Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Nested Multiparty Session Programming in Go

Author:
Benito Echarren Serrano

Supervisor:
Prof. Nobuko Yoshida

Second Marker:
Dr. Iain Phillips

June 17, 2020

Submitted in partial fulfillment of the requirements for the MEng Computing of Imperial College London

Abstract

Go is a programming language which is widely used in industry, and its concurrency primitives - *goroutines* and *shared memory channels* - make it a popular choice for development in distributed systems. Despite its inbuilt concurrency features, the language does not provide much support against concurrency errors such as deadlocks and race conditions.

Multiparty session types (MPST) provide a typing discipline for describing the interactions between processes, and they can be used to develop message-passing APIs where these kinds of concurrency bugs *cannot* happen. However, previous implementations of MPST frameworks for Go did not treat Go's concurrency primitives, and they were unable to express a large number real-world APIs, which limited their practical applications.

In order to fill this gap, we present the first implementation of the MPST theory of *nested protocols*, which makes it possible to call a subprotocol during the execution of a parent protocol, possibly involving new participants. We extend an MPST-based framework for specifying and statically verifying concurrent protocols with the nested protocol theory, introducing the definitions of nested protocols and nested protocol calls.

We design a scheme to generate APIs in Go for the roles taking part in each nested protocol which ensures that they only perform I/O actions that comply with the protocol specification. Our implementation also leverages Go's in-built concurrency primitives to implement the behaviour of the roles, using channels as the mechanism for communication.

We evaluate the expressiveness of our implementation by showing how it can be used to describe common distributed computing patterns. We then demonstrate how these patterns can be applied to implement three case studies and analyse their performance using a benchmark.

Acknowledgements

I would like to thank Prof Nobuko Yoshida and Dr David Castro-Pérez for providing me with their guidance and support throughout the project. I would also like to thank Dr Francisco Ferreira and Fangyi Zhou for helping me with my doubts and for allowing me to contribute to the development of the Scribble framework they have created.

I would also like to thank my family for supporting me during these four years I have been studying abroad.

Finally, I would like to thank all the friends I have made at Imperial, who have continuously supported and encouraged me during my time at university.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 3
- 1.3 Contributions and Report Structure . . . 3

2 Background **6**
- 2.1 Session Types . . . 6
  - 2.1.1 Overview . . . 6
  - 2.1.2 Asynchronous π-Calculus . . . 7
  - 2.1.3 Binary Session Types . . . 8
  - 2.1.4 Multiparty Session Types . . . 12
  - 2.1.5 Scribble API Endpoint Generation . . . 15
- 2.2 Nested Protocols . . . 17
  - 2.2.1 Motivations . . . 17
  - 2.2.2 Nested Session Calculus . . . 18
  - 2.2.3 Nested Session Types . . . 20
  - 2.2.4 Returning Values from Subprotocols . . . 22

3 Extending Scribble with Nested Protocols **24**
- 3.1 Syntax Extensions . . . 24
  - 3.1.1 Global Protocols . . . 25
  - 3.1.2 Scopes . . . 26
  - 3.1.3 Local Protocols . . . 28
- 3.2 Well-formedness . . . 29
- 3.3 Renaming Protocols . . . 30
  - 3.3.1 Renaming Algorithm . . . 31
  - 3.3.2 Renaming Protocol Interactions . . . 32
- 3.4 Projection . . . 35
  - 3.4.1 Projection of Global Protocols . . . 35
  - 3.4.2 Projection of Choice . . . 36
  - 3.4.3 The Full Merge Operator . . . 39

4 Design of Code Generation **40**
- 4.1 Code Generation Approach . . . 40
- 4.2 Implementation Design . . . 40
  - 4.2.1 Implementation of Roles . . . 41

