$$\frac{\Gamma \vdash \mathtt{c}^{\mathtt{io}}[T]\ \mathtt{type}}{\Gamma \vdash \mathtt{a}^T : \mathtt{c}^{\mathtt{io}}[T]}\ t\text{-}\mathtt{C} \qquad \frac{\Gamma \vdash \mathtt{c}^{\mathtt{io}}[T]\ \mathtt{type}}{\Gamma \vdash \textbf{chan()}^T : \mathtt{c}^{\mathtt{io}}[T]}\ t\text{-}\textbf{chan}$$

$$\frac{\Gamma \vdash T\ \mathtt{type}}{\Gamma \vdash \mathtt{c}^{\mathtt{io}}[T]\ \mathtt{type} \quad \Gamma \vdash \mathtt{c}^{\mathtt{i}}[T]\ \mathtt{type} \quad \Gamma \vdash \mathtt{c}^{\mathtt{o}}[T]\ \mathtt{type}}\ T\text{-}\mathtt{C}$$

$$\frac{\Gamma \vdash T \leq T'}{\Gamma \vdash \mathtt{c}^{\mathtt{io}}[T] \leq \mathtt{c}^{\mathtt{i}}[T'] \quad \Gamma \vdash \mathtt{c}^{\mathtt{i}}[T] \leq \mathtt{c}^{\mathtt{i}}[T'] \quad \Gamma \vdash \mathtt{c}^{\mathtt{io}}[T'] \leq \mathtt{c}^{\mathtt{o}}[T] \quad \Gamma \vdash \mathtt{c}^{\mathtt{o}}[T'] \leq \mathtt{c}^{\mathtt{o}}[T]}\ \leq\text{-}\mathtt{C}$$

- **Input/Output Type** can be cast on message sending and receiving process according to rules t-**send** and t-**recv** . Both type specify the channel type used and the continuation type but output type also specifies the type of message sent as of rule π-**o** and π-**i**. We can only validate them as a process instead of the general type. The subtyping is straightforward and shown in rules $\leq$-**o** and $\leq$-**i**.

$$\frac{\Gamma \vdash \mathbf{o}[S,T,U]\ \pi\texttt{-type} \quad \Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : U}{\Gamma \vdash \mathbf{send}(t_1,t_2,t_3) : \mathbf{o}[S,T,U]}\ t\text{-}\mathbf{send}$$

$$\frac{\Gamma \vdash \mathbf{i}[S,T]\ \pi\texttt{-type} \quad \Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathbf{recv}(t_1,t_2) : \mathbf{i}[S,T]}\ t\text{-}\mathbf{recv}$$

$$\frac{\Gamma \vdash S \leq \mathtt{c}^{\mathtt{o}}[T_o] \quad \Gamma \vdash T \leq T_o \quad \Gamma \vdash U\ \pi\texttt{-type}}{\Gamma \vdash \mathbf{o}[S,T,\prod()U]\ \pi\texttt{-type}}\ \pi\text{-}\mathbf{o}$$

$$\frac{\Gamma \vdash S \leq \mathtt{c}^{\mathtt{i}}[T_i] \quad \Gamma \vdash T_i \leq T \quad \Gamma, x : T \vdash U\ \pi\texttt{-type}}{\Gamma \vdash \mathbf{i}[S,\prod(\underline{x} : T)U]\ \pi\texttt{-type}}\ \pi\text{-}\mathbf{i}$$

$$\frac{\Gamma \vdash S \leq S' \quad \Gamma \vdash T \leq T' \quad \Gamma \vdash U \leq U'}{\Gamma \vdash \mathbf{o}[S,T,U] \leq \mathbf{o}[S',T',U']}\ \leq\text{-}\mathbf{o}$$

$$\frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash U \leq U'}{\Gamma \vdash \mathbf{i}[T,U] \leq \mathbf{i}[T',U']}\ \leq\text{-}\mathbf{i}$$

- **Equi-Recursive Type** can be cast to recursive binding process as discussed above. However, there's no direct rule that assign recursive binding process to recursive type. Type assignment for binding type is given in rule $\leq$-**let** and type validation rules for equi-recursive type is given in T-μ and π-μ. We can then expand the recursive type alongside coinduction on general subtyping rules to assign the equi-recursive type to the process.

$$\frac{\Gamma, x : U \vdash t : U' \quad \Gamma, x : U \vdash t' : T \quad \Gamma \vdash U' \leq U}{\Gamma \vdash \mathbf{let}\ x^U = t\ \mathbf{in}\ t' : T\{U'/\underline{x}\}}\ \leq\text{-}\mathbf{let}$$

$$\frac{\Gamma, x : \top \vdash T\ \texttt{type} \quad \underline{x} \notin \texttt{fv}(T) \quad T \notin \{U \mid \exists U', \underline{z} \in \mathbb{X} : U \equiv U' \vee \underline{z}\}}{\Gamma \vdash \mu\underline{x}.T\ \texttt{type}}\ T\text{-}\mu$$

$$\frac{\Gamma, x : \top \vdash T\ \pi\texttt{-type} \quad \underline{x} \notin \texttt{fv}(T) \quad T \notin \{U \mid \exists U', \underline{z} \in \mathbb{X} : U \equiv U' \vee \underline{z}\}}{\Gamma \vdash \mu\underline{x}.T\ \pi\texttt{-type}}\ \pi\text{-}\mu$$

Note that the error term **err** is not defined(untypable) in the type system. $\Gamma \vdash t : T$ and $t \implies t'$ implies $\exists T'.(\Gamma \vdash t' : T')$. In other words, typed terms will only reduced to typed typed terms, if $\Gamma \vdash t : T$, then t is safe.

2.3 Dotty

Dependent Object Type (DOT)[28] is a new type-theoretic foundation of Scala language. It is a core calculus for Scala's path-dependent types, abstract types and mixture of nominal and structural typing through usage of refinement types. DOT normalizes Scala's type system by unifying the constructs for type members and by providing classical intersection and union types which we will further discuss.

In Scala 3, DOT has been implemented as a foundation of Scala and Dotty is a new open source compiler which work with Scala 3. The main objectives of Dotty according to the official Dotty docs [15] are (1)simplify when possible. (2) eliminate inconsistencies and surprising behaviors. (3) build on strong foundations to ensure design hangs well together. (4) consolidate language constructs. (5) become more opinionated.

2.3.1 New Types in Dotty

We shall go through a few new types supported by Dotty, as mentioned in [18]

Intersection Types

Intersection is represented by the **&** operator. Intersection of two types, **S** and **T** can be written as **S & T** or **T & S** (commutative). Any members of an intersection type **S & T** are all the members of **S** and all the members of **T**. It's a modification of compound type **with** from Scala 2 but commutative.

Union Types

Union is represented by the | operator. Similar to Intersection Types, | is commutative such that the union of two types, **A** and **B** can be written as **A** | **B** or **B** | **A**. A member of union type **A** | **B** has as all values of type **A** and also all values of type **B**. In other words, the member can either be type **A** or type **B** at one time. Type matching can be use to check the actual type of the member during runtime. It has a lower precedence than Intersection Types.

Match Types

Type matching is one of the key constructs of Dotty. In Scala 2, pattern matching is only applicable to objects but not types. In Figure 5, we demonstrate a simple use case of type matching in the Two Buyers example. **X** is the response type from the buyer, which is the Union type **Buy** | **Cancel**. This means that the buyer can either be of type **Buy** (proceed with the purchase) or **Cancel** (cancel transaction). By using type matching, we can assign different type to our Seller's behavior based on the response type of the buyer. If **X** is of type **Buy**, the seller will have a type of sending a message of type **Confirm** through a channel of type **CConf**. Otherwise, it will has the null process type(**PNil**), which means do nothing.

```
type SellerMatch[X <: Buy|Cancel, CConf <: OutChannel[Confirm]]
<: Process = X match {
    case Buy => Out[CConf, Confirm]
    case Cancel => PNil
  }
```

Figure 5: SellerMatch Type after received response from Buyer

```

type ReceiveRequest[CReq <: InChannel[GET|Terminate],
                    CData <: OutChannel[Data]] =
                In[CReq, GET|Terminate, (x: GET|Terminate) =>
                DataServer[x.type, CReq, CData]]

type DataServer[X <: GET|Terminate,
      CReq <: InChannel[GET|Terminate], CData <: OutChannel[Data]]
<: Process = X match {
    case GET => Out[CData, Data] >>: ReceiveRequest[CReq, CData]
    case Terminate => PNil
  }
```

Figure 6: Request/Data Server Type

Type matching can also be recursive as shown in Figure 6. The server have type **ReceiveRequest** which will receive a request from the client of type **GET** or **Terminate**. It will send message of type **Data** to the client upon each receiving of **GET** type request and continue to await for request again until it receives a request of type **Terminate**.

Other Types

Type lambdas is used with the operator =>> and allow us to express a higher-kinded type directly without type definition. **Dependent Function Types** extended the dependent methods functionality from Scala 2 such we are able to turn dependent methods into function values, so that they can be passed as parameters to other functions or returned as results. Similarly, **Polymorphic Function Types** allows polymorphic functions (functions that take types as parameters) to be passed as parameters to other functions or returned as results. More details of these types can be found on Dotty's new types doc [18] and worked examples can be found on github [19].

2.4 Related Works

We shall demonstrate below related works that developed a toolchain based on session types. The toolchains can be used to generate API which contributes to real world development.

2.4.1 Scribble and Endpoint API

Scribble[29, 31] is a toolchain based on multiparty session types for distributed programming in Java. The methodology is based on the generation of protocol-specific Endpoint APIs from Scribble specifications. The stages of the Scribble toolchain is shown as below:

1. **Global Protocol Input** - The tool takes a primary input a textual description of the source protocol from a global perspective.

2. **Protocol Validation** - The tool validates the well-formedness of the global protocol. Besides the basic syntactic checks, bound role names and recursion variables, the tool also checks key conditions: role enabling, consistent external choice subjects and reachability.

3. **Endpoint FSM generation** - Endpoint Finite State Machine (EFSM) is generated for each role in the protocol. The construction is based on and extends the syntactic projection of global types to local types then translate into an EFSM. Each nodes of the EFSM represents the states in the localised view of the protocol for a target role. The arrow from one node pointing to another indicates the communication actions performed by the role between states.

4. **Endpoint API Generation** - The toolchain generates an Endpoint API for each role based on its EFSM. The generated API is consisted of two main components, Session API and State Channel API, which will be explained as below:

    - **Session API** - The frontend class of Session API is a final subclass that house the family of protocol-specific constants for type directed session programming in Java. Session API also reify abstract names(role names/ message labels) as singleton types following a singleton pattern. For each role and message operator in the source protocol, a final Java class with the same name is generated. This class has a single private constructor along a public static final field of the same type and same name as the role that forms the singleton instance of this class. In the frontend class of Session API, a public static final field of the same type and name as the role is initialised to the constant declared in the final class above.
    - **State Channel API** - This API captures the protocol-specific behaviour of a role in the global protocol as represented by its EFSM,

via static typing in Java. Each state in EFSM is implemented as a Java Class for a state-specific session channel. Each channel class only supports I/O methods that are permitted according to the EFSM state. The return type of each I/O method is the channel type for the successor state following transition from current state.

2.4.2 Scala APIs for multiparty sessions

An encoding[30] of a full-fledged multiparty session π-calculus into linear π-calculus is introduced to provide a theoretical foundation for concurrent and distributed programming. We shall not go into details about the encoding, we discuss the implementation of a practical toolchain for safe multiparty programming in Scala based on the encoding.

The main feature of the API generation is to exploit type safety and distribution provided by an existing library for binary session channels and then treat the ordering of communications across separate channels.

lchannels The toolchain utilised lchannels library which provides two important classes, `OUT[T]` and `IN[T]`. These two classes represent the sending/ receiving endpoints in the encoding and enforce the typing of I/O action via static Scala typing. lchannels is used as it promotes session type-safety through an encoding of binary session types that is close to our encoding. lchannels passes messages over multiple mediums such as local memory, TCP sockets and Actors. `IN[T]` and `OUT[T]` instances for binary session delegation can be sent through lchannels or sent remotely through distributed message transport. Hence we can obtain type-safe distributed multiparty delegation of a typed channel tuple.

Multiparty API Although we can ensure communication safety of a process based on the π-calculus labelled tuple type yielded by it's type encoding, the process does not convey any ordering to communication across channels. We can refine the class such that each multiparty channel class exposes a `send()` or `receive()` method based on the I/O action expected by the multiparty session type. We can also implement methods that uses binary channels based on our process encoding. Note that `send()` and `receive()` method are mechanical and can be automatically generated by extending Scribble.

Scribble-Scala Toolchain Scribble in Section 2.4.1 is extended to support the encoding. Scribble is augmented with a projection operator @ such that it can support the protocols. It then computes the projections/ encodings and then automates the Scala API generation.

Chapter 3

Projection from Session Types to Effpi Types

In this chapter we shall define a mapping from local session types to local Effpi types in Section 3.1. In Section 3.2, we proceed to to formalise a theory on channel generation and matching between the participants of the program to ensure each communication action is carried out through the right channels.

3.1 Local Session Types to Local Effpi Types

Since we had our local session types projected from a global session type, we can make sure the program communicates in accordance to a global protocol if each participant communicates in accordance to their assigned local session type. Effpi type can effectively specify the sequence of communication actions taken by a Scala program, hence we generate a local Effpi type for each participant in the program based on their local session type such that it represents the same sequence of communication actions per stated in the local session type. By having each participant in the program type-checked with their assigned local Effpi type, we can make sure the Scala program communicates consistently as the global protocol and hence prevents the occurrence of concurrency bugs such as deadlocks and communication mismatch during the execution of the program.

In this section, we define a projection from local session types to local Effpi types. Recall that the local session type, T of a participant was defined earlier in Definition 2.1.2. Again we assumed that the local session type is projected from a well established global session type as per defined in Definition 2.1.4, we can make sure that the local session types are well defined and mergeable such that for every send action of a participant, there will be another participant waiting to receive the message. We can also combine this projection with the projection from global session type to local session type defined in Definition 2.1.4 to define a projection from global session type to Effpi type.

Definition 3.1.1 (Projection from Local Session types to Local Effpi Types). *We define the local Effpi type for a participant with local session type T, et(T), by structural induction on T, as follows:*

$$\begin{aligned}
et(\texttt{end}) &= \boldsymbol{nil} \\
et(t) &= t \\
et(\mu \mathbf{t}.T) &= \mu \mathbf{t}.et(T) \\
et(\oplus_{i \in I} \mathsf{q}!l_i(S_i).T_i) &= \bigvee_{i \in I} \boldsymbol{o}[c^o[\vee_{j \in I} l_j], l_i, \Pi() et(T_i)] \\
et(\&_{i \in I} \mathsf{p}?l_i(S_i).T_i) &=
\end{aligned}$$

$$\boldsymbol{i}[c^i[\vee_{i \in I} l_i], \Pi(\underline{x} : \vee_{i \in I} l_i)\ \underline{\boldsymbol{match}}\ \underline{x}\ \{l_i \Rightarrow et(T_i)\}_{i \in I}]$$

where $\underline{x}$ is a fresh variable

The projection from local session types to local Effpi types is defined in Definition 3.1.1. We assume that the set of message labels within session type is the subset that as of Effpi's. We also assume that the message labels are unique within each program such that we can't override them (no two message labels have the same name with different payloads). Hence on the type level, we can decide the continuation of the communication by looking only at the label of the message sent/received. On the function/ implementation level, the compiler will check if the message sent has the correct payload, any incorrect number of payloads/ payload type will lead to compilation error.

For each communication action, there is a fixed set of message labels allowed, sender can choose which message label to send while receiver is expected to receive a message where it's label is within the set. Hence, we need a channel for that particular communication action that can transfer message of all possible labels in the set. The channel is further cast into an output channel instance for the sender and an input channel instance for the receiver, this ensures that the receiver doesn't send through the channel and vice versa, any violation shall be detected on compilation stage. Also, since the sender knows the exact message label they are sending through the channel, they can proceed directly with the continuation for that message label, receiving message however, only ensures the receiver that the label of the message received is within the set, further type matching is required in order to identify the actual label of the message for the receiver to know which continuation to proceed with. This idea can be showcase clearly in the Bank Audit example. This example consisted of three participants: **Bank**, **Audit** and **User**. Their local session types are denoted as T_{Bank}, T_{Audit} and T_{User} respectively and are shown as below:

$$\begin{aligned}
T_{Bank} = \mathsf{Audit}?Record(int).\mathsf{User}!Success().end \\
\&\mathsf{Audit}?Cancel().\mathsf{User}!Failure().end
\end{aligned}$$

$$T_{Audit} = \textsf{Bank}!Record(int) \oplus \textsf{Bank}!Cancel()$$

$$T_{User} = \textsf{Bank}?Success(int) \& \textsf{Bank}?Failure()$$

The **Audit** decides if a payment is successful, if it is, it will request the **Bank** to Record the amount of type int and the **Bank** will notify the **User** the Success of the payment, otherwise the **Audit** will Cancel the payment with the **Bank** and the **Bank** will inform the **User** the Failure of the payment. The correspond local Effpi type (projected from their local session type) for **Bank**, **Audit** and **User**are denoted as $et(T_{Bank})$, $et(T_{Audit})$ and $et(T_{User})$ respectively and are shown below:

$$et(T_{Bank}) =$$
$$\mathbf{i}[c^i[Record \vee Cancel], \Pi(\underline{x} : Record \vee Cancel)$$
$$\underline{\mathbf{match}}\ \underline{x} \left\{ \begin{array}{l} Record \Rightarrow \mathbf{o}[c^o[Success], Success, \Pi()\mathbf{nil}] \\ Cancel \Rightarrow \mathbf{o}[c^o[Failure], Failure, \Pi()\mathbf{nil}] \end{array} \right\}]$$

$$\begin{aligned} et(T_{Audit}) &= \mathbf{o}[c^o[Record \vee Cancel], Record, \Pi()\mathbf{nil}] \\ &\quad \vee \mathbf{o}[c^o[Record \vee Cancel], Cancel, \Pi()\mathbf{nil}] \end{aligned}$$

$$et(T_{User}) = \mathbf{i}[c^i[Success \vee Failure], \Pi(\underline{x} : Success \vee Failure)$$
$$\underline{\mathbf{match}}\ \underline{x} \left\{ \begin{array}{l} Success \Rightarrow \mathbf{nil} \\ Failure \Rightarrow \mathbf{nil} \end{array} \right\}]$$

After receiving a message of label type $Record \vee Cancel$ from **Audit**, **Bank** will need to perform type matching to identify the actual message label, whether the message is of label Record or Cancel. The different message labels decide the following communication, whether to send message of label Success or Failure to **User**. **Audit** has the choice to either send message of type Record or Cancel to **Bank**, we can observe that they are both sent through channel instance of the same type, $c^o[Record \vee Cancel]$, an output channel that send message of all possible labels in this sending action. In fact, we will eventually assign the same actual channel to both channel instances, which we will discuss in detail in Section 3.2.

Notice that in the Effpi type, the target receiver is not specified, this will also be resolved later in Section 3.2 to make sure that the correct channel is assigned to every communication action such the sender and receiver are allocated accurately. From the Effpi type of **Bank**, we can also observe that another valid type would be:

$$et(T_{Bank}) =$$
$$\mathbf{i}[c^i[Record \vee Cancel], \Pi(\underline{x} : Record \vee Cancel)$$
$$\underline{\mathbf{match}}\ \underline{x} \left\{ \begin{array}{l} Record \Rightarrow \mathbf{o}[c^o[Success] \vee Failure], Success, \Pi()\mathbf{nil}] \\ Cancel \Rightarrow \mathbf{o}[c^o[Success \vee Failure], Failure, \Pi()\mathbf{nil}] \end{array} \right\}]$$

We merged the two channel instances used for communication with **User** into one of it's union label, since we can observe that they shall communicate through the same actual channel. Although this type reduce the number of channel instances within the local Effpi type, the total number of actual channels needed to be generated doesn't change, it also generalise the type, losing it's clarity. After sending Cancel to **Audit**, **Bank** should only be allowed to send Failure to **User**, it doesn't make sense to send it over a channel of type Success∨ Failure. We will discuss in the Section 4.3.3 why we choose the earlier channel merging method instead of the later.

Another takeaway is by observing the Effpi types for both **Audit** and **User**, we can see that the continuations within each cases are the same **nil**. In fact, we can further reduce the Effpi type for **User** such that:

$$et(T_{User}) \;= \mathbf{i}[c^i[Success \ \vee Failure], \Pi(\underline{x} : Success \ \vee Failure)\mathbf{nil}]$$

Since that we know that no matter what message type we receive, the continuation will always be **nil**, we can eliminate the type matching process for efficiency purpose. However, to what extend shall we merge the continuation such that we maintain flexibility of our tool? And also merging continuation for output doesn't make as much sense as for input, as without type matching, there's no action to eliminate and it doesn't increase efficiency in any way. We will discuss this more in detail in Section 4.5.4. The idea we are trying to bring out here is, there may be plenty of local Effpi types that correspond to a local session type, but we choose the one that gives us the best balance between clarity, flexibility and efficiency for the program.

3.2 Channel Generation and Matching

Clearly, Effpi type doesn't include the sender and receiver when sending/receiving messages over channels, it only specifies the message label that is going to be sent/ receive and the type of the channel where this communication takes place. Hence, we provide a channel generation and matching method to assign an actual channel for each communication in the program and ensure that each message sent shall reach it's intended receiver as per described in the protocol.

Again, we shall focus on local session types instead of global session types as concrete efforts have been done in projection from global session type to local session type. Having the projected local session types, we can ensure that the global session type is well-defined, for every message sent by a participant, there will be another participant anticipating to receive it and the continuations are able to be merged.

Definition 3.2.1 (Channel Instance Generation for Local Session Type)**.** *The* *grammar of local session type T with channel instances, gen(T) is:*

$$\begin{array}{rll} gen(T) ::= & \texttt{end} & \textit{Termination} \\ & \mid\ t & \textit{Type Variable} \\ & \mid\ \mu\mathbf{t}.T & \textit{Recursive Type} \\ & \mid\ (\oplus_{i \in I} \mathbf{q}!l_i(S_i).T_i : (\mathbb{C} : U)) & \textit{Message Sending} \\ & \mid\ (\&_{i \in I} \mathbf{p}?l_i(S_i).T_i : (\mathbb{C} : U)) & \textit{Message Receiving} \end{array}$$

where $\mathbb{C} = \{a,b,c...\}$ *is the infinite set of channel name variables in Effpi and* U *is the channel type in Effpi*

The channel instance generation for a local session type T *of a participant* **r**, *gen(*T*), is defined by induction on the structure of* T*:*

$$\begin{array}{rcl} gen(\texttt{end}) & = & \texttt{end} \\ gen(t) & = & t \\ gen(\mu\mathbf{t}.T) & = & \mu\mathbf{t}.gen(T) \\ gen(\oplus_{i \in I} \mathbf{q}!l_i(S_i).T_i) & = & (\oplus_{i \in I} \mathbf{q}!l_i(S_i).gen(T_i) : (a' : c^o[\vee_{i \in I} l_i])) \\ gen(\&_{i \in I} \mathbf{p}?l_i(S_i).T_i) & = & (\&_{i \in I} \mathbf{p}?l_i(S_i).gen(T_i) : (b' : c^i[\vee_{i \in I} l_i])) \end{array}$$

where a', b' are fresh channel name in Effpi, generated with a' = chan() and b' = chan() each time.

The grammar and generation of local session type with channel instance are given in Definition 3.2.1. They are both simple and straightforward, for both message sending and receiving, we generated a fresh channel instance that transfers message of all possible label for that communication, the only difference is that we cast the Channel into an OutputChannel instance for message sending and an InputChannel instance for message receiving.

By applying the channel instance generation method to our previous Bank Audit example, we can define the local session type with channel instances for each participant. The local session type with channel instances for **Bank**, **Audit** and **User** are given as $gen(T_{Bank})$, $gen(T_{Audit})$ and $gen(T_{User})$ respectively and are shown as below:

$$\begin{aligned} gen(T_{Bank}) =& (\mathbf{Audit}?Record(int).(\mathbf{User}!Success().end : (a : c^o[Success])) \\ & \&\mathbf{Audit}?Cancel().(\mathbf{User}!Failure().end : (b : c^o[Failure])). \\ & : (c : c^i[Record \vee Cancel)) \end{aligned}$$

$$\begin{aligned} gen(T_{User}) =& (\mathbf{Bank}?Success().end\ \&\ \mathbf{Bank}?Failure().end \\ & : (d : c^i[Success \vee Failure])) \end{aligned}$$

$$\begin{aligned} gen(T_{Audit}) =& (\mathbf{Bank}!Record(int).end\ \oplus\ \mathbf{Bank}!Cancel().end \\ & : (e : c^o[Record \vee Cancel])) \end{aligned}$$

As we can observe above, channel instances with name a, b and d should share the same actual channel for **Bank** and **User** to communicate in accordance to the protocol. If we assign the same actual channel to channel instance d and e, we will encounter a communication mismatch issue, **Audit** is sends a message of label type $\mathit{Record} \lor \mathit{Cancel}$ but User is expecting a message of label type $\mathit{Success} \lor \mathit{Failure}$. If we only assign the same channel for channel instance a and d but not channel instance b, deadlock will occur when **Bank** try to send Failure through channel instance b as **User** will never received the message and hence wait for it forever. Channel matching is vital in preserving communication safety of session types to Effpi. Having the local session type with channel instances for each participant, we came up with a method to perform channel instance matching as formalised in Definition 3.2.2.

Definition 3.2.2 (Matching Channel Instance in Local Session Type With Channel Instances)**.** *For two participants* $\mathbf{p}$ *and* $\mathbf{q}$ *with local session type* T_p *and* T_q *respectively, we define the set of matching channels between them, match(gen(*T_p*), gen(*T_q*), []) by induction on the structure of local session type with channel instances:*

$$\begin{array}{rcl} match(\mathtt{end}, \_, channelSet) & = & channelSet \\ match(t, \_, channelSet) & = & channelSet \\ match(\mu \mathbf{t}.T, T', channelSet) & = & match(T, T', channelSet) \end{array}$$

$$match((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)), channelSet) \quad =$$

$$\begin{cases} \begin{cases} subMatch(K, (X \cup Y) : channelSet') & channelSet = X : Y : channelSet' \ \& \ A \in X \ \& \ B \in Y \\ subMatch(K, channelSet)) & channelSet = X : channelSet' \ \& \ A \in X \ \& \ B \in X \\ subMatch(K, (A : B : X) : channelSet') & channelSet = X : channelSet' \ \& \ (A \in X \lor B \in X) \\ subMatch(K, \{A, B\} : channelSet) & otherwise \end{cases} & \begin{array}{l} (\mathbf{r} = \mathbf{p} \ \& \ \mathbf{r'} = \mathbf{q} \\ \lor \mathbf{r} = \mathbf{q} \ \& \ \mathbf{r'} = \mathbf{p}) \\ \& K = [(T_i, T_j) | \\ i \in I \land j \in J \land \\ l_i(S_i) = l_j(S_j)] \end{array} \\ subMatch([(T_i, (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B))) | i \in I], channelSet) & \mathbf{r} \neq \mathbf{p} \ \& \ \mathbf{r} \neq \mathbf{q} \\ subMatch([((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), T_j) | j \in J], channelSet) & \mathbf{r'} \neq \mathbf{p} \ \& \ \mathbf{r'} \neq \mathbf{q} \\ undefined & otherwise \end{cases}$$

$$match((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), (\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B)), channelSet) \quad =$$

$$\begin{cases} subMatch([(T_i, (\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B)))|i \in I], channelSet) & \mathbf{r} \neq \mathbf{p} \ \& \ \mathbf{r} \neq \mathbf{q} \\ subMatch([(\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), T_j)|j \in J], channelSet) & \mathbf{r'} \neq \mathbf{p} \ \& \ \mathbf{r'} \neq \mathbf{q} \\ undefined & otherwise \end{cases}$$

$$match((\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)), channelSet) \quad =$$
$$\begin{cases} subMatch([(T_i, (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)))|i \in I], channelSet) & \mathbf{r} \neq \mathbf{p} \ \& \ \mathbf{r} \neq \mathbf{q} \\ subMatch([(\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), T_j)|j \in J], channelSet) & \mathbf{r'} \neq \mathbf{p} \ \& \ \mathbf{r'} \neq \mathbf{q} \\ undefined & otherwise \end{cases}$$

$$match(T, T', channelSet) \quad = \quad match(T', T, channelSet)$$

where all channel instances in get(T_p) and get(T_q) has different channel name.
channelSet is a list of set of channel names, each channel name is unique within each set.
The definition of subMatch(K, channelSet) is given as:

$$subMatch(K, channelSet) =$$
$$\begin{cases} match(T_i, T_j, subMatch(K', channelSet)) & K = (T_i, T_j) : K' \\ channelSet & K = [] \\ undefined & otherwise \end{cases}$$

First of all, we assume that channel instance names are different intra- and and inter- local session type. No two channel instances share the same channel instance name within a participant's local session type or with some channel instances of other participant's local session type, this will be later enforce in our implementation. Given the local session type with channel instances of two participants **p** and **q**, we match their channel instances following a certain rules, starting with an empty list of matching-channel sets, we refer to this list as channel-matching list. We only focus on sending and receiving of messages as there's where usage of channels is required. Since this local session types are projected from global session type, we can be sure that the local session types are well defined such that the matching of communication between any two participants is complete and consistent throughout the session type. This matching is not affected by recursion, the matching of each communication remains the same for each cycle. In fact, for every recursion in **p** which involves communication actions with **q**, there will be a recursion in **q** which correspond to them and vice versa.

We shall now look at the process of matching. If one of the local session type with channel instances is `end`, we shall just return the current list of matching channels as there will be no continuation to perform the matching. Same goes for $\mathtt{t}$, the recursive variable indicates the return to the start of a cycle, which we have already perform the matching once and matching is consistent for each cycle as only one actual channel shall be assigned for each channel instance. If both local session types are sending/receiving messages to/from p or q, we can match their channels. From the Bank Audit example before, say we try to match local session type with channel instances of both **Bank** and **Audit**:

$$\begin{aligned}
match((&\mathsf{Audit}?Record(int).T_1 \& \mathsf{Audit}?Cancel().T_2 \\
&: (c : c^i[Record \vee Cancel])), \\
(&\mathsf{Bank}!Record(int).T_3 \ \oplus \ \mathsf{Bank}!Cancel().T_4 \\
&: (e : c^o[Record \vee Cancel])), [])
\end{aligned}$$

$$\begin{aligned}
where\ T_1 &= (\mathsf{User}!Success().end : (a : c^o[Success])) \\
T_2 &= (\mathsf{User}!Failure().end : (b : c^o[Failure])) \\
T_3, T_4 &= \mathtt{end}
\end{aligned}$$

We can see that by definition of $match$, channel instances $c : c^i[Record(int) \vee Cancel()]$ (abbrev. C) and $e : c^o[Record(int) \vee Cancel()]$ (abbrev. E) should be matched together. Since the current list of channel-matching set is empty, we can just add a new set with C and E into the list, [{C,E}]. If one or more of the channel already exist in the set of the list, we merge or add into the sets. Say channel instance C has previously matched with a channel instance C' and channel instance E matched with a channel instance E', forming a channel-matching list of [{C',C}, {E,E'}], matching channel instance C and E will form a new list [{C',C,E,E'}] as these channel instances shall share the same actual channel. By doing so we can make sure each channel instance only appears in at most one set in the list, we can then later assign the same actual channel to each channel instance of a set.

In Definition 3.2.2, we took into account the cases where p and q try to send(receive resp.) to(from resp.) each other at the same time and also the case where we have p(or q) as the only target of both session type. We did so by labelling the output of $match$ as $undefined$, this should never occur in practice having local session types successfully projected from global session type. After matching the channels, we will perform the match again on the continuations of same message types, along with the newly generated channel-matching list. Communication from p to q after p sent(received resp.) a message of a label M to q should correspond to the communication from q to p after q received(sent resp.) a message of label M to p. From the example above, we should continue by matching T_1 to T_3 and T_2 to T_4, formalised into $match(T_1, T_3, match(T_2, T_4, [\{c : c^i[Record(int) \vee Cancel()], e : c^o[Record(int) \vee Cancel()]\}]))$. The sequence of matching doesn't affect the output as we always check the sets in the list if a channel instance already existed.

A question that one may ask is: Why do we need to consider all different cases where the channel instances may or may not exist in the current channel-matching list when we perform matching on them? Since from session type theory, we know that all message type within a send/receive must be different. It doesn't make sense to have a local session type of **Bank**?Cancel().T_3 & **Bank**?Cancel().T_4 as we can't differentiate which continuation, T_3 or T_4, should we proceed with after receiving a message of type Cancel(). Having this in mind, the matching of continuations for each sending/receiving of message should be independent of each other. In other words, there shouldn't be a case where a channel instance already existed in current channel-matching list when you perform a matching as you should never performed matching on it before. This is however, only true for protocols with only two participants, or what we referred more formally as binary session types. For protocols with more than two participants, we may encounter communication with a third participant in the local session types when we try to match the channel instances between two participants. Say that when we try to match the local session type with channel instances of two participants, **p** and **q**, one of them is a sending(receiving resp.) of messages to(from resp.) a third party participant **r**, this communication shouldn't affect the ongoing communication between **p** and **q**. This is because **p**(or **q**) does not acknowledge the communication between **r** and **q**(or **p**), **p**(or **q**) has no clue of what message type has been exchanged between **r** and **q**(or **p**). Hence all the continuations of communication between **r** and **q**(or **p**) should match with the current local session type with channel instances of **p**(or **q**). As usual, we have this property as we assume local session types are successfully projected from a mergeable global type as described in Definition 2.1.4. Let's look back our example where we try to match the local session types with channel instances between **Bank** and **User**:

$$\begin{aligned} match((&\textbf{Audit}?Record(int).T_1 \& \textbf{Audit}?Cancel().T_2 \\ &: (c : c^i[Record \vee Cancel])), \\ &(\textbf{Bank}?Success().end \ \& \ \textbf{Bank}?Failure().end \\ &: (d : c^i[Success \vee Failure])), [\,]) \end{aligned}$$

$$\begin{aligned} where\ T_1 &= (\textbf{User}!Success().end : (a : c^o[Success])) \\ T_2 &= (\textbf{User}!Failure().end : (b : c^o[Failure])) \end{aligned}$$

Bank's local session type with channel instance starts by receiving messages from **Audit** which is not acknowledged by **User**. Hence, their continuations T_1 and T_2 should match with the local session type with channel instances of **User**.

We can further reduce the formula to:

$$\begin{aligned}&match((\textsf{User}!Success().end : (a : c^o[Success])), T,\\ &\qquad match((\textsf{User}!Failure().end : (b : c^o[Failure])), T, []))\\ \\ &\qquad where\ T = (\textsf{Bank}?Success().end \ \&\ \textsf{Bank}?Failure().end\\ &\qquad : (d : c^i[Success \lor Failure]))\end{aligned}$$

which further reduces to:

$$\begin{aligned}&match((\textsf{User}!Success().end : (a : c^o[Success])),\\ &(\textsf{Bank}?Success().end \ \&\ \textsf{Bank}?Failure().end : (d : c^i[Success \lor Failure])),\\ &[\{b : c^o[Failure], d : c^i[Success \lor Failure]\}])\end{aligned}$$

Now we will have to match channel instance $a : c^o[Success]$ with $d : c^i[Success \lor Failure]$, however we already have $d : c^i[Success \lor Failure]$ in our channel-matching list because it was used to match with the channel instance from another continuation. In accordance to definition of *match*, the final output will be $[\{a : c^o[Success],\ d : c^i[Success \lor Failure], b : c^o[Failure]\}]$, indicating that we will have to assign the same channel to these three channel instances.

The type of channel to be generated for each channel instance set in the channel matching list is given as in Definition 3.2.3, which is an input output channel that transfer message of all possible labels in the communication. We built the idea on the fact that each channel instance set within the channel-matching list must contains at least one input channel instance and input channel instance must receive message of all possible labels in the communication. This is because selection can occur through multiple output channel instances, with each output channel instances sending one of the possible label. All the output channel instances of this selection will be eventually assigned to the same actual channel, the sender can choose which output channel instance they send the message through based on the message label they want to send. We can't have multiple input channel instances for branching, one for the receiving of each label, as receiver cannot wait on multiple channel instances at the same time. Hence, input channel instance is responsible to receive message of all possible labels in the communication. The actual channel assigned to all the channel instances within a channel matching set shall transfer message of the same labels as of the input channel instance in the set.

Definition 3.2.3 (Type of Actual Channel Assigned to Channel Instances)**.** *We define the channel type assigned to a set of channel instances C as assignChanType(C):*

$$assignChanType(C) = \begin{cases} c^{io}[L] & (\_ : c^i[L]) \in C \\ undefined & otherwise \end{cases}$$

Chapter 4

Design and Implementation

In this chapter we shall discuss about the implementation, design and limitation of our tool. Our tool takes in a multiparty session type(MPST)-based protocol in the form of a Scribble file as an input. This global protocol provides a bird-eye-view of the underlying communication between multiple participants. We show an example of a protocol which we will use to demonstrate the entire generation process in this chapter, the ArtSeller protocol, which can be seen in Listing 4.1. Two buyers, B1 and B2, want to buy the same art piece through a seller server Svr, the server *Start* by sending them both the initial price of the art piece and initiating the bidding process. In the bidding process, B1 and B2 will offer a *Bid* price to Svr, Svr then decides if it wants to *Continue* with the bidding process again or *Confirm* the purchase with one of the buyer and *Cancel* with the another. Each participant may have their own logic on which selection to make or what price to offer, our purpose is to make sure they communicate in accordance to the global protocol. Our tool is written in Python 3.8, integrated with Scala and Dotty to compile and run the generated Effpi-typed program. We shall also discuss about the contributions we made to the Dotty compiler. Before we go any further, we shall introduce some of the notion we used in this and upcoming chapters:

- `Effpi type` - Effpi types syntax used in the implementation
- `Effpi term` - Effpi term syntax used in the implementation
- `Comm class` - Self-defined communication class' name
- Var name - Member variables of communication classes
- func - algorithm function/ member function of communication classes

```
module ArtSeller;

global protocol ArtSeller(role B1, role Svr, role B2) {

    Start(basePrice: number) from Svr to B1;
    Start(basePrice: number) from Svr to B2;
    do Bidder(B1, Svr, B2);
}

aux global protocol Bidder(role B1, role Svr, role B2) {

     Bid(offer: number) from B1 to Svr;
     Bid(offer: number) from B2 to Svr;
     choice at Svr{
           Continue(currPrice : number) from Svr to B1;
           Continue(currPrice : number) from Svr to B2;
           do Bidder(B1, Svr, B2);
         }or{
           Cancel() from Svr to B1;
           Confirm(finalPrice:number) from Svr to B2;
         }or{
           Confirm(finalPrice:number) from Svr to B1;
           Cancel() from Svr to B2;
         }
}
```

Listing 4.1: ArtSeller Protocol

```
digraph G {
  0;1;2;3;4;

  0 -> 1 [label="Svr?Start(basePrice: number)", ];
  1 -> 2 [label="Svr!Bid(offer: number)", ];
  2 -> 1 [label="Svr?Continue(currPrice: number)", ];
  2 -> 3 [label="Svr?Cancel()", ];
  2 -> 4 [label="Svr?Confirm(finalPrice: number)", ];
  }
```

Listing 4.2: digraph generated by nuScr for B1 in ArtSeller protocol

```
digraph G {
  0;1;2;3;4;

  0 -> 1 [label="Svr?Start(basePrice: number)", ];
  1 -> 2 [label="Svr!Bid(offer: number)", ];
  2 -> 1 [label="Svr?Continue(currPrice: number)", ];
  2 -> 3 [label="Svr?Confirm(finalPrice: number)", ];
  2 -> 4 [label="Svr?Cancel()", ];
  }
```

Listing 4.3: digraph generated by nuScr for B2 in ArtSeller protocol

```
1  digraph G {
2    0;1;2;3;4;5;6;7;8;9;
3
4    0 -> 1 [label="B1!Start(basePrice: number)", ];
5    1 -> 2 [label="B2!Start(basePrice: number)", ];
6    2 -> 3 [label="B1?Bid(offer: number)", ];
7    3 -> 4 [label="B2?Bid(offer: number)", ];
8    4 -> 5 [label="B1!Continue(currPrice: number)", ];
9    4 -> 6 [label="B1!Cancel()", ];
10   4 -> 8 [label="B1!Confirm(finalPrice: number)", ];
11   5 -> 2 [label="B2!Continue(currPrice: number)", ];
12   6 -> 7 [label="B2!Confirm(finalPrice: number)", ];
13   8 -> 9 [label="B2!Cancel()", ];
14
15   }
```

Listing 4.4: digraph generated by nuScr for Svr in ArtSeller protocol

4.1 nuScr and EFSM

In order to understand the local communication for each participant, we will need to generate a Communication Finite State Machine(CFSM) that specifies the different states of the participant, the possible communication actions at each state and the consequent state after performing the action. In order to overcome this, we integrated nuScr[11] into our tool, nuScr is a toolchain implemented in OC_{AML} for MPST-based protocols. It can convert the protocols into global session types in MPST theory; global session types are then projected into local session types for each participant as defined in Definition 2.1.4, and local types are converted to their correspond CFSM. We input the ArtSeller protocol in Listing 4.1 into nuScr and it outputs the CFSM in the form of a digraph string. The digraph string output for B1 and B2 and Svr are shown in Listing 4.2, 4.3 and 4.4 respectively, we can see in Line 2 all the different states of the local CFSM of a participant, Lines below it shows the possible actions taken in each state and their respective successor state. For example, in Line 8 to 10 of Svr's digraph, Svr is at state 4 and can either choose to send to B1 a label with payload of *Continue(currPrice: number)*, *Cancel()* or *Confirm(finalPrice:number)* and will go to state 5, 6 or 8 respectively.

Working with a digraph string format, however, can be very troublesome as it is hard to access the actions and states in the form of strings. To do so, we define our own `EFSM`,`State` and `Action` classes. `EFSM` stores all states in the CFSM in a list of `State`s, each `State` stores the id of the state it's representing and the actions that can be performed in the state in a list of `Action`s. All `Action` in the list must be all of type `SendAction` or `ReceiveAction` but not a mix of both. Each `Action` instance has a sender, receiver, message label, message payloads and a successor `State`. Note that all this information are retrievable from the digraph, Line 5 of Svr's digraph represents a single `SendAction` instance with sender Svr, receiver B2, message label *Start*, payloads ['basePrice: number'] and successor `State` with id 2. To construct the `EFSM` from digraph

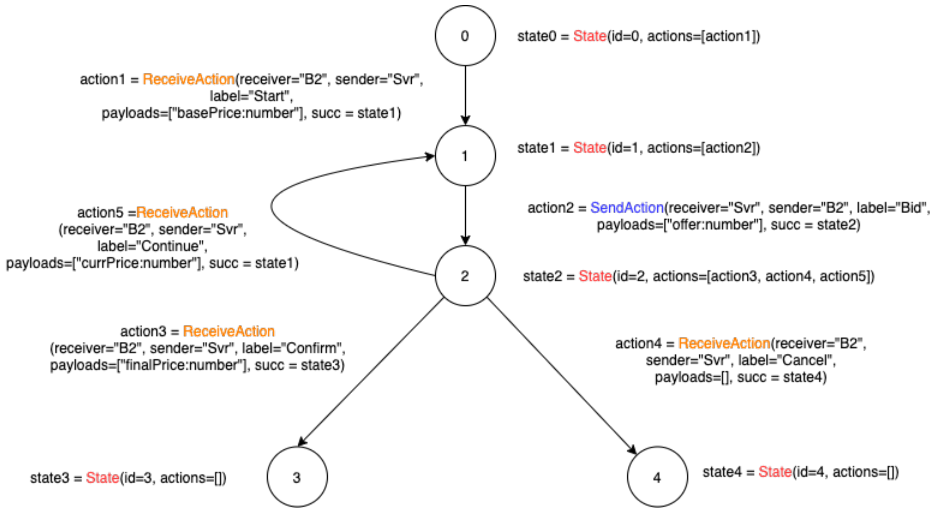

0
state0 = State(id=0, actions=[action1])
action1 = ReceiveAction(receiver="B2", sender="Svr", label="Start", payloads=["basePrice:number"], succ = state1)
1
state1 = State(id=1, actions=[action2])
action5 =ReceiveAction (receiver="B2", sender="Svr", label="Continue", payloads=["currPrice:number"], succ = state1)
action2 = SendAction(receiver="Svr", sender="B2", label="Bid", payloads=["offer:number"], succ = state2)
2
state2 = State(id=2, actions=[action3, action4, action5])
action3 = ReceiveAction (receiver="B2", sender="Svr", label="Confirm", payloads=["finalPrice:number"], succ = state3)
action4 = ReceiveAction(receiver="B2", sender="Svr", label="Cancel", payloads=[], succ = state4)
state3 = State(id=3, actions=[])
3
4
state4 = State(id=4, actions=[])

Figure 7: EFSM for B1 in ARTSELLER protocol

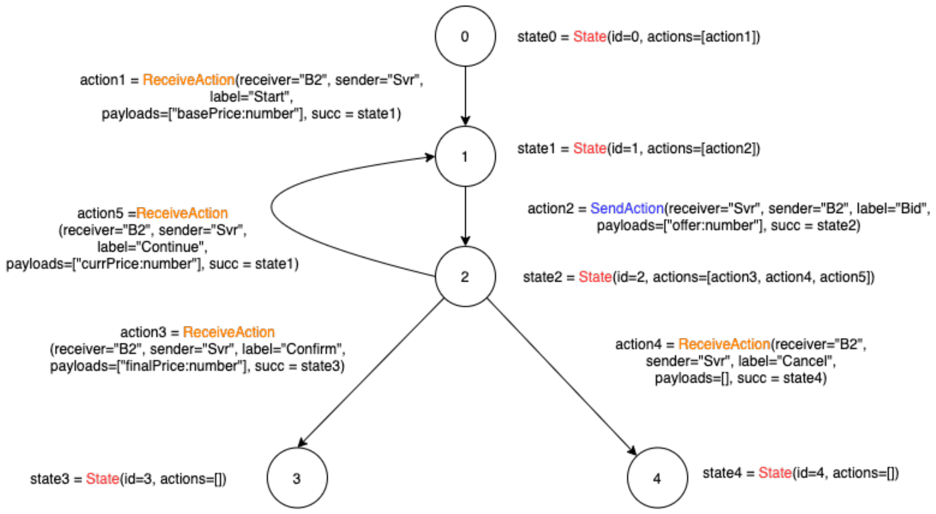

0
state0 = State(id=0, actions=[action1])
action1 = ReceiveAction(receiver="B2", sender="Svr", label="Start", payloads=["basePrice:number"], succ = state1)
1
state1 = State(id=1, actions=[action2])
action5 =ReceiveAction (receiver="B2", sender="Svr", label="Continue", payloads=["currPrice:number"], succ = state1)
action2 = SendAction(receiver="Svr", sender="B2", label="Bid", payloads=["offer:number"], succ = state2)
2
state2 = State(id=2, actions=[action3, action4, action5])
action3 = ReceiveAction (receiver="B2", sender="Svr", label="Confirm", payloads=["finalPrice:number"], succ = state3)
action4 = ReceiveAction(receiver="B2", sender="Svr", label="Cancel", payloads=[], succ = state4)
state3 = State(id=3, actions=[])
3
4
state4 = State(id=4, actions=[])

Figure 8: EFSM for B2 in ARTSELLER protocol

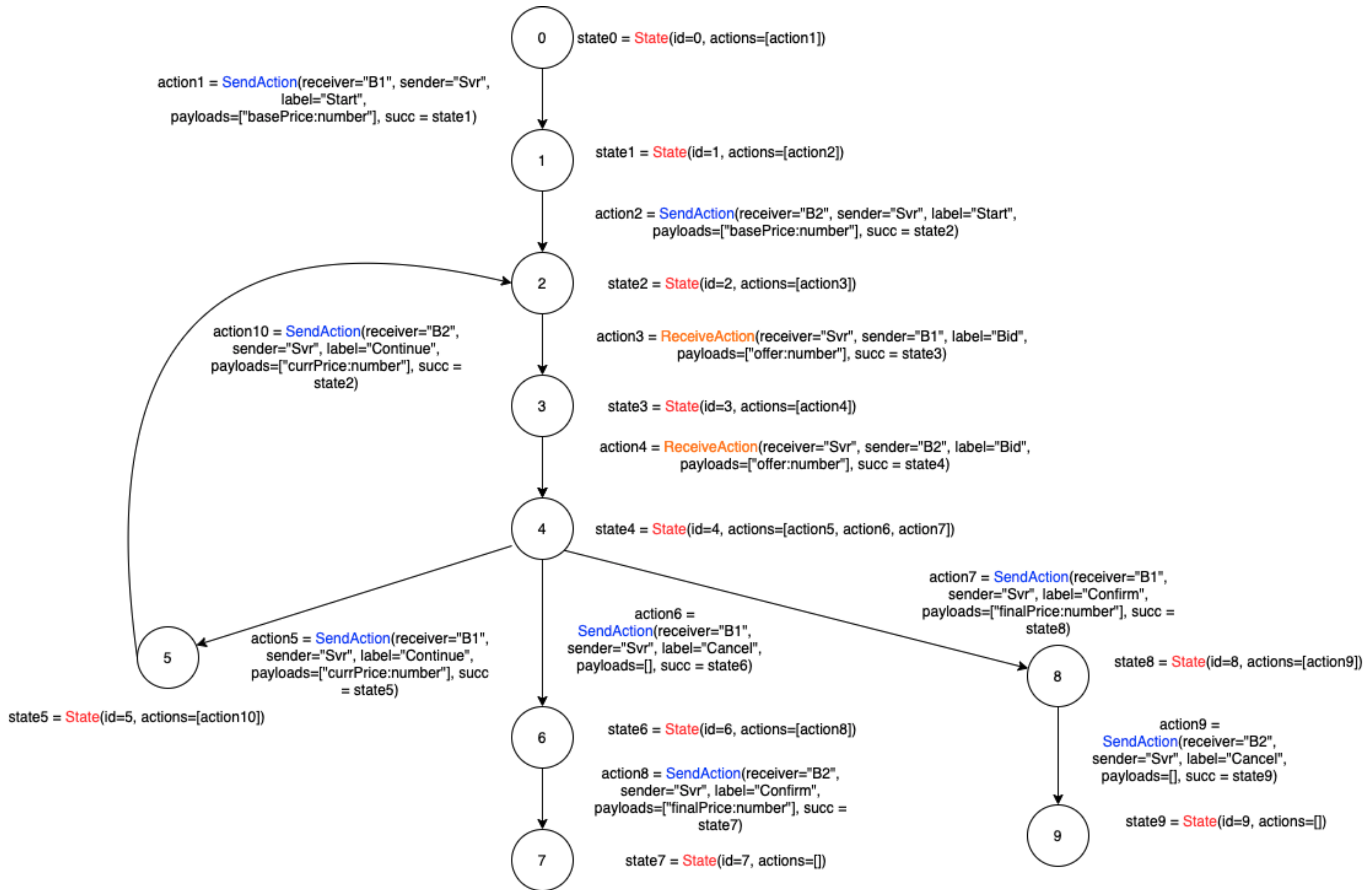

0
state0 = State(id=0, actions=[action1])
action1 = SendAction(receiver="B1", sender="Svr", label="Start", payloads=["basePrice:number"], succ = state1)
1
state1 = State(id=1, actions=[action2])
action2 = SendAction(receiver="B2", sender="Svr", label="Start", payloads=["basePrice:number"], succ = state2)
2
state2 = State(id=2, actions=[action3])
action10 = SendAction(receiver="B2", sender="Svr", label="Continue", payloads=["currPrice:number"], succ = state2)
action3 = ReceiveAction(receiver="Svr", sender="B1", label="Bid", payloads=["offer:number"], succ = state3)
3
state3 = State(id=3, actions=[action4])
action4 = ReceiveAction(receiver="Svr", sender="B2", label="Bid", payloads=["offer:number"], succ = state4)
4
state4 = State(id=4, actions=[action5, action6, action7])
action7 = SendAction(receiver="B1", sender="Svr", label="Confirm", payloads=["finalPrice:number"], succ = state8)
action6 = SendAction(receiver="B1", sender="Svr", label="Cancel", payloads=[], succ = state6)
action5 = SendAction(receiver="B1", sender="Svr", label="Continue", payloads=["currPrice:number"], succ = state5)
5
state5 = State(id=5, actions=[action10])
8
state8 = State(id=8, actions=[action9])
action9 = SendAction(receiver="B2", sender="Svr", label="Cancel", payloads=[], succ = state9)
6
state6 = State(id=6, actions=[action8])
action8 = SendAction(receiver="B2", sender="Svr", label="Confirm", payloads=["finalPrice:number"], succ = state7)
9
state9 = State(id=9, actions=[])
7
state7 = State(id=7, actions=[])

Figure 9: EFSM for Svr in ARTSELLER protocol

string, we get a digraph from the digraph string using *graph_from_dot_data()* in pydot library[16]. We then loop through the edges of the digraph and build a `SendAction` or `ReceiveAction` based on their action type, they are then added into the `EFSM`. The `EFSM` record the states and actions in that state, it also provides some very useful member functions for us to check if a `State` is terminal, initial, sending or receiving. A terminal state is defined as a `State` with empty `Action` list and the initial state is defined as the `State` with the smallest state id, there may be zero or more terminal state in the `EFSM` but exactly one initial state. EFSM for B1, B2 and Svr are shown in Figure 7, 8 and 9 respectively. Looking at the EFSM for Svr, State 7 and 9 are terminal states as they have empty action list, state 0 is initial state as it has the smallest state id. State 4 has three actions, action5, action6 and action7 in it's action list as there are three possible actions that can take place in the state. Receiver, sender, label and payloads of each action is self-explanatory, we can always check the states and actions in the EFSM in accordance to the digraph or even the original global protocol. We shall use these EFSMs throughout the generation process.

4.2 Channel Instances Assignment and Matching

Since we have the local EFSM for each participant, we can further match the communications between them as described in Definition 3.2.2. Prior to this, channel instances need to be generated for each communication based on Definition 3.2.1. In other words, we will need to assign a channel instance to each non terminal state in the EFSM. We can opt to generate channel instances for each EFSM state when building communication classes from the EFSM in the next section and perform channel instance matching between each participant during a later stage. However, we choose to do them in this stage as we can perform some optimisation on channel instance generation at the same time which will affect the communication classes generated right after, the details of optimisation will be discuss in Chapter 5.

By combining Definition 3.2.1 and 3.2.2, we formalise an algorithm for channel instance assignment and channel matching for every two participants in Listing 4.5. Note that we add a `name` variable to the `State` class in EFSM to record the channel name generated for it. The same EFSM will be used in communication class generation later and we can get the desired channel instance name for each state directly.

The recursive match_state() function takes the name of both participants, the states to be matched, channel count for both participant, list of visited states and the current channel-matching list. On initialisation, both states are set to the initial state of both EFSM, counts are set to 1, visited and channel-matching

```
def match_state(role1, role2, state1, state 2, visited,
                channelMap, count1, count2){
    states1 = remove_third_party(role2, {state1}, visited)
    states2 = remove_third_party(role1, {state2}, visited)

    if states1 == {} or states2 == {}:
        return count1, count2

    channelSet = {}
    for state in states1 ⋃ states2:
        if state.name != null and channelMap = X: subChannelMap
           and state.name ∈ X:
           channelMap = subChannelMap
           channelSet = channelSet ⋃ X
        else if state.name != null:
           channelSet = channelSet ⋃ {state.name}
        else:
          if state ∈ states1:
             channelName = gen_chan_name(role1, role2, count1)
             count1 = count1 + 1
          else:
             channelName = gen_chan_name(role2, role1, count2)
             count2 = count2 + 1
          channelSet = channelSet ⋃ channelName
          state.name = channelName

    channelMap = channelMap ⋃ channelSet
    visited = visited ⋃ states1 ⋃ states2

    for action1 in [state.action | state ∈ states1]:
       for action2 in [state.action | state ∈ states2]:
           if action1.label == action2.label:
             count1, count2 = match_state(role1, role2,
             action1.succ, action2.succ, visited,
                channelMap, count1, count2)

    visited = visited \ states1 \ states2
    return count1, count2
}


def remove_third_party(target, states, visited){
    isValid = True
    for state in states:
       if state in visited or state.is_terminal:
           states = states \ state
       else if state.actions[0].sender != target and
               state.actions[0].sender != target:
           isValid = False
           states = states \ state
           visited = visited ⋃ {state}
           for action in state.actions:
                states = {action.succ} ⋃ states
    if isValid:
       return states
    return remove_third_party(target, states, visited)
}

def gen_chan_name(origin, target, count):
    return "c_{origin}_{target}_{count}"
```

Listing 4.5: Algorithm for Channel Name Assignment and Matching for Two Participant

list are empty. This algorithm assumes that all the EFSMs are generated from local session types that was projected from a same global session types, hence we don't have to check the validity of communication between participants. First, it calls a recursive remove_third_party to remove visited, terminal or third party states. As we mentioned before, a third party state is a `State` in the EFSM where the communication involve some other participant. Remove_third_party() operates in a breadth first search(BFS) manner, whenever it encounter a third party state, it will remove the state from the list, add it into the visited list, add all the continuation states of the state into then state list and repeat the check again. It repeat until the list is free from third party, terminal and visited states (it may be an empty list). If we get non empty list of `State`s after applying remove_third_party() on the state of both participants, we know that these states are meant to communicate through the same actual channel, otherwise we shall return early as there's no more channel instance matching to perform. As described in the definition, we need to check whether the state to be match already has a channel instance name and exists in the current channel-matching list. If so, we will retrieve the set that contains the channel instance name from the list and add into our current building set. For states that yet to have a channel instance name, we generate a new channel instance name through gen_-chan_name(), assign it to the state and add it into our building set, the set is later added into the channel_matching list. Our channel instance naming convention is fairly simple, the channel instance name takes the form of c_{origin participant name}_{target participant name}_{channel count}. For example, if participant Svr wants to generate a channel instance name for it's input or output channel to participant B1, the channel count from Svr to B1 is 2, the channel instance name generated will be c_Svr_B1_2. We keep track of the channel count from one participant to another, every time we generate a new channel instance name, we will increment the count, the counts will eventually be returned at the end of the function. By doing so we can make sure that no two channel share the same name throughout the entire program. Note that at this point we only match channel instances through their names, channel instance types will be assigned at the communication class generation stage. Channel instance types let us know what type of actual channel to be generated on the main class level, it will not affect the channel instance matching. After assigning the channel instances, we will perform the recursive call of match_state() on successor states of actions from each set of states that share the same label.

Applying this algorithm to initial state of B1's EFSM in Figure 7 and initial state of Svr's EFSM in Figure 9, state0, state2, state4 of Svr's EFSM will be assigned channel name c_Svr_B1_1, c_Svr_B1_2 and c_Svr_B1_3 respectively. State0, state1 and state2 of B1 are assigned channel name c_B1_Svr_1, c_B1_Svr_2 and c_B1_Svr_3 respectively. State3 and state4 of B1 are not assigned any names as they are terminal states, no communication is expected to take place. The channel matching list generated is [{c_Svr_B1_1, c_B1_Svr_1}, {c_Svr_B1_2, c_B1_Svr_2}, {c_Svr_B1_3, c_B1_Svr_3}] which is self-explanatory. Even though B2 has a seemingly similar EFSM to B1, but when we perform the same matching procedure for B2 and Svr, state1, state3, state5, state6 and state8 of Svr's EFSM are assigned names c_Svr_B2_{1-5} respectively, which is two channels more than that of B1. State0, state1 and state2 of B2 are assigned channel names c_B2_Svr_{1-3}, the same manner as of B1. This is because from Svr's perspective, communication to B2 in state5, state6 and state8 are perform through three separate channel instances but from B2 perspective these three messages are received through the same channel instance in state2. The channel matching list generated, [{c_Svr_B2_1, c_B2_Svr_1}, {c_Svr_B2_2, c_B2_Svr_2}, {c_Svr_B2_3, c_Svr_B2_4, c_Svr_B2_5, c_B2_Svr_3}] showcased this matching, the same channel will eventually be assigned to these channel instances. When we match between B1 and B2, no channel instance names are generated and the channel matching list generated is empty as there's no communication between them.

4.3 Communication Packages

In this section, we shall discuss various communication packages and the self-defined communication classes within them. For each communication class, we show the Effpi type they represent and the function body that corresponds to the type. We shall first introduce the basic packages and dive into more complex packages that handle sending/ receiving of messages, type matching and recursion. Note that we only show class methods that add functionality to the class, trivial methods such as get methods are not shown. Some Effpi types syntax and terms syntax are defined differently from Section previously to increase clarity for programmers. For example, the union type operator $\vee$ is replaced with |, other replacements will be mentioned throughout this section.

4.3.1 Base package

The base package consists of the most basic base classes that build up the program. This package is imported and used or extended by other communication classes .

`abstract class` **CommunicationBase**

- `abstract method` String get_type()
- `abstract method` String get_function_body()
- `abstract method` List[CommunicationBase] get_continuations()

This abstract class is meant to be extended by all communication class. Each communication class must have it's built in logic to construct it's own Effpi type and the correspond function body in the form of a string and return them on get_type() and get_function_body() call respectively. Each communication class will also have to keep track of it's successive communication classes, we can picture it as a node in the tree knowing it's children nodes, get_continuations() return the list of of successive communication classes of a communication class. It is used in Section 7.1 later to check that the communication classes are generated in accordance to our EFSM. All communication classes that extends this base class will have to implement all three functions otherwise a `NotImplementedError` will be thrown.

`class` **Label**

- `var` String name
- `var` List[String] payloads
- `method` String get_payload_string()
- `method` String get_default_payloads()

This is not a communication class but it's used by almost every communication class. It represent the label of a communication action by record the label's name and it's list of payloads. Payloads can come in different format, with or without assigned name and also various types from nuScr. For example a label *Buy(amount: number, String, date)*, it has payloads of ["amount:number", "String", "date"]. The first payload is assigned a name "amount" but the other two other payloads are not assigned any name, "number" and "date" are also not valid types in Scala. get_payload_string() assign default payload name to unnamed payloads and replace nuScr types to Scala types. Call of get_payload_string() on the payloads above will return "amount:Int, x1:String, x2:Date". get_default_payloads() assign default values for each type in the payload. These two functions are extremely helpful when generating default function body for a communication class, different labels from large number of actions can be handled easily. We hash this class based on the name but not the payload as we assume overloading of labels are not allowed (we can't have two labels of the same name and different payloads), we shall discuss this in the limitation section later.

`class` **Termination** `extends` **CommunicationBase**

`Termination` indicates the end of the program. It has an empty continuation list as no more communication action is expected to take place after this. It returns the Effpi Termination type, `PNil`(shown as **nil** in original Effpi type syntax) with correspond function body `nil` (shown as **end** in original Effpi term syntax), both seen in Listing 4.6.

4.3.2 Channel Instances package

Channels are the communication backbone in Effpi programs, participants send and receive messages through specific channels and we use Effpi channel type

```
def get_type():
    return PNil

def get_function_body():
    return nil
```

Listing 4.6: Effpi type and fuction body generation for Termination

to make sure the sending/receiving of the right message occurs through the expected channel in the desired sequence. In this subsection we shall introduce the abstract `ChannelInstance` class and the concrete `InChannel` and `OutChannel` class that is used to send/received single messages. On the local participant level, each channel instance used for message exchange is of type `InChannel` or `OutChannel`. It is very important to learn that the "channels" we used in participant level are channel instances but not actual channel, the actual channels are generated on the main class level and assigned to every channel instances of each participant. From the channel matching section we seen before, we can see why and how the same channel are assigned to multiple channel instances within a participant. We shall discuss this further on the generated Effpi types after we introduced Selection and Branch package. All we have to know now is that the actual channel generated on the main class level is of type `InOutChannel` but will be casted into `InChannel` or `OutChannel` instances at the participant level based on it's role from the participant's perspective. We do this as we want the purpose of each channel instance to be as specific as possible. This will add a level of type safety to the program as sending a messages through an `InChannel` will cause a compilation error as well as receiving messages through an `OutChannel`. There are some other communication classes that extends `ChannelInstance` but not used for sending/receiving single message, we shall discuss them in later sections.

`abstract class` **ChannelInstance** `extends` **CommunicationBase**

- `var` String channelName
- `var` Set[Label] labels
- `var` String sender
- `var` String receiver
- `method` String get_channel_type()

Every channel instance has it's own name, we hash channel instances based on their names, we showed in the channel matching algorithm in Listing 4.5 that no two channel instances in our program share the same name. Other than that, we can view channel instance as an action of message receiving/sending. When we send/receive a message, we need to know the sender and receiver of the message. We also need to know the labels of message we are expected to send/receive. get_channel_type() returns the channel instance's type. A detail worth mentioning is each channel instance variable in the function correspond to a channel instance type variable in the Effpi type the function type checked with. Both variable are generated during the get_function_body() and get_type() call respectively. For simplicity purpose, the channel instance type variable in the Effpi type has the same name as the channel instance name(The first 'C' in the name is uppercase by default), the channel instance variable in the function body will has the first 'c' lowercase. A channel instance variable c_Svr_Client_3 in a function will correspond to type variable C_Svr_Client_3 in it's Effpi type. The communication behaviour of c_Svr_Client_3 in the function must conform to C_Svr_Client_3's communication type. We link these two variables in a very simple way and will explain in Section 4.3.6 later.

`class` **OutChannel** `extends` **ChannelInstance**

- `var` CommunicationBase continuation

`OutChannel` is a communication class that represents channel instance for sending single message. `OutChannel` has exactly one message label in labels as we already know the actual label of a message we are about to send. Since it's a single message sending, we don't have to choose which message to send, we are expected to have exactly one communication class as our continuation. get_continuation() will return a list with continuation as it's only element. The Effpi type it is representing is $\mathbf{o}[c^o[\vee_{j\in I} l_j], l_i, \Pi()et(T_i)]$ where $i \in I$. For clarity, the Effpi type syntax is modified such that $\mathbf{o}$ is replaced with `Out`, continuation operator, >>: is introduced and c^o in replaced with `OutChannel`. The corresponding type syntax is `Out[OutChannel[`$|_{j\in I} l_j], l_i]$ >>: $\Pi()et(T_i)$ where $i \in I$. The syntax of Effpi term that correspond to this Effpi type is $\mathbf{send}(t_1, t_2, t_3)$ where t_1 is a channel instance variable with type $c^o[\vee_{j\in I} l_j]$, t_2 is an object of instance l_i and t_3 is a continuation term of type $\Pi()et(T_i)$. Similarly, a continuation term >>, is added for clarity, the correspond new term syntax is **send**(t_1, t_2) >> t_3. The Effpi type and function body generated for `OutChannel` is shown in Listing 4.7. first_char_lower() is a built it function for string typed value to convert the first character in the string to lower case. We use get_default_type() here to generate a default payloads for the message we are going to sent, the programmer can easily modify these value.

The get_channel_type() function returns the channel type for `OutChannel`, it assumes that the channel instance only takes in a single label. We can only call this function on `OutChannel` class instance if we know that this `OutChannel` is not used for making selection. Otherwise, we only know the channel instance's name and it's an `OutChannel`, but we don't know the full type of the channel instance(We doesn't know what $\vee_{j\in I} l_j$ is). For example, we know that we are going to send message with label Hello through channel c_P1_P2_1, however we don't know if c_P1_P2_1 has a type of `OutChannel[Hello]` or

```
assert(len(labels) == 1)

def get_type():
    return Out[channelName, labels[0]] >>: continuation.get_type()

def get_channel_type():
    return OutChannel[labels[0]]

def get_function_body():
    return send(channelName.first_char_lower(), labels[0].get_default_type()) >>
            {continuation.get_function_body()}
```

Listing 4.7: Effpi type and fuction body generation for OutChannel

`OutChannel[Hello | Bye]` or even some `OutChannel[Hello | ...]`. This will done later when we build communication classes from EFSM, where we record the channel type for each channel instances, we can see if this channel instance is used for selection or single message sending. We plan to keep the responsibility at the communication class level as small as possible, `OutChannel` doesn't need to know the channel instance's full type at it's level. As long we maintain the consensus of naming channel instance variable as described above, the process of communication class generation from EFSM will eventually assign the type $c^o[\vee_{j\in I} l_j]$(`OutChannel`$[|_{j\in I} l_j]$ in our defined syntax) to our channel instance variables.

`class` **InChannel** `extends` **ChannelInstance**

– `var` String param

– `var` CommunicationBase continuation

InChannel is a channel instance for receiving single message. By single message, we mean a single message instance, but not message of single label. Different from sending message, we don't know the actual label of message that we will receive, hence we may have one or more message labels in labels. For example, we know that the message we are going to receive may be of type *Buy*, *Cancel* or *Retry*. We are hence expecting to receive a single message of type *Buy* | *Cancel* | *Retry*, further type matching is required to determine the actual type of the message. Similarly, `InChannel` doesn't need to know whether further type matching is required, this will be handled in the process of generating communication classes from EFSM, the correct continuation will be assigned to this `InChannel`. get_continuation() will return a list with continuation as it's only element. param is the name for the variable used to store the message received through the input channel, it has a name "X" by default if no name is given. The Effpi type it's representing here is $\mathbf{i}[c^i[\vee_{i\in I} l_i], \Pi(\underline{x} : \vee_{i\in I} l_i)U]$. Identical to `OutChannel`, the correspond replaced type syntax is `In[InChannel`$[|_{i\in I} l_i], (\underline{x} : |_{i\in I} l_i) \Rightarrow U]$.

The original Effpi term that correspond to this type is $\mathbf{recv}(t_1, t_2)$ where t_1 is a channel instance of type $c^i[\vee_{i\in I} l_i]$ and t_2 is the continuation term of type $\Pi(\underline{x} : \vee_{i\in I} l_i)U$, the modified syntax is `receive`$(t_1)\{t_2\}$. The Effpi type and function body generated for `InChannel` is shown in Listing 6.8.

As we observe, param is handled in a similar manner as of how we did for channelName to differentiate between Effpi type varaible and function variable. Oppose to OutChannel, we know exactly the channel type that is going to be assigned to our channel instances. For example, we have a channelName of C_Svr_B1_1 and labels of [*Hello*, *Bye*], we know that channel instance C_Svr_B1_1 would have a type of `InChannel[Hello|Bye]` and we could have an Effpi type of `In[InChannel[Hello|Bye], ...]` instead of having the channel variable. This however, lost specification on the channel instance, we can't keep track of of the communication through a channel instance. Say we want to specify receiving a message of type *Hello*|*Bye* through the channel instance c_Svr_B1_1, we can easily type it as `In[C_Svr_B1_1, ...]`. However, if we use the previous type, we can receive the message through another channel instance of the same type and the program will compiles, this will however leads to concurrency bugs such as deadlocks as the sender and receiver communicate through different channels.

```
n = len(labels)

def get_type():
    return In[channelName, (param:labels[0]|labels[1]|...|labels[n-1])
               ⇒ continuation.get_type()]

def get_channel_type():
    return InChannel[labels[0]|labels[1]|...|labels[n-1]]

def get_function_body():
    return receive(channelName.first_char_lower() )
            {(param.first_char_lower():labels[0]|labels[1]|...|labels[n-1]) ⇒
            continuation.get_function_body()}
```

Listing 4.8: Effpi type and fuction body generation for InChannel

One may ask: Why do we include the parameter assignment at this communication class but not delegate the responsibility to the continuation. The Effpi type generated would be `In[channelName, continuation.get_type()]` and the correspond function body is `receive(channelName.first_char_lower() ){ continuation.get_function_body()}`. As we can see, in order to do parameter assignment, we need to declare the type of param which is the union of all message label in labels. We design it such that continuation doesn't need to know about the message labels received at this stage for simplicity. This is true if we only have one message label in labels. For multiple message labels, continuation need to know what are the possible message labels in order to perform type matching. The continuation will be instead a function call to function that performs the type matching, this function call doesn't have to know about the message labels, the function that performs the type matching will be generated separately. More details on this will be explained in Branch package and communication class generation later, we do this to reduce the keep the communication classes simple and reduce the dependency between them, making it much easier for modification and extension.

4.3.3 Selection package

`class` **Selection** `extends` **ChannelInstance**

- `var` List[OutChannels] output_channels

`Selection` is the only communication class in the communication package, it is the communication action of making a choice of the message type to send. The output_channels is a non empty list of `OutChannel`s which represent the possible sending actions that can be performed in this selection. The type it's representing here is $\bigvee_{i \in I} \mathbf{o}[c^o[\vee_{j \in I} l_j], l_i, \Pi()et(T_i)]$ or in our modified type syntax, $\Big|_{i \in I} \texttt{Out[OutChannel[}|_{j \in I} l_j], l_i] >>: \Pi()et(T_i)$. The Effpi term that correspond to this type is:

$$
\begin{aligned}
&\textbf{if } b_1 \textbf{ then send}(t_{1,1}, t_{1,2}, t_{1,3}) \textbf{ else}\\
&(\textbf{if } b_2 \textbf{then send}(t_{2,1}, t_{2,2}, t_{2,3}) \textbf{ else}\\
&(... \textbf{ if } b_{n-1} \textbf{ send}(t_{n-1,1}, t_{n-1,2}, t_{n-1,3}) \textbf{ else send}(t_{n,1}, t_{n,2}, t_{n,3}))...)\\
&\textit{where } I = \{1, 2, ..., n\}; \forall k \in I[t_{k,1} : c^o[\vee_{i \in I} l_i] \ \wedge \ t_{k,2} : l_k \ \wedge \ t_{k,3} : \Pi()et(T_k)];\\
&\qquad\qquad b_1, b_2, ..., b_{n-1} \in \mathbb{B}
\end{aligned}
$$

Nested if-else statement is a lot of pain to read, especially in cases where we have many possible message labels, so we shall as well use the built in 'else if' statement in Scala. The modified syntax is:

```
if($b_1$) {send($t_{1,1}$, $t_{1,2}$) >> {$t_{1,3}$}}
else if($b_2$) {send($t_{2,1}$, $t_{2,2}$) >> {$t_{2,3}$}}
...
else if($b_{n-1}$) {send($t_{n-1,1}$, $t_{n-1,2}$) >> {$t_{n-1,3}$}}
else {send($t_{n,1}$, $t_{n,2}$) >> {$t_{n,3}$} }
```

Effpi type and function body generation for Selection is shown in Listing 4.9. We can see how `Selection`'s Effpi type and function is generated from a composition of `OutChannel`s. The Effpi type and function body generated take account into all the possible sending actions. In the function body, the message label to sent is picked by random, the programmer can easily modify the condition.

```
n = len(output_channels)

def get_type():
    return output_channels[0].get_type()|output_channels[1].get_type()|
            ...|output_channels[n-1].get_type()

def get_channel_type():
    return OutChannel[output_channels[0].get_label()|
             output_channels[1].get_label()|...|output_channels[n-1].get_label()]

def get_function_body():
    return val random = Random(currentTime())
            val index = random.nextInt(n)
            if(index == 0){output_channels[0].get_function_body()}
            else if(index == 1){output_channels[1].get_function_body()}
            ...
            else{output_channels[n-1].get_function_body()}
```

Listing 4.9: Effpi type and fuction body generation for Selection

If an OutChannel is used for Selection, it will share the same channelName and channel type with the Selection it resides in. However, this violates our rule where all channel instances must has different channelNames. So we deemed Selection as a wrapper around multiple OutChannels such that the OutChannels are seen as a sending action instead of a channel instance, Selection takes over all the responsibility of a channel instance. get_channel_type() shall not be call on OutChannel instance but Selection instance if the OutChannel is used for Selection. In simpler terms, Selection provides a channel instance that handles all possible message labels that can be sent. For each possible message label, there is a correspond OutChannel in output_channels that represent the sending of the specific message label over the channel. We will follow these rules when generating OutChannels for Selection in the communication class generation section.get_continuation() will return a list of continuations, consisted of continuation of each OutChannel in output_channels, output_channels[0].get_continuation() ∪ output_channels[1].get_continuation() ∪ ... ∪ output_channels[n-1].get_continuation().

We shall now discuss about a mapping choice we made in Section 3.1. Local session type for messages sending, $\oplus_{i\in I}\mathbf{q}!l_i(S_i).T_i$ is mapped to an Effpi type of $\bigvee_{i\in I}\mathbf{o}[c^o[\vee_{j\in I}l_j], l_i, \Pi()et(T_i)]$ but not $\bigvee_{i\in I}\mathbf{o}[c^o[l_i], l_i, \Pi()et(T_i)]$. Both type has the same level of precision but the later will have more channel instances in the local participant's program, it also doesn't give a clear picture on the underlying communication from the local partcipant's perspective. For example, Svr is making a selection to Client, it may send a message of with label *Buy*, *Cancel* or *Negotiate*. The Effpi type (of our syntax) generated for our defined mapping:

$$\begin{aligned}
&\texttt{Out[c\_Svr\_Client\_1, Buy] >> } T_{Buy}| \\
&\texttt{Out[c\_Svr\_Client\_1, Cancel] >> } T_{Cancel}| \\
&\texttt{Out[c\_Svr\_Client\_1, Retry] >> } T_{Retry} \\
\textit{where } &\texttt{c\_Svr\_Client\_1:OutChannel[Buy|Cancel|Retry]}
\end{aligned}$$

The other mapping would give us:

$$\begin{aligned}
&\texttt{Out[c\_Svr\_Client\_1, Buy] >> } T_{Buy}| \\
&\texttt{Out[c\_Svr\_Client\_2, Cancel] >> } T_{Cancel}| \\
&\texttt{Out[c\_Svr\_Client\_3, Retry] >> } T_{Retry} \\
\textit{where } &\texttt{c\_Svr\_Client\_1:OutChannel[Buy]} \\
&\texttt{c\_Svr\_Client\_2:OutChannel[Cancel]} \\
&\texttt{c\_Svr\_Client\_3:OutChannel[Retry]}
\end{aligned}$$

For the later case, two more channel instances are required. Even though the channel mapping will eventually cast a channel of type `Channel[Buy|Cancel|Retry]` to the three channel instances, it is being done on main class level and not known to the local participant. Hence from the local participant's perspective, it is making a choice over three separate channel instance which is much more confusing to the programmer.

4.3.4 Branch package

In this package we handle the type matching of messages received. There are multiple communication classes here as we have to delegate the responsibility of type matching to another function. Otherwise, there will be tons nested type matching within a function, making code unreadable for protocols with huge amount of branching.

`class` **TypeMatchParam** `extends` **ChannelInstance**

`TypeMatchParam` is not an actual channel instance but a variable used to store the message type that needed to be type matched. This variable is then passed down to the type match function to retrieve the actual type of the variable. We extend `ChannelInstance` all the parameters of a function are of type `ChannelInstance`, unifying the message type variable as a `ChannelInstance` makes handling data much easier. It also shares a lot of similarity with channel instances, labels store all the possible labels of this message, channelName represents it's variable name. get_continuation() will return an empty list.

The effpi type and function body for TypeMatchParam is straightforward, shown in Listing 4.10. We can see that it's just a variable overall, it's type variable name is the uppercase of it's channelName and it's function variable name is the lowercase of it. get_channel_type() returns the message type the variable holds, which is the union of all possible message labels.

```
n = len(labels)

def get_type():
    return channelName

def get_channel_type():
    return labels[0]|labels[1]|...|labels[n-1]

def get_function_body():
    return channelName.first_char_lower()
```

Listing 4.10: Effpi type and fuction body generation for TypeMatchParam

`class` **FunctionCall** `extends` **CommunicationBase**

- `var` String functionName
- `var` List[ChannelInstance] channels

As we mentioned, this class is expected to be the continuation for `InChannel` when the message received required type matching, it calls the function that perform the type checking. functionName is the name of the function to be called. channels is the list of `ChannelInstance`s to be passed into the function as parameters. There must be exactly one `TypeMatchParam` in our channels for the function to perform type match, we implement it such that the first element of the channels will be the `TypeMatchParam` instance. This class has an empty continuation list.

```
n = len(channels)
assert n >= 1
assert(channels[0] isInstanceOf TypeMatchParam)

def get_type():
    return functionName[channels[0].channelName.type, channels[1].channelName,
               ..., channels[n-1].channelName]

def get_function_body():
    return functionName.first_char_lower()(channels[0].channelName.first_char_lower(),
                                    channels[1].channelName.first_char_lower(),
                                ..., channels[n-1].channelName.first_char_lower())
```

Listing 4.11: Effpi type and fuction body generation for FunctionCall

The Effpi type and function body of `FunctionCall` is self-explanatory in Listing 4.11. Note that based on our design, each function name has it's first character as a lowercase and that function will correspond to an Effpi type of the same name with the first character being an uppercase. Both the function and it's Effpi type takes in the same channel instances as their parameter in the same sequence. For `TypeMatchParam` in the Effpi type, we need to append a `.type` to it to retrieve it's type in order for type matching to be performed in the Effpi type level.