4.2.2 Callbacks . . . 41
4.2.3 Nested Protocol Calls . . . 41
4.2.4 Returning Results from Protocols . . . 42
4.2.5 Entry-Point Protocol Setup . . . 42
4.3 Implementation Restrictions . . . 43
4.3.1 Code Organisation in Go . . . 43
4.3.2 Naming Restrictions . . . 44
4.3.3 Package Organisation Restrictions . . . 45
4.4 Implementation Structure . . . 45

5 Project Implementation **47**
5.1 Naming and Notation . . . 47
5.2 Imports . . . 49
5.3 Package `messages` . . . 49
5.4 Package `channels` . . . 50
5.5 Package `invitations` . . . 51
5.5.1 Invitation Structs . . . 51
5.5.2 Protocol Setup Structs . . . 52
5.6 Package `results` . . . 53
5.7 Package `callbacks` . . . 53
5.7.1 Callback Generation . . . 54
5.7.2 Initial State of Dynamic Participants . . . 57
5.7.3 Package Design Restrictions . . . 57
5.8 Package `protocol` . . . 57
5.8.1 Protocol Setup Environment . . . 58
5.8.2 Protocol Setup Function . . . 58
5.8.3 Initial Roles Setup . . . 59
5.9 Package `roles` . . . 60
5.9.1 Protocol Setup Functions . . . 60
5.9.2 Role Implementation Functions . . . 62
5.9.3 Package Design Restrictions . . . 62

6 Implementation of Local Protocols **64**
6.1 Message Exchanges . . . 64
6.2 Protocol Calls . . . 65
6.3 Choice . . . 67
6.4 Recursion . . . 68
6.4.1 Choice and Recursion . . . 68
6.4.2 Recursion as Nested Protocols . . . 69
6.5 End . . . 70

7 Evaluation **72**
7.1 Distributed Computation Patterns . . . 72
7.1.1 Ring Protocol . . . 72
7.1.2 Pipeline Protocol . . . 73
7.1.3 Fork-Join Protocol . . . 74

7.2 Case Studies 76
7.2.1 Fibonacci 76
7.2.2 Fannkuch-redux 77
7.2.3 Bounded Prime Sieve 78
7.3 Performance 80
7.4 Expressiveness and Limitations 82

8 Conclusion **84**
8.1 Contributions 84
8.2 Future Work 85

Chapter 1

Introduction

1.1 Motivation

From the multiple cores in a CPU or GPU to the large server clusters in data centers, concurrency and parallelism have become an inherent part of computers and how they are used. This distributed computation model gives much higher performance and scalability, as multiple tasks can be executed at the same time, but it also makes reasoning about the software much harder, giving rise to concurrency bugs such as race conditions and deadlocks.

In order to reason about these concurrent and parallel programs, researchers have come up with different memory models which offer different guarantees. The two most important ones are shared memory and message passing. Shared memory is an abstraction where all the components can read and write to a single piece of memory, and the changes that anyone makes become visible to the other processes. On the other hand, message passing expresses the communications between different components as a series of message exchanges, with each process having its own address space.

When writing a concurrent program, understanding the concurrency model that you are using is vital to writing correct code, but unlike the type system, which provides some correctness guarantees about the program at compile time (type safety of assignments, method calls, etc.), programming languages generally offer little to no support when it comes to statically detecting concurrency bugs (deadlocks, race conditions, etc.). Although separate tools for different programming languages have been developed to do this, like FindBugs[5] or Jlint[3] for Java, they do not scale well with the size of the program and may not find all the concurrency bugs in the implementation[20]. Session types[16] provide an alternative approach for reasoning about inter-process communication in a message passing setting. They formalise the structured interactions between the participants as a protocol, ensuring that there are no communication errors or concurrency bugs in the program. Instead of trying to find bugs in an existing implementation, session types guarantee that the implementation will be free of concurrency bugs. Unlike data types, which

express the type of information that will be stored in memory during execution, session types describe the communication exchanges between processes.

Session types have been extended in different directions in order to be more expressive by providing more control over the interactions through means such as logical assertions[7]. The theory has also been extended to provide greater flexibility on how participants can join or leave a session[13] or to define protocols with parametric number of participants acting a particular role (e.g. n workers)[9].