`class` **TypeMatch** `extends` **CommunicationBase**

- `var` TypeMatchParam param
- `var` List[Label] labels
- `var` List[CommunicationBase] continuations

Here is where the actual operation for type matching resides. param indicates the variable to be type matched, labels contains the non empty list of possible message labels that can be matched and continuations contains the continuation class for each labels matched which will be return on get_continuation(). labels and continuations are ordered in the same mannered such that each label in labels correspond to the continuation class in continuations of the same position.

```
assert(len(labels) == len(continuations))
assert(len(labels) >= 1)
n = len(labels)

def get_type():
    return param.channelName match{
            case labels[0] ⇒ continuations[0].get_type()
            case labels[1] ⇒ continuations[1].get_type()
            ...
            case labels[n-1] ⇒ continuations[n-1].get_type()}

def get_function_body():
    return param.channelName.first_char_lower() match{
            case labels[0] ⇒ continuations[0].get_function_body()
            case labels[1] ⇒ continuations[1].get_function_body()
            ...
            case labels[n-1] ⇒ continuations[n-1].get_function_body()}
```

Listing 4.12: Effpi type and fuction body generation for TypeMatch

The Effpi type we are representing here is $\underline{\textbf{match}}\ \underline{x}\ \{l_i \Rightarrow T_i\}_{i\in I}$ and the correspond term $\textbf{match}\ x\ \{l_i \Rightarrow t_i\}_{i\in I}$ where t_i is a term of type T_i. Since the Effpi type and term are very similar, we shall adopt the same syntax for both cases, using the built it 'case' in Scala. The modified syntax for Effpi type and term are $\underline{x}\ \underline{\textbf{match}}\ \{\underline{\textbf{case}}\ l_i \Rightarrow T_i\}_{i\in I}$ and $x\ \textbf{match}\ \{\textbf{case}\ l_i \Rightarrow t_i\}_{i\in I}$ respectively. The Effpi type and function body generation is shown in Listing 4.12

4.3.5 Recursion package

Recursions can be viewed as a cycle, all we need to know is where the cycle starts and ends and connect them together. This package consisted of two communication classes, `Loop` and `GoTo`.

`class` **Loop** `extends` **CommunicationBase**

- `var` String recVar
- `var` CommunicationBase continuation

This class represent the start of the cycle, or $\mu\mathtt{t}.T$ in Effpi types. recVar represents the recurse variable name `t` and continuation is the following communication of type T. Our syntax for type and term are `Rec[t, T]` and `rec(t){t}` respectively. Unlike channel instance names and `TypeMatchParam`s, recurse variable has the same name in the type and function. get_continuation shall return a list with continuation with it's only element. Also note that no two cycle within the same program shall same recurse variable name, this is again enforced during communication class generation. The effpi type and function body generation of this class is easily understood and shown in Listing 4.13.

```
def get_type():
    return Rec[recVar, continuation.get_type()]

def get_function_body():
    return rec(recVar){continuation.get_function_body()}
```

Listing 4.13: Effpi type and fuction body generation for Loop

`class` **GoTo** `extends` **CommunicationBase**

- `var` String recVar

This class represent the end of a cycle or the recursion variable `t` in terms of Effpi type, it will need to redirect the program to the start of the cycle. recVar represents the recurse variable that points to the start of a cycle. The correspond new syntax for Effpi type and function are `Loop[t]` and `loop(t)` respectively as shown in Listing 4.14, it assumes that exactly one Loop has already used this recVar in it's type/function generation.It has an empty continuation list.

4.3.6 Function package

`class` **Function** `extends` **CommunicationBase**

- `var` Boolean isMain
- `var` String functionName

```
def get_type():
    return Loop[recVar]

def get_function_body():
    return loop(recVar)
```

Listing 4.14: Effpi type and fuction body generation for GoTo

- var Set[ChannelInstance] channels
- var CommunicationBase continuation

Function is the only communication class within the package. We need a main function for each participant to execute the program, a function may have a few sub functions to perform type matching. funcName indicates the name of the function, it should be unique throughout the entire program. isMain says whether the function is the main function of a participant or not, any function that is not a main function is a sub function that is used for type matching. continuation is the communication class that represents the function body and channels is a non empty set of distinct of ChannelInstances where the function takes in as a parameter. We can view channels as the channel instances required by continuation to execute. get_continuation() will return a list with continuation as it's only element.

The Effpi type and function generation for this class is shown in Listing 4.15. We can now see how functions are linked to their Effpi types and how channel instance variables in function bodies correspond to their type variables in the Effpi type. As we mentioned before, each function name will has it's first char as lowercase and it's correspond Effpi type has the same name with first character being an uppercase (See Line 6 and 14). In Line 19 we can see how we bind the function to it's Effpi type. Channel instance variables also has the same naming convention as of functions, we can see from Line 19 - 22 how each channel instance variable is cast into a channel instance type variable in the Effpi type with the correct sequence. In Line 6-10 and 15-18, it is clear that each channel instance variable and their channel instance type variable has it's channel type assigned on the parameter level. Hence we can generate an Input Output Channel(InOutChannel) on the main class level and pass it to the function as a parameter, it will be further cast into the channel instance's channel type which is a subtype of the general InOutChannel. For example a channel type **InOutChannel**[YES|NO] maybe cast into a a channel c_Svr_Client_1 of type OutChannel[YES] for Svr and c_Client_Svr_1 of type InChannel[YES|NO] for Client. Also note from Line 3 and 11, a subfunction's Effpi type must extends the type Process and it's a process of performing type matching. Also, all non main functions should not be called outside of this file hence they must be declared as private.

```
assert(len(channels) > 0)
n = len(channels)
processVar = isMain? "" : "<:  Process"
accessType = isMain? "" : "private"

def get_type():
   return type functionName[
             channels[0].channelName <:  channels[0].get_channel_type()
             channels[1].channelName <:  channels[1].get_channel_type()
             ...
            channels[n-1].channelName <:  channels[n-1].get_channel_type()
            ] processVar = continuation.get_type()

def get_function_body():
 return accessType def functionName.first_char_lower()(
          channels[0].channelName.first_char_lower():channels[0].get_channel_type()
          channels[1].channelName.first_char_lower():channels[1].get_channel_type()
             ...
      channels[n-1].channelName..first_char_lower():channels[n-1].get_channel_type()
            ):functionName[channels[0].channelName.first_char_lower().type
                           channels[1].channelName.first_char_lower().type
                           ...
                           channels[n-1].channelName.first_char_lower().type]
        = { continuation.get_function_body() }
```

Listing 4.15: Effpi type and fuction generation for Function

4.4 Code Generation from EFSM

After introducing EFSM and communication classes, we shall generate communication classes from EFSM. We then show how various part in the program are generated.

4.4.1 EFSM to Communication Classes

We build the communication classes in a depth first search (DFS) manner, starting from the initial state of the EFSM. We shall introduce, a recursive function, `generate_communication_class()` which takes the following four parameters:

- **efsm** - The `EFSM` where **state** resides in, contains helper functions that provide information on each state
- **state** - The `State` in the EFSM which communication class is to be generated
- **visited** - A map from integer to boolean, indicate if a state has been visited in the DFS
- **function_list** - List of `Function`(s) generated

We execute the program, having **state** as the initial state in the EFSM, **function_list** as empty list and **visited** as an empty map. The function will return the communication class generated for the **state** and a list of ChannelInstance(s) required by the communication class. We shall now show in-sequence the various checks the function perform and the corresponding communication class it returns.

Termination State

This is the first check performed by the function and can be easily done so using the built in is_terminal_state() function of **efsm** (a **state** is considered terminal if it has an empty action list). If it is a terminal state, then it indicates the end of the program, no action can be performed further, the function returns early a Termination() instance along an empty channel list. For example, state 3 and 4 from 7 are termination states, where B1 has no more actions after receiving a Confirm or Cancel message from Svr.

Visited State

A **state** is considered visited if it's id is in the **visited**'s keys list(doesn't matter if the value it maps to is true or false). With that being said, a visited state already has it's communication class generated, or in simpler terms it's the start of a cycle, we will have to direct the program to the start of the cycle. For all cycles that start from the same state, they shall share the same recurse variable to be directed back to the same starting state. Hence, we shall consistently name our recurse variable based on the starting state of a cycle. If a **state** is visited, it shall set the value that the **state** id correspond to in **visited** to True. It then returns early a Goto(recVar="Rec_{**efsm**.role}_{**state**.id}") instance alongside an empty channel list, **efsm**.role indicates the participant's name which the **efsm** is representing. If the **state** is not visited yet, we will add the **state**'s id into **visited** as a key with False as it's value. From Figure 8, if we visit state 1 after state 2(B2 received Continue request from Svr and go back to the state where it awaits to send a Bid to Svr), we should expect state 1 to be visited before hence having "1" in our **visited** keys, we then set **visited**[1] = True and the correspond communication class generated is GoTo(recVar="Rec_B2_1"). Note that we include participant name to avoid overlapping of recurse variable name with other participant.

Single Message Sending State

We can use the built-in is_sending_state() function of **efsm** to check if **state** is a sending state, a sending state is where all the actions in the **state**'s action list are sending action. If **state** is a sending state and only has one action in it's action list, it is require to send a message of a specific label (without choice), we refer to it as single message sending state. In Listing 4.16, we demonstrate how communication class is generated for single message sending state through a helper function generate_single_send() that takes the same parameters. We will need to know the communication class correspond to the successive state of the sending action. Since we know there's only one action in the **state**'s action list, we can do so by recursive call on generate_communication_class() on the first element of the list directly. As we mentioned in `OutChannel` description (4.3.2), it can be seen as the action of sending single message instance, the variable assignments to the `OutChannel` instance in Line 4 and 5 are self explanatory, **state**.name is the channel instance name assigned to each non terminal state when we perform channel assignment in Section 4.2. We shall ignore sender and receiver as they are trivial and used mainly for testings. Note that we include the `OutChannel` we built into the list of channels required. An example we can see is through Svr's EFSM in Figure 9, where we try to generate a communication class at state 8, where Svr is required to send a single message of label *Cancel* to B2 to inform him about his failure of bid call and terminate. Our function would return an `OutChannel(`channelName=**state8**.name, labels={Cancel}, continuation=`Termintion())`.

```
1  def generate_single_send(efsm, state, visited, function_list):
2        continuation, channels
3          = generate_communication_class(efsm, state.actions[0].succ, visited, function_list)
4        out_channel = OutChannel(channelName=state.name,
5                     labels={state.actions[0].label}, continuation=continuation)
6        return out_channel, {out_channel} ∪ channels
```

Listing 4.16: Communication Class Generation for Single Message Sending State

Single Message Receiving State

Very similar to single message sending state, we use a built in is_receiving_state() in **efsm** to check if **state** is a receiving state, a receiving state is where all the actions in the **state**'s action list are receiving action. We then check the size of the **state**'s action list to determine if a state is single message receiving state. A single message receiving state is expected to receive a message of single type(no further type matching is required). Again we show in Listing 4.17 how the correspond communication class is generated through a helper generate_single_receive() that takes the same parameters. The only difference with single message sending state is that it generates an `InChannel` instead of an `OutChannel`, `InChannel` requires an extra param variable, which we set to "x". We dont have to know the payload values of the message received to determine the following communications, the programmer can easily modify this variable name in the generated program if they wish to perform some action on the message received.

Selection State

A selection state is where the participant can make a choice on with message label to send. The **state** is a sending state and has more than one action in it's action list. We mentioned in description for `Selection`(4.3.3) that it

```
def generate_single_receive(efsm, state, visited, function_list):
    continuation, channels
      = generate_communication_class(efsm, state.actions[0].succ, visited, function_list)
    in_channel = InChannel(channelName=state.name, param='x',
                    labels={state.actions[0].label}, continuation=continuation)
    return in_channel, {in_channel} ∪ channels
```

Listing 4.17: Communication Class Generation for Single Message Receiving State

is a wrapper over multiple sending actions(`OutChannel` instances). Hence, in Listing 4.18 we show how an `OutChannel` is generated for each sending action of the **state** and how all of them are added to a `Selection`. labels are not used in the Effpi type and function generation for `Selection`, we include it here for clarity purpose since `Selection` extends `ChannelInstance` where labels is required. Observe how all the `OutChannel`s share the same channelName as the `Selection` they resides in. The `OutChannel`s are also not added into the list of channels generated, the list is composed of the channels generated from their successive states and the **state**'s `Selection` instance instead. This is because the `OutChannel`s within `Selection` are seen as possible send actions through a channel instance rather an actual channel instance, `Selection` takes the responsibility as a channel instance. Note that single message sending state can be deemed as a selection state with one `OutChannel` in it's out_channels, we split it into two separate operations to contrast the functionality between `Selection` and `OutChannel`.

```
def generate_selection(efsm, state, visited, function_list):
    out_channels = []
    new_channels = []
    labels = []
    for action in state.actions:
        continuation, channels
          = generate_communication_class(efsm, action.succ, visited, function_list)
        out_channel = OutChannel(channelName=state.name,
                    labels={action.label}, continuation=continuation)
        out_channels = out_channels ∪ out_channel
        new_channels = new_channels ∪ channels
        labels = labels ∪ {action.label}

    selection = Selection(channelName=state.name,
                    labels=labels, output_channels=out_channels)
    return selection, {selection} ∪ channels
```

Listing 4.18: Communication Class Generation for Selection State

Branching State

A **state** is a branching state if it's a receiving state and has more than one action in it's action list. The participant is expected to receive a message which is union-typed, further type matching is required to determine the actual message label received and which continuation to proceed with. The algorithm is shown in Lisiting 4.19. What we are trying to achieve here is fairly simple, once received a message through an `InChannel`, which message's type needed to be further determined, we use a `FunctionCall` to delegate all the remaining responsibilities to another `Function`, the parameters of the `Function` are consisted of the channels required(to continue execute the program) and a `TypeMatchParam` that stores the message received. The `Function` has a function body of `TypeMatch` that match the actual type of the message and proceed with the correspond continuation. We provide a simple naming convention for our `TypeMatchParam` and `Function` as shown in Line 2 and 3. Since both communication classes are only generated once on each type match function, we include the current number of functions in the naming to prevents overlap of naming. We include participant name for function naming but not type match param naming as no two functions in the program shall share the same name. Naming of type match param doesn't affect the program, we name it such that each type match param within the same participant has different name, just for clarity purpose. In Line 4 - 13, we generate the correspond communication class for each actual message label received. The sequence between labels and continuations must be maintained and passed into `TypeMatch` (Line 17 and 18) for it to link each type matched to the correct continuation. In Line 21 - 27, we generate a `Function` that performs the type matching and the `FunctionCalls` that calls the function after receiving a message that required type matching. isMain is always set to false here as the function generated is a helper function used for type matching and not a main function. A `TypeMatchParam` is added into the channel instance list before passing it as a parameter to both classes, this is required for the `FunctionCall` to pass the message received as a parameter to the `Function`. The channels will needed to be deep copied by both classes, the `TypeMatchParam` will be eventually removed from the channel instance list as it is not deemed as an actual channel instance. Finally in Line 30-33, we shall return an `InChannel` with the `FunctionCall` as it's continuation.

An example we can see is through state 2 of B1's EFSM in Figure 7. B1 is waiting a message of type *Confirm* | *Cancel* | *Continue* from Svr, if the actual type received is *Confirm* or *Cancel*, B1 will reach a termination state, otherwise it will goto state 1. Say state 1 is visited but not state 2 and with an initially empty **function_list**, calling generate_communication_class() on state 2 will return:

```
def generate_branching(efsm, state, visited, function_list):
    function_name = "{efsm.role}_{len(function_list) + 1}"
    param_name = "X_{len(function_list) + 1}"
    labels = []
    continuations = []
    new_channels = []
    for action in state.actions:
        continuation, channels
          = generate_communication_class(efsm, action.succ, visited, function_list)
        continuations = continuations ∪ continuation
        new_channels = new_channels ∪ channels
        labels = labels ∪ {action.label}

    type_match_param = TypeMatchParam(channelName=param_name,
                                      labels=labels)
    type_match = TypeMatch(param=type_match_param,
                           labels=labels, continuations=continuations)


    new_channels = {type_match_param} ∪ new_channels
    function_call = FunctionCall(functionName=function_name,
                                 channels=new_channels.copy)
    function = Function(functionName=function_name, continuation=type_match
                        channels=new_channels.copy, isMain=False)
    function_list = function_list ∪ {function}
    new_channels = new_channels \ type_match_param


    in_channel = InChannel(channelName=state.name, param=param_name,
                           labels= labels, continuation=function_call)

    return in_channel, {in_channel} ∪ new_channels
```

Listing 4.19: Communication Class Generation for Branching State

$$\begin{aligned}
&\texttt{InChannel(}\mathsf{channelName} = state2.name, \mathsf{param} = ”X1”,\\
&\mathsf{labels} = \{Cancel, Confirm, Continue\}, \mathsf{continuation} =\\
&\texttt{FunctionCall(}\mathsf{functionName} = ”B1\_1”, \mathsf{channels} = [\\
&\texttt{TypeMatchParam(}\mathsf{channelName} = ”X\_1”,\\
&\mathsf{labels} = \{Cancel, Confirm, Continue\}\texttt{)}]\texttt{))}
\end{aligned}$$

with **function_list** as:

$$[\texttt{Function(}\mathsf{functionName} = \text{"}B1\_1\text{"}, \mathsf{isMain} = False,$$
$$\mathsf{channels} = [type\_match\_param], \mathsf{continuation} =$$
$$\texttt{TypeMatch(}\mathsf{param} = type\_match\_param, \mathsf{labels} = [Cancel, Confirm, Continue],$$
$$\mathsf{continuation} = [\texttt{Termination()}, \texttt{Termination()}, \texttt{GoTo(}\mathsf{recVar} = \text{"}Rec\_B1\_1\text{"}\texttt{)}]\texttt{))}]$$

$$where\ type\_match\_param = \texttt{TypeMatchParam(}\mathsf{channelName} = \text{"}X\_1\text{"},$$
$$\mathsf{labels} = \{Cancel, Confirm, Continue\}\texttt{)}$$

The channel list returned would only contain the InChannel. Also in both FunctionCall and Function, TypeMatchParam is the only ChannelInstance required for the Function to perform type match on the message received and continue with the execution of the program. This is because after the receiving of message at state 2, there's no more new send/receive action (exclude loop) is required hence no more new channels are needed. The functionName is "B1_1" and TypeMatchParam has channelName "X_1", the trailing 1 in both cases is a result of initial size of **function_list**(which is 0). The 1 in recVar="Rec_B1_1" of GoTo indicates the program is returning to state 1.

Start of a Cycle

Now we shall explain here why **visited** is a map instead of a list, when a list of state id is a much simpler way to keep track of which state has been visited through the DFS. The answer is, we not only want to know whether a state is being visited, but also if it's a start of a cycle. If we reach a state that is yet to be visited, we add it into **visited** with **state**'s id as the key and value as False, we then continue DFS on the continuation states, if the **state** is eventually visited again somewhere in it's continuation, the correspond value would be set to True, indicating that **state** is the initial state for one or more cycles. The generated communication class is Loop(recVar="Rec{**efsm.role**}_{**state.id**}", continuation=continuation) where continuation is the send/receive action at **state**. The channels returned are the channels required by continuation. **state**'s id is then removed from **visited**.

Combining the different checks for EFSM states and their correspond output, we summarize it into an algorithm in Listing 4.20. generate_functions() shown in Line 31 - 37 is called by the main class to generate the channels and communication classes for the entire EFSM and store in in a list of Functions. In Line 34-35, the main Function of the participant is generated, it is given the same name as the participant's name, there shall be exactly one main Function per participant, the remaining Functions in the list are for type matching purpose. With this list, we can easily generate the Effpi type and default functions for the participant.

```
def generate_communication_class(efsm, state, visited, function_list):

    if efsm.is_terminal_state(state):
         return Termination()
    if state.id ∈ visited:
         visited[state.id] = True
         return GoTo(recVar="Rec_{efsm.role}_{state.id}")
    visited[state.id] = False

    if efsm.is_sending_state(state):
        if len(efsm.actions) == 1:
            com_class, channels =
                generate_singe_send(efsm, state, visited, function_list)
        else:
            com_class, channels =
                generate_selection(efsm, state, visited, function_list)
     else:
         if len(efsm.actions) == 1:
            com_class, channels =
                generate_singe_receive(efsm, state, visited, function_list)
        else:
            com_class, channels =
                generate_branching(efsm, state, visited, function_list)

     if visited[state.id]:
         com_class =
             Loop(recVar="Rec{efsm.role}_{state.id}", continuation=com_class)
     visited = visited \ state.id
     return com_class, channels

def generate_functions(efsm):
      function_list = []
      com_class, channels =
      generate_communication_class(efsm, efsm.initial_state, {}, function_list)
      function_list = function_list ∪ Function(functionName=efsm.role,
                      continuation=com_class, channels=channels, isMain=True)
      return function_list
```

Listing 4.20: Communication Class Generation Algorithm

4.4.2 Code Generation

Our program consisted of multiple parts, we shall briefly discuss below how the codes for each part is generated.

Import Statements

Multiple libraries and packages needed to be imported for the program to executes. For example, Effpi terms and types needed to be imported from the Effpi package and Date type is required to be imported from `java.util.Date`. These imports are fixed in a template file and is copied to the program file. Some designs such as user interface and error detection(which we will discuss) require more imports and will be copied from another template file.

Type/Function Generation

By using generate_functions() in Lisitng 4.20, we can get a list of Functions for each participant. We then generate the Effpi types and functions for each participant by running get_type() and get_function() respectively on each Function in the list. No further work is needed to be done to link each function to their respective Effpi type as it's being handled by Function with our naming convention. No two function or Effpi type shall share the same name throughout the entire program. We shall leave the discussion of the Effpi type and function generated to our case study.

Case Classes

Labels and their payloads are stored in the program in the form of a case class. Since we assume we can't have labels of the same name with different payloads, we use a set of Labels to keep track of all the message labels in the program when generating communication classes from EFSM. For each sending/ receiving action we encounter, we build a Label instance for the message label and add it to the set. We merge the set of labels from each participant together and add it into the program file in the form of case classes. The payload types are converted into Scala types. Note that in Effpi types, the payload of each message labels are not considered, we only check the case class instance. The payloads are checked on the function level using case class to make sure the required payloads are included during sending of a message. The case class generated for the Art Seller Protocol in Listing 4.1 is shown in Listing 4.21.

```
case class Bid(offer:Int)
case class Cancel()
case class Continue(currPrice:Int)
case class Confirm(finalPrice:Int)
case class Start(basePrice:Int)
```

Listing 4.21: Case Classes Generated for ArtSeller Protocol

Recurse Variables

We can observe from the Recursion package in Section 4.3.5 and communication class algorithm in Listing 4.20 that we need to generate a recurse variable recVar for each cycle of a participant in the program. This recurse variable is used in both type and function level to specify the starting of a cycle, such that we know where to go on the end of the loop. These recurse variables will need to be a part of Effpi, extending the RecVar on the type level and Unit on the function level. Since we may have different recurse varaibles for each programs, our tool keep track of all the recurse variable name generated during communication class generation from EFSMs. It then write them into effpi/src/main/scala/RecurseExtend.scala of package effpi.recurse, which is later imported by the program. The recurse variables generated for our ArtSeller protocol is shown in Listing 4.22.

```
package effpi.recurse

sealed abstract class RecB2_1[A]() extends RecVar[A]("B2_1")
case object RecB2_1 extends RecB2_1[Unit]

sealed abstract class RecB1_1[A]() extends RecVar[A]("B1_1")
case object RecB1_1 extends RecB1_1[Unit]

sealed abstract class RecSvr_2[A]() extends RecVar[A]("Svr_2")
case object RecSvr_2 extends RecSvr_2[Unit]
```

Listing 4.22: Effpi Type Generated for participant B

Main Class

Main Class is where the main function of the program is define, where it execute all the participant's main function in parallel. Here is also where the assignment of actual channel to each participant is generated. We have to combine the channel matching list between each two participant. Then each matching set within the list shall be assigned an actual channel of InOutChannel(c^{io}[] in Effpi type) with it's message type as the union of all channel's label type within the set. Then, each channel instance of a participant's main `Function` shall be assign the correspond channel based on the set it resides in. Note that channel instances are stored in channel matching list in the form of `ChannelInstance` which contains helper function get_labels() to retrieve the channel instance's label types. The algorithm is shown in Listing 4.23 where **match_list** is the combined channel matching list and **channels** is a list of channel instances of each participant.

Again, we shall look at the ArtSeller protocol as an example, the combined channel matching list is:

$$[\{c\_Svr\_B1\_1, c\_B1\_Svr\_1\}, \{c\_Svr\_B1\_2, c\_B1\_Svr\_2\},$$
$$\{c\_Svr\_B1\_3, c\_B1\_Svr\_3\}, \{c\_Svr\_B2\_1, c\_B2\_Svr\_1\}, \{c\_Svr\_B2\_2, c\_B2\_Svr\_2\},$$
$$\{c\_Svr\_B2\_3, c\_Svr\_B2\_4, c\_Svr\_B2\_5, c\_B2\_Svr\_3\}]$$

Hence, there are six channels needed to be generated as there are 6 sets within the list. The main Effpi function's parameter for each participant is shown in Listing 4.24 where the message labels of each channel instance can be seen clearly.

By using our algorithm, we assign a channel to be generated for each set in

```
def assign_channels(match_list, channels):
    channel_assign = {}
    channel_gen=[]
    for channel_set in match_list:
        n = len(channel_set)
        channel = InOutChannel[channel_set[0].get_labels()|
                              channel_set[1].get_labels()|
                              ...
                              channel_set[n-1].get_labels()]
        channel_gen = channel_gen ∪ {(channel_set, channel)}

    for channel in channels
      for channel_set, new_chan in channel_gen:
          if channel ∈ channel_set:
              channel_assign[channel] = new_chan
    return channel_assign
```

Listing 4.23: Channel Generation and Assignment

```
def b1(c_B1_Svr_1: InChannel[Start],
       c_B1_Svr_2: OutChannel[Bid],
       c_B1_Svr_3: InChannel[Continue|Cancel|Confirm])

def b2(c_B2_Svr_1: InChannel[Start],
       c_B2_Svr_2: OutChannel[Bid],
       c_B2_Svr_3: InChannel[Continue|Confirm|Cancel])

def svr(c_Svr_B1_1: OutChannel[Start],
       c_Svr_B2_1: OutChannel[Start],
       c_Svr_B1_2: InChannel[Bid],
       c_Svr_B2_2: InChannel[Bid],
       c_Svr_B1_3: OutChannel[Continue|Cancel|Confirm],
       c_Svr_B2_3: OutChannel[Continue],
       c_Svr_B2_4: OutChannel[Confirm],
       c_Svr_B2_5: OutChannel[Cancel])
```

Listing 4.24: Function parameters for each participant of ArtSeller Protocol

the channel matching list:

$$\begin{aligned} [\{c\_Svr\_B1\_1, c\_B1\_Svr\_1\} &\Rightarrow \texttt{InOutChannel[Start]},\\ \{c\_Svr\_B1\_2, c\_B1\_Svr\_2\} &\Rightarrow \texttt{InOutChannel[Bid]},\\ \{c\_Svr\_B1\_3, c\_B1\_Svr\_3\} &\Rightarrow \texttt{InOutChannel[Continue|Cancel|Confirm]},\\ \{c\_Svr\_B2\_1, c\_B2\_Svr\_1\} &\Rightarrow \texttt{InOutChannel[Start]},\\ \{c\_Svr\_B2\_2, c\_B2\_Svr\_2\} &\Rightarrow \texttt{InOutChannel[Bid]},\\ \{c\_Svr\_B2\_3, c\_Svr\_B2\_4, c\_Svr\_B2\_5, c\_B2\_Svr\_3\}&\\ &\Rightarrow \texttt{InOutChannel[Continue|Cancel|Confirm]}] \end{aligned}$$

We can now generate the six channels and assign them to each channel instance of the participant in accordance the assignment above. The resulting main class generated is shown in Listing 4.25. `InOutChannel` is renamed as

Channel here for simplicity, we can easily check that the channel assignment to each participant correspond to our assignment and the parameters in Listing 4.24. par() and eval() functions evaluates multiple process in parallel. We added a sleep() in the end to allow some time for communication between each participant to take place before we terminate the program. We developed an automated testing system that test our implementation on more than thirty various protocols and make sure the generation process and generated program executes without failure.

```scala
object Main {
  def main(): Unit = main(Array())
  def main(args: Array[String]) = {
    implicit val ps = effpi.system.ProcessSystemRunnerImproved()

    val(c1, c2, c3, c4, c5, c6) =
     (Channel[Start](), Channel[Bid](),
     Channel[Cancel|Continue|Confirm](),
     Channel[Start](),
     Channel[Confirm|Continue|Cancel](),
     Channel[Bid]())

    eval(par(
     svr(c4, c1, c6, c2, c5, c3, c3, c3),
     b2(c1, c2, c3),
     b1(c4, c6, c5)))

    Thread.sleep(1000)
    ps.kill()
  }
}
```

Listing 4.25: Main Class of Generated Program from ArtSeller Protocol

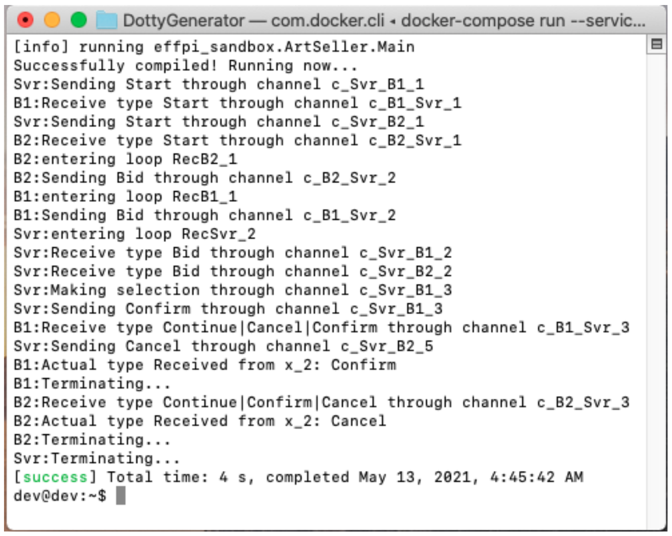
4.5 Design Features

We have added some features to make it more convenient for both the programmers and users. In this section we shall discuss about the design features we included in our code generation.

4.5.1 Print Statements and Indentation

We added indentation in the output function body as we often have loads of nested continuations. The function body may be unreadable for large program with multiple if else/ case statement on the same indentation level. Indentation is not added for Effpi types as they are comparatively much smaller in size and for clarity purpose, we seldom break them into different lines.

We added a print statement during function body generation for each communication. It shows the programmer the real time communication during the execution of the program, the programmer can easily checks and identify any


DottyGenerator — com.docker.cli ◂ docker-compose run --servic...

```
[info] running effpi_sandbox.ArtSeller.Main
Successfully compiled! Running now...
Svr:Sending Start through channel c_Svr_B1_1
B1:Receive type Start through channel c_B1_Svr_1
Svr:Sending Start through channel c_Svr_B2_1
B2:Receive type Start through channel c_B2_Svr_1
B2:entering loop RecB2_1
B2:Sending Bid through channel c_B2_Svr_2
B1:entering loop RecB1_1
B1:Sending Bid through channel c_B1_Svr_2
Svr:entering loop RecSvr_2
Svr:Receive type Bid through channel c_Svr_B1_2
Svr:Receive type Bid through channel c_Svr_B2_2
Svr:Making selection through channel c_Svr_B1_3
Svr:Sending Confirm through channel c_Svr_B1_3
B1:Receive type Continue|Cancel|Confirm through channel c_B1_Svr_3
Svr:Sending Cancel through channel c_Svr_B2_5
B1:Actual type Received from x_2: Confirm
B1:Terminating...
B2:Receive type Continue|Confirm|Cancel through channel c_B2_Svr_3
B2:Actual type Received from x_2: Cancel
B2:Terminating...
Svr:Terminating...
[success] Total time: 4 s, completed May 13, 2021, 4:45:42 AM
dev@dev:~$
```

Figure 10: Output of ArtSeller generated program

possible error. The output of ArtSeller's program is shown in Figure 10. We can easily tell the communication action taken by each participant, through what channel and in which sequence. Note that for the default function body generated, output may be different on each execution as the message sent on choice is picked by random.

4.5.2 File Structure

Having all the code generated in the same file works, but it ends up with an obscure huge chunk of code and also violate our purpose of generating a distributed program. Each participant doesn't have to know about the other participants' Effpi types nor function. All it needs to do is to implement the function body which type checks to the Effpi type assigned to it. Hence, we build a folder for each participant, the folder has the same name as the participant, it contains two files, the Effpi type file and the function file. Again only the function body within the function file shall be modified. A case_class file that stores all message labels are being shared by each participant where they import the message labels for their using. A main file that contains the main class imports the main function from each participant, generates and assign actual channels to each participant, then executes them in parallel. The main file shares the same name and package name as the protocol. The output file structure for the generated ArtSeller's program is shown in Figure 11. The programmer can always choose to generate the program into a single file if they wish.

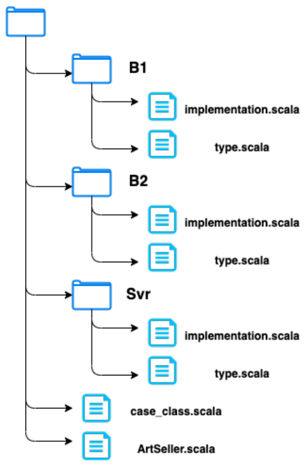


Figure 11: File Structure of ArtSeller generated program

4.5.3 Asynchronous/Synchronous Communication

In asynchronous communication, the sender can proceed with the continuation without waiting for the receiver to receive the message it just sent. The program we generated communicates synchronously by default. However, programmer may wish to generate asynchronous program on occasions where waiting for the receiver to receive the message may affect the performance of the program. For example, a live game server that updates the clients about their status every few seconds, it may have to send out thousands of messages every minute hence it is not ideal to wait for a client to receive their update before sending to another client. Some client may have poor connection or low bandwidth, causing slow receive and eventually congest the entire sending process to other clients. Hence, we provide an option for the programmer to generate asynchronous program using our tool, we do so by generating asynchronous actual channels for the program instead of synchronous ones.

4.5.4 Type/ Function Merging

Large amount of the protocols contain branching states where each branch has a Termination state as it's continuation. This makes the type matching process futile as the continuations are the same, this affect the performance of our tool when it comes to protocols with huge amount of such branching states. Hence, we remove the type matching process from both type and function level if all the continuations of the branch are Termination.

In Listing 4.26 and 4.27, we show the Effpi type and function generated for a participant **q** without type/function merging. **q** is expected to **p** a message of label *Hello*, *Bye* or *Cancel*, we can see that there is an extra subfunction q_2

```
type q_2[ X_2 <: Hello|Bye|Cancel] <: Process =
 X_2 match {
    case Hello => PNil
    case Bye => PNil
    case Cancel => PNil }

type q[ C_q_p_1 <: InChannel[Hello|Bye|Cancel]] =
 In[C_q_p_1, Hello|Bye|Cancel, (X_2:Hello|Bye|Cancel) => q_2[X_2.type]]
```

Listing 4.26: Effpi Type generated for **q** without Type/Function Merging

```
private def q_2(x_2: Hello|Bye|Cancel
   ):q_2[x_2.type] =
      x_2 match {
         case x_2 : Hello => { nil }
         case x_2 : Bye => { nil }
         case x_2 : Cancel => { nil }}

   def q(c_q_p_1: InChannel[Hello|Bye|Cancel]
   ):q[c_q_p_1.type] ={
      receive(c_q_p_1) {
         (x_2:Hello|Bye|Cancel) => q_2(x_2)}}
```

Listing 4.27: Function generated for **q** without Type/Function Merging

```
type q[C_q_p_1 <: InChannel[Hello|Bye|Cancel]] =
 In[C_q_p_1, Hello|Bye|Cancel, (x:Hello|Bye|Cancel) => PNil]
```

Listing 4.28: Effpi Type generated for **q** with Type/Function Merging

```
def q(c_q_p_1: InChannel[Hello|Bye|Cancel]
   ):q[c_q_p_1.type] ={
      receive(c_q_p_1) {
         (x:Hello|Bye|Cancel) => nil}}
```

Listing 4.29: Function generated for **q** with Type/Function Merging

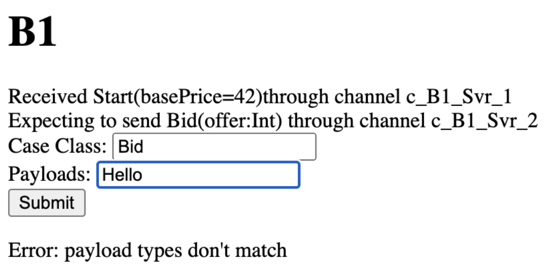
is the function and an extra nested type q_2 in the Effpi type, they are both needed for type matching. With type/ function merging, the generated Effpi type and function are shown in Listing 4.28 and 4.29, there's no nested type nor subfunction in both. We considered to perform type/function merging on branching state where all branches are of the same type instead of limiting them to termination only. However, we decided to abolish this idea as we thought that programmers may wish to add different business logic based on the message labels they received. If we perform merging on non terminal branches, it will confuse the programmer on the underlying communication. Hence, we limit type/ function merging to termination branches and we believe that programmers can perform refactoring based on their preferences for non terminal branches.

4.5.5 User Interface

User interface is one of the main features we implemented in our tool. For server based participants, the programmer can easily implement it's logic on which message label to send on each occasion. However for client based participant(actual human), we may want to prompt input from them during selection/ message sending such that they can choose which message labels and what payload values to send. Hence, we include a functionality such that the programmer can choose which participants in the protocol to generate a user interface for. Of course, they can choose to generate for any number of participants as they wish.

This feature is built on the existing program by adding a simple HTML form that retrieve the message label and payload values from the user. Semaphore is also added to make sure the the messages are only sent when it's asked to. We also make use of the third party HttpExchange and HttpHandler library to host the user interface on localhost. Extra functionality is added for the convenience of the user. Display messages are added to inform the user what message labels and payload values it has received from which user, it also shows what are the message labels and payload types expected to be sent out by the user at that point. After submitting the form, we will check if the message label and payload type is as expected before sending. Otherwise, an error message will be shown to the user, notifying the cause of failure to send the message. By doing so we can make sure that user inserting invalid values will not affect the communication between participants.

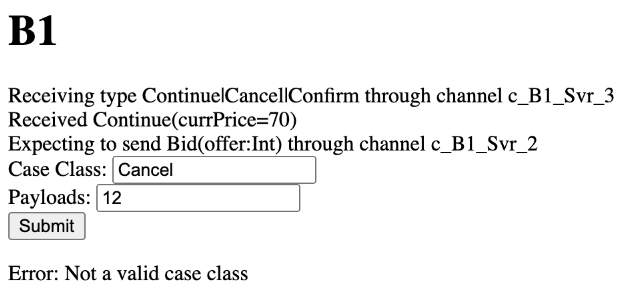
For example, say we design Svr and B2 to be automated in the ArtSeller Protocol (B2 may be a robot bidder with it's own bidding logic), we have B1 as the human bidder. The programmer can generate the user interface for B1 using our tool. Figure 12 and 13 show the user interface of B1, both at different stages of the protocol. Figure 12 shows where B1 received *Start* with base price £42 from Svr and is expected to offer a *Bid*, even though sending the correct message label, B1 sent a string as the payload instead of an integer hence the error message is shown. Figure 13 is at the state where B1 received *Continue* request from Svr, notifying that the new price is £70 and ask B1 to **Bid** again, the error message is shown as B1 sent an unexpected message label *Cancel*.


B1

Received Start(basePrice=42)through channel c_B1_Svr_1
Expecting to send Bid(offer:Int) through channel c_B1_Svr_2
Case Class: Bid
Payloads: Hello

Error: payload types don't match

Figure 12: User interface 1 of B1 in ArtSeller generated program


B1

Receiving type Continue|Cancel|Confirm through channel c_B1_Svr_3
Received Continue(currPrice=70)
Expecting to send Bid(offer:Int) through channel c_B1_Svr_2
Case Class: Cancel
Payloads: 12

Error: Not a valid case class

Figure 13: User interface 2 of B1 in ArtSeller generated program

4.6 Limitations

Our tool aims to convert any valid protocol in nuScr to a functioning Effpi typed Scala program. However, there's some minor limitations for this and we shall discuss below.

4.6.1 Message Label Overloading

We have mentioned this assumption multiple times throughout the project. In nuScr, we are allowed to have more than one message labels of the same name but with different payload types/ names and nuScr is able to differentiate between them. This is however not achievable in our code generation as we represent each message labels and their payloads in the form of a case class. We couldn't have two case class of the same name with different payloads as this will lead to redefinition and reference error. Also, in Effpi type level, we only consider the case class instance without it's payload, we can't differentiate between overloaded case classes. If overload message labels are provided, some of them will get eliminated such that only one message label of that name is reserved, the output program may not send the desired message label as described in the protocol.

4.6.2 Message Label Name

There are some message label name that are reserved for Effpi's term syntax, such as send, receive, eval and par. As we represent message labels as case class in our program, having case class that has the same name as our Effpi term name will cause reference error. Additional check if perform on EFSM generation level to make sure no such names existed in the protocol.

4.6.3 Payloads type

Payload types must be supported by Scala, we currently included String, Integer and Date. Unrecognised types will cause error during code generation as type replacement couldn't find the correspond Scala type to replace.

4.7 Contributions to Dotty

Dotty was introduced to the public in 2015, which makes it a very young compiler. Hence, a lot of work has been undergoing to improve it[49], including resolve compiler bugs, improve compilation speed, embed Scala tools etc. Hence, the Scala/ Dotty community[50] plays a very important role in suggesting new features and reporting issues to the team. Throughout our project, we have encountered bugs in Dotty which led to compilation failures in our program, we have communicated with Dotty members to resolved these issues.

4.7.1 Nested Type Match

Dotty failed to compiled nested type matching. Participant B in NestedMatch protocol (shown in Listing), has a generated Effpi type in Listing 4.30 with default function generated in Listing 4.32. Participant B has a nested type match, it is required to check if a message received is of label *Hello* or *Bye*. If it's Hello, it will need to further receive a message and check if it's of label *Ok* or *Cancel*. However, the compiler managed to compile the type but failed to compile the second type matching function, b_3 to it's Effpi type, B_3, the error log is shown in Figure 14.

```
global protocol NestedMatch(role A, role B) {
    choice at A {
        Hello()   from A to B;
        choice at A {
            Ok()   from A to B;
            Confirm()  from B to A;
        } or { Cancel()  from A to B;  }
    } or { Bye()  from A to B; }
}
```

Listing 4.30: NestedMatch Protocol

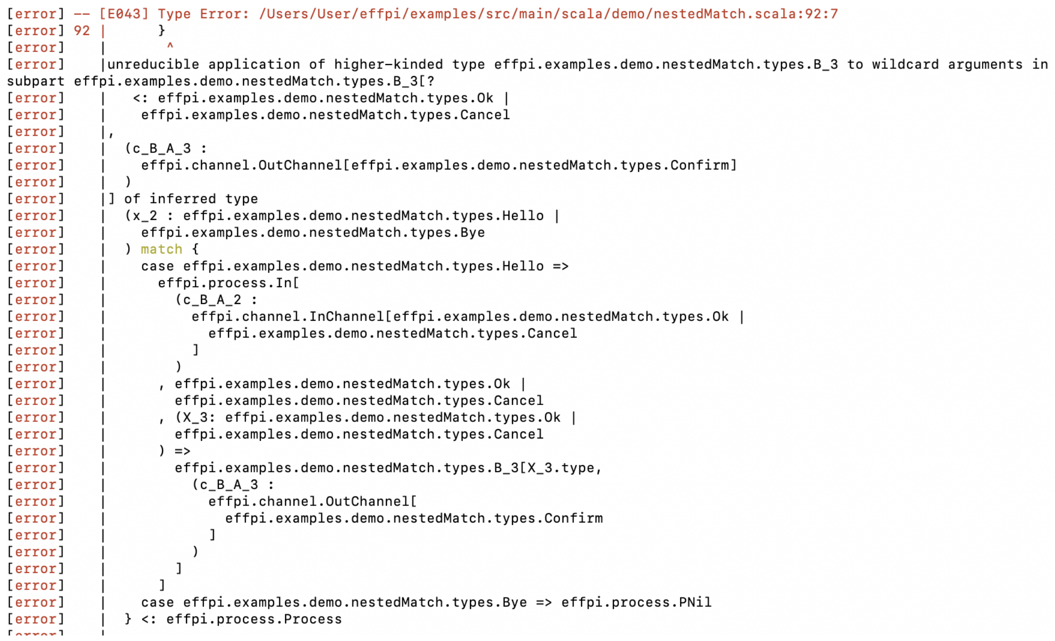
```
type B_3[X_3 <: Ok|Cancel,
          C_B_A_3 <: OutChannel[Confirm]] <: Process =
 X_3 match {
     case Ok => Out[C_B_A_3,Confirm] >>: PNil
     case Cancel => PNil }

type B_2[X_2 <: Hello|Bye,
         C_B_A_2 <: InChannel[Ok|Cancel],
         C_B_A_3 <: OutChannel[Confirm]] <: Process =
 X_2 match {
     case Hello => In[C_B_A_2, Ok|Cancel, (X_3:Ok|Cancel)
                    => B_3[X_3.type,C_B_A_3]]
     case Bye => PNil }

type B[C_B_A_1 <: InChannel[Hello|Bye],
       C_B_A_2 <: InChannel[Ok|Cancel],
       C_B_A_3 <: OutChannel[Confirm]] =
 In[C_B_A_1, Hello|Bye, (X_2:Hello|Bye) => B_2[X_2.type,C_B_A_2,C_B_A_3]]
```

Listing 4.31: Effpi Type Generated for participant B

```
     def b_3(x_3: Ok|Cancel,
             c_B_A_3: OutChannel[Confirm]
             ):B_3[x_3.type,c_B_A_3.type] =
      x_3 match {
         case x_3 : Ok => {
            send(c_B_A_3,Confirm()) >> { nil }}
         case x_3 : Cancel => { nil }}

   def b_2(x_2: Hello|Bye,
           c_B_A_2: InChannel[Ok|Cancel],
           c_B_A_3: OutChannel[Confirm]
          ):B_2[x_2.type,c_B_A_2.type,c_B_A_3.type] =
      x_2 match {
         case x_2 : Hello => {
            receive(c_B_A_2) {
               (x_3:Ok|Cancel) => b_3(x_3,c_B_A_3)}}
         case x_2 : Bye => { nil }}

   def b(c_B_A_1: InChannel[Hello|Bye],
         c_B_A_2: InChannel[Ok|Cancel],
         c_B_A_3: OutChannel[Confirm]
        ):B[c_B_A_1.type,c_B_A_2.type,c_B_A_3.type] ={
      receive(c_B_A_1) {
         (x_2:Hello|Bye) => b_2(x_2,c_B_A_2,c_B_A_3)}}
```

Listing 4.32: Functions Generated for participant B

We reported this bug and was deemed as high priority by the Dotty community. It broke some core functionality and correctness of Dotty: (1) Composition of Match types (2) Composition of Match types with Dependent type (3)Type Aliases. The issue was due to Dotty flagging match type as unreducible, the bug was fixed on the 3.0.0-RC2 release. More information on this bug and fix is available on the bug's Github report page[51].



```
[error] -- [E043] Type Error: /Users/User/effpi/examples/src/main/scala/demo/nestedMatch.scala:92:7
[error] 92 |      }
[error]    |       ^
[error]    |unreducible application of higher-kinded type effpi.examples.demo.nestedMatch.types.B_3 to wildcard arguments in
subpart effpi.examples.demo.nestedMatch.types.B_3[?
[error]    |   <: effpi.examples.demo.nestedMatch.types.Ok |
[error]    |    effpi.examples.demo.nestedMatch.types.Cancel
[error]    |,
[error]    |  (c_B_A_3 :
[error]    |    effpi.channel.OutChannel[effpi.examples.demo.nestedMatch.types.Confirm]
[error]    |  )
[error]    |] of inferred type
[error]    |  (x_2 : effpi.examples.demo.nestedMatch.types.Hello |
[error]    |    effpi.examples.demo.nestedMatch.types.Bye
[error]    |  ) match {
[error]    |    case effpi.examples.demo.nestedMatch.types.Hello =>
[error]    |      effpi.process.In[
[error]    |        (c_B_A_2 :
[error]    |          effpi.channel.InChannel[effpi.examples.demo.nestedMatch.types.Ok |
[error]    |            effpi.examples.demo.nestedMatch.types.Cancel
[error]    |          ]
[error]    |        )
[error]    |      , effpi.examples.demo.nestedMatch.types.Ok |
[error]    |        effpi.examples.demo.nestedMatch.types.Cancel
[error]    |      , (X_3: effpi.examples.demo.nestedMatch.types.Ok |
[error]    |        effpi.examples.demo.nestedMatch.types.Cancel
[error]    |      ) =>
[error]    |        effpi.examples.demo.nestedMatch.types.B_3[X_3.type,
[error]    |          (c_B_A_3 :
[error]    |            effpi.channel.OutChannel[
[error]    |              effpi.examples.demo.nestedMatch.types.Confirm
[error]    |            ]
[error]    |          )
[error]    |        ]
[error]    |      ]
[error]    |    case effpi.examples.demo.nestedMatch.types.Bye => effpi.process.PNil
[error]    |  } <: effpi.process.Process
```

Figure 14: Compilation Error of NestedMatch Protocol

4.7.2 Nested Choice Runtime Error

After fixing nested type matching issue, we encountered another issue with Dotty compiler. The compiler fails to compile the function when there is two or more different nested choice right after a choice of the same sender. The error protocol example BuyerChoice is shown in Listing 4.33. Participant A choose Item1 or Item2 then decides to *Buy1*/*Buy2* or *Cancel1*/*Cancel2* right after. This issue is hardly spotted as there are a few requirements to reproduce it:

1. There must be one or more continuation of the main choice that contains a nested choice.
2. The nested choices must be different with each other. If it's *Buy* or *Cancel* for both nested choices the program compiles successfully.
3. The sender of the nested choice must be the same as the main choice.
4. Nested choice must come right after main choice without any operation in between. Participant A can't have additional communication with B between sending Item1 and the nested choice.

Effpi Type and function generated for participant A is shown in Listing 4.34 and Listing 4.35 respectively. Again the type compiles fine but not the function. A runtime error is thrown on compilation as shown in Figure 15. The error log consisted of more than 1000 lines, where the compiler repeats type comparison

```
global protocol BuyerChoice(role A, role B) {
 choice at A {
        Item1()   from A to B;
        choice at A {
            Buy1()   from A to B;
          } or { Cancel1()   from A to B; }
    } or {
        Item2()   from A to B;
        choice at A {
        Buy2()   from A to B;
        } or { Cancel2()   from A to B;}
    }}
```

Listing 4.33: BuyerChoice Protocol

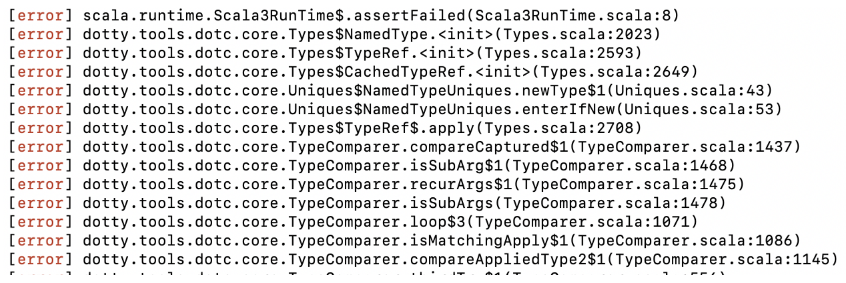
```
type A[ C_A_B_1 <: OutChannel[Item1|Item2],
        C_A_B_2 <: OutChannel[Buy1|Cancel1],
        C_A_B_3 <: OutChannel[Buy2|Cancel2]] =
 ((Out[C_A_B_1,Item1] >>:
 ((Out[C_A_B_2,Buy1] >>: PNil)|(Out[C_A_B_2,Cancel1] >>: PNil)))|
 (Out[C_A_B_1,Item2] >>:
 ((Out[C_A_B_3,Buy2] >>: PNil)|(Out[C_A_B_3,Cancel2] >>: PNil))))
```

Listing 4.34: Effpi Type Generated for participant A

```
def a(c_A_B_1: OutChannel[Item1|Item2],
      c_A_B_2: OutChannel[Buy1|Cancel1],
      c_A_B_3: OutChannel[Buy2|Cancel2]
   ):A[c_A_B_1.type,c_A_B_2.type,c_A_B_3.type] ={
      if(True){
         send(c_A_B_1,Item1()) >> {
            if(True){ send(c_A_B_2,Buy1()) >> { nil }}
            else{ send(c_A_B_2,Cancel1()) >> { nil }}}}
      else{
         send(c_A_B_1,Item2()) >> {
            if(True){ send(c_A_B_3,Buy2()) >> { nil }}
            else{ send(c_A_B_3,Cancel2()) >> { nil }}}}}
```

Listing 4.35: Functions Generated for participant A

on the main if else statement until it exceeds the limited runtime. This is caused by indefinite type comparison on the nested if else statement. A fix is being made for , by alternating the sequence of type comparison for if else statement. More information of the issue and the fix can be seen of the issue's page[52].



```
[error] scala.runtime.Scala3RunTime$.assertFailed(Scala3RunTime.scala:8)
[error] dotty.tools.dotc.core.Types$NamedType.<init>(Types.scala:2023)
[error] dotty.tools.dotc.core.Types$TypeRef.<init>(Types.scala:2593)
[error] dotty.tools.dotc.core.Types$CachedTypeRef.<init>(Types.scala:2649)
[error] dotty.tools.dotc.core.Uniques$NamedTypeUniques.newType$1(Uniques.scala:43)
[error] dotty.tools.dotc.core.Uniques$NamedTypeUniques.enterIfNew(Uniques.scala:53)
[error] dotty.tools.dotc.core.Types$TypeRef$.apply(Types.scala:2708)
[error] dotty.tools.dotc.core.TypeComparer.compareCaptured$1(TypeComparer.scala:1437)
[error] dotty.tools.dotc.core.TypeComparer.isSubArg$1(TypeComparer.scala:1468)
[error] dotty.tools.dotc.core.TypeComparer.recurArgs$1(TypeComparer.scala:1475)
[error] dotty.tools.dotc.core.TypeComparer.isSubArgs(TypeComparer.scala:1478)
[error] dotty.tools.dotc.core.TypeComparer.loop$3(TypeComparer.scala:1071)
[error] dotty.tools.dotc.core.TypeComparer.isMatchingApply$1(TypeComparer.scala:1086)
[error] dotty.tools.dotc.core.TypeComparer.compareAppliedType2$1(TypeComparer.scala:1145)
```

Figure 15: Compilation Error Snippet of BuyerChoice Protocol

Chapter 5

Clone Channel Detection and Elimination

We further extends our program to reduce the number of channel instances used by a local participant and the actual channels generated on the main class level. This can largely increase clarity of the program and slightly improve execution and compilation time for large programs where the same message labels are reused a lot in the protocol. In this chapter, we shall clearly show the problem we are trying to solved. We then introduce the modification/ extension to existing channel instance generation and assignment algorithm. We will then show the improved program.

5.1 Clone Channels

In this section we show four protocols and their generated programs, we discuss what minimisation are we trying to achieve. We shall only show the functions and the main class of the generated program as they clearly showcase how an actual channel/ channel instance can be modified. The correspond Effpi type will be modified in accordance to the minimised program.

5.1.1 Reusing Channel Instances

Instead of generating a new channel instance for each communication action in the local session type, we may reuse a previous channel instance of the exact same type and target. In other words, no channel instance of the same sender, receiver and type shall appear for than once within a participant's program. We can store a map from channel type to channel instance when generating the channel instances between two participants. However, this approach is only complete when no actual channel is assigned to a subtyped OutChannel instance.

The protocol ReuseChannel is shown in Listing 5.1. The generated function for the protocol is shown in Listing 5.2. Channel instances c_A_B_1 and

```
global protocol ReuseChannel(role A, role B) {

    OK() from A to B;
    choice at A{
       Buy() from A to B;
       choice at B{
          Cancel() from A to B;
       } or {
          Buy() from A to B;
       }
    } or {
       Cancel() from A to B;
       OK() from A to B;
    }
}
```

Listing 5.1: ReuseChannel Protocol

```
      def a(
      c_A_B_1: OutChannel[OK],
      c_A_B_2: OutChannel[Buy|Cancel],
      c_A_B_3: OutChannel[Cancel|Buy],
      c_A_B_4: OutChannel[OK]
   ):A[c_A_B_1.type,c_A_B_2.type,c_A_B_3.type,c_A_B_4.type] ={
      send(c_A_B_1,OK()) >> {
         if(True){send(c_A_B_2,Buy()) >> {
                  if(True){ send(c_A_B_3,Cancel()) >> { nil }}
                  else{ send(c_A_B_3,Buy()) >> { nil }}}}
         else{ send(c_A_B_2,Cancel()) >> {
                 send(c_A_B_4,OK()) >> { nil }}}}}

   def b_2(
      x_2: Buy|Cancel,
      c_B_A_3: InChannel[Cancel|Buy],
      c_B_A_4: InChannel[OK]
   ):B_2[x_2.type,c_B_A_3.type,c_B_A_4.type] =
      x_2 match {
         case x_2 : Buy => { receive(c_B_A_3) {(x:Cancel|Buy) => nil }}
         case x_2 : Cancel => { receive(c_B_A_4) {(x:OK) => nil }}}


   def b(
      c_B_A_1: InChannel[OK],
      c_B_A_2: InChannel[Buy|Cancel],
      c_B_A_3: InChannel[Cancel|Buy],
      c_B_A_4: InChannel[OK]
   ):B[c_B_A_1.type,c_B_A_2.type,c_B_A_3.type,c_B_A_4.type] ={
      receive(c_B_A_1) {
         (x:OK) =>
         receive(c_B_A_2) {(x_2:Buy|Cancel) => b_2(x_2,c_B_A_3,c_B_A_4)}}}
```

Listing 5.2: Functions Generated for ReuseChannel Protocol

c_A_B_4 of participant A's function are used to send message of type OK to participant B, hence we can replace c_A_B_4 with c_A_B_1 throughout the program or vice versa. Similarly, both c_A_B_2 with c_A_B_3 are used to send message of type Buy|Cancel to participant B, we can replace of them with another. The

```
val(c1, c2, c3, c4) = (Channel[OK](), Channel[Cancel|Buy](),
                       Channel[Cancel|Buy](), Channel[OK]())

eval(par( b(c1, c2, c3, c4), a(c1, c2, c3, c4)))
```

Listing 5.3: Channels generated for ReuseChannel/DuplicateInOut Protocol

same rules can be applied to participant B, even though it has InChannels instead of OutChannels. By doing so, we can reduce two channel instances for each participant. Looking at the actual channel generation in 5.3, we could generate just two channels instead of four.

5.1.2 InChannel and OutChannel Assignment

We take the channel assignment further on the main class level such that for each two participant and each message type, there can be at most one channel that handles the message transfer. In simpler terms, channel instances that used for sending or receiving the same message label to/from the same participant shall be assigned the same actual channel.

```
global protocol DuplicateInOutChannel(role A, role B) {

    OK() from A to B;
    choice at A{
       Buy() from A to B;
       choice at B{
       Cancel() from B to A;
       } or {
        Buy() from B to A;
       }
    } or {
       Cancel() from A to B;
       OK() from B to A;
    }

}
```

Listing 5.4: DuplicateInOut Protocol

We slightly modified the sender/receiver of ReuseChannel protocol into DuplicateInOut protocol in Listing 5.4. The function generated is shown in Listing 5.5, we only show the channels parameter required by each participant as the function body is trivial. As we may observe, the channel instances for both participant couldn't be replaced as they are of different types. They have same message labels but different InChannel/OutChannel. The channels generated is the same as for ReuseChannel protocol, as in Listing 5.3. We can also

```
def a(
   c_A_B_1: OutChannel[OK],
   c_A_B_2: OutChannel[Buy|Cancel],
   c_A_B_3: InChannel[Cancel|Buy],
   c_A_B_4: InChannel[OK]
):A[c_A_B_1.type,c_A_B_2.type,c_A_B_3.type,c_A_B_4.type] = ...

def b_2(
   x_2: Buy|Cancel,
   c_B_A_3: OutChannel[Cancel|Buy],
   c_B_A_4: OutChannel[OK]
):B_2[x_2.type,c_B_A_3.type,c_B_A_4.type] = ...

def b(
   c_B_A_1: InChannel[OK],
   c_B_A_2: InChannel[Buy|Cancel],
   c_B_A_3: OutChannel[Cancel|Buy],
   c_B_A_4: OutChannel[OK]
):B[c_B_A_1.type,c_B_A_2.type,c_B_A_3.type,c_B_A_4.type] = ...
```

Listing 5.5: Functions Generated for DuplicateInOut

reduce the number of channels generated by generating a single `InOutChannel` for each message label OK and Cancel|Buy and assign them to the participant based on the message label. The same channel may be used to send message from participant A to B or vice versa. It shall be cast into an `InChannel` instance for one participant and an `OutChannel` instance for another during each communication. Our channel instance generation and assignment algorithm guarantee this property as long the channels are assigned correctly on the main class level.

5.1.3 OutChannel casting

We shall now explain why the naive approach of removing duplicate channel instance within a program doesn't work. This is because we may cast actual channels into `OutChannel`s of their subtype. We shall look at the ThirdParty protocol in Listing 5.6 for better explanation. From Line 3-5 and 8-9, we can see that participant A make a selection to participant B then proceeds to send Test1 or Test2 to a third participant C. C has no clue which selection A made to B on the earlier stage, so it's waiting on a single message of type Test1|Test2, but from A's perspective, it's two separate communication. The function generated for the program is given in Listing 5.7, we can see in Line 15 that participant C is waiting for a message of type Test1|Test2 through a single `InChannel` c_C_A_1 while participant A sends Test1 and Test2 through two separate `OutChannel`s, c_A_C_1 and c_A_C_2 respectively(as seen in Line 28 and 34). From the actual channel assignment in Listing 5.8, we can see that these three channel instances are assigned the same channel of type `InOutChannel[Test1|Test2]`.

We have a nested choice from A to B when we call SubThirdParty protocol in Line 6 where eventually A will send Test1 or Test3 to C. Following the same rule, we shall have another two OutChannel instances that send Test1 and Test3 respectively. Now, look back at the functions, we can see that two channel instances of participant A, c_A_C_1 and c_A_C_3 have the same type OutChannel[Test1]. However, different channels are assigned to it as one of it is part of sending message type Test1|Test2 while another is for Test1|Test3. We can't replace the channel instance with each other as we can't assign the same actual channel to it. Of course, if we replace Test3 with Test2 in the protocol, we can unify the channel instances and the channel generated. Note that casting to subtype channel instance only applies to OutChannel but not InChannel, we can have sender choosing which message to send but we can't have receiver waiting on multiple channels.

One may question: Why don't we just replace c_A_C_1 and c_A_C_2 with a single channel instance of type OutChannel[Test1|Test2]? By doing so we can reduce the number of channel instances within the participant and removing duplicate channel instances won't be an issue anymore. We abandoned this design idea as this lose clarity in terms of Effpi type and function. After sending Hello to B, A must send Test1 to C, hence providing an OutChannel[Test1|Test2] instance doesn't make much sense. The participant needs to be very clear about their own responsibility. It's the main class' task to connect the channels between participants.

```
global protocol ThirdParty(role A, role B, role C) {

    choice at A{
       Hello() from A to B;
       Test1() from A to C;
       do SubThirdParty(A, B, C);
    } or {
       Bye() from A to B;
       Test2() from A to C;
    }
}

aux global protocol SubThirdParty(role A, role B, role C){
   choice at A{
           Hello() from A to B;
           Test1() from A to C;
        } or {
           Bye() from A to B;
           Test3() from A to C;
        }
}
```

Listing 5.6: ThirdParty Protocol

We shall now show another example of minimisation before we proceed with the algorithm. As we mentioned, an actual channel may be cast to OutChannel

```
def c_2(
      x_2: Test2|Test1,
      c_C_A_2: InChannel[Test3|Test1]
   ):C_2[x_2.type,c_C_A_2.type] =
      x_2 match {
         case x_2 : Test2 => { nil }
         case x_2 : Test1 => {
            receive(c_C_A_2) {
               (x:Test3|Test1) => nil }}}

   def c(
      c_C_A_1: InChannel[Test2|Test1],
      c_C_A_2: InChannel[Test3|Test1]
   ):C[c_C_A_1.type,c_C_A_2.type] ={
      receive(c_C_A_1) { (x_2:Test2|Test1) => c_2(x_2,c_C_A_2)}}

def a(
      c_A_B_1: OutChannel[Hello|Bye],
      c_A_C_1: OutChannel[Test1],
      c_A_B_2: OutChannel[Hello|Bye],
      c_A_C_3: OutChannel[Test1],
      c_A_C_4: OutChannel[Test3],
      c_A_C_2: OutChannel[Test2]
   ):A[c_A_B_1.type,c_A_C_1.type,c_A_B_2.type,
       c_A_C_3.type,c_A_C_4.type,c_A_C_2.type] ={
      if(True){
         send(c_A_B_1,Hello()) >> {
            send(c_A_C_1,Test1()) >> {
               if(True){send(c_A_B_2,Hello()) >> {
                         send(c_A_C_3,Test1()) >> { nil }}}
               else{ send(c_A_B_2,Bye()) >> {
                     send(c_A_C_4,Test3()) >> { nil }}}}}}
      else{ send(c_A_B_1,Bye()) >> {
            send(c_A_C_2,Test2()) >> { nil }}}}
```

Listing 5.7: Functions Generated for participant A and C in ThirdParty Protocol

```
val(c1, c2, c3, c4) = (Channel[Hello|Bye](),
                       Channel[Hello|Bye](),
                       Channel[Test2|Test1](),
                       Channel[Test3|Test1]())

eval(par( b(c1, c2), a(c1, c3, c2, c4, c4, c3), c(c3, c4)))
```

Listing 5.8: Channels generated for ThirdParty Protocol

instances of it's subtype. We can reduce the number of actual channels generated by combining them on the main class level if the actual channels generated for two participants are of the same type. This is not so straightforward when it comes to replacing channel instances on the local participant level. We shall look at MultipleCast protocol in Listing to further discuss this.

We shall focus on participant's A function as there is where all message sending take place and actual channels get cast onto `OutChannel` instances of it's subtype. Again, we don't cast actual channels to `InChannel` subtype, so the

```
 1  global protocol MultipleCast(role A, role B, role C) {
 2  
 3      choice at A{
 4         Hello() from A to B;
 5         choice at A{
 6            Test1() from A to C;
 7         } or {
 8            Test2() from A to C;
 9         }
10      } or {
11         Ok() from A to B;
12         Test4() from A to C;
13      } or {
14         Bye() from A to B;
15         Test3() from A to C;
16         do SubCast(A, B, C);
17      }
18  }
19  
20  aux global protocol SubCast(role A, role B, role C){
21       choice at A {
22            Hello() from A to B;
23            Test1() from A to C;
24         } or {
25            Bye() from A to B;
26            choice at A{
27                Test2() from A to C;
28               } or {
29                Test3() from A to C;
30                 }
31         } or {
32           Ok() from A to B;
33           Test4() from A to C;
34         }
35  }
```

Listing 5.9: MultipleCast Protocol

channel instances of participant B and C cannot be further replaced/ removed. The actual channnels generation and assignment is shown in Listing 5.10. We can see that actual channel c3 and c4 used to pass message of type Bye|Hello|Ok between A and B can be combine into one. So does c1 and c2 that is used to pass message of type Test1|Test2|Test3|Test4 between A and C. We shall focus on the later as c1 and c2 get cast into an `OutChannel` subtype. The function generated for participant A is shown in Listing 5.11. From both the protocol(Line 5-9, 12, 15) and function(Line 14-15, 18, 20), we can observe that the first actual channel of type `InOutChannel[Test1|Test2|Test3|Test4]` that is used to perform the first send of message of label Test1|Test2|Test3|Test4 from A to C, gets cast into three `OutChannel` instances of message labels Test1|Test2, Test3 and Test4 respectively. We then perform a second sending of message of label Test1|Test2|Test3|Test4 from A to C in a nested choice, as shown in Sub-

Cast protocol. Even though the actual channel assign for this communication has the same type as of the previous channel, however it's cast to different channel instance types, which are three `OutChannel` instances of message labels Test2|Test3, Test1 and Test4 respectively. Since we combined the two actual channels on the main class level, we know that these six `OutChannel` instances are assigned the same actual channel. Hence it is obvious that we can combine the two channels, c_A_C_2 and c_A_C_6, of type `OutChannel[Test4]` into one. We will have to keep track of the channel instance type that has been generated for each actual channel to know if there's a generated channel instance that can be reuse.

```
val(c1, c2, c3, c4) = (Channel[Test4|Test1|Test3|Test2](),
                       Channel[Test1|Test2|Test3|Test4](),
                       Channel[Bye|Hello|Ok](),
                       Channel[Bye|Hello|Ok]())

eval(par( c(c1, c2), a(c3, c1, c1, c1, c4, c2, c2, c2),
          b(c3, c4)))
```

Listing 5.10: Channels generated for MultipleCast Protocol

```
 def a(
      c_A_B_1: OutChannel[Hello|Ok|Bye],
      c_A_C_1: OutChannel[Test1|Test2],
      c_A_C_2: OutChannel[Test4],
      c_A_C_3: OutChannel[Test3],
      c_A_B_2: OutChannel[Hello|Bye|Ok],
      c_A_C_4: OutChannel[Test1],
      c_A_C_5: OutChannel[Test2|Test3],
      c_A_C_6: OutChannel[Test4]
   ):A[c_A_B_1.type,c_A_C_1.type,c_A_C_2.type,c_A_C_3.type,
       c_A_B_2.type,c_A_C_4.type,c_A_C_5.type,c_A_C_6.type] ={
      if(True){
         send(c_A_B_1,Hello()) >> {
            if(True){ send(c_A_C_1,Test1()) >> { nil }}
            else{ send(c_A_C_1,Test2()) >> { nil }}}}
      else if(True){
            send(c_A_B_1,Ok()) >> {
                send(c_A_C_2,Test4()) >> { nil }}}
      else{ send(c_A_B_1,Bye()) >> {
                send(c_A_C_3,Test3()) >> {
                   if(True){
                      send(c_A_B_2,Hello()) >> {
                         send(c_A_C_4,Test1()) >> { nil }}}
                   else if(True){
                      send(c_A_B_2,Bye()) >> {
                         if(True){ send(c_A_C_5,Test2()) >> { nil }}
                         else{ send(c_A_C_5,Test3()) >> { nil }}}}
                   else{
                      send(c_A_B_2,Ok()) >> {
                         send(c_A_C_6,Test4()) >> { nil }}}}}}}
```

Listing 5.11: Functions Generated for A in MultipleCast Protocol

5.2 Channel Generation and Matching with Clone Detection and Elimination

In this section we shall extend the channel generation and matching we defined in Section 3.2 earlier to include channel detection and elimination. First, we assign a channel instance for each communication action within a local session type with the gen() method we defined in Definition 3.2.1 earlier. Then, for every two participants, we perform these four steps in order to get the final channel-matching list: Clone Channel Instance Detection, Clone Channel Instance Replacement, Channel Matching, In/Out Channel Merging. Channel Matching is defined earlier in 3.2.2, we shall discuss in the following subsections about the rest.

5.2.1 Clone Channel Instance Detection

For every two participants, we shall define the only channel instances required between them. For each `OutChannel` instance, we will need to know the actual `InOutChannel`'s message labels that is going to be assigned to it. We utilise the fact that `InOutChannel`s are not cast into `InChannel`s subtype. So, for each `InChannel` instance that is used to receive a message from an `OutChannel` instance, the `OutChannel` instance shall be assigned an actual `InOutChannel` of message labels the same as the `InChannel` instance. For example, participant **p** has an `OutChannel[Hello]` instance that is used to send message of label Hello to participant **q**, which is expected to received by their `InChannel[Hello|Bye|Ok]` instance. We can deduce that the `OutChannel[Hello]` instance will be eventually assigned to an actual `InOutChannel[Hello|bye|Ok]`. For every two participant, if two channel instances have the exact same type, target(must be one of the two participants) and the type of the actual `InOutChannel` assigning to it, then they must be the same channel instance.

We define *getchan* in Definition 5.2.1 in a same manner we did for *match* in Def 3.2.2. It takes the local session type with channel instances of two participants and an empty list, it then returns a list consisting of the unique channel instances required for the communication between these two participants, one for each target, channel instance type and actual assigned channel type. It done so by checking all the communication between the two participants and build the channel instance list, if there is already a channel instance of the same type, target and type of actual channel assigned, it will ignore it, otherwise it will add it into the list. The approach it took to handle third party participants and continuations between send and receive are the same as of *match*. The list it output consisted of three element tuples, the first is the message labels of the actual `InOutChannel` assigned to the channel instance, second is the target of the channel instance, third is the channel instance which contains the name and type of the channel instance. The example above would give us a tuple of (Hello|Bye|Ok, **q**, c_P_Q_1 : $c^o[Hello]$) in the list. This indicates that any communication that has a channel instance of type $c^o[Hello]$, target **q**(sender must be **p** as we only consider about two participants) and actual channel assigned of type $c^{io}[Hello|Bye|Ok]$, it must be assigned the channel instance c_P_Q_1 : $c^o[Hello]$.

Definition 5.2.1 (Generation of Unique Channel Instance List From Local Session Type With Channel Instances)**.** *For two participants* **p** *and* **q** *with local session type with channel instances* T_p *and* T_q *respectively, we generate the list of unique channel instances between them, getChan(*T_p*,* T_q*, []) by induction on the structure of local session type with channel instances:*

$$\begin{array}{rcl} getChan(\texttt{end}, \_\_, replaceList) & = & replaceList \\ getChan(t, \_\_, replaceList) & = & replaceList \\ getChan(\mu\mathbf{t}.T, T', replaceList) & = & getChan(T, T', replaceList) \end{array}$$

$$getChan((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)), replaceList) \quad =$$

$$\begin{cases} \begin{cases} subChanGen(K, replaceList) & (L_2, \mathbf{r}, (c_3 : c^o[L_1])) \in replaceList \; \& (L_2, \mathbf{r'}, (c_4 : c^i[L_2])) \in replaceList \\ subChanGen(K, (L_2, \mathbf{r'}, (c_2 : c^i[L_2])) : replaceList) & (L_2, \mathbf{r}, (c_3 : c^o[L_1])) \in replaceList \\ subChanGen(K, (L_2, \mathbf{r}, (c_1 : c^o[L_1])) : replaceList) & (L_2, \mathbf{r'}, (c_4 : c^i[L_2])) \in replaceList \\ subChanGen(K, (L_2, \mathbf{r}, (c_1 : c^o[L_1])) : (L_2, \mathbf{r'}, (c_2 : c^i[L_2])) : replaceList) & otherwise \end{cases} & \begin{array}{l} (\mathbf{r} = \mathbf{p} \;\&\; \mathbf{r'} = \mathbf{q} \\ \vee \mathbf{r} = \mathbf{q} \;\&\; \mathbf{r'} = \mathbf{p}) \\ \& K = [(T_i, T_j) | \\ i \in I \wedge j \in J \wedge \\ l_i(S_i) = l_j(S_j)] \\ \& A = c1 : c^o[L_1] \\ \& B = c2 : c^i[L_2] \end{array} \\ subChanGen([(T_i, (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B))) | i \in I], replaceList) & \mathbf{r} \neq \mathbf{p} \;\&\; \mathbf{r} \neq \mathbf{q} \\ subChanGen([(\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), T_j) | j \in J], replaceList) & \mathbf{r'} \neq \mathbf{p} \;\&\; \mathbf{r'} \neq \mathbf{q} \\ undefined & otherwise \end{cases}$$

$$chanGen((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), (\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B)), replaceList) \quad =$$

$$\begin{cases} subChanGen([(T_i, (\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B))) | i \in I], replaceList) & \mathbf{r} \neq \mathbf{p} \;\&\; \mathbf{r} \neq \mathbf{q} \\ subChanGen([(\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), T_j) | j \in J], replaceList) & \mathbf{r'} \neq \mathbf{p} \;\&\; \mathbf{r'} \neq \mathbf{q} \\ undefined & otherwise \end{cases}$$

$$chanGen((\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)), replaceList) \quad =$$

$$\begin{cases} subChanGen([(T_i, (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B))) | i \in I], replaceList) & \mathbf{r} \neq \mathbf{p} \;\&\; \mathbf{r} \neq \mathbf{q} \\ subChanGen([(\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), T_j) | j \in J], replaceList) & \mathbf{r'} \neq \mathbf{p} \;\&\; \mathbf{r'} \neq \mathbf{q} \\ undefined & otherwise \end{cases}$$

$$getChan(T, T', replaceList) \quad = \quad getChan(T', T, replaceList)$$

c1, c2, c3, c4 are arbitary channel names, L_1 and L_2 are arbitary message types

The definition of subChanGen(K, replaceList) is given as:

$$subChanGen(K, replaceList) =$$
$$\begin{cases} chanGen(T_i, T_j, subChanGen(K', replaceList)) & K = (T_i, T_j) : K' \\ replaceList & K = [] \\ undefined & otherwise \end{cases}$$

5.2.2 Clone Channel Instance Replacement

After generating the list of unique channel instances between each two participants, we will have to replace the channel instance for each communication action in the session types with their unique instance in the list. We did it on a separate function as we are returning two local session types with channel instances here while we were returning a list before. The local session types input are not commutative here as it will affect the order of session type in the output. The *replace* method is shown in Definition 5.2.2 where we took two local session types with channel instances and the unique channel instance list we generated from the previous step, it then return the two local session types with their channel instances replaced to their unique channel instances. For each send receive action, we look up for the correspond channel instance in the list and replace it for the session type. *subReplace* is a helper function to help with the continuations and third party participants. It takes a local session type with channel instances which channel instance needed to be replaced and a list of local session types with channel instances which it need to perform *replace* with. It will return the local session type with it's channel instance replaced, those replacement made to the session types in the list are ignored.