Despite this, as protocols have grown and become more complex, different branches of the session type theory have been unable to fully express some of their behaviour. Often, protocols will have a highly modular structure, with a protocol calling or depending on other protocols. Moreover, different protocols may also share some common structure, with interactions between different participants following the same pattern. For instance, a protocol where a client wants to communicate with a server might initially involve authenticating with a different authentication server in order to get a valid token. This set of interactions with the authentication server could be considered a different subprotocol, which might also be used by many different applications in their authentication process.

Demangeon and Honda [12] extended the existing session type theory in order to define nested protocols using subsessions and invitations. This new theory structures such protocols in a more intuitive way and even provides the ability to be able to reuse subprotocols for different use cases. It also parametrises protocols so that multiple calls to the same protocol can be made with potentially different parameters, making protocol declarations more concise and readable. Subprotocols can also invite new participants to participate in them, making it possible to define protocols with a dynamic number of participants. We develop the first implementation of this theory to give a session type-based framework for the specification and safe implementation of distributed programs in Go.

Go is a programming language which has become widely used in industry[2], especially for development of distributed systems, and it has been used to develop frameworks like Kubernetes and Docker. Go defines two concurrency primitives which simplify the development of programs involving local concurrency[1]: the ability to spawn thousands of lightweight threads called *goroutines* and *channels*, which enable inter-process communication and synchronisation through message passing. Despite this, Go offers little support against concurrency bugs such as deadlocks and race conditions. Session types can address this issue by generating implementations of concurrent programs which are guaranteed to be correct.

Unfortunately, existing session type-based frameworks have two major issues which limit their practical applications: first, they require the number of participants to fixed at the start of the session, and second, the Go implementations they generate do not take advantage of the language's concurrency primitives[9]. In nested pro-

tocols, participants can be introduced into the session dynamically, enabling them to be used to model more complex systems. This pattern of dynamically assigning tasks is common practice in Go, where processes can easily spawn new goroutines to perform different jobs in parallel. This makes this programming model highly suitable for implementing nested protocols.

1.2 Objectives

The aim of this project was to produce an end-to-end solution for **statically** verifying and generating the implementation of a protocol specification with nested protocols. We extend Scribble[29], a protocol description language based on the multiparty session types (MPST), with constructs for defining nested protocols and nested protocol calls. These extensions are based on the MPST theory presented in [12]. We also define a code generation scheme to generate APIs for the different roles in a protocol from a Scribble protocol specification. These APIs can then be used by the developers to generate an implementation of the protocol, and they guarantee that the implementation of the role will follow the behaviour specified in their protocol declaration without a need for any runtime checks.

1.3 Contributions and Report Structure

The result of this project is an end-to-end solution for **statically** verifying nested protocols and generating an API implementation in Go for all the roles participating in the protocols. To the best knowledge of the author, this is the first practical implementation of the nested protocols MPST theory presented in [12]. With this new extended framework, a significant subset of message-passing APIs can be generated.

The workflow for the verification and API generation of nested protocols is shown in Figure 1.1. The parts shown in green denote steps of the workflow which only need to be carried out when generating the implementation for the protocol specification.

In Chapter 2, we introduce the typing discipline of session types and the current approach for generating the implementation of protocols from their session types using Scribble[29]. We also describe the MPST theory of nested protocols which we use to develop our implementation.

In Chapter 3, we describe the extensions we have introduced to Scribble[25] in order to model nested protocols. We describe the new syntactic constructs we define based on the theory presented in [12] and how we have extended the projection definitions in Scribble to take into account the addition of nested protocols. We have based our definition of projection on the one presented in [12], modifying it to be able to represent a wider range of protocols.