Definition 5.2.2 (Clone Channel Instance Replace)**.** *For two participants* **p** *and* **q** *with local session type with channel instances T_p and T_q respectively, with the unique channel instance list replaceList, we replace the channel instances in both session types, replace(T_p, T_q, replaceList) by induction on the structure of local session type with channel instances:*

$$\begin{aligned} replace(\mathtt{end}, T, replaceList) &\quad = \quad (\mathtt{end}, T) \\ replace(T, \mathtt{end}, replaceList) &\quad = \quad (T, \mathtt{end}) \\ replace(t, T, replaceList) &\quad = \quad (t, T) \\ replace(T, t, replaceList) &\quad = \quad (T, t) \end{aligned}$$

$$
\begin{aligned}
\mathit{replace}(T', \mu\mathbf{t}.T, \mathit{replaceList}) &= (T'', \mu\mathbf{t}.T''') \\
\mathit{where} T'', T''' &= \mathit{replace}(T', T, \mathit{replaceList}) \\
\mathit{replace}(\mu\mathbf{t}.T, T', \mathit{replaceList}) &= (\mu\mathbf{t}.T'', T''') \\
\mathit{where} T'', T''' &= \mathit{replace}(T, T', \mathit{replaceList})
\end{aligned}
$$

$$
\mathit{replace}((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)), \mathit{replaceList}) \quad =
$$

$$
\begin{cases}
\begin{array}{l}
((\oplus_{i\in I}\mathbf{r}!l_i(S_i).\mathit{subReplace}(T_i, [T_j | j \in J \wedge l_i(S_i) = l_j(S_j)], \\
\qquad\qquad\qquad \mathit{replaceList}) : (c3 : c^o[L_1])), \\
(\&_{j\in J}\mathbf{r'}?l_j(S_j).\mathit{subReplace}(T_j, [T_i | i \in I \wedge l_i(S_i) = l_j(S_j)], \\
\qquad\qquad\qquad \mathit{replaceList}) : (c4 : c^i[L_2])))
\end{array}
&
\begin{array}{l}
(\mathbf{r} = \mathbf{p} \ \& \ \mathbf{r'} = \mathbf{q} \\
\vee \ \mathbf{r} = \mathbf{q} \ \& \ \mathbf{r'} = \mathbf{p}) \\
\& A = c1 : c^o[L_1] \\
\& B = c2 : c^i[L_2] \\
\& (L_2, \mathbf{r}, c3 : c^o[L_1]) \\
\in \mathit{replaceList} \\
\& (L_2, \mathbf{r'}, c4 : c^i[L_2]) \\
\in \mathit{replaceList}
\end{array}
\\[2ex]
\begin{array}{l}
((\oplus_{i\in I}\mathbf{r}!l_i(S_i).\mathit{subReplace}(T_i, [(\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B))], \\
\qquad\qquad\qquad \mathit{replaceList}) : (A)), \\
\mathit{subReplace}((\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)), [T_i | i \in I], \mathit{replaceList}))
\end{array}
& \mathbf{r} \neq \mathbf{p} \ \& \ \mathbf{r} \neq \mathbf{q}
\\[2ex]
\begin{array}{l}
(\mathit{subReplace}((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), [T_j | j \in J], \mathit{replaceList}), \\
(\&_{j\in J}\mathbf{r'}?l_j(S_j).\mathit{subReplace}(T_j, [(\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A))], \\
\qquad\qquad\qquad \mathit{replaceList}) : (B)))
\end{array}
& \mathbf{r'} \neq \mathbf{p} \ \& \ \mathbf{r'} \neq \mathbf{q}
\\[2ex]
\mathit{undefined} & \mathit{otherwise}
\end{cases}
$$

$$
\mathit{replace}((\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), (\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B)), \mathit{replaceList}) \quad =
$$

$$
\begin{cases}
\begin{array}{l}
((\&_{i\in I}\mathbf{r}?l_i(S_i).\mathit{subReplace}(T_i, [T_j | j \in J \wedge l_i(S_i) = l_j(S_j)], \\
\qquad\qquad\qquad \mathit{replaceList}) : (c3 : c^i[L_1])), \\
(\oplus_{j\in J}\mathbf{r'}!l_j(S_j).\mathit{subReplace}(T_j, [T_i | i \in I \wedge l_i(S_i) = l_j(S_j)], \\
\qquad\qquad\qquad \mathit{replaceList}) : (c4 : c^o[L_2])))
\end{array}
&
\begin{array}{l}
(\mathbf{r} = \mathbf{p} \ \& \ \mathbf{r'} = \mathbf{q} \\
\vee \ \mathbf{r} = \mathbf{q} \ \& \ \mathbf{r'} = \mathbf{p}) \\
\& A = c1 : c^i[L_1] \\
\& B = c2 : c^o[L_2] \\
\& (L_1, \mathbf{r}, c3 : c^i[L_1]) \\
\in \mathit{replaceList} \\
\& (L_1, \mathbf{r'}, c4 : c^o[L_2]) \\
\in \mathit{replaceList}
\end{array}
\\[2ex]
\begin{array}{l}
((\&_{i\in I}\mathbf{r}?l_i(S_i).\mathit{subReplace}(T_i, [(\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B))], \\
\qquad\qquad\qquad \mathit{replaceList}) : (A)), \\
\mathit{subReplace}((\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B)), [T_j | j \in J], \mathit{replaceList}))
\end{array}
& \mathbf{r} \neq \mathbf{p} \ \& \ \mathbf{r} \neq \mathbf{q}
\\[2ex]
\begin{array}{l}
(\mathit{subReplace}((\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), [T_j | j \in J], \mathit{replaceList}), \\
(\oplus_{j\in J}\mathbf{r'}!l_j(S_j).\mathit{subReplace}(T_j, [(\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A))], \\
\qquad\qquad\qquad \mathit{replaceList}) : (B)))
\end{array}
& \mathbf{r'} \neq \mathbf{p} \ \& \ \mathbf{r'} \neq \mathbf{q}
\\[2ex]
\mathit{undefined} & \mathit{otherwise}
\end{cases}
$$

$$replace((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), (\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B)), replaceList) \quad =$$

$$\begin{cases} ((\oplus_{i\in I}\mathbf{r}!l_i(S_i).subReplace(T_i, [(\oplus_{j\in J}\mathbf{r'}!l_j(S_j).T_j : (B))], \\ \qquad\qquad replaceList) : (A)), & \mathbf{r} \neq \mathbf{p}\ \&\ \mathbf{r} \neq \mathbf{q} \\ subReplace((\oplus_{j\in J}\mathbf{r}!l_j(S_j).T_j : (B)), [T_i | i \in I], replaceList)) & \\ & \\ (subReplace((\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A)), [T_j | j \in J], replaceList), & \\ (\oplus_{j\in J}\mathbf{r'}!l_j(S_j).subReplace(T_j, [(\oplus_{i\in I}\mathbf{r}!l_i(S_i).T_i : (A))], & \mathbf{r'} \neq \mathbf{p}\ \&\ \mathbf{r'} \neq \mathbf{q} \\ \qquad\qquad replaceList) : (B))) & \\ undefined & otherwise \end{cases}$$

$$replace((\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), (\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B)), replaceList) \quad =$$

$$\begin{cases} ((\&_{i\in I}\mathbf{r}?l_i(S_i).subReplace(T_i, [(\&_{j\in J}\mathbf{r'}?l_j(S_j).T_j : (B))], \\ \qquad\qquad replaceList) : (A)), & \mathbf{r} \neq \mathbf{p}\ \&\ \mathbf{r} \neq \mathbf{q} \\ subReplace((\&_{j\in J}\mathbf{r}?l_j(S_j).T_j : (B)), [T_i | i \in I], replaceList)) & \\ & \\ (subReplace((\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A)), [T_j | j \in J], replaceList), & \\ (\&_{j\in J}\mathbf{r'}?l_j(S_j).subReplace(T_j, [(\&_{i\in I}\mathbf{r}?l_i(S_i).T_i : (A))], & \mathbf{r'} \neq \mathbf{p}\ \&\ \mathbf{r'} \neq \mathbf{q} \\ \qquad\qquad replaceList) : (B))) & \\ undefined & otherwise \end{cases}$$

c1, c2, c3, c4 are arbitary channel names, L_1 and L_2 are arbitary message types

The definition of subReplace(T, K, replaceList) is given as:

$$subReplace(T, K, replaceList) =$$
$$\begin{cases} T' & K = T'' : K' \\ T & K = [] \\ undefined & otherwise \end{cases}$$
$$where\ T', \_\_ = replace(subReplace(T, K', replaceList), T'', replaceList)$$

5.2.3 Merging Channel of Same Message Label

As we mentioned, one actual channel can be generated between two participant for both way communication of same message label. After generating the channel matching list between two participants, we further combine any set within the list that has `InChannel` of the same type as `InChannel`'s message type are the same as of the actual `InOutChannel` assign to the set. This shown in the *mergeIn* method as defined in Definition 5.2.3

Definition 5.2.3 (Merge InChannel of same type)**.** *We further combine actual channels that are used for passing of the same message type between two participants by matching InChannels of the same message type. This is done through mergeIn(channelSet), where channelSet is the generated channel matching list between the two participant.*

$$mergeIn(channelSet) = \begin{cases} mergeIn((X \cup Y) : channelSet') & \begin{array}{l} channelSet = X : Y : channelSet' \\ \& \ \exists L[\ (\_ : c^i[L]) \in X \ \wedge \ (\_ : c^i[L]) \in Y] \end{array} \\ \\ channelSet & otherwise \end{cases}$$

5.2.4 Algorithm for Channel Name Assignment and Matching

We shall modify the channel name assignment and matching algorithm we defined earlier in Section 4.2 to include clone channel detection and elimination. The modified algorithm is shown as in Listing 5.12, get_chan_name() and remove_third_party() are not modified and hence not shown. We remove the original channel_map and replace it with a generated channels map per participant. For each participant, this map mapped the actual assigned channel message labels and channel instance type to a channel instance name. At each state, the participant can look up the map to check whether a channel instance of it's desired type has been generated and can be reused. Otherwise, a new channel instance will needed to be generated and insert to the map. All the information of channel instance type. labels of actual channel assigned can be easily extracted from the states. After having the full generated channel instances map of the two participants, we can easily check that the channel instances that are within the same assigned channel labels shall be assign the same actual channel. This algorithm combined all the steps we mentioned earlier in one go. In our implementation, we generate the program using clone channel elimination by default, but we also provide an option for the users to generate without channel elimination.

```
 def match_state(role1, role2, state1, state 2, visited,
                 count1, count2, generated1, generated2){
     states1 = remove_third_party(role2, {state1}, visited)
     states2 = remove_third_party(role1, {state2}, visited)

     if states1 == {} or states2 == {}:
         return count1, count2

     channelSet = {}
     assigned_labels = get_assigned_labels(states1 ⋃ states2)
     state_labels = [action.label |  action ∈ state.actions]

     for state in states1 ⋃ states2:
         if state.name = null:
             if state ∈ states1:
                 channelName, count1 =
                   get_chan(state_labels,assigned_labels,count1,
                             generated1,role1,role2)
             else:
                 channelName, count2 =
                   get_chan(state_labels,assigned_labels,count2,
                             generated2,role2,role1)
             state.name = channelName
     visited = visited ⋃ states1 ⋃ states2

     for action1 in [state.action | state ∈ states1]:
        for action2 in [state.action | state ∈ states2]:
            if action1.label == action2.label:
              count1, count2 = match_state(role1, role2,
              action1.succ, action2.succ, visited,
                 count1, count2, generated1, generated2)

     visited = visited \ states1 \ states2
     return count1, count2
 }

def get_assigned_labels(states):
   for state in states:
       if state.is_receive_state():
           return [action.label |  action ∈ state.actions]
   return []

def get_chan(state_labels, labels, count,
             generated, role1, role2):
   send_rec = state.is_receive_state() ? "Rec" : "Send"
   channelName = generated[labels][send_rec][state_labels]
      if channelName = null:
         channelName = gen_chan_name(role1, role2, count1)
         generated[labels][send_rec][state_labels] = channelName
         count = count + 1
   return channelName, count
```

Listing 5.12: Algorithm for Channel Name Assignment and Matching for Two Participant With Clone Channel Detection and Elimination

Chapter 6

Error Handling

In real life communications, receivers are not guarantee to receive a message. Sender may have crashed or the message is lost during transmission. The receiver may wait forever to receive the message and deadlock will occur. Hence, we provide a solution for this situation by introducing an alternative protocol that will be activated by the receiver if it waited for a message for more than a certain period. The communications we use in this chapter are asynchronous as we allow the sender to send to a crashed receiver. Global protocol for error handling is still work in progress, hence we shall demonstrate them in terms of local session types throughout this chapter.

6.1 Local Session Type With Error Handling

We modified the local session type in Definition 2.1.2 to include error handling. The extended definition is given in Definition 6.1.1. The syntax of this local session type is slightly different from the original. We include the error handling branch for all message receiving such that we will have to consider the case of failure during every receive. The error handling type receives messages in the same manner as of normal message receiving/branching. The only difference is that it provides a **crash** branch that contains the alternative continuation if the expected message is not received after certain period. Note that we only introduce error handling for message receiving but not message sending as we assume the communications are asynchronous. Sender can proceed with the continuation immediately after sending a message without waiting for the receiver to receive it. Receiver however, is required to wait until the message is received as continuation may depend of the message label/value received. Also we assume that failure is caused by process failure instead of bad communication. This being said, a crashed process is assumed to be crashed throughout the remaining communications, we shouldn't waste time communicating with the process in the future. However, we don't restrict participants from attempting (and failing) to communicate with the crashed participant. `stop`is included to distinguish between a crash and a benign end, for the sake of semantics. It is intended as an internal system affair and the programmer cannot write it.

Definition 6.1.1 (Local Session Types With Error Handling)**.** *Grammar for Local Session Types With Error Handling, ranged over by T:*

$$
\begin{array}{lll}
T ::= & \texttt{end} & \textit{End (Success State)} \\
 & \mid\ \texttt{stop} & \textit{End (Crash State)} \\
 & \mid\ t & \textit{Type Variable} \\
 & \mid\ \mu \mathbf{t}.T & \textit{Recursive Type} \\
 & \mid\ \mathsf{q} \oplus \{l_i(S_i).T_i\}_{i \in I} & \textit{Message Sending/ Selection} \\
 & \mid\ \mathsf{p}\&\{l_i(S_i).T_i, \textbf{\textit{crash}} : T_{crash}\}_{i \in I} & \textit{Message Receiving/} \\
 & & \textit{Branching With Error Handling}
\end{array}
$$

where:

– p *and* q *are the participants/parties*

– $\forall i, j \in I\{i \neq j \implies l_i \neq l_j\}$

– *Recursion is guarded and session types are closed unless specified*

6.1.1 Digraph and EFSM

In our implementation, local session types are represented in the form of digraphs and efsms. We shall then generate the code for the program based on them. We extend the digraph and EFSM we introduced in Chapter 4 earlier to include error handling. First of all, we introduce the symbol '#' in digraph to represent the error handling action. Then, we introduce a an error_detection branch and ReceiveErrorAction in EFSM state to represent a state that contains error handling in the digraph. This can be better understood with a simple example, consider a participant p with local session type with error handling, T_p below:

$$
T_p = \mathsf{q}\& \begin{cases} Confirm(int).\texttt{end} \\ Cancel().\texttt{end} \\ \textbf{crash} : \texttt{end} \end{cases}
$$

The digraph and EFSM generated for T_p are shown in Listing 6.1 and Figure 16 respectively. Note that we only consider about the origin and target state in the error handling branch, labels and payloads are not required. We provide a built-in is_error_detection_state method in EFSM to check whether a state is an error handling state. A state with error handling must be a receive state. Although we provide an option in our implementation where receive may or may not contain error handling, we shall not discuss about it in this chapter as we assume every receives contain error handling.

```
digraph G {
  0;1;2;3;

  0 -> 1 [label="q?Confirm(number)", ];
  0 -> 2 [label="Svr!Cancel()", ];
  0 -> 3 [label="Svr#", ];
  }
```

Listing 6.1: digraph generated for T_p

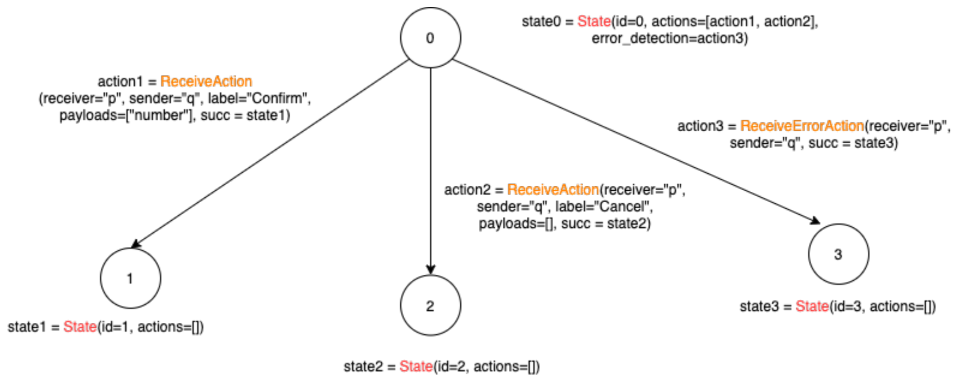

0
state0 = State(id=0, actions=[action1, action2], error_detection=action3)
action1 = ReceiveAction (receiver="p", sender="q", label="Confirm", payloads=["number"], succ = state1)
action3 = ReceiveErrorAction(receiver="p", sender="q", succ = state3)
action2 = ReceiveAction(receiver="p", sender="q", label="Cancel", payloads=[], succ = state2)
1
state1 = State(id=1, actions=[])
2
state2 = State(id=2, actions=[])
3
state3 = State(id=3, actions=[])

Figure 16: EFSM generated from T_p

6.2 Effpi with Error Handling

In this section, we shall explain how we modified Effpi to include error handling. We first extend the syntax and type of Effpi in Section 6.2.1, then we implement receiving with error handling in Section 6.2.2. Finally in Section, we define a projection from local session type with error handling to Effpi types with error handling.

6.2.1 Type and Syntax

We extend the syntax and types of Effpi defined in Section 2.2.2 to include error handling. The extended type and process syntax for error handling in Effpi are shown in Figure 17 and 18 respectively. In $\mathbf{i_{err}}$[S,T,U], S represents the channel type which this receive action is taking place, T indicates the continuation type once the message is received and U indicates the continuation type of the error-handling continuation. Similarly in $\texttt{recv}_{\texttt{err}}(t, t', t'')$, t in the channel term, t' is the normal continuation term and t" is the continuation in the error handling branch.

$$S, T, U, \ldots ::= \quad \ldots$$
$$\mid \mathbf{i_{err}}[S, T, U] \quad \text{Input With Error Handling}$$

Figure 17: Extended Type of Effpi $\lambda^{\pi}_{\leq}$ With Error Handling

$$\mathbb{P} \ni p, q, \ldots ::= \quad \ldots$$
$$\mid \quad \mathtt{recv}_{\mathtt{err}}(t, t', t'') \quad \text{Message Receiving With Error Handling}$$

Figure 18: Extended Syntax of Effpi $\lambda^{\pi}_{\leq}$ Process With Error Handling

6.2.2 Implementation

We extended the code in the process file `ProcessDSL.scala` in Effpi to include error handling. Implementation for both error handling type `InErr` and syntax `receiveErr` are shown in Listing 6.2 and Listing 6.3 respectively. Both the type and syntax are implemented in manner to the normal receive, the only difference is that we provide an extra `Process` which is the continuation to be activated if the receive has failed. Also, the timeout parameter is explicit instead of implicit, so that the programmers can specify the duration the input channel shall wait for the message before it throw a RuntimeException, receiveErr shall proceed with the alternative continuation once it caught the error. This can be seen from the extended method of eval() in Listing 6.4, which is called during the execution of Effpi program to handle different communication types. If we successfully receive a message, we will call eval on the original continuation, otherwise we will call eval on the error handling process if a RuntimeException is caught. We can clearly see how `InErr` (modified type syntax) is design based on the Effpi type $\mathbf{i_{err}}$. The last parameter input in `receiveErr` is the duration timeout which is not included in the Effpi syntax $\mathtt{receive}_{\mathtt{err}}$ as it doesn't modify communication sequences, it may be helpful to add in the future when we work on the semantics.

```
case class InErr[C <: InChannel[A], A,
                 P <: A => Process,
                 Q <: Throwable => Process]
                 (channel : C, cont : P, err : Q, timeout : Duration)
                       extends Process
```

Listing 6.2: Error handling type InErr in Effpi

```
def receiveErr[C <: InChannel[A], A,
               P <: Process, F <: A => P,
               Q <:  Process, G <: Throwable => Q]
         (c : C)(cont : F, err : G, timeout : Duration) : InErr[C,A,F,G] =
                          InErr[C,A,F,G](c, cont, err, timeout)
```

Listing 6.3: Error handling receive function receiveErr in Effpi

```
def eval(env: Map[ProcVar[_], (_) => Process],
         lp: List[() => Process], p: Process): Unit =
 p match {
   ...
   case ie: InErr[_,_,_,_] => {
        receiveMessageErr(ie) match {
          case Success(cont) => eval(env, lp, cont)
          case Failure(e) => eval(env, lp, ie.err(e))
          }
    }
   ...
}

def receiveMessageErr(i : InErr[_,_,_,_]) : Try[Process] = {
  try {
    val ic = i.channel
    val v: Any = ic.receive()(i.timeout)
    val cont = if (ic.synchronous) {
      val (v2, ack) = v.asInstanceOf[Tuple2[Any, OutChannel[Unit]]]
      ack.send(())
      i.cont.asInstanceOf[Any => Process](v2)
    } else {
      i.cont.asInstanceOf[Any => Process](v)
    }
    Success(cont)
  } catch {
    case e: RuntimeException => Failure(e)
  }
}
```

Listing 6.4: Execution of InErr in eval in Effpi

6.2.3 Local Session Types With Error Handling to Effpi Types With Error Handling

We modify our projection from local session type to Effpi type in Definition 3.1.1 to include error handling. The modified projection is shown in Definition 6.2.1. We don't include stophere as it is not part of the program that the programmer can implement.

Definition 6.2.1 (Projection from Local Session Types with Error Handling to Effpi Types With Error Handling)**.** *We define the local Effpi type (with error handling) for a participant with local session type (with error handling) T, et(T), by structural induction on T, as follows:*

$$\begin{aligned} et(\mathsf{end}) &= \boldsymbol{nil} \\ et(t) &= t \end{aligned}$$

$$et(\mu \mathbf{t}.T) = \mu \mathbf{t}.et(T)$$

$$et(\mathbf{q} \oplus \{l_i(S_i).T_i\}_{i\in I}) = \bigvee_{i\in I} \boldsymbol{o}[c^o[\vee_{j\in I} l_j], l_i, \Pi()et(T_i)]$$

$$et(\mathbf{p}\&\{l_i(S_i).T_i, \boldsymbol{crash}: T_{crash}\}_{i\in I}) =$$
$$\boldsymbol{i_{err}}[c^i[\vee_{i\in I} l_i], \Pi(\underline{x} : \vee_{i\in I} l_i)\ \underline{\boldsymbol{match}}\ \underline{x}\ \{l_i \Rightarrow et(T_i)\}_{i\in I}, \Pi(\underline{x}' : \boldsymbol{err})et(T_{crash})]$$

where $\underline{x}$ and $\underline{x}'$ are fresh variables

6.3 Channel Handling

With an extra error handling branch, we need a very different approach to match the channel instances between participants. We shall then explain the `InChannel` subtype issue and propose a solution to it.

6.3.1 Channel Name Assignment and Matching

We extend the channel assignment method in Definition 3.2.1 to include error handling. The extended method is given in Definition 6.3.1. Channel instance assignment for error handling is done in the same manner as of message receiving by assigning a new input channel instance to it.

Definition 6.3.1 (Channel Instance Generation for Local Session Type With Error Handling)**.** *The grammar of local session type with error and channel instances, gen(T) is:*

$$
\begin{array}{rll}
gen(T) ::= & \mathtt{end} & \textit{Termination} \\
 & \mid\ t & \textit{Type Variable} \\
 & \mid\ \mu \mathbf{t}.T & \textit{Recursive Type} \\
 & \mid\ (\mathbf{q} \oplus \{l_i(S_i).T_i\}_{i\in I} : (\mathbb{C} : U)) & \textit{Message Sending} \\
 & \mid\ (\mathbf{p}\&\{l_i(S_i).T_i, \boldsymbol{crash}: T_{crash}\}_{i\in I} : (\mathbb{C} : U)) & \textit{Message Receiving}
\end{array}
$$

where $\mathbb{C} = \{a,b,c...\}$ is the infinite set of channel name variables in Effpi and U is the channel type in Effpi

The channel instance generation for a local session type with error handling T of a participant $\mathbf{r}$, gen(T), is defined by induction on the structure of T:

$$
\begin{array}{rl}
gen(\mathtt{end}) & = \mathtt{end} \\
gen(t) & = t \\
gen(\mu \mathbf{t}.T) & = \mu \mathbf{t}.gen(T) \\
gen(\mathbf{q} \oplus \{l_i(S_i).T_i\}_{i\in I}) & = (\oplus_{i\in I} \mathbf{q}!l_i(S_i).gen(T_i) : (a' : c^o[\vee_{i\in I} l_i]))
\end{array}
$$

$$gen(\mathbf{p}\&\{l_i(S_i).T_i, \textbf{\textit{crash}} : T_{crash}\}_{i\in I}) \quad = \quad (\mathbf{p}\&\{l_i(S_i).gen(T_i), \textbf{\textit{crash}} : gen(T_{crash})\}_{i\in I} : (b' : c^i[\vee_{i\in I} l_i]))$$

where a' and b' are fresh channel name in Effpi, generated with a' = chan() and b' = chan() each time.

Definition 6.3.2 (Matching Channel Instance in Local Session Type With Error Handling an Channel Instances)**.** *types is a list of participants along their respective local session type with error handling and channel instances. We define the set of matching channels of this program, matchErr(types, [], []) by induction on the structure of local session type with error handling and channel instances:*

$$matchErr(types, channelSet, crashedSet) \quad = $$

$$\begin{cases}
channelSet & \forall(\mathbf{p}, T_p) \in types \\ & [T_p = \mathtt{end} \vee T_p = t \vee \mathbf{p} \in crashedSet] \\[2ex]
matchErr((\mathbf{p}, T_p) : types', & types = (\mathbf{p}, \mu\mathbf{t}.T_p) : types' \\ \quad channelSet, crashedSet) & \\[2ex]
subMatchErr([(\mathbf{p} : T_i) : types' | i \in I], & types = (\mathbf{p}, T_p) : types' \;\&\; \mathbf{p} \in crashedSet \\ \quad channelSet, crashedSet) & \&\; T_p = (\mathbf{q} \oplus \{l_i(S_i).T_i\}_{i\in I} : (A)) \\[2ex]
matchErr((\mathbf{p} : T_p) : (\mathbf{q}, T_{crash}) : types', & types = (\mathbf{p}, T_p) : (\mathbf{q}, T_q) : types' \\ \quad channelSet, crashedSet) & \&\; (T_p = \mathtt{end} \vee \mathbf{p} \in crashedSet) \\ & \&\; T_q = (\mathbf{p}\&\{l_i(S_i).T_i, \textbf{\textit{crash}} : T_{crash}\}_{i\in I} : (A)) \\[2ex]
subMatchErr([(\mathbf{p} : T_p) : (\mathbf{q}, T_i) : types' | i \in I], & types = (\mathbf{p}, T_p) : (\mathbf{q}, T_q) : types' \\ \quad channelSet, crashedSet) & \&\; T_p = \mathtt{end} \;\&\; T_q = (\mathbf{p} \oplus \{l_i(S_i).T_i\}_{i\in I} : (A)) \\[2ex]
subMatchErr(types'', matchChan(A, B, & types = (\mathbf{p}, T_p) : (\mathbf{q}, T_q) : types' \\ \quad subMatchErr(types''', channelSet, \mathbf{q} : & \&\; T_p = (\mathbf{q}\&\{l_i(S_i).T_i, \textbf{\textit{crash}} : T_{crash}\}_{i\in I} : (A)) \\ \quad crashedSet)), crashedSet) & \&\; T_q = (\mathbf{p} \oplus \{l_j(S_j).T_j\}_{j\in J} : (B)) \\ & \&\; types'' = [(\mathbf{p}, T_i) : (\mathbf{q}, T_j) : types' | \\ & \quad i \in I \wedge j \in J \wedge l_i(S_i) = l_j(S_j)] \\ & \&\; types''' = [(\mathbf{p}, T_{crash}) : (\mathbf{q}, T_j) : types' | j \in J] \\[2ex]
undefined & otherwise
\end{cases}$$

where

- *all channel instances have different channel name.*

- *channelSet is a list of set of channel names, each channel name is unique within each set.*
- **p** *and* **q** *are arbitary participants and* T_p *and* T_q *are their local session type with error handling and channel instances respectively.*

The definition of matchChan(A, B, channelSet) is given as:

$$matchChan(A, B, channelSet) \quad = \\ \begin{cases} (X \cup Y) : channelSet' & channelSet = X : Y : channelSet' \\ & \& \ A \in X \ \& \ B \in Y \\ \\ channelSet & channelSet = X : channelSet' \\ & \& \ A \in X \ \& \ B \in X \\ \\ (A : B : X) : channelSet' & channelSet = X : channelSet' \\ & \& \ (A \in X \vee B \in X) \\ \\ \{A, B\} : channelSet & otherwise \end{cases}$$

The definition of subMatchErr(K, channelSet, crashedSet) is given as:

$$subMatchErr(K, channelSet) = \\ \begin{cases} matchErr(types, subMatchErr(K', channelSet), crashedSet) & K = types : K' \\ channelSet & K = [] \\ undefined & otherwise \end{cases}$$

The method for channel instance matching is given in Definition 6.3.2. This approach is very different from our previous channel instance matching approaches. Before this, our method (see Definition 3.2.2) matched the channel instances between two participants and ignored any communications with third participant. This method is unapplicable when error handling is taken into consideration as we assume that failure is permanent, we need to know the sequence of each message being received. For example, say we are trying to match the local session type with error handling and channel instances between two participants, **p** and **q**. At this point, they are both waiting to receive a message from a third participant **r** and the message receiving included error handling such that if they didn't receive from **r** after certain period, they assume that **r** has crashed and proceed with the alternative continuation. We can't just ignore the local session type of **r** as we need to know the sequence of sending from the perspective of **r**. If **r** sent to **q** first before **p**, then if the sending process from **r** to **q** failed, we assume that **r** has crashed and the sending process from **r** to **p** must failed too. However, if the sending process from **r** to **q** succeed, we can't guarantee that the sending process from **r** to **q** will success. Also, since communications are asynchronous, participants are allowed to send to a crashed participant even though the crashed participant may not receive the message.

Looking at our definition now, we need to keep track of the local session type with error handling and channel instances of each participant in the protocol. We also used a *crashedSet* to keep track of the participants that have already crashed. We shall now explain the ideation of our channel instance matching method. If all the participants has crashed, terminated or restarting a cycle(go to), no more channel instance matching is expected to take place, we shall return the current channel-matching list. If a participant is trying to receive (with error handling) from a crashed or terminated participant, it will proceed to the alternative error handling continuation as the message will never be received. If a participant has crashed but it's trying to send to some other participant, we shall ignore the sending actions and continue to match the continuations. The same action is taken if a participant is trying to send to a terminated participant. Now we shall consider the case of sending and receiving of message to/from each other, including the case where we send to a crashed participant. Since all message receiving contains error handling, in addition to the normal send/receive matching, we will have to add the sender into the crashed list and match the continuation in the error handling branch of the receiver to all continuations of the sender. This is because the receiver may still send the sender messages even though it knows the sender has crashed.

We formalised an algorithm for the definition in Listing 6.5. *counts* is a map that keep tracks of the channel instance name generated between every two participants. *is_end*() check if all the states are visited, terminated or crashed. *update_channel_map*() generate a new channel instance name for the communication and update the *counts*, it then update the channelMap in the same manner we defined in the channel matching algorithm in Definition 4.5. Clone channel detection and elimination was integrated into our implementation. We don't provide a formalisation for it as it is very similar to as of the formalisation we provided in Section 5.2, just that we have to keep track of the channel instances we generated for each two participant.

6.3.2 InChannel Subtype

We shall explain an issue with `InChannel` that come up during channel instance generation and matching. We do this by showing an example, consisting of three

```
def match_states(states, counts, crashed, visited, channelMap):

    if is_end(states, visited, crashed):
       return

    for (role1, state1) ∈ states:
      for (role2, state2) ∈ states and role1 ≠ role2:

          if state1 ∈ visited or state2 ∈ visited:
              continue

          else if state1.is_send and (role1 ∈ crashed or
                  state1.target == role2 and role2 ∈ crashed):
                visited = visited ∪ {state1}
                for act ∈ state1.actions:
                   match_states(states[role1]=act.succ, counts,
                                crashed, visited, channelMap)
                visited = visited \ state1
                return

           else if state1.is_err_handle
                   and state1.target == role2
                   and role2 ∈ crashed:
               visited = visited ∪ {state1}
               match_states(states[role1]=state1.err.succ,counts,
                                crashed, visited, channelMap)
               visited = visited \ state1
               return

           else if state1.is_send and state1.target == role 2
                   and state2.is_receive
                   and state2.target == role1:

                 update_channel_map(role1, role2, state1,
                                    state2, counts, channelMap)

                 visited = visited ∪ {state1, state2}

                 for action1 ∈ state1.actions:
                     for action2 ∈ state2.actions:
                       if action1.label == action2.label:
                           states[role1] = action1.succ
                           states[role2] = action2.succ
                           match_states(states,counts,
                                crashed, visited, channelMap)

                 if state2.is_err_handle:
                   states[role2]=state2.err.succ
                   for act in state1.actions:
                     match_states(states[role1]=act.succ,counts,
                                crashed, visited, channelMap)

                 visited = visited \ state1 \ state2
```

Listing 6.5: Algorithm for Channel Name Assignment and Matching for Participants with Error Handling

participants p, q and r, with their respective local session type T_p, T_q and T_r:

$$T_p = \mathsf{q} \oplus hi().\mathsf{r} \oplus hi().\mathtt{end}$$

$$T_q = \mathsf{p}\& \begin{cases} hi().\mathsf{r}\& \begin{cases} Success().\mathtt{end} \\ Fail().\mathtt{end} \\ \mathbf{crash} : \mathtt{end} \end{cases} \\ \mathbf{crash} : Fail().\mathtt{end} \end{cases}$$

$$T_r = \mathsf{p}\& \begin{cases} hi().\mathsf{q} \oplus Success().\mathtt{end} \\ \mathbf{crash} : \mathsf{q} \oplus Fail().\mathtt{end} \end{cases}$$

As we may observe p sends *hi* to q then r. r sends to q *Success* or *Fail* depends on whether it successfully received *hi* from p. Again, these local session types are expected to generated from a global session type (which is still work in progress), so they are correct in the sense there is no communication mismatch. As we mentioned, failure is permanent, a crashed participant shall remain crashed forever. Since, p sends to q before r, if q didn't receive *hi* from p, it will expect that r doesn't receive it as well. Hence, in the first **crash** branch of T_q, it only expect to receive *Fail* from r. If q successfully received *hi* from p, it doesn't know whether if r will receive it, hence it expect to receive *Success* or *Fail* from r. As we can observe from the local session types, it is obvious that the exchange of *Success* and *Fail* between q and r occurs through the same actual channel of type `InOutChannel[Success|Fail]`. However, based on our channel instance generating algorithm, we will generate a channel instance of type `InChannel[Fail]` for the receiving of *Fail* in the first **crash** branch of T_q. `InChannel[Fail]` is however not a subtype of `InOutChannel[Success|Fail]` hence cannot be cast to.

To resolve this issue, we must have `InChannel[Fail]` as `InChannel[Success|Fail]` instead. To do so, we modify the EFSM to allow receive of *Success* that will lead to an "Unreachable" Termination state, which will throw an exception if the program ever reach the state. In order to identify such states where extra receive actions need to be added, we perform the normal channel instance matching method without clone channel elimination, we then check the channel matching list if there's any `InChannel` that receive fewer message labels than other channel instances in the set. Any state that use these channel instances are the states that require extra receive action, we then add the respective receive action for those states respectively, which all lead to the "Unreachable" Termination state. We add a boolean variable `is_unreachable` to the `State` class to identify the "Unreachable" Termination state. We then re-perform channel instance generation and matching on the modified EFSM again (with or without clone channel detection). This same EFSM is used for communication class generation later. We extend the `Termination` class to include a boolean variable is_unreachable to indicate if it is representing a "Unreachable" Termination state. The modified function generation is shown in Listing 6.6, we will throw an exception. We extend the communication class generation algorithm in Listing 4.20, shown in Listing 6.7 to take ”Unreachable” Termination state into consideration. Note that these states should never be reached as our local session types are well defined and assuming failure is permanent, we only do this to solve the `InChannel` subtyping issue, actual generated code can be seen in the simple broadcast example in Section 6.5.1.

```
def get_type():
    return PNil

def get_function_body():
    if is_unreachable:
         return throw Exception("error") nil
    return nil
```

Listing 6.6: Effpi type and fuction body generation for Termination with Unreachable state

```
def generate_communication_class(efsm, state, visited, function_list):
    ...
     if efsm.is_terminal_state(state):
         return Termination(is_unreachable=state.is_unreachable)
    ...
```

Listing 6.7: Communication Class Generation Algorithm With Unreachable State

6.4 Communication Package and Code Generation

In this section we shall discuss how we extended the Channel Instance package in Section to include a new InErrChannel class that generate the Effpi type and function for receiving with error handling. Then we discuss how we identify a receive with error handling from an EFSM state and generate the codes for it.

6.4.1 Channel Instances package with Error Handling

`class` **InErrChannel** `extends` **ChannelInstance**

- `var` String param
- `var` CommunicationBase continuation
- `var` CommunicationBase errContinuation

We choose to extend the Channel Instance package as receive with error handling is also an action of receive single message. The InErrChannel name here may be slightly misleading as we still use an `InChannel` to receive message, just that we call `receiveErr` on it which will catch the RuntimeException thrown. Different from the InChannel class, it takes an extra errContinuation as it's parameter which describe the continuation in the error handling branch. The Effpi type it's representing here is $\mathbf{i_{err}}[c^i[\vee_{i\in I} l_i], \Pi(\underline{x} : \vee_{i\in I} l_i)U, \Pi(\underline{x}' : \mathbf{err})T]$. The correspond replaced type syntax is $\texttt{In}[\texttt{InErrChannel}[|_{i\in I} l_i], (\underline{x} : |_{i\in I} l_i) \Rightarrow U, (\underline{x}' : \texttt{err}) \Rightarrow T]$. As we mentioned,the original Effpi term that correspond to this type is $\mathbf{recv}(t_1, t_2, t_3)$ where t_1 is a channel instance of type $c^i[\vee_{i\in I} l_i]$, t_2 is the normal continuation term of type $\Pi(\underline{x} : \vee_{i\in I} l_i)U$ and t_3 is the error branch continuation term of type $\Pi(\underline{x}' : \mathbf{err})T$. the modified syntax is $\texttt{receive}(t_1)(\{t_2\}, \{t_3\}, \_)$. The duration for each timeout period is 5 seconds by default, the programmer can later modify it based on requirements.

```
n = len(labels)

def get_type():
    return In[channelName, (param:labels[0]|labels[1]|...|labels[n-1])
               ⇒ continuation.get_type(),
               (err:Throwable) ⇒ errContinuation.get_type() ]

def get_channel_type():
    return InChannel[labels[0]|labels[1]|...|labels[n-1]]

def get_function_body():
    return receiveErr(channelName.first_char_lower() )
           ({(param.first_char_lower():labels[0]|labels[1]|...|labels[n-1]) ⇒
           continuation.get_function_body()},
           {(err:Throwable) ⇒ errContinuation.get_function_body()},
             Duration("5 seconds"))
```

Listing 6.8: Effpi type and fuction body generation for InErrChannel

6.4.2 Communication Class from EFSM

We shall identify states in the EFSM where error handling is required and generate the respective communication classes for it. The idea is very simple, we replace all receiving states in Section 4.4.1 with receiving states with error handling, all usage of `InChannel` class shall be replaced with `InErrChannel` class.

Single Message Receiving State with Error Handling

This is the replacement state to the original Single Message Receiving State. The only difference is that we replace `InChannel` with `InErrChannel`, evaluate the error continuation, pass it as the errContinuation variable to the class and combine the channel instances generated. The generation algorithm modified from Listing 4.17 is shown in Listing 6.9.

```
def generate_single_receive_w_err(efsm, state, visited, function_list):
      continuation, channels1
        = generate_communication_class(efsm, state.actions[0].succ, visited, function_list)
      errContinuation, channels2
        = generate_communication_class(efsm, state.err_detect.succ, visited, function_list)

      in_channel = InErrChannel(channelName=state.name, param='x',
                   labels={state.actions[0].label}, continuation=continuation
                   errContinuation=errContinuation)
      return in_channel, {in_channel} ∪ channels1 ∪ channels2
```

Listing 6.9: Communication Class Generation for Single Message Receiving State With Error Handling

Branching State with Error Handling

This is the replacement state to the original Branching State. Similarly, we only replace InChannel with InErrChannel, evaluate the error continuation, pass it as the errContinuation variable to the class and combine the channel instances generated. This is because there's only one error handling continuation for each receiving state. The generation algorithm modified from Listing 4.19 is shown in Listing 6.10.

6.4.3 Code Generation

We can easily include this in our Effpi Type/Function code generation by slightly modifying the communication classs generation algorithm in Listing 4.20. We add an extra check to see whether a state is an error handling state.

```
def generate_branching_w_err(efsm, state, visited, function_list):
     function_name = "{efsm.role}_{len(function_list) + 1}"
     param_name = "X_{len(function_list) + 1}"
     labels = []
     continuations = []
     channels1 = []
     for action in state.actions:
         continuation, channels
           = generate_communication_class(efsm, action.succ, visited, function_list)
         continuations = continuations ∪ continuation
         channels1 = channels1 ∪ channels
         labels = labels ∪ {action.label}

     type_match_param = TypeMatchParam(channelName=param_name,
                                        labels=labels)
     type_match = TypeMatch(param=type_match_param,
                            labels=labels, continuations=continuations)


     channels1 = {type_match_param} ∪ channels1
     function_call = FunctionCall(functionName=function_name,
                                  channels=channels1.copy)
     function = Function(functionName=function_name, continuation=type_match
                         channels= channels1.copy, isMain=False)
     function_list = function_list ∪ {function}
     channels1 = channels1 \ type_match_param


     errContinuation, channels2
       = generate_communication_class(efsm, state.err_detect.succ, visited, function_list)

     in_channel = InErrChannel(channelName=state.name, param=param_name,
                 labels={labels}, continuation=function_call,
                 errContinuation=errContinuation)

     return in_channel, {in_channel} ∪ channels1 ∪ channels2
```

Listing 6.10: Communication Class Generation for Branching State With Error Handling

```
def generate_communication_class(efsm, state, visited, function_list):
    ...
    if efsm.is_sending_state(state):
        ...
     else if efsm.is_err_handling_state(state):
        if len(efsm.actions) == 1:
          com_class, channels =
              generate_singe_receive_w_err(efsm, state, visited, function_list)
       else:
          com_class, channels =
              generate_branching_w_err(efsm, state, visited, function_list)
    ...
```

Listing 6.11: Communication Class Generation Algorithm With Error Handling

6.5 Examples

In this section, we shall demonstrate how our tool generate Effpi-typed Scala program with error handling. We apply our tool on some well-known protocol in distributed system with error handling. Again, since global types are still work in progress, we shall show the protocols in term of local session types. We convert those local session types into digraphs manually and input into our tool, our tool will convert them into EFSMs and generate the code based on it. We don't show the digraph and EFSM of some examples as they are huge in size, they can be found in our github repository.

6.5.1 Simple Broadcast

In this example, we have three participants, **p**, **q** and **q**. **p** tries to broadcast some message to **q**and **r**. Should **p** crashes after successfully sending it's message to **q** but before sending it to **r**, **r** can handle the failure of **p** by requesting the data from **q**. The communication diagram for this case is shown in Figure 19.