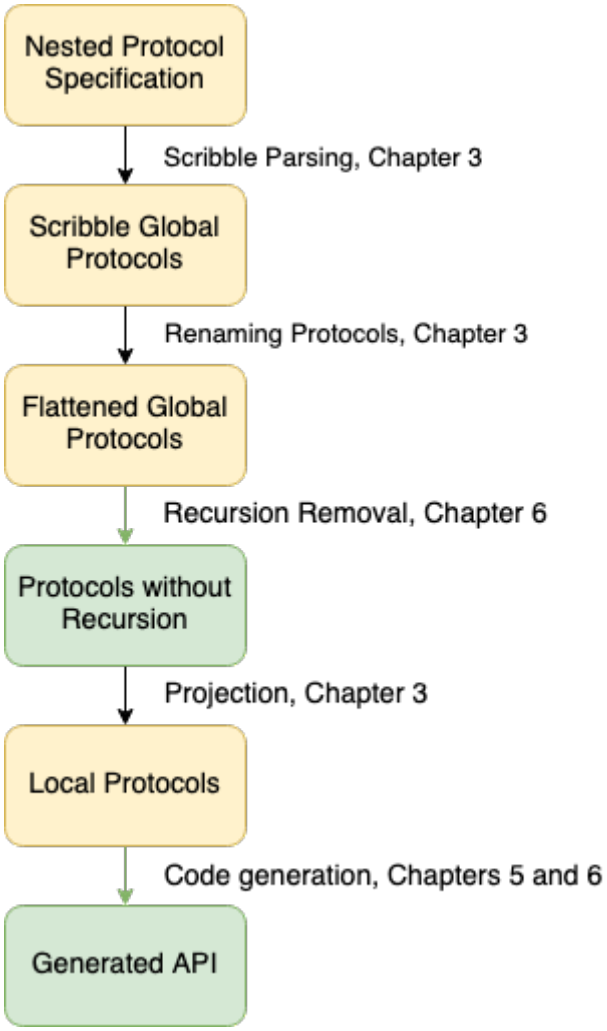

Figure 1.1: Workflow for Verification and API Generation of Nested Protocols

In Chapter 4, we discuss our design for the implementation of the Go APIs and the factors which have affected the choices we have made when coming up with this design. In particular, we describe how some of the features of Go, our target language, have affected the approach we follow in our implementation and how we structure it.

In Chapter 5, we discuss how we structure our implementation and describe how we generate the all the different components which it is comprised of from a given protocol specification.

In Chapter 6 we describe the correspondence between the different constructs of a Scribble protocol declaration and the implementation of the role Go APIs. Developers can use the code that we generate to implement protocols which are correct by construction. The type system will guarantee that the messages will always be of the correct type and our implementation will ensure that the roles follow the behaviour specified by the protocol.

In Chapter 7, we evaluate our work through case studies implemented as nested protocols. We discuss the increase in expressiveness that nested protocols provide and evaluate the performance of our implementation design by running a benchmark on our case studies.

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

Our work on extending Scribble with nested protocols has all been implemented on top of an existing OCaml implementation of the framework called `nuscr`[1]. Our extension is currently only available a fork of the repository[2], but it should be integrated into the main repository once it has been reviewed and approved.

[1] https://github.com/nuscr/nuscr
[2] https://github.com/becharrens/nuscr

Chapter 2

Background

In this chapter we introduce the typing discipline of session types and describe a branch of the session types theory which defines nested protocols. We also present Scribble[29], a protocol description language based on the theory of session types.

2.1 Session Types

2.1.1 Overview

The two main models to reason about concurrent programs are shared memory and message passing. In shared memory, components communicate by reading and writing to a shared part of memory. This is how CPU threads and processes in a computer communicate with each other. However, reasoning about concurrent programs with this model can be tricky, as compiler and hardware optimisations can reorder the program's instructions[6]. On the other hand, in message passing, inter-process communication is carried out by exchanging data in the form of explicit messages. This model closely resembles how communication is carried out in distributed systems.

Message passing can be encoded using the π-calculus, a process algebra based on name passing which was developed by Milner[23]. In this calculus, processes communicate by sending channel names over named channels. We present an asynchronous variant of the π-calculus in Section 2.1.2. Session types[16] introduce a typing discipline for formalising the communication exchanges between processes in the π-calculus. The initial theory was defined only for two participants, but it was later extended by Honda et al.[17, 18] to include multiple parties. This enabled the session types to encode a larger number of protocols. Session types improve the reliability of distributed systems by guaranteeing the correctness of the programs. If the processes are well-typed, they will have session fidelity and will not suffer from communication errors such as deadlocks, type mismatches or protocol violations.

There are already multiple implementations of session types for popular programming languages such as Java[19], C[27], Go[9], Python[24], Erlang[14], etc. Due

to the set of features available to different programming languages, the implementation of MPST varies from one language to another. Some languages rely on the type system to verify the correctness of the session's implementation at compile-time, while others may require run-time checks in order to detect protocol violations.

2.1.2 Asynchronous π-Calculus

The π-calculus is a process algebra for encoding communicating systems. Processes communicate with one another through name passing, by sending and receiving channel names over named channels. The π-calculus was originally proposed by Robin Milner[23], but different variants have been proposed since. It is a powerful model which has been shown to be Turing-complete[22], as it can encode the λ-calculus. The first asynchronous π-calculus theory was presented in [8], and we present a variant based on this theory, as defined in [30].

$P,\ Q$	::=		Processes
		0	Nil Process
	|	$P \mid Q$	Parallel Composition
	|	$(\nu\ a)\ P$	Scope Restriction
	|	$!P$	Replication
	|	$\overline{u}\langle v \rangle$	Output
	|	$u(x).P$	Input
$u,\ v$	::=		Identifiers
		$a,\ b,\ c$	Names
	|	$x,\ y,\ z$	Variables

Figure 2.1: Syntax of monadic asynchronous π-calculus

The syntax of the asynchronous π-calculus is defined in Figure 2.1:

- 0 is the nil process, which represents the process with no actions.
- $P \mid Q$ is the parallel composition of two processes. These processes can execute in any order.
- $(\nu\ a)\ P$ is the scope restriction operation. It creates a new named channel a that can only be used within process P and will not interfere with any other existing names.
- $!P$ is process replication. It represents the infinite parallel composition of process P: $P \mid P \mid P \mid ...$
- $\overline{u}\langle v \rangle$ is the output operation, which sends v over u.

- $u(x).P$ is the input operation. It receives value over channel u. After receiving the value it will continue executing process P, replacing all references to x in P with the received value.

Structural congruence is an equivalence relation which expresses that two processes are equivalent/interchangeable. We define the structural congruence of asynchronous π-calculus relation in Figure 2.2.

$$
\begin{array}{rcrcll}
 & & P & \equiv & P & \text{Reflexivity} \\
P \equiv Q & \Longrightarrow & Q & \equiv & P & \text{Symmetry} \\
P \equiv Q \ \wedge \ Q \equiv R & \Longrightarrow & P & \equiv & R & \text{Transitivity} \\
P \equiv Q & \Longrightarrow & (\nu\ a)\ P & \equiv & (\nu\ a)\ Q & \text{Cong of Restriction} \\
P \equiv Q & \Longrightarrow & P \mid R & \equiv & Q \mid R & \text{Cong of Parallel Comp} \\
P \equiv Q & \Longrightarrow & u(x).P & \equiv & u(x).Q & \text{Cong of Input} \\
P \equiv Q & \Longrightarrow & !P & \equiv & !Q & \text{Cong of Replication} \\
P =_\alpha Q & \Longrightarrow & P & \equiv & Q & \alpha-\text{equivalence} \\
 & & P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \text{Associativity} \\
 & & P \mid Q & \equiv & Q \mid P & \text{Commutativity} \\
 & & P \mid 0 & \equiv & P & \text{Zero} \\
 & & !P & \equiv & P \mid !P & \text{Replication} \\
 & & (\nu\ a)\ 0 & \equiv & 0 & \text{Res of Nil} \\
 & & (\nu\ a)(\nu\ b)\ P & \equiv & (\nu\ b)(\nu\ a)\ P & \text{Res of Restriction} \\
a \notin fn(P) & \Longrightarrow & P \mid (\nu\ a)\ Q & \equiv & (\nu\ a)(P \mid Q) & \text{Res over Parallel Comp}
\end{array}
$$