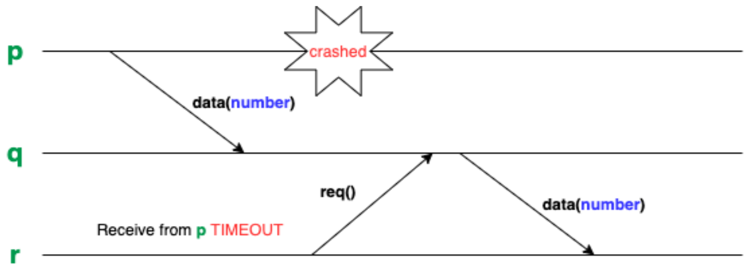


Figure 19: Error handling on **p** Failure

We shall now show the local session type for each participants:

1. **p**: T_p merely broadcasts to **q** and **r**. It does not detect any crashes should **q** or **r** fail.

$$T_p = \mathbf{q} \oplus data(int).\mathbf{r} \oplus data(int).\mathtt{end}$$

2. **q**: If T_q detects that **p** has crashed, it waits to receive(the inevitable) request from **r**, informing **r** that it cannot provide the requested data. Should **r** has failed, **q** merely exits.

$$T_q = \mathbf{p}\&\{data(int).T'_q, \mathbf{crash} : T''_q\}$$

$$T'_q = \mathbf{r}\& \begin{cases} ok().\mathtt{end} \\ req().\mathbf{r} \oplus data(int).\mathtt{end} \\ \mathbf{crash} : \mathtt{end} \end{cases}$$

$$T''_q = \mathbf{r}\&\{req().\mathbf{r} \oplus ko().\mathtt{end}, \mathbf{crash} : \mathtt{end}\}$$

3. **r**: Likewise, if T_r detects that **p** has crashed, it will request the data from **r**. Since **p** could have crashed before of after sending its data to **q**, **q** will either deny or fulfill our request, respectively. Should **q** have failed, **r** merely exists.

$$T_r = \mathbf{p}\&\{data(int).\mathbf{q} \oplus ok().\mathsf{end}, \mathbf{crash} : T'_r\}$$
$$T'_r = \mathbf{q} \oplus req().T''_r$$
$$T''_r = \mathbf{q}\& \begin{cases} ko().\mathsf{end} \\ data(int).\mathsf{end} \\ \mathbf{crash} : \mathsf{end} \end{cases}$$

The digraphs we generated (manually) for participant **p**, **q** and **r** are shown in Listing 6.12 , 6.13 and 6.14 respectively, the EFSMs generated by our tool is shown in Figure 20,21 and 21 respectively.

```
digraph G {
  0;1;2;

  0 -> 1 [label="q!data(number)", ];
  1 -> 2 [label="r!data(number)", ];}
```

Listing 6.12: digraph generated for **p** in Simple Broadcast

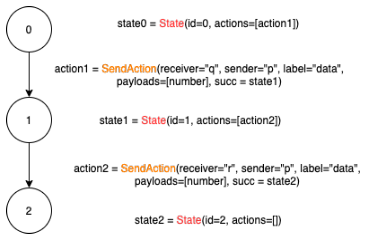
```
digraph G {
  0;1;2;3;4;5;6;7;8;9;

  0 -> 1 [label="p?data(number)", ];
  1 -> 2 [label="r?ok()", ];
  1 -> 5 [label="r#",];
  1 -> 3 [label="r?req()", ];
  3 -> 4 [label="r!data(number)", ];
  0 -> 6 [label = "p#",];
  6 -> 7 [label="r?req()", ];
  7 -> 8 [label="r!ko()", ];
  6 -> 9 [label = "r#",];}
```

Listing 6.13: digraph generated for **q** in Simple Broadcast

We shall now look at the Effpi type and function generated by our tool. The Effpi type generated for each participant can be seen in Listing 6.15, 6.17 and **??**. The Scala function for each participant can be seen in Listing 6.18, 6.19 and 6.20. The generated Effpi type and function for **p** is self-explanatory. For **q**, from Line 18-19 and 21-22 of it's Effpi type and Line 22 and 24 of it's function, we can observe that **q** is expected to receive a message of label ok or request from **r** whether **q** themselves had successfully received the data from **p**. However in the later case, we know that **r** shouldn't have receive from **p** if **q** didn't receive from **p**. This is because **p** sends the data to **q** before **r** and

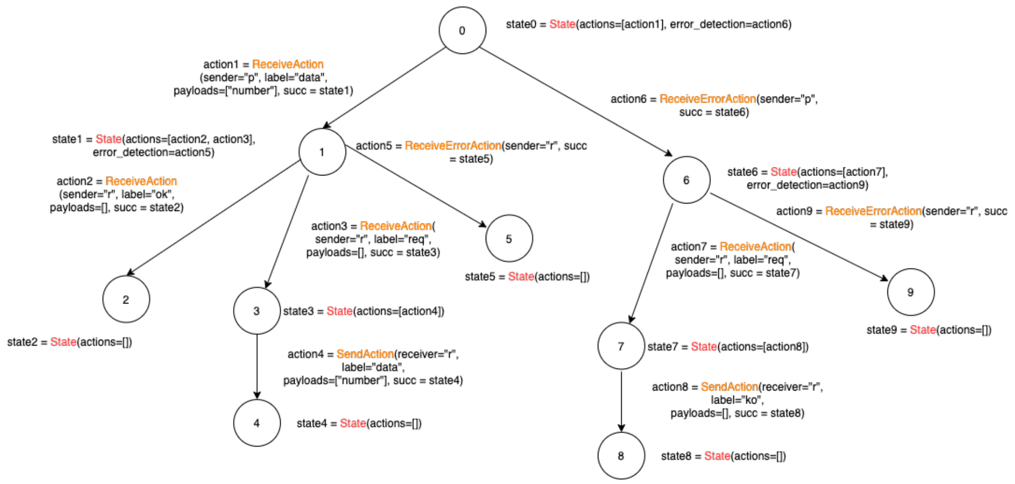
```
digraph G {
  0;1;2;3;4;5;6;7;

  0 -> 1 [label="p?data(number)", ];
  1 -> 2 [label="q!ok()", ];

  0 -> 3 [label = "p#",];
  3 -> 4 [label="q!req()", ];
  4 -> 5 [label="q?ko()", ];
  4 -> 6 [label="q?data(number)", ];
  4 -> 7 [label = "q#",];}
```

Listing 6.14: digraph generated for **r** in Simple Broadcast



0
state0 = State(id=0, actions=[action1])
action1 = SendAction(receiver="q", sender="p", label="data", payloads=[number], succ = state1)
1
state1 = State(id=1, actions=[action2])
action2 = SendAction(receiver="r", sender="p", label="data", payloads=[number], succ = state2)
2
state2 = State(id=2, actions=[])

Figure 20: EFSM generated for **p** in Broadcast Example



0
state0 = State(actions=[action1], error_detection=action6)
action1 = ReceiveAction (sender="p", label="data", payloads=["number"], succ = state1)
action6 = ReceiveErrorAction(sender="p", succ = state6)
1
state1 = State(actions=[action2, action3], error_detection=action5)
action5 = ReceiveErrorAction(sender="r", succ = state5)
action2 = ReceiveAction (sender="r", label="ok", payloads=[], succ = state2)
action3 = ReceiveAction(sender="r", label="req", payloads=[], succ = state3)
5
state5 = State(actions=[])
2
state2 = State(actions=[])
3
state3 = State(actions=[action4])
action4 = SendAction(receiver="r", label="data", payloads=["number"], succ = state4)
4
state4 = State(actions=[])
6
state6 = State(actions=[action7], error_detection=action9)
action9 = ReceiveErrorAction(sender="r", succ = state9)
action7 = ReceiveAction(sender="r", label="req", payloads=[], succ = state7)
9
state9 = State(actions=[])
7
state7 = State(actions=[action8])
action8 = SendAction(receiver="r", label="ko", payloads=[], succ = state8)
8
state8 = State(actions=[])

Figure 21: EFSM generated for **q** in Broadcast Example

we assume failure occurs on process and is permanent. Also,from the function generated for **r**, we can tell that they send ok to **q** if it successfully received from **p**, otherwise it will send req. Hence, in Line 14 of **q**'s function, we can see that it throws an Exception if it receives an ok from **r** and it failed to receive from **p** earlier. Note that channel instances c_r_q_1, c_r_q_2 and c_q_r_1 where all the exchange of messages of labels ok—req take place are assign the same actual

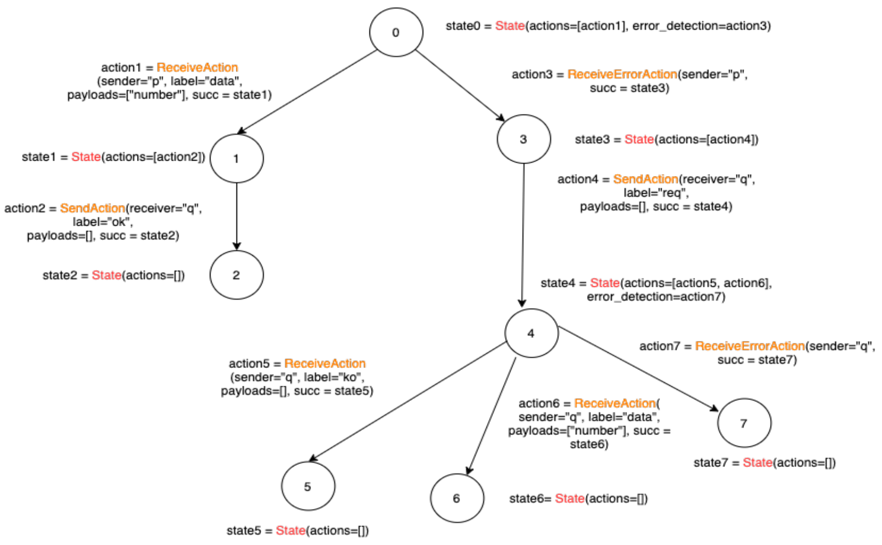

state0 = State(actions=[action1], error_detection=action3)
action1 = ReceiveAction (sender="p", label="data", payloads=["number"], succ = state1)
action3 = ReceiveErrorAction(sender="p", succ = state3)
state1 = State(actions=[action2])
state3 = State(actions=[action4])
action2 = SendAction(receiver="q", label="ok", payloads=[], succ = state2)
action4 = SendAction(receiver="q", label="req", payloads=[], succ = state4)
state2 = State(actions=[])
state4 = State(actions=[action5, action6], error_detection=action7)
action5 = ReceiveAction (sender="q", label="ko", payloads=[], succ = state5)
action7 = ReceiveErrorAction(sender="q", succ = state7)
action6 = ReceiveAction(sender="q", label="data", payloads=["number"], succ = state6)
state7 = State(actions=[])
state6= State(actions=[])
state5 = State(actions=[])
0
1
2
3
4
5
6
7

Figure 22: EFSM generated for **r** in Broadcast Example

channel c_3 in Listing 6.21.

```
type p[ C_p_q_1 <: OutChannel[data],
        C_p_r_1 <: OutChannel[data]] =
 Out[C_p_q_1,data] >>: Out[C_p_r_1,data] >>: PNil
```

Listing 6.15: Effpi Type Generated for **p** in Broadcast Example

We also implemented a feature for the programmers. For every sending action that correspond to a error handling receive action, we add the code "if(false)throw Exception("Some exception")" before it(This is not shown in the Listings above). The programmer can easily set the condition to true to simulate the case where the participant crashed on that point and observe what are the following communication that take place. That being said, there exist this line of code right before Line 5 of **p**'s function in Listing 6.18. We set the condition to true to simulate the example we demonstrated in Figure 19. The output of the program is seen as in Figure 23.

6.5.2 Simple Consensus

This example is based on Flooding Consensus[53], it has four participants, **c**, **p**, **q** and **r**. **c** is a client and the rest are members of the system with **p** being the leader (only one that can propose and communicate with **c**). For consensus to be reached, we need at least two of **p**, **q** and **r** to be in agreement. Ultimately, this means that we want either **q** or **r** to respond with an acceptance message(The

```
type q_2[ X_2 <: ok|req,
          C_q_r_2 <: OutChannel[data]] <: Process =
 X_2 match {
    case ok => PNil
    case req => Out[C_q_r_2,data] >>: PNil }

type q_3[ X_3 <: req|ok,
          C_q_r_3 <: OutChannel[ko]] <: Process =
 X_3 match {
    case req => Out[C_q_r_3,ko] >>: PNil
    case ok => PNil }

type q[
C_q_p_1 <: InChannel[data],
C_q_r_1 <: InChannel[ok|req],
C_q_r_2 <: OutChannel[data],
C_q_r_3 <: OutChannel[ko]] =
 InErr[C_q_p_1, data, (x:data) =>
      InErr[C_q_r_1, ok|req, (X_2:ok|req) =>
              q_2[X_2.type,C_q_r_2], (err:Throwable) => PNil],
              (err:Throwable) => InErr[C_q_r_1, req|ok,
              (X_3:req|ok) => q_3[X_3.type,C_q_r_3],
              (err:Throwable) => PNil]]
```

Listing 6.16: Effpi Type Generated for **q** in Broadcast Example

```
type r[ C_r_p_1 <: InChannel[data],
        C_r_q_1 <: OutChannel[ok],
        C_r_q_2 <: OutChannel[req],
        C_r_q_3 <: InChannel[ko|data]] =
 InErr[C_r_p_1, data, (x:data) => Out[C_r_q_1,ok] >>: PNil,
       (err:Throwable) => Out[C_r_q_2,req] >>:
           InErr[C_r_q_3, ko|data, (x:ko|data) => PNil,
                 (err:Throwable) => PNil]]
```

Listing 6.17: Effpi Type Generated for **r** in Broadcast Example

```
def p(c_p_q_1: OutChannel[data],
      c_p_r_1: OutChannel[data]
   ):p[c_p_q_1.type,c_p_r_1.type] = {
      send(c_p_q_1,data(50)) >> {
         send(c_p_r_1,data(43)) >> { nil }}}
```

Listing 6.18: Scala Function Generated for **p** in Broadcast Example

number of acceptors can be extended as one may wish). In the case where **p** has failed and **q** has already the propose from them previously, it will take over **p**'s role to communicate with **r** and inform **c**regarding the outcome.Figure 24 shows the case where **p** successfully sent **q** the propose and crashed without sending it to **r**. **q** was not aware that **p** has crashed and send it's choice, reject to it. Until **r** failed to receive from **p** and notified **q** about it. **q** informs **r** that it has received the propose and **r** notify **q**about it's choice accept. On the other hand, **c** failed to receive the result from **p** and proceed to wait for **q**. **q** then combines everyone's selection and notify **c** about the outcome.

```
private def q_2(x_2: ok|req, c_q_r_2: OutChannel[data]
   ):q_2[x_2.type,c_q_r_2.type] =
      x_2 match {
         case x_2 : ok => { nil}
         case x_2 : req => {
            send(c_q_r_2,data(84)) >> { nil }}}


private def q_3(x_3: req|ok, c_q_r_3: OutChannel[ko]
         ):q_3[x_3.type,c_q_r_3.type] =
      x_3 match {
         case x_3 : req => {
            send(c_q_r_3,ko()) >> { nil }}
         case x_3 : ok => { throw Exception("error")
                            nil }}

   def q( c_q_p_1: InChannel[data], c_q_r_1: InChannel[ok|req],
          c_q_r_2: OutChannel[data], c_q_r_3: OutChannel[ko]
   ):q[c_q_p_1.type,c_q_r_1.type,c_q_r_2.type,c_q_r_3.type] ={
      receiveErr(c_q_p_1) ({
         (x:data) =>
         receiveErr(c_q_r_1) ({(x_2:ok|req) => q_2(x_2,c_q_r_2)},
         {(err : Throwable) => nil }, Duration("5seconds"))},
      {(err : Throwable) => receiveErr(c_q_r_1) ({
            (x_3:req|ok) => q_3(x_3,c_q_r_3)},
         {(err : Throwable) => nil}, Duration("5seconds"))
      }, Duration("5seconds"))
   }
```

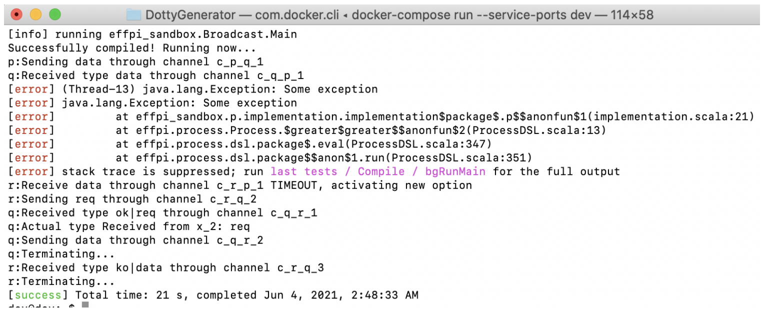
Listing 6.19: Scala Function Generated for **q** in Broadcast Example

```
def r(c_r_p_1: InChannel[data], c_r_q_1: OutChannel[ok],
      c_r_q_2: OutChannel[req], c_r_q_3: InChannel[ko|data]
   ):r[c_r_p_1.type,c_r_q_1.type,c_r_q_2.type,c_r_q_3.type] ={
      receiveErr(c_r_p_1) ({
         (x:data) => send(c_r_q_1,ok()) >> { nil }  },
        {(err : Throwable) => send(c_r_q_2,req()) >> {
            receiveErr(c_r_q_3) ({
               (x:ko|data) => nil },
            {(err : Throwable) => nil}, Duration("5seconds"))}
      }, Duration("5seconds"))}
```

Listing 6.20: Scala Function Generated for **r** in Broadcast Example

```
val(c1, c2, c3, c4) =
     (Channel.async[data](),
     Channel.async[data](),
     Channel.async[req|ok](),
     Channel.async[ko|data]())

    eval(par(
     p(c1, c2),
     r(c2, c3, c3, c4),
     q(c1, c3, c4, c4)))
```

Listing 6.21: Channels generated for Broadcast Example


DottyGenerator — com.docker.cli ◂ docker-compose run --service-ports dev — 114×58

```
[info] running effpi_sandbox.Broadcast.Main
Successfully compiled! Running now...
p:Sending data through channel c_p_q_1
q:Received type data through channel c_q_p_1
[error] (Thread-13) java.lang.Exception: Some exception
[error] java.lang.Exception: Some exception
[error]         at effpi_sandbox.p.implementation.implementation$package$.p$$anonfun$1(implementation.scala:21)
[error]         at effpi.process.Process.$greater$greater$$anonfun$2(ProcessDSL.scala:13)
[error]         at effpi.process.dsl.package$.eval(ProcessDSL.scala:347)
[error]         at effpi.process.dsl.package$$anon$1.run(ProcessDSL.scala:351)
[error] stack trace is suppressed; run last tests / Compile / bgRunMain for the full output
r:Receive data through channel c_r_p_1 TIMEOUT, activating new option
r:Sending req through channel c_r_q_2
q:Received type ok|req through channel c_q_r_1
q:Actual type Received from x_2: req
q:Sending data through channel c_q_r_2
q:Terminating...
r:Received type ko|data through channel c_r_q_3
r:Terminating...
[success] Total time: 21 s, completed Jun 4, 2021, 2:48:33 AM
```

Figure 23: Output of Broadcast Example with **p** Failure

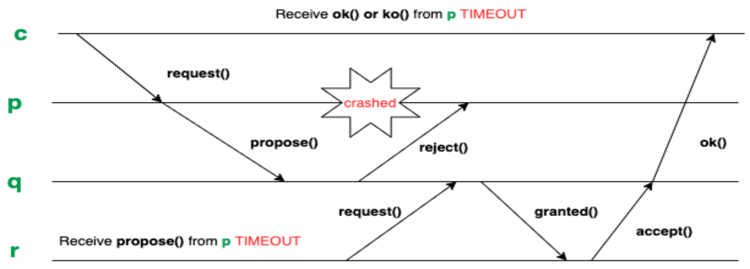

Receive ok() or ko() from p TIMEOUT
c
p
q
r
request()
crashed
propose()
reject()
ok()
request()
granted()
accept()
Receive propose() from p TIMEOUT

Figure 24: Simple Consensus with **p** Failure

We shall now show the local session type for each participants:

1. **c**: It only sends the request to the leader **p** and proceed to wait for the result from it, if it failed to do so, it will proceed to wait for the result from **q**.

$$T_c = \mathbf{p} \oplus request().\mathbf{p}\& \begin{cases} ok().\texttt{end} \\ ko().\texttt{end} \\ \mathbf{crash} : \mathbf{q}\& \begin{cases} ok().\texttt{end} \\ ko().\texttt{end} \\ \mathbf{crash} : \texttt{end} \end{cases} \end{cases}$$

2. **p**: T_p broadcasts to **q** and **r**, receives the result from them and notify **c** about the outcome. Note that **p** always accept, since they are the one who propose. As long one other of **p** or **q** accept, the proposal is deemed

as success.

$$T'_p = \mathbf{c}\&\{request().\mathbf{p} \oplus propose().\mathbf{r} \oplus propose().T'_p, \mathbf{crash} : \mathbf{c} \oplus ko()\}$$

$$T_p = \mathbf{q}\& \begin{cases} accept().\mathbf{r}\& \begin{cases} accept().\mathbf{c} \oplus ok().\mathsf{end} \\ reject().\mathbf{c} \oplus ok().\mathsf{end} \\ \mathbf{crash} : \mathbf{c} \oplus ok().\mathsf{end} \end{cases} \\ reject().\mathbf{r}\& \begin{cases} accept().\mathbf{c} \oplus ok().\mathsf{end} \\ reject().\mathbf{c} \oplus ko().\mathsf{end} \\ \mathbf{crash} : \mathbf{c} \oplus ko().\mathsf{end} \end{cases} \\ \mathbf{crash} : \mathbf{r}\& \begin{cases} accept().\mathbf{c} \oplus ok().\mathsf{end} \\ reject().\mathbf{c} \oplus ok().\mathsf{end} \\ \mathbf{crash} : \mathbf{c} \oplus ok().\mathsf{end} \end{cases} \end{cases}$$

3. **q**: It receives from **p** and either return an accept or reject. If **p** crashes before sending to **q**, it will crash for **r** too, we can't do anything but inform **r** when it inevitably requests the value from us. If we received the propose from **p**, but **r** didn't, we will notify **r** and **r** will tell us whether they accept or reject the proposal, we then notify **c** about the outcome, since we know **p** will not be sending to **c** anymore.

$$T_q = \mathbf{p}\&\{propose().\mathbf{p} \oplus \begin{cases} accept().T'_q \\ reject().T''_q \end{cases}, \mathbf{crash} : T'''_q\}$$

$$T'_q = \mathbf{r}\& \begin{cases} request().\mathbf{r} \oplus granted().\mathbf{r}\& \begin{cases} accept().\mathbf{c} \oplus ok().\mathsf{end} \\ reject().\mathbf{c} \oplus ok().\mathsf{end} \\ \mathbf{crash} : \mathbf{c} \oplus ok().\mathsf{end} \end{cases} \\ nothing().\mathsf{end} \\ \mathbf{crash} : \mathsf{end} \end{cases}$$

$$T''_q = \mathbf{r}\& \begin{cases} request().\mathbf{r} \oplus granted().\mathbf{r}\& \begin{cases} accept().\mathbf{c} \oplus ok().\mathsf{end} \\ reject().\mathbf{c} \oplus ko().\mathsf{end} \\ \mathbf{crash} : \mathbf{c} \oplus ko().\mathsf{end} \end{cases} \\ nothing().\mathsf{end} \\ \mathbf{crash} : \mathsf{end} \end{cases}$$

$$T'''_q = \mathbf{r}\&\{request().\mathbf{r} \oplus declined().\mathbf{c} \oplus ko().\mathsf{end}, \mathbf{crash} : \mathsf{end}\}$$

4. **r**: We will receive from **p** and return an accept or reject. If **p** crashes, we request the proposal from **q** and inform **q** if we accept of reject it. If **q** is

unable to fulfill our request, or has itself crashed, we end the protocol.

$$T_r = \mathbf{p}\&\{propose().\mathbf{p} \oplus \begin{cases} accept().\mathbf{q} \oplus nothing().\mathtt{end} \\ reject().\mathbf{q} \oplus nothing().\mathtt{end} \end{cases}, \mathbf{crash} : T'_r\}$$

$$T'_r = \mathbf{q} \oplus request().\mathbf{q}\& \begin{cases} granted().\mathbf{q} \oplus \begin{cases} accept().\mathtt{end} \\ reject().\mathtt{end} \end{cases} \\ declined().\mathtt{end} \\ \mathbf{crash} : \mathtt{end} \end{cases}$$

6.5.3 Byzantine Generals Problem

We adopt the version of Byzantine General where we have one betraying commander and three loyal lieutenants. The ideation is simple, a general may ask a lieutenant to launch an attack on the enemy or do nothing. If majority of the lieutenants launch the attack at the same time, the enemy will be defeated, otherwise the lieutenants will be killed. Since the commander is a traitor, it may send different command to each lieutenant. This can be easily solve if the lieutenants are able to communicate within themselves, reaching a consensus. For example, commander request Lieutenant B to launch an attack while requesting the other two lieutenants to do nothing, Lieutenant B will be killed if he launch the attack alone. Each lieutenants broadcast the command they received with each other, in the sequence Lieutenant A to B to C. Lieutenant B noticed that attack is the minority command hence stop the attack. However, this problem is unsolvable if communication failure may occur. Look at Figure ,say Lieutenant A crashed and failed to send Lieutenant B and C it's message, both of them couldn't decide which command is the majority. They are also not sure whether A received their broadcast and what action shall A take. We shall demonstrate how this example can be represented in local session types and hence our tool can generate a program from it.

We have four participants, **c** is the commander and **p**,**q** and **r** are the lieutenants. Message receiving and broadcasting prioritise the order of **p** to **q** to **r**. If the lieutenants failed to receive from the commander, they will send nothing to the other lieutenants. Their local session types are shown as below:

1. **c**: We have a very straightforward type for the commander, it sends to **p** then to **q** finally **r**. At each stage it can choose either attack or nothing to

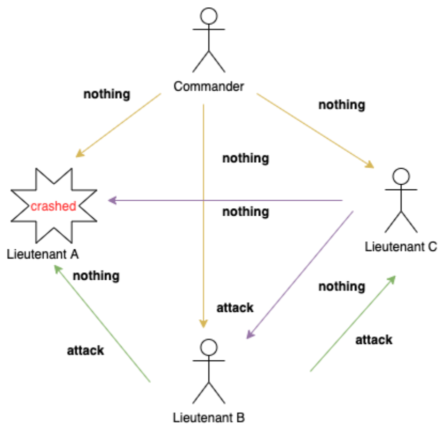


Figure 25: Byzantine Problem with Lieutenant A crashed

send.

$$T_c = \mathbf{p} \oplus \begin{cases} attack().\mathbf{q} \oplus \begin{cases} attack().\mathbf{r} \oplus \begin{cases} attack().\mathtt{end} \\ nothing().\mathtt{end} \end{cases} \\ nothing().\mathbf{r} \oplus \begin{cases} attack().\mathtt{end} \\ nothing().\mathtt{end} \end{cases} \end{cases} \\ nothing().\mathbf{q} \oplus \begin{cases} attack().\mathbf{r} \oplus \begin{cases} attack().\mathtt{end} \\ nothing().\mathtt{end} \end{cases} \\ nothing().\mathbf{r} \oplus \begin{cases} attack().\mathtt{end} \\ nothing().\mathtt{end} \end{cases} \end{cases} \end{cases}$$

2. **p**: It is the first one to receive and send among all lieutenants. It will send the message as it received from **c** to **q** and **r**. If it failed to receive from **c**, it will expect to receive nothing from both **q** and **r** as they shouldn't

receive from **c** as well.

$$T_p = \mathbf{c}\& \begin{cases} attack().\mathbf{q} \oplus attack().\mathbf{r} \oplus attack().T'_p \\ nothing().\mathbf{q} \oplus nothing().\mathbf{r} \oplus nothing().T'_p \\ \mathbf{crash} : \mathbf{q} \oplus nothing().\mathbf{r} \oplus nothing(). \end{cases}$$

$$T'_p = \mathbf{q}\& \begin{cases} attack().T''_p \\ nothing().T''_p \\ \mathbf{crash} : T'''_p \end{cases}$$

$$T''_p = \mathbf{r}\& \begin{cases} attack().\mathtt{end} \\ nothing().\mathtt{end} \\ \mathbf{crash}.\mathtt{end} \end{cases}$$

$$T'''_p = \mathbf{q}\& \begin{cases} nothing().\mathbf{r}\& \begin{cases} nothing().\mathtt{end} \\ \mathbf{crash}.\mathtt{end} \end{cases} \\ \mathbf{crash} : \mathbf{r}\& \begin{cases} nothing().\mathtt{end} \\ \mathbf{crash}.\mathtt{end} \end{cases} \end{cases}$$

3. **q**: It is the second lieutenant to receive and send among all lieutenants. It will send the message as it received from **c** to **p** and **r**. If it failed to receive from **c**, it will send nothing to **q** and expect to receive nothing from **r** as they shouldn't receive from **c** as well.

$$T_q = \mathbf{c}\& \begin{cases} attack().T'_q \\ nothing().T''_q \\ \mathbf{crash} : T'''_q \end{cases}$$

$$T'_q = \mathbf{p}\& \begin{cases} attack().\mathbf{p} \oplus attack().\mathbf{r} \oplus attack().T''''_q \\ nothing().\mathbf{p} \oplus attack().\mathbf{r} \oplus attack().T''''_q \\ \mathbf{crash} : \mathbf{p} \oplus attack().\mathbf{r} \oplus attack().T''''_q \end{cases}$$

$$T''_q = \mathbf{p}\& \begin{cases} attack().\mathbf{p} \oplus nothing().\mathbf{r} \oplus nothing().T''''_q \\ nothing().\mathbf{p} \oplus nothing().\mathbf{r} \oplus nothing().T''''_q \\ \mathbf{crash} : \mathbf{p} \oplus nothing().\mathbf{r} \oplus nothing().T''''_q \end{cases}$$

$$T'''_q = \mathbf{p}\& \begin{cases} attack().\mathbf{p} \oplus nothing().\mathbf{r} \oplus nothing().T'''''_q \\ nothing().\mathbf{p} \oplus nothing().\mathbf{r} \oplus nothing().T'''''_q \\ \mathbf{crash} : \mathbf{p} \oplus nothing().\mathbf{r} \oplus nothing().T'''''_q \end{cases}$$

$$T''''_q = \mathbf{r}\& \begin{cases} attack().\mathtt{end} \\ nothing().\mathtt{end} \\ \mathbf{crash} : \mathtt{end} \end{cases}$$

$$T'''''_q = \mathbf{r}\& \begin{cases} nothing().\mathtt{end} \\ \mathbf{crash} : \mathtt{end} \end{cases}$$

4. **r**: It is the final lieutenant to receive and send among all lieutenants. It will send the message as it received from **c** to **p** and **q**. If it failed to receive from **c**, it will send nothing to **q** and **q**.

$$T_r = \mathbf{c}\& \begin{cases} attack().T'_r \\ nothing().T''_r \\ \mathbf{crash} : T''_r \end{cases}$$

$$T'_r = \mathbf{p}\& \begin{cases} attack().T'''_r \\ nothing().T'''_r \\ \mathbf{crash} : T'''_r \end{cases}$$

$$T''_r = \mathbf{p}\& \begin{cases} attack().T''''_r \\ nothing().T''''_r \\ \mathbf{crash} : T''''_r \end{cases}$$

$$T'''_r = \mathbf{q}\& \begin{cases} attack().\mathbf{p} \oplus attack().\mathbf{q} \oplus attack().\texttt{end} \\ nothing().\mathbf{p} \oplus attack().\mathbf{q} \oplus attack().\texttt{end} \\ \mathbf{crash} : \mathbf{p} \oplus attack().\mathbf{q} \oplus attack().\texttt{end} \end{cases}$$

$$T''''_r = \mathbf{q}\& \begin{cases} attack().\mathbf{p} \oplus nothing().\mathbf{q} \oplus nothing().\texttt{end} \\ nothing().\mathbf{p} \oplus nothing().\mathbf{q} \oplus nothing().\texttt{end} \\ \mathbf{crash} : \mathbf{p} \oplus nothing().\mathbf{q} \oplus nothing().\texttt{end} \end{cases}$$

6.5.4 Paxos Algorithm

The Paxos Algorithm is a fault tolerant algorithm in distributed system, we use this example from Leslie[54]. Similar to the Simple Consensus example, we have a few nodes in a distributed system and we need all of them to come to a consensus for a proposal. Each node has a current proposal id and the value of the last proposal it has accepted. The client send the proposer(which is also a node) a request, the proposer will need to get the proposal accepted by majority of the nodes in the system (including itself) for the proposal to be seemed as successful. We also have a listener that will record the values for each nodes. This process can be split into two phase.

Phase one can be seen from Figure 26, the client sends the proposer a request and the proposer broadcast the proposal to each nodes along a proposal id. Each node then checks if their current proposal id is lower than the one received from the proposer, if it is, they will send promise to the proposer, along the value of the last proposal it has accepted, promising the proposer that it will not accept any proposal with it's id lower than the one the proposer sent. Otherwise, if the current proposal id of the node is higher than of the one received from proposer, it will send a Nack to it, notifying it their current proposal id so that it will not send any proposal with id lower than that value anymore. It will then notify the listener to cancel the operation. Of course, we will also have to consider the case where the proposer propose to crashed nodes as the proposer has no clue on which nodes have crashed.

Phase two can be seen from Figure 27, the proposer sends accept to the nodes which have replied with promise in the earlier stage. The proposer include two values in the accept payloads, one number that indicates the proposal id and another which is the largest value of proposal it has received from the nodes in the previous stage. Note that the nodes may have promise on some other proposal with an higher id between sending promise and receiving accept from the proposer. So it will have to check it current proposal id again, if it's larger than the one in the accept message, it sends accepted to the proposer, updates the value of the last proposal it has accepted and notify the listener to record the new value. Otherwise, it will send Nack to the proposer to inform it about it's current proposal id and tell listener to cancel the operation. Proposer will then send Succeed to the client if majority of the nodes responded with accepted, otherwise it will send failed.

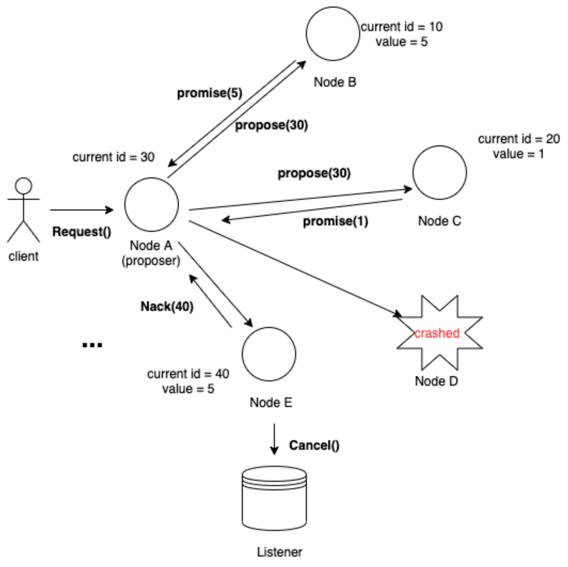


Figure 26: Phase 1 of Paxos Algorithm

We shall use an example with only 3 nodes, **p**,**q** and **r**, where **p** is the proposer in this case and communicates directly with a client **c**. We also have a listener **l** that records the value from the nodes on request. We shall now show the local session type for each participants:

1. **c**: Client has a very straightforward type, it sends a request to the proposer

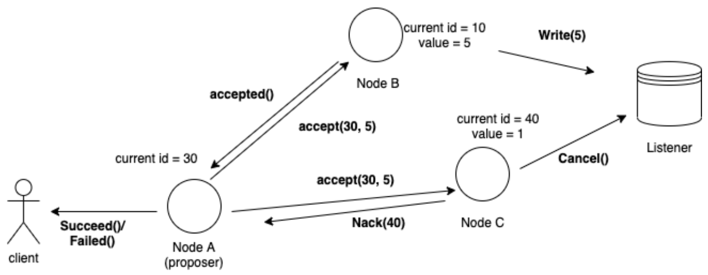


Figure 27: Phase 2 of Paxos Algorithm

and wait for it's reply.

$$T_c = \mathbf{p} \oplus request().\mathbf{p}\& \begin{cases} succeed().\mathtt{end} \\ failed().\mathtt{end} \\ \mathbf{crashed} : \mathtt{end} \end{cases}$$

2. **q**,**r**: The member nodes receive a propose from the proposer. It compares it with it's current id and decides whether to send promise (lower than proposal id) or nack (higher, notify proposer about current id). If it sends promise, it will expect to receive an accept request from the proposer. It then compares it with it's current id again and decides whether to send accepted to accept the proposal or nack to reject the proposal. Note that we only ask the listener to write when we accepted a proposal, otherwise we will always end up cancelling the operation with it. If they failed to receive from proposer at any point, it will just cancel operation with listener and terminate.

$$T_q = \mathbf{p}\&\{propose(int).Tq', \mathbf{crashed} : \mathbf{l} \oplus Cancel().\mathtt{end}\}$$

$$T_q' = \mathbf{p} \oplus \begin{cases} promise(int).T_q'' \\ nack(int).\mathbf{l} \oplus Cancel().\mathtt{end} \end{cases}$$

$$T_q'' = \mathbf{p}\&\{accept(int, int).T_q''', \mathbf{crashed} : \mathbf{l} \oplus Cancel().\mathtt{end}\}$$

$$T_q''' = \mathbf{p} \oplus \begin{cases} accepted().\mathbf{l} \oplus Write(int).\mathtt{end} \\ nack(int).\mathbf{l} \oplus Cancel().\mathtt{end} \end{cases}$$

3. **l**: The listener's type is also self explanatory, it receive requests from **q** and **r** simultaneously.

$$T_l = \mathbf{q}\& \begin{cases} write(int).T_l' \\ cancel().T_l' \\ \mathbf{crash} : T_l' \end{cases}$$

$$T_l' = \mathbf{r}\& \begin{cases} write(int).\mathtt{end} \\ cancel().\mathtt{end} \\ \mathbf{crash} : \mathtt{end} \end{cases}$$

4. p: The proposer has a much complex type than the others, it is due to the large amount of cases which we need to consider. We start of by sending propose to both q and r, we will have to deal with cases where one of them crashed or send nack in either phase. As long as we received accepted from one of q or r, we can send succeed to the client, otherwise we will send failed.

$$T_p = \mathsf{c}\& \begin{cases} \mathsf{q} \oplus propose(int).\mathsf{r} \oplus propose(int).\mathsf{q}\& \begin{cases} promise(int).T_p''' \\ nack(int).T_p' \\ \mathbf{crash} : T_p' \end{cases} \\ \mathbf{crash} : \mathsf{c} \oplus failed().\mathtt{end} \end{cases}$$

$$T_p' = \mathsf{r}\& \begin{cases} promise(int).\mathsf{r} \oplus accept(int, int).T_p'' \\ nack(int).\mathsf{c} \oplus failed().\mathtt{end} \\ \mathbf{crashed} : \mathsf{c} \oplus failed().\mathtt{end} \end{cases}$$

$$T_p'' = \mathsf{r}\& \begin{cases} accepted().\mathsf{c} \oplus succeed().\mathtt{end} \\ nack(int).\mathsf{c} \oplus failed().\mathtt{end} \\ \mathbf{crashed} : \mathsf{c} \oplus failed().\mathtt{end} \end{cases}$$

$$T_p''' = \mathsf{r}\& \begin{cases} promise(int).\mathsf{q} \oplus accept(int, int).\mathsf{r} \oplus accept(int, int).T''''' \\ nack(int).\mathsf{q} \oplus accept(int, int).T_p'''' \\ \mathbf{crash} : \mathsf{q} \oplus accept(int, int).T_p'''' \end{cases}$$

$$T_p'''' = \mathsf{q}\& \begin{cases} accepted().\mathsf{c} \oplus succeed().\mathtt{end} \\ nack(int).\mathsf{c} \oplus failed().\mathtt{end} \\ \mathbf{crashed} : \mathsf{c} \oplus failed().\mathtt{end} \end{cases}$$

$$T_p''''' = \mathsf{q}\& \begin{cases} accepted().\mathsf{r}\& \begin{cases} accepted().\mathsf{c} \oplus succeed().\mathtt{end} \\ nack(int).\mathsf{c} \oplus succeed().\mathtt{end} \\ \mathbf{crash} : \mathsf{c} \oplus succeed().\mathtt{end} \end{cases} \\ nack(int).\mathsf{r}\& \begin{cases} accepted().\mathsf{c} \oplus succeed().\mathtt{end} \\ nack(int).\mathsf{c} \oplus failed().\mathtt{end} \\ \mathbf{crash} : \mathsf{c} \oplus failed().\mathtt{end} \end{cases} \\ \mathbf{crash} : \mathsf{r}\& \begin{cases} accepted().\mathsf{c} \oplus succeed().\mathtt{end} \\ nack(int).\mathsf{c} \oplus failed().\mathtt{end} \\ \mathbf{crash} : \mathsf{c} \oplus failed().\mathtt{end} \end{cases} \end{cases}$$

Chapter 7

Evaluation

In this section we shall present the unit tests performed on our tool to showcase our backend on similar mapping from various nuScr types to their Effpi types. Then, we introduce some metrics for our tool and evaluate them on some existing examples. We also measure how the runtime performance of our tool is affected with the increase in complexity of the input protocol. Finally, we will demonstrate the usage of our tool on some real life case studies to present the strength of our tool.