Figure 2.2: Structural congruence of monadic asynchronous π-calculus

The operational semantics of the π-calculus shown in Figure 2.3 define how the communication between the processes happens. The most important rule is COMM, which shows how the value sent by the output process is received by the input process and used in its continuation P by replacing the references to the variable x in P.

It is important to notice that the output process $\bar{u}\langle v\rangle$ does not have any continuation, which reflects the asynchronous nature of the calculus. The asynchronous π-calculus presented above is very expressive; it is possible to encode other variants of π-calculus using it, including synchronous π-calculus, which permits continuation after output, and polyadic π-calculus, where it is possible to exchange vectors of messages at once.

2.1.3 Binary Session Types

A *session* is a unit of conversation between participants. Session types provide a structured way to define and reason about the communications which take place

$$\frac{}{\overline{a}\langle v\rangle \mid a(x).P \longrightarrow P\{^{v}/_{x}\}}\ \textsc{Comm}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \textsc{Par}$$

$$\frac{P \longrightarrow P'}{(\nu\ a)\ P \longrightarrow (\nu\ a)\ P'}\ \textsc{Res}$$

$$\frac{P \equiv Q \qquad Q \longrightarrow Q' \qquad Q' \equiv P'}{P \longrightarrow P'}\ \textsc{Struct}$$

Figure 2.3: Operational semantics of monadic asynchronous π-calculus

between participants. In binary session types, these interactions happen between two participants. A session results from the binary composition of the processes of each participant. There are multiple similar formulations for session types in the literature, but Figure 2.4 shows the syntax for processes in binary session types as presented in [30].

- 0 is the nil process, which represents no actions.
- $\overline{p}\langle e\rangle.P$ is the output operation, which sends the value e to participant p with continuation P (the session calculus is synchronous).
- $p(x).P$ is the input operation, which waits for participant p to send a value x. Upon receiving the value, execution continues with P, with the received value replacing the references to x in P.
- $p \triangleright \{l_i : P_i\}_{i \in I}$ is the branching operation. The process waits for participant p to send a label $l = l_i$, for some $i \in I$. After receiving the label, execution continues with process P_i.
- $p \triangleleft l.P$ is the selection operation. The process sends label l to participant p, then continues executing P.
- `if` e `then` P `else` Q is the conditional operation. Expression e should evaluate to a boolean. If e evaluates to true then process P is executed, otherwise, Q is executed.
- $\mu X.P$ and X are used to express recursion in the processes. The recursive process $\mu X.P \equiv P\{^{\mu X.P}/_{X}\}$, that is, you are able to replace all the instances of the recursion variable X in P with $\mu X.P$. This enables you to carry out potentially infinite unfoldings of the process.
- $e \oplus e'$ expresses a non-deterministic choice between two expressions of the same type, so the expression could evaluate to the result of either of the two expressions.

v	::=	$\underline{n}$	Integers
		$\mid$ `true` $\mid$ `false`	Booleans
		$\mid$ `"str"`	Strings
$e,\ e'$	::=	v	Values
		$\mid\ x$	Variables
		$\mid\ e + e' \mid e - e' \mid -e$	Arithmetic
		$\mid\ e = e' \mid e < e' \mid e > e'$	Relational
		$\mid\ e \wedge e' \mid e \vee e' \mid \neg e$	Logical
		$\mid\ e \oplus e'$	Non-determinism
p	::=	$Alice \mid Bob$	Participant
$P,\ Q$	::=		Processes
		0	Nil Process
		$\mid\ \overline{p}\langle e\rangle.P$	Output
		$\mid\ p(x).P$	Input
		$\mid\ p \triangleright \{l_i : P_i\}_{i \in I}$	Branching
		$\mid\ p \triangleleft l.P$	Selection
		$\mid$ `if` e `then` P `else` Q	Conditional
		$\mid\ \mu X.P$	Recursive Process
		$\mid\ X$	Recursive Variable
M	::=	$p :: P \mid q :: Q$	Binary Composition

Figure 2.4: Processes in the binary session calculus

- The evaluation of expressions is only defined when the types match: logical expressions are only defined on expressions which evaluate to booleans, arithmetic expressions and $e < e'$ and $e > e'$ are only defined on integer values, and $e = e'$ is only defined if e and e' have the same type.

This session calculus is based on the π-calculus, so we can observe a great amount of similarities between the constructs in both calculi. The main differences are the introduction of the branching and selection operations (which can be encoded in the π-calculus), the definition of recursion and the conditional process.

We introduce the definition of the session types in Figure 2.5. We will not give the formal typing judgments for the processes, these can be found in [30, 16, 10]. Instead, we will give an intuition behind the correspondence between processes and their session types:

S	::=		Session Type
		$\texttt{end}$	Termination
	|	$p!\![U];\ S$	Value Send
	|	$p?[U];\ S$	Value Receive
	|	$p \bigoplus \{l_i : S_i\}_{i \in I}$	Selection
	|	$p \& \{l_i : S_i\}_{i \in I}$	Branching
	|	t	Type Variable
	|	$\mu t.S$	Recursive Type
U	::=	$\texttt{int} \mid \texttt{bool} \mid \texttt{string}$	Sorts for Expressions

Figure 2.5: Syntax of binary session types

- $Bob(x).\overline{Bob}\langle x+1\rangle.0 : Bob?[\texttt{int}];\ Bob!\![\texttt{int}];\ \texttt{end}$ - In order to infer the sort of the input variable it may be necessary to see how it is used in the process.
- $Alice \lhd choice.\overline{Alice}\langle\texttt{"hello"}\rangle.0 : Alice \bigoplus \{choice : Alice!\![\texttt{str}];\ \texttt{end}\}$ - similarly for branching.
- In order to be able to specify a type for a conditional process $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$, both branches P and Q must have the same type S, and therefore the type of the conditional process will also S.
- $\mu t.S$ and t are used to specify recursive types. They work much in the same way as recursive processes; recursive types can also unfolded by replacing instances of t in S by $\mu t.S$.

$$
\begin{aligned}
\overline{\texttt{end}} &= \texttt{end} \\
\overline{p!\![U];\ S} &= p?[U];\ \overline{S} \\
\overline{p?[U];\ S} &= p!\![U];\ \overline{S} \\
\overline{p \bigoplus \{l_i : S_i\}_{i \in I}} &= p \& \{l_i : \overline{S_i}\}_{i \in I} \\
\overline{p \& \{l_i : S_i\}_{i \in I}} &= p \bigoplus \{l_i : \overline{S_i}\}_{i \in I} \\
\overline{\mu t.S} &= \mu t.\overline{S} \\
\overline{t} &= t
\end{aligned}
$$

Figure 2.6: Duality of binary session types

A key concept in binary session types is *duality*. When two processes are composed in a session, in order for the session to progress correctly, both processes must be carrying out complimentary operations. For instance, if *Alice* is trying to send a string

to *Bob*, *Bob* must be waiting to receive a string from *Alice*, otherwise the protocol is stuck. Similarly, if *Bob* is waiting for a label from *Alice* to decide which branch to execute, *Alice*'s process must send one of the expected labels. Not only must the constructs be complimentary, but the types of the messages being sent must also match. Therefore, in order for a session to be correct, the processes involved must be duals of one another. This will ensure that the binary session will always make progress and that the session continues to be well-typed as the protocol is carried out. The duality of binary session types is shown in Figure 2.6.