It is necessary to outline the generation process of our tool. It can be comprised of four main steps:

1. Our tool takes a Scribble file as the input, the file contains a global protocol which describe the communication between multiple parties.
2. We use nuScr to generate the local types for each participants in the form of FSM.
3. We translate each FSM to a tree-like instance that specify the local communication behaviour of each participant, with each nodes and leafs comprising of our self-defined communication class, each branch represents a possible action the participant may take.
4. Generate a Scala file containing the local Effpi type for each participant and template functions that communicate according to the types.

7.1 Conformance and Adherence of Tool Output to Design

We want to check the mapping from nuScr Types to Effpi Type aligned with our theories. In order to do so, we designed multiple unit test cases to test the mapping of the states of the FSM to our self-defined communication class instances from which we generate Effpi code. We obtain the FSM and communication class instance from two separate runs, having the same global protocol as the input, to showcase that our mapping from nuScr types to Effpi types remain the same despite being generated from different runs. We also assume the correctness of FSM generated by nuScr[11].

From the table in Figure 28, we show how various types in nuScr are being represented in FSM states and how they correspond to our self-defined class instances. Given a FSM state and a class instance, other than checking the class instance correspond to the expected class type as shown in the table, we will also need to to make sure the outgoing edges(actions) of the state matches the elements in the continuation array of the class instance. The criteria we check are: number of outgoing edges and size of continuation array, payload and labels of each actions, sender and recipient of the action. Note that continuation array of a class instance is an array of class instances which represent the possible following communication class instances.

nuScr Type	**FSM state**	**Class Instance**
termination	no outgoing edges	`Termination`
send (single message)	one outgoing edge of sending action	`OutChannel`
receive (single message)	one outgoing edge of receiving action	`InChannel`
selection	multiple outgoing edges of sending action	`Selection` (comprised of multiple `OutChannel`s)
branch	multiple outgoing edges of receiving action	`InChannel` that leads to a `Function` with function body `TypeMatch`
start recurse/loop	entry point of a cycle	`Loop`
continue/go to start of a loop	final state in a cycle, next state is the entry point of current cycle	`GoTo`

Figure 28: nuScr Types and Correspond Effpi Types

Before running our test, we perform depth first search(DFS) on the FSM to identify states that are entry point of a loop, to make sure they map to the `Loop` class instance later. We start our test by performing DFS at the initial state of the FSM and the root class instance. Given a FSM state and a class instance, once we checked one correspond to another (based on the method we described above), for each outgoing edge of the FSM state, we find a corresponding new class instance from the continuation array based on the payload and label of the outgoing edge, we then perform the check between the state on the other end of the outgoing edge and the new class instance an so on, until we reached a terminal or visited state. Note that we keep track of the visited state throughout the tests so we can check that a state shall correspond to a `GoTo` instance if it has been visited before.

7.2 Benchmarks

In this section, we will showcase the strength and expressiveness of our tool by introducing a few metrics and evaluate them on some well-known Scribble examples. We will then evaluate the runtime performance of our tool with respect to the complexity of the input protocol. Finally, we will compare the runtime of the output Effpi-typed Scala functions to manually written channel based Akka. All benchmarks are are measured on a MacBookPro running macOS Catalina version 10.15.7, having a 2.2 Ghz 6-Core Intel i7 processor and 16GB RAM and Python version 3.8.5.

7.2.1 Standard Benchmarks

We defined a few metrics for our tool alongside their respective motivation:

- **Total generation time** - By observing the overhead to generate the output, we can measure the efficiency of our tool.
- **Number of Case Classes** - The number of distinct case classes should match with the number of different message labels in the global protocol. It showcases the general complexity of the program as the larger the number of different case classes the more different channels we need to use in our program and hence the harder it takes to implement.
- **Total lines of code generated for functions** - Implementing functions are deemed as the main responsibility by a programmer. Function body is always required for the program to perform a task regardless of having a type checking mechanism. By measuring this we can have an idea on the complexity of the minimal function implementations for the program the executes in accordance to the protocol. Note that we only measure the lines of code inside the function body, ignoring lines that only contains print statement, brackets and function declaration. We do so to focus on codes that contribute to the core functionality of the program.
- **Number of Type Definitions** - On the very rare simple case with only sending and receiving single messages, we may have only one type definition for each participant. However, for most cases we have one or more nested choice in our local protocol, each branch in the local protocol will require a new type definition for match type checking. Type checking ensures communication safety but at the same time can be very time consuming and error prone to implement manually with complex protocols. The number of type definitions generated gives us an indirect measure on the complexity of the generated Effpi type for the program.

- **Largest number of continuation for a participant's type** - The motivation is same as the one above. Complexity of each participants type within the same protocol varies a lot based on their communication role in the protocol. Nevertheless, the type with the most continuations usually hardest to implement.

- **Number of channels generated** - Our tool generates channel of different type for participants to communicate through. Channel generation is deemed to be one of the biggest challenge in channel-based languages, a single channel plays different roles in different participant, it may be also be used within the same participant multiple times, but having it cast to different channel types. For example a Channel[YES|NO] may be cast into OutChannel[YES] and OutChannel[NO] within the same participant as the participant is required to send a message of type YES or NO based on a previous choice. However, the recipient is expecting a message of type YES or NO through an InChannel[YES|NO] (also cast from the same Channel[YES|NO]) as it has no knowledge regarding the previous choice between the sender and other participant. A programmer must have a great picture of the local type of each participants in order to perform the correct channel generation and casting. Our tool does this by implementing a merging theory to assign channels generated each participant correctly. This metric tells us how many channels are required to run the program and again showcase the complexity of the output program.

We shall now execute our tool on some examples from the session type literature: Two Buyer[41], Calculator[39], Ticket[42], Travel Agency[43], SH[44], Fibonacci[39], Negotiate[45], Online Wallet[46] and Http[47].We measure the various metrics we describe above for each example. The total generation time is measured by taking the sum of time taken for various generation stage, which we will discuss in detail in the runtime performance section later. The time is recorded using the built in Python time library, we get the average generation time from 10 runs of our tool for each example. Other metrics can be easily observed from the output Scala file.

The result is shown as in the table in Figure 29. Note that Travel Agency and SH generated around 56-57 lines of code for their function bodies, which is more than double of the average of the other examples (26.75). This is because Travel Agency and SH have one nested choice while the others have none, more functions are needed for each participants to perform type matching for branching. We can also observe that the average number of type definitions for Two Buyer, Travel Agency and SH (6) is double the average of the other examples(3). Other than Travel Agency and SH having nested choice, these three examples have three participants while the remaining only have two. It is obvious that the more the participant, the more the type definition as we will have to generate at least one type definition for each participant. We take into account the channels when measuring the largest number of lines for a participant, this gives a clearer picture on the complexity of a participant's type. This measurement varies a lot

Example	LCF	NTD	LNPT	NCG	NCC	TGT
Two Buyer[41]	32	5	7	6	8	0.089s
Calculator[39]	30	3	6	5	7	0.071s
Ticket[42]	21	3	4	2	3	0.045s
Travel Agency[43]	58	5	8	7	8	0.122s
SH[44]	98	6	14	11	11	0.090s
Fibonacci[39]	24	3	6	3	4	0.050s
Negotiate[45]	24	4	7	3	5	0.075s
Online Wallet[46]	73	7	13	10	10	0.133s
Http[47]	48	4	13	7	8	0.099s

NTD - Number of Type Definitions **NCC** - Number of Case Classes
NCG - Number of Channels Generated **TGT** - Total Generation Time
LNPT - Largest Number of lines for a Participant's Type
LCF - Lines of Code generated for Function bodies

Figure 29: Selected Examples From Literature

based on the participant's role in the protocol, for example in the Travel Agency protocol as shown in Listing 7.1, participant A has a type of 26 lines while participant S only have 10, this is due to A having to communicate back and forth between participant B and S while S only needs to communicate with A, A also has a more complicated communication pattern, a huge loop and a branch within another branch, the later leads to two type definitions being generated for A to perform type matching. A channel is required for each communication action in a protocol, however in some cases some channels have to be merged together for the program to execute and terminate successfully, our tool do so by checking the local protocol for each participant and deduce which channels need to be merged. Looking at the Travel Agency protocol again, we have 2 choices and 6 sending/receiving of message, one may conclude that 8 channels are needed based on naive observation. However, Quote in Line 9 and Full in Line 19 have to be merged into Channel[Quote|Full] while Confirm in Line 12 and Reject in Line 15 have to be merged into Channel[Confirm|Reject]. This is because the recipients of the actions has no knowledge regarding the prior choice action, recipient of Quote in Line 9 and Full in Line 19 is participant B, however the prior choice in Line 7 occurs between participant S and A, hence from B's point of view, he's receiving a choice from A hence the communication proceeds through a same channel. The number of case classes generated is the same as the number of distinct labels in the protocol. We can also observe the total generation time for each example is low, with each not exceeding 0.2 seconds.

```
module TravelAgency;

global protocol TravelAgency(role A, role B, role S) {

    Suggest(string) from B to A;  // friend suggests place
    Query(string) from A to S;
    choice at S {
        Available(number) from S to A;
        Quote(number) from A to B; // check quote with friend
        choice at B {
            OK(number) from B to A;
            Confirm(number) from A to S;
        } or {
            No() from B to A;
            Reject() from A to S;
        }
    } or {
        Full() from S to A;
        Full() from A to B;
        do TravelAgency(A, B, S);
    }

}
```

Listing 7.1: TravelAgency Protocol

7.2.2 Benchmark for Clone Channel Elimination

We demonstrate the total number of channel instances generated for the participants' local functions and the total number of actual channels generated, with and without clone channel detection and elimination. We execute our tool on some examples from our self defined examples and session type literature: ReuseChannel (Listing 5.3), DuplicateInOut (Listing 5.5), MultipleCast (Listing 5.9), Calculator[39], Travel Agency[43], SH[44], Negotiate[45], Online Wallet[46], Http[47] and Battleship [40].

The results can be seen in Figure 30. We only show examples that have it number of actual channels generated or number of channel instances generated reduced by clone channel detection and elimination. Ticket[42], Two Buyer[41] and Fibonacci[39] are not reducible as no same message labels are reused between the communication of any two participants. We may also observe that for all reducible examples, the number of actual channels generated are always reducible but not the number of local channel instances generated. This is due to the case where the same message label is used for both-way communication between two participants, the actual channels can be combined but the local channel instances of In/Output channel instances cannot be further combined. Taking the Negotiate protocol as an example, participant P and C send each other a message of type accpt|counter|reject at different stages of the communication. We can also note that the more the labels are being reused between

	Without CCDE		**With CCDE**	
Example	**TCIG**	**TACG**	**TCIG**	**TACG**
ReuseChannel	8	4	4	2
DuplicateInOut	8	4	8	2
MultipleCast	12	4	10	3
Calculator[39]	12	6	10	5
Travel Agency[43]	20	9	16	7
SH[44]	29	13	25	11
Negotiate[45]	10	5	10	3
Online Wallet[46]	24	11	22	10
Http[47]	20	10	20	7
Battleships[40]	40	14	22	8

CCDE - Clone Channels Detection and Elimination
TCIG - Total Number of Local Channel Instances Generated
TACG - Total Number of Actual Channels Generated

Figure 30: Channels Comparison With Selected Examples From Literature

the communication of any two participants, the larger the amount of generated actual channels/ channel instances can be reduced. In the Battleships example, the number of channel instances generated almost doubled without clone channel elimination, this is due to players receiving multiple messages of label Hit, Miss, Sunk from the server throughout the protocol, a single channel instance is suffice for each receive.

7.2.3 Generation Time

We evaluate the run-time performance of our tool by measuring the time taken in different stages of code generation: (1)Communication Class generation—From the FSM generated, we build a tree-like structure, consisting of our self-defined communication classes which specify the local communication behaviour of each participant. (2)Type generation and (3) Function generation—generate local Effpi Types and function that correspond to the types for each participants. (4)Channel generation/merging—We identify for each two participants, which communication between them uses the same channel. We aim to measure how the run-time of our tool is being affected by the growing complexity/size of the protocol, we do so by testing our tool on three different Scribble communication patterns:

- **Send/Receive** - We execute a loop of 100 runs, for each i^{th} run we repeat 10 times: Generate a protocol of two participants with i send/receive of different messages between them, with each message consisting of random payload values(as shown in Listing 7.2). Then we input the protocol into our tool and record the time taken for each stage of generation. At the end, we will then take the average time taken over the 10 runs for each stage of generation respectively to smooth out the graph.

- **Nested Choices** - We execute a loop of 100 runs and measure the time taken using the exact same method we used for send/receive. The only difference is, instead of having a protocol of purely i send/receive each i^{th} run, we have a protocol consisted of i nested choices, as shown in Listing 7.3

- **Loop** - We execute a loop of 10 runs, again we measure the time taken using the same method. During each i^{th} run, we have a protocol consisted of i nested choices, with each choice being a entry point of a loop and the first selection containing a goto(continue) to the entry of the loop, as shown in Listing 7.4

```
global protocol Test(role Client, role Svr) {
   HELLO55(number, string, string, number) from Svr to Client;
   HELLO96(string, number, number, string) from Svr to Client;
   HELLO31(number, string, number, string) from Client to Svr;
   ......
}
```

Listing 7.2: SEND/RECEIVE Test Protocol Snippet

```
global protocol Test(role Client, role Svr) {

choice at Client {
    HELLO77(number, string, number) from Client to Svr;
    choice at Svr {
        HELLO31(number, number, number) from Svr to Client;
        choice at Svr {
           ....
           }or {
           HELLO70(number, string) from Client to Svr;
        }
   or {
     HELLO91(string, number, number) from Svr to Client;
  }
or {
 HELLO86(number, string, string) from Svr to Client;
 } }
```

Listing 7.3: NESTED CHOICES Test Protocol Snippet

We eliminate the time taken for nuScr and EFSM generation as it is not part of our implementation and the time taken for them is much larger than other generations, mainly because a last portion of generation time is used by nuScr to map from a global protocol to local protocol, also as a part of FSM

```
global protocol Test(role Client, role Svr) {

rec Loop13 {
  choice at Svr {
    HELLO81(string, string, number, number) from Svr to Client;
    continue Loop13;
  } or {
    HELLO39(number, number, number, string) from Svr to Client;
    rec Loop12 {
        choice at Svr {
            HELLO60(string, string, string) from Svr to Client;
            continue Loop12;
         } or {
            HELLO44(string, string, string) from Svr to Client;
             rec Loop11 {
             ......
             }}}}}}
```

Listing 7.4: Loop Test Protocol Snippet

generation, a digraph is generated from the dot data by using a third party library pydot, this function is notoriously slow as it spent a lot of time parsing, a graph with 50 nodes took 1.88s to generate[48]. In comparison, the graph generated from nuScr has 48 nodes during the 10th iteration of the loop protocol. In Figure 31 we repeat the send/ receive and loop test above but including the generation time for nuScr and FSM generation to showcase the generation time differences between them and other generations. It also shows why we only took 10 iterations for the loop test, the generation time for FSm and nuScr combined grew exponentially and almost vertically after the 7th iteration, any generation beyond the 10th iteration would take hours to complete.

The result can be seen as shown in Figure 32. The class generation for each test grow exponentially with growing complexity as there's a huge amount of checks it need to perform and assign different instances of communication classes. Function generation time grows linearly for send receive but exponentially for the other two as more type matching functions needed to be generated with increasing amount of choices. Type generation and channel generation/merging on the other hand grows linearly and almost stationary for all of the tests. Type matching mechanism for type generation is comparatively simpler and more straightforward compared to that of function generation, much fewer lines of code is required. Channel generation/merging on the other hand is a more lightweight process to the other. Nevertheless, the total generation time excluding nuScr and FSM generation is still low during the 100th iteration of both send/receive and nested choices test, accounting to 0.057s and 0.243s respectively, we expect the same for the loop test if it ever reached the 100th iteration. For protocols with multiple participants, the total generation time will increase but the tendency of growth for each generation time of different tests is expected to reamin the same as of two participants.

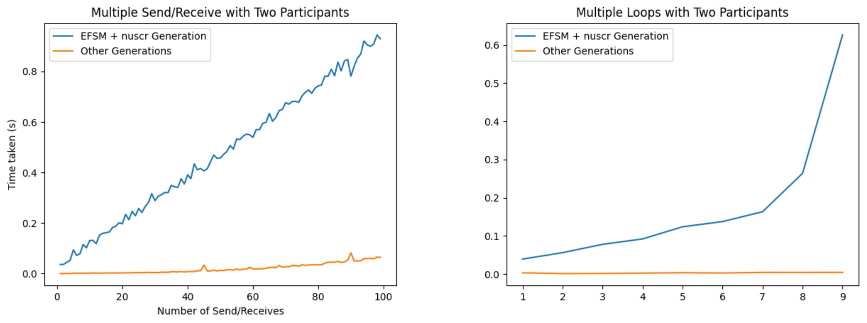
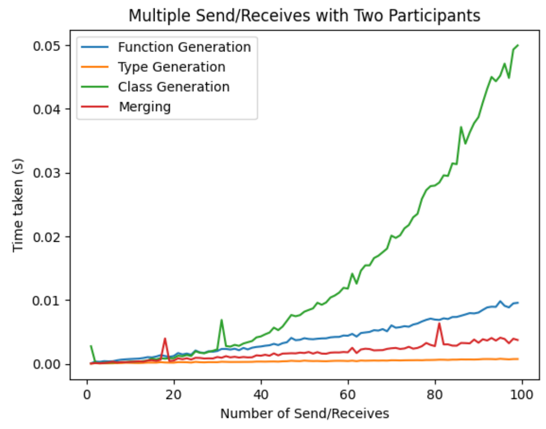
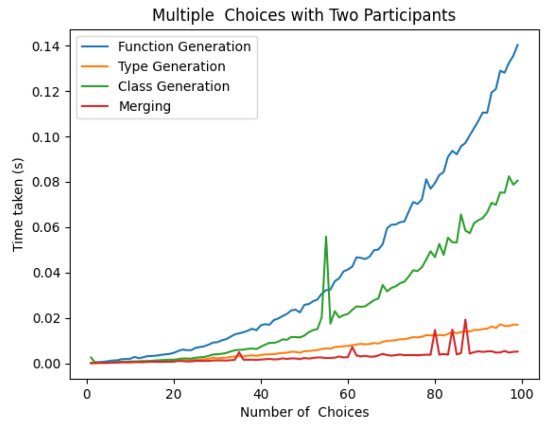


Figure 31: Time taken to generate Effpi type + functions from protocol





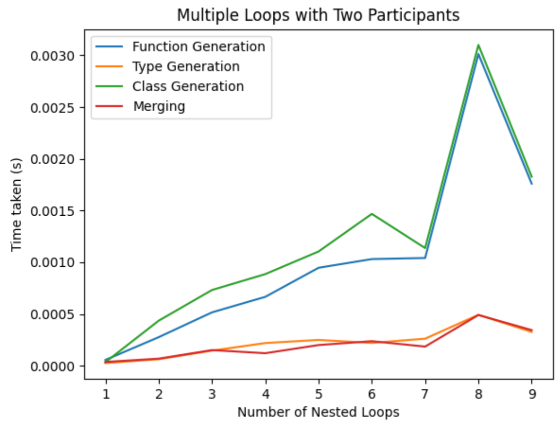


Figure 32: Time Taken for Different Stages of Generation

7.3 Case studies

In this section, we will demonstrate the expressiveness and functionality of our tool by using them on some real life case studies. We discuss in detail how the input protocol relates to the output and how it benefits the programmer.

7.3.1 Two Buyer with Negotiation Protocol

We consider the two buyer with negotiation protocol in Listing 7.5 with the local FSM of each participant shown in Figure 33. This protocol is comprised of a Travel Agent(denoted as S) and two buyers, A and B. Buyer A sends the travel agent a location where he or she wishes to travel and the travel agent pairs he or her up with another buyer, B, who wishes to to travel to the same destination. There's no direct communication between A and B, any communication between them is handled by the travel agent, which acts as a moderator. B doesn't have the privilege of cancelling or buying the plan, they can only offer a price which he or she is willing to pay (A variation where B can also buy and cancel the plan can be easily implemented bu having a nested choice). After receiving the price offer from B through the travel agent, A may request the cancellation of the purchase, proceed with the purchase, or request a price negotiation with B. If A decides to proceed with the purchase, the travel agent will send a confirmation to both parties alongside the departure date, every party then terminates. If A requests for a negotiation, B offers A a new price and A proceeds with the selection again. Otherwise, if A chooses to cancel the purchase, the travel agent will notify B of the cancellation and every party terminates.

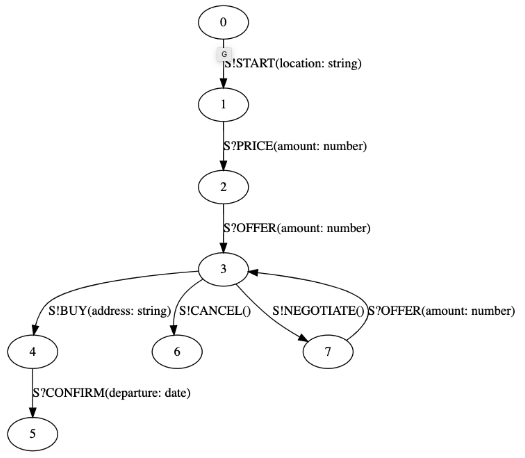
Type Generation

From Listing 7.6, we can observe that a case class is generated for each label in the protocol along with their respective arguments. The Effpi type generated for travel agent S, Buyer A and Buyer B is shown in Listing 7.7, 7.8 and 7.9 respectively. Sending/ Receiving of a single message is straightforward, for example, Buyer A has an output channel C_A_S_1 of type OutChannel[START] which is used to send the travel location to the travel agent, as seen in Line 6 of Buyer's A type. Conversely, the travel agent has an input channel C_S_A_1 of type InChannel[START] which receives the requests from Buyer A, as seen in Line 8 of the travel agent's type. Channels intended for selection/ branching have multiple labels in it's type, taking channel C_A_S_4 in Buyer A as an example, it has type OutChannel[NEGOTIATE|BUY|CANCEL] as it is use for selection to the travel agent, it may send out a message of either type NEGOTIATE, BUY or CANCEL. This can be seen in Line 9-11 of Buyer A's type, each line shows a possible selection A made through the channel C_A_S_4, follows by their respective continuation type. Take Line 9 as an example, Buyer A sends BUY request to the travel agent, the continuation type shows that Buyer A is expecting a message of type CONFIRM from the travel agent through channel C_S_A_5.

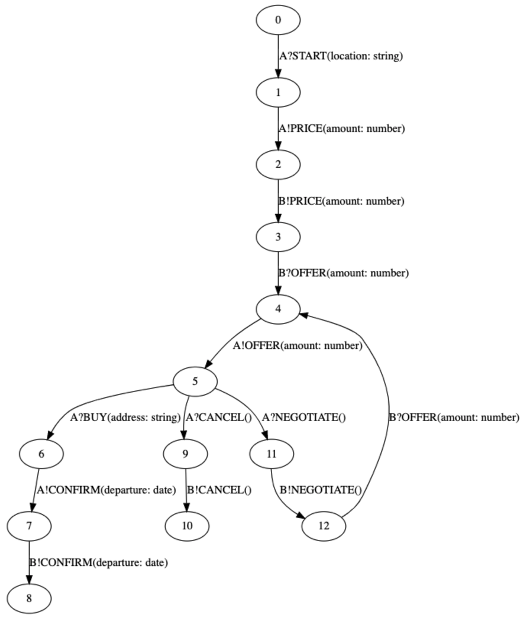
```
module TwoBuyerNegotiate;

global protocol TwoBuyerNegotiate(role A, role B, role S)
{
  START(location: string) from A to S;
  PRICE(amount: number) from S to A;
  PRICE(amount: number) from S to B;

  rec Loop {
     OFFER(amount: number) from B to S;
     OFFER(amount: number) from S to A;
     choice at A
     {
       BUY(address: string) from A to S;
       CONFIRM(departure: date) from S to A;
       CONFIRM(departure: date) from S to B;
     }
     or
     {
       CANCEL() from A to S;
       CANCEL() from S to B;
     }
     or
     {
       NEGOTIATE() from A to S;
       NEGOTIATE() from S to B;
       continue Loop;
     }
  }
```

Listing 7.5: Two Buyers With Negotiation Protocol

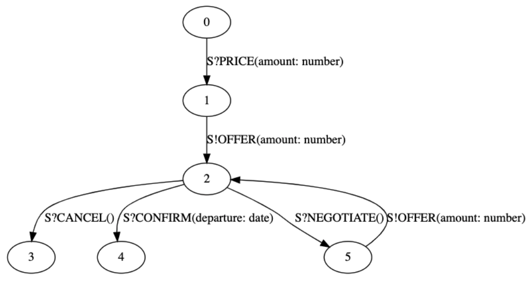
Now we shall look at how branching is being handled through type matching. Say Buyer A has sent a CANCEL request through channel C_A_S_4, we can see that this channel corresponds to the channel C_S_A_4 in the travel agent's type, with a channel type of InChannel[NEGOTIATE|BUY|CANCEL]. In Line 11 of the travel agent's type, we can see that right after the travel agent receive a message through channel C_S_A_4, it called the type definition S2 immediately in Line 12, alongside the message type and some channels which are required for the type continuation as the parameter. S2 is what we called as a type matching definition, which is responsible for identifying the actual type of a variable (message in our case) and proceeds with the correct type continuation. In Line 19, we can see that S2 tries to match the message type received in S earlier, Line 20-22 indicates the possible types of the message and their corresponding type continuation. For our example where the travel agent received a message of type CANCEL, it will proceed with Line 21, send CANCEL message to Buyer B through channel C_S_B_4 before it terminates. Type for travel agent and Buyer B has one type matching definition each as they have branching in their local protocol, we can easily observe this from the FSM of the participants.



(a) Buyer A



(b) Travel Agent S



(c) Buyer B

Figure 33: FSM for each participant in Two Buyers With Negotiation Protocol

Understanding how message sending/receiving, selection and branching is being shown in Effpi type, most part of the type body is clear and self-explanatory. We will now look at how recursion is being handled in each participant's type. From Line 9 and 27 of the global protocol in Listing 7.5, we can observe that each time Buyer A sends a NEGOTIATE request to the travel agent, travel agent will inform B about it and every participant will restart the process where Buyer B sends an OFFER price to the travel agent and the travel agent notify Buyer A about it. In each participant's type, Rec[X, ...] indicates the entry of a loop and Loop[X] means going to the start of that loop. From travel agent's type, we can see that the travel agent returns to the point where they await to receive an OFFER from Buyer B through channel C_S_B_2 in Line 10, after they received a NEGOTIATE request from Buyer A through channel C_S_A_4 and sent a NEGOTIATE request to Buyer B through channel C_S_B_5, as seen in Line 22 and 10. For Buyer B, it returns to the state in Line 6, where he or she is about to send an OFFER to the travel agent through channel C_B_S_2, after receiving a NEGOTIATE request from the travel agent through channel C_B_S_3, as seen in Line 4 and 14. Lastly, once Buyer A sent a NEGOTIATE request to travel agent through channel C_A_S_4 in Line 9, it returns to Line 8 where he or she awaits an OFFER from the travel agent through channel C_A_S_3. Again, these types can be checked accordingly to the FSM of each participant.

```
case class OFFER(amount:Int)
case class CONFIRM(departure:Date)
case class PRICE(amount:Int)
case class NEGOTIATE()
case class BUY(address:String)
case class START(location:String)
case class CANCEL()
```

Listing 7.6: Case Classes Generated for Two Buyer With Negotiation Protocol

```
1   type S[C_S_A_1 <: InChannel[START], C_S_A_2 <: OutChannel[PRICE],
2          C_S_B_1 <: OutChannel[PRICE], C_S_B_2 <: InChannel[OFFER],
3          C_S_A_3 <: OutChannel[OFFER],
4          C_S_A_4 <: InChannel[BUY|CANCEL|NEGOTIATE],
5          C_S_A_5 <: OutChannel[CONFIRM], C_S_B_3 <: OutChannel[CONFIRM],
6          C_S_B_4 <: OutChannel[CANCEL], C_S_B_5 <: OutChannel[NEGOTIATE]] =
7
8    In[C_S_A_1, START,(x:START) =>
9        Out[C_S_A_2,PRICE] >>: Out[C_S_B_1,PRICE] >>:
10         Rec[RecS4, In[C_S_B_2, OFFER, (x:OFFER) => Out[C_S_A_3,OFFER] >>:
11             In[C_S_A_4, BUY|CANCEL|NEGOTIATE, (x:BUY|CANCEL|NEGOTIATE) =>
12                  S2[x.type,C_S_A_5,C_S_B_3,C_S_B_4,C_S_B_5]]]]]
13
14
15  type S2[X2 <: BUY|CANCEL|NEGOTIATE, C_S_A_5 <: OutChannel[CONFIRM],
16          C_S_B_3 <: OutChannel[CONFIRM], C_S_B_4 <: OutChannel[CANCEL],
17          C_S_B_5 <: OutChannel[NEGOTIATE]] <: Process =
18
19  X2 match {
20    case BUY => Out[C_S_A_5,CONFIRM] >>: Out[C_S_B_3,CONFIRM] >>: PNil
21    case CANCEL => Out[C_S_B_4,CANCEL] >>: PNil
22    case NEGOTIATE => Out[C_S_B_5,NEGOTIATE] >>: Loop[RecS4]}
```

Listing 7.7: Effpi Type Generated for Travel Agent S

```
type A[C_A_S_1 <: OutChannel[START], C_A_S_2 <: InChannel[PRICE],
       C_A_S_3 <: InChannel[OFFER],
       C_A_S_4 <: OutChannel[NEGOTIATE|BUY|CANCEL],
       C_A_S_5 <: InChannel[CONFIRM]] =

  Out[C_A_S_1,START] >>:
     In[C_A_S_2, PRICE, (x:PRICE) =>
        Rec[RecA3, In[C_A_S_3, OFFER, (x:OFFER) =>
          ((Out[C_A_S_4,NEGOTIATE] >>: Loop[RecA3])|
          (Out[C_A_S_4,BUY] >>: In[C_A_S_5, CONFIRM, (x:CONFIRM) => PNil])|
          (Out[C_A_S_4,CANCEL] >>: PNil))]]]
```

Listing 7.8: Effpi Type Generated for Buyer A

```
type B[C_B_S_1 <: InChannel[PRICE],
       C_B_S_2 <: OutChannel[OFFER],
       C_B_S_3 <: InChannel[NEGOTIATE|CONFIRM|CANCEL]] =

   In[C_B_S_1, PRICE, (x:PRICE) =>
       Rec[RecB2, Out[C_B_S_2,OFFER] >>:
          In[C_B_S_3, NEGOTIATE|CONFIRM|CANCEL,
              (x:NEGOTIATE|CONFIRM|CANCEL) => B2[x.type]]]]


type B2[X2 <: NEGOTIATE|CONFIRM|CANCEL] <: Process =

X2 match {
  case NEGOTIATE => Loop[RecB2]
  case CONFIRM => PNil
  case CANCEL => PNil}
```

Listing 7.9: Effpi Type Generated for Buyer B

Function Generation

The functions generated for Buyer A, Buyer B and the travel agent can be seen in Listing 7.10, 7.11 and 7.12 respectively. We can easily check that the function bodies conform to their communication type we described right before, send and receive methods are used to pass messages through channels, type matching functions are called to check the actual received type when it's a branching, selection consisted of if else cases, rec and loop and self explanatory.

Hence we shall discuss about the possible extensions/modifications a programmer can apply to the function. Actual selection made for each choice is picked on random by default, the programmer can modify the condition for each selection to be made. For example we have modified the selection in Buyer's A function as seen in Line 10 - 16, Buyer A cancels the purchase if the total price exceeds 1000, or else if Buyer B offers a price more than half of the original price, Buyer A will proceeds with the purchase, otherwise Buyer A will ask to negotiate with Buyer B. The programmer can also modify the argument values of the case classes/labels. For example in Line 5 of Buyer A's function, they requested to go to Paris, this can be easily modified to some other location names. A programmer can also add some operations in between, which is highly flexible. For example, in Line 14 and 15 of travel agent's function, it is modified such that the travel agent sends Buyer A an OFFER of the exact amount offered by Buyer B, another variation of this is having the travel agent to deduct a percentage of the amount offered by Buyer B as their own commission, before sending the reduced amount to Buyer A. Also in travel agent's function, a map can be added to store the price for different locations, from Line 10 - 12, the travel agent looks up the location in the map after receiving the request from Buyer A, then proceed to send the price to both buyers. Overall there's a huge variety of extensions a programmer can implement,as long they don't modify the communication between participants. A programmer doesn't have to worry much about adding modifications to the functions that will break the communication protocol as any code that violates the protocol won't type check successfully nor compile.

```
def a(c_A_S_1: OutChannel[START],
   c_A_S_2: InChannel[PRICE], c_A_S_3: InChannel[OFFER],
   c_A_S_4: OutChannel[NEGOTIATE|BUY|CANCEL], c_A_S_5: InChannel[CONFIRM])
   :A[c_A_S_1.type,c_A_S_2.type,c_A_S_3.type,c_A_S_4.type,c_A_S_5.type] ={
   send(c_A_S_1,START("Paris")) >> {
       receive(c_A_S_2) {
         (x:PRICE) => rec(RecA3){
           receive(c_A_S_3) {
               (x1:OFFER) =>
               if(x.amount > 1000){
                  send(c_A_S_4,CANCEL()) >> {nil}
               }
               else if(x1.amount < x.amount / 2){ send(c_A_S_4,NEGOTIATE())
                                 >> { loop(RecA3)}}
               else{send(c_A_S_4,BUY("Street")) >> {
                         receive(c_A_S_5) { (x:CONFIRM) => nil}}}}}}}
```

Listing 7.10: Function Generated for Buyer A

```
def b(c_B_S_1: InChannel[PRICE], c_B_S_2: OutChannel[OFFER],
   c_B_S_3: InChannel[NEGOTIATE|CONFIRM|CANCEL]
):B[c_B_S_1.type,c_B_S_2.type,c_B_S_3.type] ={
   receive(c_B_S_1) {(x:PRICE) =>
      rec(RecB2){ send(c_B_S_2,OFFER(35)) >> {
            receive(c_B_S_3) { (x:NEGOTIATE|CONFIRM|CANCEL) => b2(x)}}}}}


def b2(x2: NEGOTIATE|CONFIRM|CANCEL):B2[x2.type] =
   x2 match {
      case x2 : NEGOTIATE => {loop(RecB2)}
      case x2 : CONFIRM => {nil}
      case x2 : CANCEL => {nil}}
```

Listing 7.11: Function Generated for Buyer B

```
 def s( c_S_A_1: InChannel[START], c_S_A_2: OutChannel[PRICE],
        c_S_B_1: OutChannel[PRICE], c_S_B_2: InChannel[OFFER],
        c_S_A_3: OutChannel[OFFER], c_S_A_4:InChannel[BUY|CANCEL|NEGOTIATE],
        c_S_A_5: OutChannel[CONFIRM], c_S_B_3: OutChannel[CONFIRM],
        c_S_B_4: OutChannel[CANCEL], c_S_B_5: OutChannel[NEGOTIATE]
   ):S[c_S_A_1.type,c_S_A_2.type,c_S_B_1.type,
       c_S_B_2.type,c_S_A_3.type,c_S_A_4.type,
       c_S_A_5.type,c_S_B_3.type,c_S_B_4.type,c_S_B_5.type] ={
   receive(c_S_A_1) {
      (x:START) =>
      send(c_S_A_2,PRICE(50)) >> {
         send(c_S_B_1,PRICE(50)) >> {
            rec(RecS4){
               receive(c_S_B_2) {
                  (x:OFFER) =>send(c_S_A_3,OFFER(x.amount)) >> {
                     receive(c_S_A_4) {(x:BUY|CANCEL|NEGOTIATE) =>
                        s2(x,c_S_A_5,c_S_B_3,c_S_B_4,c_S_B_5)}}}}}}}


def s2(x2: BUY|CANCEL|NEGOTIATE,
       c_S_A_5: OutChannel[CONFIRM], c_S_B_3: OutChannel[CONFIRM],
       c_S_B_4: OutChannel[CANCEL], c_S_B_5: OutChannel[NEGOTIATE]
):S2[x2.type,c_S_A_5.type,c_S_B_3.type,c_S_B_4.type,c_S_B_5.type] =
   x2 match {
      case x2 : BUY => {send(c_S_A_5,CONFIRM(new Date())) >> {
            send(c_S_B_3,CONFIRM(new Date())) >> {nil}}}
      case x2 : CANCEL => {send(c_S_B_4,CANCEL()) >> {nil}}
      case x2 : NEGOTIATE => {send(c_S_B_5,NEGOTIATE()) >> {loop(RecS4)}}}
```

Listing 7.12: Function Generated for Travel Agent S

Channel Generation

Our tool has a built in channel generating/merging functionality, the channels for our program are generated and assigned accordingly as shown in Listing 7.13. We can see that channel c_A_S_1 of buyer A and channel c_S_A_1 of travel agent S are assigned the same channel of type START as this channel is meant for sending/ receiving the travel location. Other channel assignments are assigned with the same method, we can check them by looking at the parameters of the functions of each participant. We then execute the program and check that the communication sequence between participants follow that as in the global protocol, we also check that each participant terminates eventually.

Metrics

For this program, we have generated 7 different case classes, 5 different type definitions, 32 lines of function bodies and the total generation time is 0.152 s. The participant with the longest type definition is the travel agent with 23 lines.

7.3.2 Bank Server with Microservices

We shall now demonstrate the program generated for the Bank protocol, the protocol along it's sub protocols can be seen in Listing 7.14, 7.15 and 7.16. The

```

val(cStart, cPrice1, cPrice2, cOffer1, cOffer2, cMatch,
    cSMatch2, cConfirm1)
    = (Channel[START](), Channel[PRICE](), Channel[PRICE](),
       Channel[OFFER](), Channel[OFFER](),
       Channel[BUY| CANCEL| NEGOTIATE](),
       Channel[CONFIRM | CANCEL | NEGOTIATE](),
       Channel[CONFIRM]())

eval(par(a(cStart,cPrice1,cOffer2, cMatch, cConfirm1),
         b(cPrice2, cOffer1, cSMatch2),
         s(cStart, cPrice1, cPrice2, cOffer1, cOffer2,
           cMatch,cSMatch2, cConfirm1)))
```

Listing 7.13: Channels generated for Two Buyers With Negotiation Protocol

communication sequence diagrams for each sub-protocols are shown in Figure 34. This protocol involves four participants: Client, Svr, SvrVer and SvrAct. Svr represents the main server of a bank and Client is the client of the bank, Svr is responsible for all communications with Client. SvrVer and SvrAct are microservice servers of the bank which handle users verification and users transaction respectively, they also communicate with Svr directly. We can observe from both the protocol and communication sequence diagram, the program starts with Client initiating a connection/session with Svr by sending message of label Connect. It then sends the Login details to Svr which Svr will forward to SvrVer for further verification. SvrVer then notifies Svr whether the verification is successful or not, an error code will be sent along if the SvrVer failed to verify the user, informing Svr the reason of the failure. With this error code, Svr decides if the user shall try to login again or terminate all operation. Otherwise if the verification is successful, Client is allowed to proceed by withdraw, deposit money or cancel all operation. Similarly, Svr will pass the request to SvrAct and SvrAct shall notify Svr regarding the success or failure of the transaction. If the transaction suceed, Svr will allow Client which action to perform again, otherwise it will decide to cancel the operation or ask Client to retry transaction based on the error code received from SvrAct.

Modification

We shall demonstrate how a working example can be built by modifying the default program generated without altering the communication between participants. We shall first define a few rules on how SvrVer and Svr decides which message label to send after verifying the user credentials:

1. If the user doesn't exist, SvrVer sends Svr Fail and Svr shall ask the Client to retry login in.

2. If the user exists but is blacklisted, Svr sends Svr Fail and Svr shall terminate the session with Client.

```
global protocol Bank(role Client, role Svr,
                     role SvrVer, role SvrAct) {
    Connect() from Client to Svr;
    do login(Client, Svr, SvrVer, SvrAct);
}

aux global protocol login(role Client, role Svr,
                          role SvrVer, role SvrAct) {
   Login(username: string, pw: string) from Client to Svr;
   Login(username: string, pw: string) from Svr to SvrVer;
   choice at SvrVer{
     Success(message:String) from SvrVer to Svr;
     Success(message:String) from Svr to Client;
     Continue() from Svr to SvrAct;
     do action(Client, Svr, SvrVer, SvrAct);
} or {
     Fail(errorCode: number, errorMessage: string)
                                        from SvrVer to Svr;
     choice at Svr{
     Cancel() from Svr to Client;
     Cancel() from Svr to SvrVer;
     Cancel() from Svr to SvrAct;
  } or {
     Retry(message:String) from Svr to Client;
     Retry(message:String) from Svr to SvrVer;
     Retry(message:String) from Svr to SvrAct;
     do login(Client, Svr, SvrVer, SvrAct);
  }}
}
```

Listing 7.14: BANK Protocol and LOGIN sub-Protocol

3. If user has more than three failed login attempts, with the latest failed login attempt occur within 10 minutes ago, Svr sends Svr Fail and Svr shall terminate the session with Client.

4. If latest login attempt occur more than 10 minutes ago from now, failed attempt count is clear to 0.

5. If the user matched the password, failed attempts count is clear to 0, SvrVer sends Success to Svr and Svr shall redirect it to Client.

6. Otherwise, failed attempts count is incremented and latest failed login time is updated, SvrVer sends Svr Fail and Svr shall ask the Client to retry login in.

We then define the rules on how SvrAct and Svr decides which message labels to send after received transaction request from the Client:

1. If the user has already performed more than 10 transactions within 24 hours, SvrAct sends Svr Fail and Svr terminates the operation with Client.

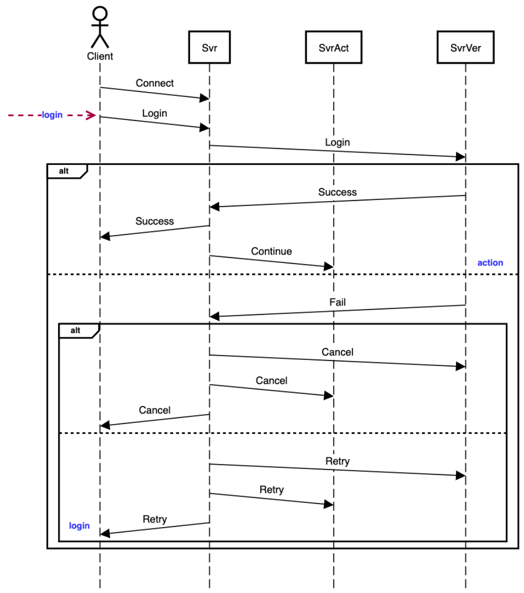
```
aux global protocol action(role Client, role Svr,
                           role SvrVer, role SvrAct) {
   choice at Client{
       Withdraw1(amount:number) from Client to Svr;
       Withdraw2(username:string, amount:number)
                                 from Svr to SvrAct;
       Continue() from Svr to SvrVer;
       do response(Client, Svr, SvrVer, SvrAct);
   }
    or{
      Deposit1(username:number) from Client to Svr;
      Deposit2(username:string, amount:number)
                                 from Svr to SvrAct;
      Continue() from Svr to SvrVer;
      do response(Client, Svr, SvrVer, SvrAct);
    }or{
      Cancel() from Client to Svr;
      Cancel() from Svr to SvrVer;
      Cancel() from Svr to SvrAct;
    }
}
```

Listing 7.15: ACTION sub-Protocol

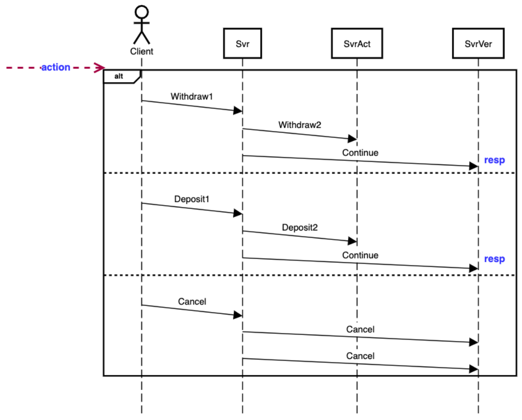
```
aux global protocol response(role Client, role Svr,
                             role SvrVer, role SvrAct) {
   choice at SvrAct{
     Success(message:String) from SvrAct to Svr;
     Success(message:String) from Svr to Client;
     Continue() from Svr to SvrVer;
     do action(Client, Svr, SvrVer, SvrAct);
} or {
     Fail(errorCode: number, errorMessage: string)
                                 from SvrAct to Svr;
     choice at Svr{
     Cancel() from Svr to Client;
     Cancel() from Svr to SvrVer;
     Cancel() from Svr to SvrAct;
  } or {
     Retry(message:String) from Svr to Client;
     Continue() from Svr to SvrVer;
     Retry(message:String) from Svr to SvrAct;
     do action(Client, Svr, SvrVer, SvrAct);
  }
}
}
```

Listing 7.16: RESPONSE sub-Protocol

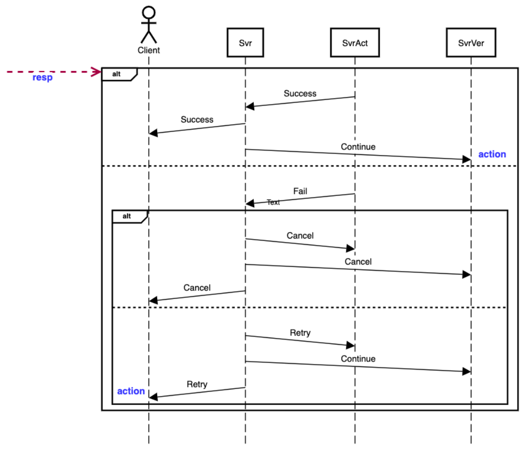
2. If the user tried to withdraw more than $500 per transaction, SvrAct sends Svr Fail and Svr terminates the operation with Client.



(a) Communication Sequence Diagram for BANK Protocol and LOGIN sub-Protocol



(b) Communication Sequence Diagram for ACTION sub-Protocol



(c) Communication Sequence Diagram for RESPONSE sub-Protocol

Figure 34: Communication Sequence Diagram for BANK Protocol and it's sub-Protocols

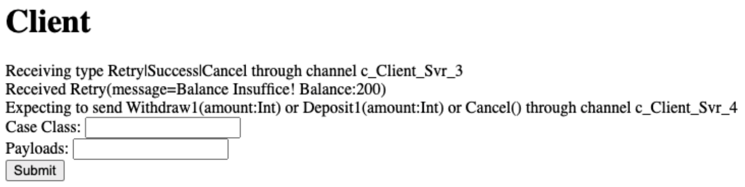
3. If the user exceed the withdraw limit of \$1000 within a week (including current withdraw action), SvrAct sends Svr Fail and Svr request Client to retry.

4. If the user exceed the deposit limit of $5000 within a week (including current deposit action), SvrAct sends Svr Fail and Svr request Client to retry.

5. If the user tries to withdraw but has insufficient balance, SvrAct sends Svr Fail and Svr request Client to retry.

6. Otherwise, SvrAct performs the desired action and update the transaction history and balance of the user, it notifies Svr about the Success and Svr allows Client to proceed with the transaction.

Note that we design such that the Svr terminates the operation with Client when Client's actions are deemed as fraudulent. Based on our design, SvrVer will need to keep a database which need to record for each user the username, password, number of failed login attempts, last failed login time and whether the user is blacklisted. SvrAct will need a database to keep track of the transaction history and balance for each user. We implemented them as Json files in their folder which they can freely read and write. Client is generated with a user interface using our tool to allow the user to choose which message labels and payloads to be sent. A snippet of the user interface is shown in Figure 35, where Svr requested Client to Retry a transaction due to a failed transaction caused by trying to withdraw an amount larger than the account balance. The user can choose to withdraw an amount, deposit an amount or cancel the operation.


Client

Receiving type Retry|Success|Cancel through channel c_Client_Svr_3
Received Retry(message=Balance Insuffice! Balance:200)
Expecting to send Withdraw1(amount:Int) or Deposit1(amount:Int) or Cancel() through channel c_Client_Svr_4
Case Class:
Payloads:

Figure 35: User Interface for Client

We shall now discuss how the default body of the function body is modified without affecting the communication between the participants. We shall only implement the business logic for Svr, SvrAct and SvrVer. Client can decides itself which message labels to send. Function svrVer of SvrVer is shown in Listing 7.18, this is where SvrVer receives the login details of the user through Svr and proceeds with the verification. In Line 11-12, we use a helper method checkCredentials that verifies the username and password and return a boolean variable suceed that indicate if the login if successful, an error code of type integer that indicates the status of the login and a message of type string that describes the result of verification. In Line 13, we check if the verification succeed and send Success to Svr alongside with the message in line 14. If the verification is unsuccessful, we send Fail to Svr in Line 20 along the error code and message. Function svrAct_3 of SvrAct is shown in Listing 7.17, this is where SvrAct receives transaction request of Client through Svr. Similarly, we use a helper function performAction to try to execute the transaction and return the results, as seen in Line 8 and 9, 20 and 21. It then sends Success or Fail to Svr based on the transaction result, as seen in Line 10 and 22. We shall now look at a few modified functions of Svr. Function svr_4 of Svr is shown in Listing 7.19 where Svr received the results for the transaction from SvrAct and notify Client. If the response is Success, we will redirect the message to Client in Line 14. Otherwise, Svr will check the error code received in Line 18, if the error code is 202 (Daily transaction limit reached) or 203 (Withdraw too much per transaction), it will terminate the operation with Client, otherwise it will send the Client a error message,requesting them to retry a transaction in Line 25. Similarly for svr_2 (shown in Listing 7.21), where Svr receives verification result from SvrVer and proceeds to notify Client about it. In the case where verification failed, Svr will terminate operation with Client if the error code is 403 (Blacklisted user) or 404 (Too many failed attempts within 10 minutes), otherwise it will request Client to retry login. svr_3 function in Lisitng 7.20 is where Svr redirect Client's transaction request to SvrVer. Note that the transaction request from Client to Svr doesn't include username but Svr to SvrVer does, this is to prevent the user trying to access someone else's account after login in. Hence, the Svr needs to keeps track of the username used for login, we can see in the first parameter of svr_3 and svr_2 how we include a username parameter to deal with this. Also, in the original function generated, there's another svr_5 function that is called at Line 30 of svr_3. Result received for withdraw is handled by s_4 while result received for deposit is handled by s_5, both function share the same communication type. Since we handle the result for deposit and withdraw in the same approach, we can replace s_4 with s_5 and hence removing both s5 function and S5 type from our program.

Clone Channel Elimination

The clone channel detection and elimination feature of our tool is applicable to this example. The channel generation and assignment of the program with and without clone channel elimination are shown in Listing 7.23 and Listing 7.22 respectively. We observe that without clone channel elimination, 17 actual channels are generated with total of 51 local channel instances. With clone channel elimination. 12 actual channels are generated with 32 local channel instances. We can find out the mergeable channels from the channels generated without clone channel elimination:

1. c3, c4, c5 are used by Svr to send Client message of type Success|Retry|Cancel, informing Client about the outcome of the verification or transaction.
2. c15 and c16 are used by Svr to notify SvrVer to Continue with the execution or Cancel the operation.
3. c10 and c12 are used by Svr to notify SvrAct whether the Client is expected

```
private def svrAct_3(
   action: Withdraw2|Deposit2|Cancel,
   c_SvrAct_Svr_3: OutChannel[Success|Fail],
   c_SvrAct_Svr_4: InChannel[Retry|Cancel]
):SvrAct_3[action.type,c_SvrAct_Svr_3.type,c_SvrAct_Svr_4.type] =
   action match {
      case action : Withdraw2 => {
         val (succeed, errorCode, errorMessage) =
               performAction(action.username, action.amount, true)
         if(succeed){
            send(c_SvrAct_Svr_3,Success(errorMessage)) >> {
               loop(RecSvrAct_6)}
         }
         else{
            send(c_SvrAct_Svr_3,Fail(errorCode,errorMessage)) >> {
               receive(c_SvrAct_Svr_4) {
                  (x_4:Retry|Cancel) => svrAct_4(x_4) }}}}

      case action : Deposit2 => {
         val (succeed, errorCode, errorMessage) =
             performAction(action.username, action.amount, false)
         if(succeed){
            send(c_SvrAct_Svr_3,Success(errorMessage)) >> {
               loop(RecSvrAct_6)
            }
         }
         else{
            send(c_SvrAct_Svr_3,Fail(errorCode,errorMessage)) >> {
               receive(c_SvrAct_Svr_4) {
                  (x_5:Retry|Cancel) => svrAct_5(x_5)}}}
      }
      case action : Cancel => { nil }
   }
```

Listing 7.17: Function svrAct_3 of SvrAct

```
def svrVer(
   c_SvrVer_Svr_1: InChannel[Login],
   c_SvrVer_Svr_2: OutChannel[Success|Fail],
   c_SvrVer_Svr_3: InChannel[Continue|Cancel],
   c_SvrVer_Svr_4: InChannel[Retry|Cancel]
):SvrVer[c_SvrVer_Svr_1.type,c_SvrVer_Svr_2.type,
         c_SvrVer_Svr_3.type,c_SvrVer_Svr_4.type] ={
   rec(RecSvrVer_0){
      receive(c_SvrVer_Svr_1) {
         (loginDetails :Login) =>
         val (succeed, errorCode, message) =
            checkCredentials(loginDetails.username, loginDetails.pw)
         if(succeed){
            send(c_SvrVer_Svr_2,Success(message)) >> {
               rec(RecSvrVer_4){
                  receive(c_SvrVer_Svr_3) {
                     (x_2:Cancel|Continue) =>
                        svrVer_2(x_2,c_SvrVer_Svr_3)}}}}
         else{
            send(c_SvrVer_Svr_2,Fail(errorCode,message)) >> {
               receive(c_SvrVer_Svr_4) {
                  (x_4:Retry|Cancel) => svrVer_4(x_4)}}}}}}
```

Listing 7.18: Function svrVer of SvrVer

```
   private def svr_4(
      response: Success|Fail,
      c_Svr_Client_3: OutChannel[Success],
      c_Svr_SvrVer_3: OutChannel[Continue],
      c_Svr_Client_4: OutChannel[Cancel|Retry],
      c_Svr_SvrVer_4: OutChannel[Cancel],
      c_Svr_SvrAct_8: OutChannel[Cancel],
      c_Svr_SvrAct_9: OutChannel[Retry]
   ):Svr_4[response.type,c_Svr_Client_3.type,c_Svr_SvrVer_3.type,
           c_Svr_Client_4.type,c_Svr_SvrVer_4.type,c_Svr_SvrAct_8.type,
           c_Svr_SvrAct_9.type] =
      response match {
         case response : Success => {
            send(c_Svr_Client_3,response) >> {
               send(c_Svr_SvrVer_3,Continue()) >> {
                  loop(RecSvr_8)}}}
         case response : Fail => {
            if(response.errorCode == 202 || response.errorCode == 203){
               send(c_Svr_Client_4,Cancel()) >> {
                  send(c_Svr_SvrVer_4,Cancel()) >> {
                     send(c_Svr_SvrAct_8,Cancel()) >> { nil }}}}
            else{
               send(c_Svr_Client_4,Retry(response.errorMessage)) >> {
                  send(c_Svr_SvrVer_3,Continue()) >> {
                     send(c_Svr_SvrAct_9,Retry(response.errorMessage)) >> {
                        loop(RecSvr_8)}}}}}}
```

Listing 7.19: Function svr_4 of Svr

to Retry the transaction or Cancel the operation, one for each withdraw and deposit case.

4. c9 and c11 are used by SvrAct to notify Svr whether the transaction has Success or Fail, one for each withdraw and deposit case.

Merging the channels into one for each case, we shall reduce 5 total actual channels generated.

```
private def svr_3(
  username: String, action: Withdraw1|Deposit1|Cancel,
  c_Svr_SvrAct_4: OutChannel[Withdraw2],
  c_Svr_SvrAct_7: InChannel[Success|Fail],
  c_Svr_Client_3: OutChannel[Success],
  c_Svr_SvrVer_3: OutChannel[Continue],
  c_Svr_Client_4: OutChannel[Cancel|Retry],
  c_Svr_SvrVer_4: OutChannel[Cancel],
  c_Svr_SvrAct_8: OutChannel[Cancel],
  c_Svr_SvrAct_9: OutChannel[Retry],
  c_Svr_SvrAct_5: OutChannel[Deposit2],
  c_Svr_SvrAct_6: OutChannel[Cancel]
   ):Svr_3[action.type,c_Svr_SvrAct_4.type,c_Svr_SvrAct_7.type,
           c_Svr_Client_3.type,c_Svr_SvrVer_3.type,c_Svr_Client_4.type,
           c_Svr_SvrVer_4.type,c_Svr_SvrAct_8.type,c_Svr_SvrAct_9.type,
           c_Svr_SvrAct_5.type,c_Svr_SvrAct_6.type] =
      action match {
         case action : Withdraw1 => {
            send(c_Svr_SvrAct_4,Withdraw2(username, action.amount)) >> {
               send(c_Svr_SvrVer_3,Continue()) >> {
                  receive(c_Svr_SvrAct_7) {
                    (x_4:Success|Fail) =>
                    svr_4(x_4,c_Svr_Client_3,c_Svr_SvrVer_3,c_Svr_Client_4,
                          c_Svr_SvrVer_4,c_Svr_SvrAct_8,c_Svr_SvrAct_9)}}}}
         case action : Deposit1 => {
            send(c_Svr_SvrAct_5,Deposit2(username,action.amount)) >> {
               send(c_Svr_SvrVer_3,Continue()) >> {
                  receive(c_Svr_SvrAct_7) {
                    (x_5:Success|Fail) =>
                    svr_4(x_5,c_Svr_Client_3,c_Svr_SvrVer_3,c_Svr_Client_4,
                          c_Svr_SvrVer_4,c_Svr_SvrAct_8,c_Svr_SvrAct_9)}}}}
         case action : Cancel => {
            send(c_Svr_SvrVer_4,Cancel()) >> {
               send(c_Svr_SvrAct_6,Cancel()) >> { nil }}}}
```

Listing 7.20: Function svr_3 of Svr

```
private def svr_2(
      username : String,
      verified: Success|Fail,
      c_Svr_SvrAct_1: OutChannel[Continue],
      c_Svr_Client_5: InChannel[Withdraw1|Deposit1|Cancel],
      c_Svr_SvrAct_4: OutChannel[Withdraw2],
      c_Svr_SvrAct_7: InChannel[Success|Fail],
      c_Svr_Client_3: OutChannel[Success],
      c_Svr_SvrVer_3: OutChannel[Continue],
      c_Svr_Client_4: OutChannel[Cancel|Retry],
      c_Svr_SvrVer_4: OutChannel[Cancel],
      c_Svr_SvrAct_8: OutChannel[Cancel],
      c_Svr_SvrAct_9: OutChannel[Retry],
      c_Svr_SvrAct_5: OutChannel[Deposit2],
      c_Svr_SvrAct_6: OutChannel[Cancel],
      c_Svr_SvrVer_5: OutChannel[Cancel],
      c_Svr_SvrAct_2: OutChannel[Cancel],
      c_Svr_SvrVer_6: OutChannel[Retry],
      c_Svr_SvrAct_3: OutChannel[Retry]
   ):Svr_2[verified.type,c_Svr_SvrAct_1.type,c_Svr_Client_5.type,
           c_Svr_SvrAct_4.type,c_Svr_SvrAct_7.type,c_Svr_Client_3.type,
           c_Svr_SvrVer_3.type,c_Svr_Client_4.type,c_Svr_SvrVer_4.type,
           c_Svr_SvrAct_8.type,c_Svr_SvrAct_9.type,c_Svr_SvrAct_5.type,
           c_Svr_SvrAct_6.type,c_Svr_SvrVer_5.type,c_Svr_SvrAct_2.type,
           c_Svr_SvrVer_6.type,c_Svr_SvrAct_3.type] =
      verified match {
         case verified : Success => {
            send(c_Svr_Client_3, verified) >> {
               send(c_Svr_SvrAct_1,Continue()) >> {
                  rec(RecSvr_8){
                     receive(c_Svr_Client_5) {
                        (x_3:Withdraw1|Deposit1|Cancel) =>
                        svr_3(username, x_3,c_Svr_SvrAct_4,c_Svr_SvrAct_7,
                              c_Svr_Client_3,c_Svr_SvrVer_3,c_Svr_Client_4,
                              c_Svr_SvrVer_4,c_Svr_SvrAct_8,c_Svr_SvrAct_9,
                              c_Svr_SvrAct_5,c_Svr_SvrAct_6)}}}}}
         case verified : Fail => {
            if(verified.errorCode == 403 || verified.errorCode == 404 ){
               send(c_Svr_Client_4,Cancel()) >> {
                  send(c_Svr_SvrVer_5,Cancel()) >> {
                     send(c_Svr_SvrAct_2,Cancel()) >> { nil }}}}
            else{
               send(c_Svr_Client_4,Retry(verified.errorMessage)) >> {
                  send(c_Svr_SvrVer_6,Retry(verified.errorMessage)) >> {
                     send(c_Svr_SvrAct_3,Retry(verified.errorMessage)) >> {
                        loop(RecSvr_1) }}}}}}
```

Listing 7.21: Function svr_2 of Svr

```
val(c1, c2, c3, c4, c5, c6, c7, c8, c9, c10,
    c11, c12, c13, c14, c15, c16, c17) =
     (Channel[Connect](),
     Channel[Login](),
     Channel[Success|Retry|Cancel](),
     Channel[Success|Retry|Cancel](),
     Channel[Success|Retry|Cancel](),
     Channel[Withdraw1|Cancel|Deposit1](),
     Channel[Continue|Retry|Cancel](),
     Channel[Withdraw2|Cancel|Deposit2](),
     Channel[Success|Fail](),
     Channel[Retry|Cancel](),
     Channel[Success|Fail](),
     Channel[Retry|Cancel](),
     Channel[Login](),
     Channel[Success|Fail](),
     Channel[Continue|Cancel](),
     Channel[Continue|Cancel](),
     Channel[Retry|Cancel]())

    eval(par(
     svr(c1, c2, c13, c14, c3, c7, c6, c8, c15, c9, c4, c16,
         c4, c16, c10, c16, c10, c8, c15, c11, c5, c16, c5,
         c16, c12, c16, c12, c15, c8, c3, c17, c7, c17, c7),
     client(c1, c2, c3, c6, c4, c5),
     svrAct(c7, c8, c9, c10, c11, c12),
     svrVer(c13, c14, c15, c16, c17)))
```

Listing 7.22: Channels generated for Bank Protocol Without Clone Channel Elimination

```
val(c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12) =
     (Channel[Connect](),
     Channel[Login](),
     Channel[Retry|Cancel|Success](),
     Channel[Withdraw1|Deposit1|Cancel](),
     Channel[Login](),
     Channel[Success|Fail](),
     Channel[Cancel|Continue](),
     Channel[Retry|Cancel](),
     Channel[Retry|Cancel|Continue](),
     Channel[Withdraw2|Cancel|Deposit2](),
     Channel[Success|Fail](),
     Channel[Retry|Cancel]())

    eval(par(
     client(c1, c2, c4, c3),
     svrVer(c5, c6, c7, c8),
     svrAct(c9, c10, c11, c12),
     svr(c1, c2, c5, c6, c9, c4, c10, c11, c3, c7, c3, c7,
         c12, c12, c10, c10, c8, c9, c8, c9)))
```

Listing 7.23: Channels generated for Bank Protocol With Clone Channel Elimination

7.4 Implementation Time Comparison

In this section, we compare the time taken to implement the Effpi type and function of a given protocol, with and without using our tool. We implemented the types, functions and channels for two protocols, with and without using our tool, then we record the time taken in different instances of the process using a stopwatch, to the precision of seconds.This includes the time taken for compilation check between implementations and bugs fixing. This was done by a fourth year Computing student from Imperial College alone. The programmer has the experience of working on more than 10,000 lines of Scala code, most of them which involves Effpi types. He also grasp the fundamentals of multiparty session types(MPST), including basic communication types between participants and the merging of types.

7.4.1 Adder Protocol

The Adder protocol is shown as in Listing 7.24. It consisted of two participants, the Client and the Server. The Client continuously send an ADD request to the Server and the Server responds with a Res message, until the Client decided to terminate by sending QUIT to Server. We propose a case where the Client asks the Server to add the numbers from 1 to 10 and finally print out the value. The result in being shown in Figure 36. Modifying the default function generated by our tool took 192 seconds. Generating the Effpi types, functions and channels manually took 561, 675 and 115 seconds respectively. Almost no time is spent in debugging the code, the time taken for the tool to generate the default program is 0.085 seconds(average of 10 runs).

```
global protocol Adder(role Client, role Svr) {
    choice at Client {
        ADD(number, number)   from Client to Svr;
        RES(number)           from Svr to Client;
        do Adder(Client, Svr);
    } or {
        QUIT()                from Client to Svr;
    }
}
```

Listing 7.24: Adder Protocol

7.4.2 Two Buyer with Negotiation Protocol

We propose a variation to the functionality of the program of Two Buyer With Negotiation protocol in Listing 7.5: Buyer A agree to to buy if Buyer B is offering to pay more than Buyer A, otherwise Buyer A requests for a negotiation with Buyer B until it has requested 10 times, then he or she will ask to cancel the purchase. Buyer B will increase his or her offer by £10 for each negotiation

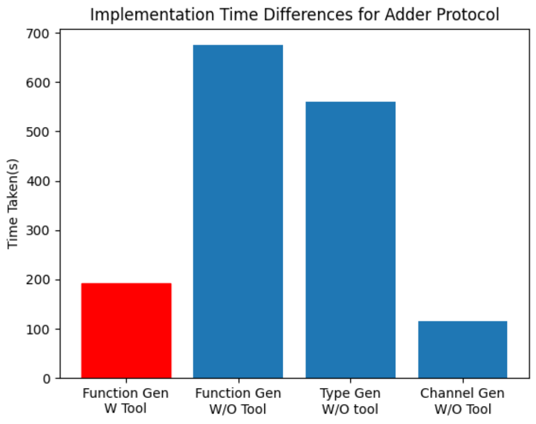


Figure 36: Implementation Time Differences of Adder Protocol With and Without tool

request, and for simplicity purpose, travel agent S sends the same price to buyer A and buyer B on start and will not deduct any extra fees throughout the communication. Comparing the code generated with and without using the tool, the core functionality, type assignment, channel generation and communication pattern are the same. However, the programmer provides a better naming convention for the channels such that it showcase the purpose of the channel more clearly. For example, InChannel[START] c_S_A_1 of Travel Agent(seen in Line 1 Listing 7.12) and OutChannel[START] c_A_S_1 of Buyer A(seen in Line 1 Listing 7.10) where both used to send/receive message of type START from each other, is named as cStart in the both cases by the programmer, the tool didn't adopt this naming convention as it can't guarantee clashing of channel name within the same participant in the general case. However, the output generated from the tool included print statement for each communication operation, the programmer can easily understand the undergoing communication when they executes the program. The manually generated code doesn't include print statements as this would increase the time taken to generate the code by a large percentage.

The results can be seen as in Figure 37. It took 6940 seconds in total to implement without tool and 323 seconds with the tool. In this particular protocol, the local communication types and functions for each participant is hard to work out as involves more than two participants and the communication between them is complex. By looking at the protocol as shown in Listing 7.5, Line 16, 21 and 26, the three messages sending: CONFIRM, CANCEL and NEGOTIATE from travel agent to Buyer B, seems to be three separate communications. However, by looking closely, we can see that it's dependant on the value the travel agent received from Buyer A prior to this. From S' FSM, we can see that these three message sending doesn't occur as a selection, however from Buyer B's perspective(FSM), it is expecting a branching from S. Without using our tool this behaviour is hard to observe and a large portion of time is invested to implement the respective channel communication. Having 8 channels and 7 labels, without using the tool, the implementation of this program is more vulnerable to errors such as type mismatch, wrong channel assignment and wrong case class type. Out of the 3467 seconds and 3167 seconds used to implement types and functions from Two Buyer protocol without using the tool, 1278 seconds and 1166 seconds are invested respectively to fix the errors.

Notice that by using our tool, we don't have to spend any time on channel generation and type generation, little work is required to add functionality and modify the payload values in the case classes, generation time using tool is 21 times faster than without using it, the time taken for the tool to generate the types, channels and default function bodies is 0.118 seconds, which is relatively insignificant compare to our manual generation time hence we didn't include in the bar chart. In order to be more efficient than the tool in implementing this specific functionality, one will have to implement 0.557 lines of code every second, or in the case of a basic executable program(default version without any extension) based on the protocol, 1525 lines of code need to be implemented every second.

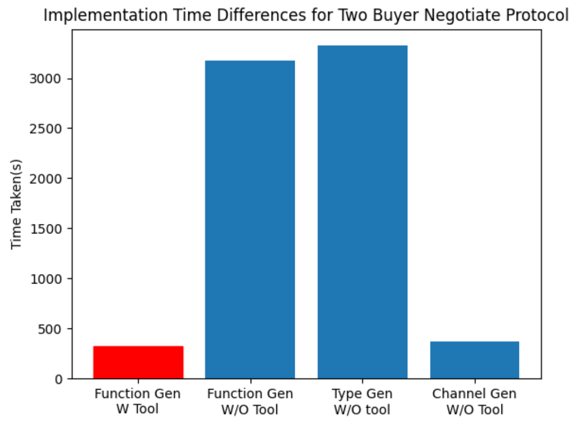


Figure 37: Implementation Time Differences of Two Buyer Negotiate Protocol With and Without tool

Chapter 8

Conclusion

8.1 Contributions

The aim of our project was to implement a tool that generates Effpi-typed Scala program from a Scribble protocol. The generated program is designed to be easily extended and understood by the developer. By implementing their applications on top of the generated program, developers enjoy communication safety guarantees in their endpoint applications by construction.

In Chapter 3, we have defined a projection of local session types from multiparty session types theory (MPST) to Effpi types. With Effpi being a channel-based framework, we also formalised a theory on how channels are generated and assigned between the participants to make sure each communication in the program occur through the correct channel. We designed and implemented our tool in Chapter 4 in accordance to these theory. We have implemented multiple features to suit various requirements of the developer, including user interface, asynchronous communication and distributed code. We also contributed to the Dotty compiler by identifying some of the bugs.

In Chapter 5, We further optimised our program by reducing the number of local channel instances required by each participant in the protocol and also the number of channels generated fro the program. We propose a method to identify the channels between participants that can be reused without affecting the communication behaviour.

In order to more closely represent real-life distributed system where participants may crash during communication, we extended local session types and Effpi types to include error handling in Chapter 6. We extended our tool to generate Effpi-typed Scala program with error handling. Instead of waiting for a crashed sender forever and cause deadlock, receiver may proceed with an alternative protocol after waiting for a message for a period of time. We also proposed a new channel matching method to take the error-handling protocol into consideration. We showcase how some well-known protocol in distributed system can be represented in our extended local session types and how our tool generate the program from it.

In Chapter 7, we evaluate our tool on some well known protocols in the session type literature, demonstrating the benchmarks for each generated program. We also compare the number of channels generated, with and without using our clone channel elimination method. Following that, we use our tools on two real life examples to showcase how the generated program can be easily extended to add business logic without modifying the underlying communication between participants. We then compare the time required for a programmer to implement an Effpi-typed Scala program from a given protocol, with and without using our tool, to showcase how our tool may assist a developer.

Overall, we offer an end-to-end solution for integrating multiparty session type theory into modern distributed system, in a way actively encourages developers to design their application with communication safety in mind. Our formalism of session type with error handling enable our tool to support a wider range of protocols.

8.2 Ethical Issue

Our project doesn't require prior collection of data from human/animal and doesn't involve any human/animal participants throughout the process. Hence, ethical issues involving humans/animals and protection of personal data are not the main issues in our project.

The general aim of our project is to make programs all around the world safer without targeting any specific country. Our project is an exclusive civilian application focus, it may have the potential to be used in the military to validate software but it's not intended to serve the military. It doesn't affect current standards in military ethics. We are working through theories and implementing software, we don't require any resources from the environment nor usage of any elements that can be harmful to humans.

The potential for any malevolent/criminal/terrorist abuse is extremely low. The only information we generated is an API/ compilation for one's program and the result is not publicised. The very worst case is when an individual accidentally or intentionally publicised their own protocol and someone used our tool to identify the flaws in the protocol and launch attacks to their program. However our tools are meant to be publicised so the individual may as well use our tool to identify the flaws in their program and fix the issue.

An ethical issue that may come up is during testing. We may test our tools on some open source/ proprietary program. If we decided to test on proprietary program, we may need to gain permission and licensing to gain access or publish our findings. We may also later publish our software as an open-source on github, we'll need to license it so that others are free to use, change, and distribute the software.

8.3 Future Work

There are many interesting future works that we would like to implement to extend the funtionality of our tool. We will select some of them to introduce:

- **Global session type with error handling.** Because of the time constraints, we have yet to define the global session type with error handling. In the future, we would like to define the grammar for it and the projection to our defined local session type with error handling. We shall also extend the `nuscr` toolkit to handle protocols with error handling and generate the local session types and communication digraph for each participant from it. By doing so, we no longer need to manually key in the digraph for each participant into our tool but instead have `nuscr` to generate for us.

- **Contribute to improve Dotty compiler.** Currently, there are some large protocols which program generated took forever to compile and used up all the available memory on our machine. This is identified to be a Dotty compiler issue instead of Effpi's or our tools. We shall continue to work closely with the developer of Dotty to break down and identify the underlying issues. We aim to be able to execute any program that successfully generated by our tool.

- **Refinement types in Effpi.** Refinement types are in the form of logical constraints and can be used to express the preconditions for the message payloads. Ongoing work has been done to include refinement types in multiparty session type. We look to extend Effpi in the future to include refinement types for our tool to generate refinement-typed program for the developers.

Bibliography

[1] Akka Documentation, Classic Actors
`https://doc.akka.io/docs/akka/current/actors.html`

[2] Lightbend Akka
`https://akka.io/`

[3] Erlang
`https://www.erlang.org/`

[4] How and Why Twitter Uses Scala
`https://sysgears.com/articles/how-and-why-twitter-uses-scala/`

[5] Big Companies that uses Scala `https://datarootlabs.com/blog/big-companies-use-scala`

[6] HONDA, K., VASCONCELOS, V. T., AND KUBO, M.
Language primitives and type discipline for structured communication-basedm programming. In Programming Languages and Systems (Berlin, Heidelberg, 1998), C. Hankin, Ed., Springer Berlin Heidelberg,
pp. 122–138.

[7] D. Engler and K. Ashcraft,
"RacerX: effective, static detection of race conditions and deadlocks," in Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)
pp. 237–252.

[8] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in Proc. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07), 2007,
pp. 205–214.

[9] KESTER, D., MWEBESA, M., AND BRADBURY, J. S.
How Good is Static Analysis at Finding Concurrency Bugs? In 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (2010),
pp. 115–124.

[10] A Newbie's Guide to Scala and Why It's Used for Distributed Computing
https://blog.insightdatascience.com/a-newbies-guide-to-scala-and-why-it-s-used-for-distri

[11] nuScr
https://github.com/nuscr/nuscr

[12] Effpi: verified message-passing programs in Dotty
https://github.com/alcestes/effpi

[13] DottyGenerator
https://github.com/jarredlim/DottyGenerator

[14] The Go Programming Language Specification, Channel Types
https://golang.org/ref/specChannel_types

[15] Dotty Documentation
http://dotty.epfl.ch/docs/

[16] Pydot GraphViz Module
https://pydotplus.readthedocs.io/reference.html

[17] Nobuko Yoshida, Lorenzo Gheri, A Very Gentle Introduction to Multipart Session Types
16th International Conference on Distributed Computing and Internet Technology (ICDCIT 2020)
p. 73 - 93

[18] Dotty Documentation, New Types
http://dotty.epfl.ch/docs/NewTypes/index.html

[19] sbt project compiled with Dotty
https://github.com/knoldus/dotty-examples

[20] C.Hoare. *Communicating Sequential Processes.* [*Communications of the ACM*], 21(8):666–677, 1978.

[21] R.Milner [*A Calculus of Communicating Systems*], volume 92. Springer-Verlag, 1980

[22] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Process,Part I. [*Information and Computation*], pages 1-77, 1992

[23] Ng, N, Yoshida, N, Static deadlock detection for concurrent go by global session graph synthesis 2016

[24] Lange, J and Ng, CWN and Toninho, B and Yoshida, N, A static verification framework for message passing in go using behavioural types 2018

[25] Scalas, A and Yoshida, N and Benussi, E, Verifying Message-Passing Programs with Dependent Behavioural Types 2019

[26] Yoshida, N, Pi2 Lecture Notes in 70008 Concurrent Processes, Accessed on 28th December 2020 page 2,4

[27] Benjamin C. Pierce and Davide Sangiorgi. 1996. Typing and Subtypingfor Mobile Processes. Mathematical Structures in Computer Science 6, 5 (1996).

[28] N. Amin and A. Moors and Martin Odersky Dependent Object Types Towards a foundation for Scala's type system, 2012

[29] YOSHIDA, N., HU, R., NEYKOVA, R., AND NG, N. The Scribble Protocol Language. 8th International Symposium on Trustworthy Global Computing (TGC 2013). p. 22 - 41

[30] Alceste Scalas and Ornela Dardha and Raymond Hu and Nobuko Yoshida, A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming, 31st European Conference on Object-Oriented Programming, 2017

[31] Raymond Hu, Distributed Programming Using Java APIs Generated from Session Types, Behavioural Types: from Theory to Tools, pages 287 - 308, 2017

[32] Companies using Golang `https://www.gowitek.com/golang/blog/companies-using-golang`

[33] Ahmad Elshareif. Which companies use the Erlang language?, 2018-06- 10. `https://www.quora.com/Which-companies-use-the-Erlang-language`

[34] Intel's GearPump Real-Time Streaming Engine Using Akka `https://www.lightbend.com/case-studies/intels-gearpump-real-time-streaming-engine-using-a`

[35] Nobuko Yoshida and Lorenzo Gheri, A Very Gentle Introduction to Multiparty Session Types, 16th International Conference on Distributed Computing and Internet Technology, pages 73 - 93, 2020

[36] Unreducible app of HK type when nesting dependent func + match types `https://github.com/lampepfl/dotty/issues/9999`

[37] Effpi: verified message-passing programs in Dotty `https://github.com/alcestes/effpi`

[38] Effpi-copy: original effpi with added new buyer examples `https://gitlab.doc.ic.ac.uk/jql17/effpi-copy/-/tree/master/examples/src/main/scala/demo`

[39] Raymond Hu and Nobuko Yoshida, Hybrid Session Verification through Endpoint API Generation, 19th International Conference on Fundamental Approaches to Software Engineering, pages 401 -418, 2016

[40] Anson Miu and Francisco Ferreira and Fangyi Zhou and Nobuko Yoshida, Generating Interactive WebSocket Applications in TypeScript, 12th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, pages 12 - 22, 2020

[41] Kohei Honda, Nobuko Yoshida, and Marco Carbone, Multiparty Asynchronous Session Types. J. ACM 63 (2016), pages 1–67.Issue 1-9, 2016 https://doi.org/10.1145/2827695

[42] Laura Bocchi, Romain Demangeon, and Nobuko Yoshida, A Multiparty Multi-session Logic. In Trustworthy Global Computing, Catuscia Palamidessi and Mark D. Ryan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, pages 97–111, 2013

[43] Raymond Hu, Nobuko Yoshida, and Kohei Honda.Session-Based Distributed Programming in Java, In ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings (Lecture Notes in Computer Science), Jan Vitek (Ed.), Vol. 5142. Springer, 516–541. 2008 https://doi.org/10.1007/978-3-540-70592-5_-22

[44] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal, A Session Type Provider: Compile-time API Generation of Distributed Protocols with Refinements in F. In Proceedings of the 27th International Conference on Compiler Construction (CC 2018), ACM, New York, NY, USA, page 128–138, 2018 https://doi.org/10.1145/3178372.3179495

[45] Romain Demangeon and Kohei Honda,
2012. Nested Protocols in Session Types. In CONCUR 2012 - Concurrency Theory -23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings (Lecture Notesin Computer Science), Maciej Koutny and Irek Ulidowski (Eds.), Vol. 7454. Springer,
page 272–286.
https://doi.org/10.1007/978-3-642-32940-1_20
2012

[46] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu,
SPY: Local Verification of Global Protocols. In Runtime Verification, Axel Legay and Saddek Bensalem (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg,
page 358–363
2013

[47] Roy T. (Ed.) Fielding and Julian F. (Ed.) Reschke,
Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing RFC 7230. RFC Editor
page 1–89
https://www.rfc-editor.org/info/rfc7230
2014

[48] Slow graph_from_dot_data `https://github.com/pydot/pydot/issues/221`

[49] Scala blog: We got liftoff `https://dotty.epfl.ch/blog/2015/10/23/dotty-compiler-bootstraps.htm`

[50] Scala Community `https://scala-lang.org/community/`

[51] Unreducible app of HK type when nesting dependent func + match types 9999 `https://github.com/lampepfl/dotty/issues/9999`

[52] Scala compile error: invalid prefix TypeBound 12141 `https://github.com/lampepfl/dotty/issues/12141`

[53] Christian Cachin, Rachid Guerraoui, Luís Rodrigues,
Introduction to Reliable and Secure Distributed Programming
page 169
2011

[54] Paxos Made Simple Leslie Lamport
`https://lamport.azurewebsites.net/pubs/paxos-simple.pdf`
2001