2.1.4 Multiparty Session Types

Binary session types have limited applications to real-world problems, as standard protocols tend to be much more complex and involve more than two participants. These interactions cannot always be correctly expressed as the composition of binary sessions between different pairs of participants.

Multiparty session types[18] (MPST) extend the theory of binary session types to enable protocols to have multiple participants, thus overcoming these limitations. The main idea is to introduce global types, which describe the multiparty interactions within a protocol from a global perspective, as well as deriving a specification of the behaviour of each participant as local session type through the projection operation.

There are multiple variants of processes and their respective MPST in the literature. We present the syntax for synchronous multiparty session types defined in [28] in Figure 2.7. Although the syntax has some minor differences when compared to the one showed in Figure 2.4, the semantics are essentially the same. The main difference is that processes now send/receive a label with the value they communicate. Because sessions now involve multiple participants, it is necessary to specify the other participant for the exchange (it would have been possible to omit this information in binary session types).

In multiparty session types, when deriving the types for protocols we consider both the global type, which formally specifies the interactions between all the participants in the protocol from a global perspective and local session types, which characterise the interactions that each of the participants carry out within the global protocol. The parallel composition of processes satisfying the session types implements the behaviour of the global type. In this way, the implementations of the different participants are decoupled from one another, so they could be implemented separately and the overall implementation would be correct as long as they all follow their respective session types.

We present the syntax for global and local types in Figure 2.8. Like the syntax of processes, the definitions for multiparty session types are very similar to those of binary session types. Global types introduce a single construct for branching/selection, $p \longrightarrow q : \{l_i(S_i).G_i\}_{i \in I}$, in which participant p sends to q the label (and corresponding

$P,\ Q$	::=	Processes
	0	Nil Process
	$\mid$ `if` e `then` P `else` Q	Conditional
	$\mid \mu X.P$	Recursive Process
	$\mid X$	Recursive Variable
	$\mid p!l\langle e\rangle.P$	Output
	$\mid \sum_{i\in I} p?l_i(x_i).P_i$	Branching
M	::=	Multiparty Session
	$\mid p :: P$	Process
	$\mid M \mid M$	Parallel Composition
	$p,\ q,\ r,\ \ldots$	Participants
	$e,\ e',\ \ldots$	Expressions
	$x,\ y,\ z,\ \ldots$	Expression variables

Figure 2.7: Processes in multiparty session calculus

value) which will determine what the continuation is.

Each participant in the protocol enacts a role in the global type. The session type for each role is derived by projecting the global type onto that role, which essentially ignores all the interactions in which the role does not take part, leaving only the behaviour that needs to be implemented for that role within the protocol. The projection operation is defined in Figure 2.9.

Projecting a role q on `end` and t (type variable) has no impact, since q is not participating in any exchange. When projecting on $p \longrightarrow p' : \{l_i(S_i).G_i\}_{i\in I}$, depending on which role q is undertaking in the exchange, the projected session type will change.

- If the $q = p$, then the projection is a selection, as the process decides which label and value to send to p'. The continuations G_i must be projected onto q as well.
- Similarly, if $q = p'$, q will be waiting for a message from p, therefore the projected session type is a branching operation.
- When q does not participate in the label exchange, the projection is less trivial. There are multiple approaches to resolve this situation; the one shown in Figure 2.9 is known as plain merge. Since q does not know which branch will have been chosen by p, all branches must have the same continuation from q's viewpoint, otherwise q would not know which behaviour to implement. A different approach called the full merge provides a less restrictive definition[28].

S	::=	`int` | `bool` | `string`	Sorts
GLOBAL TYPES			
G	::=	`end`	Termination
	|	$p \longrightarrow q : \{l_i(S_i).G_i\}_{i \in I}$	Branching
	|	t	Type Variable
	|	$\mu t.G$	Recursive Type
LOCAL TYPES			
T	::=	`end`	Termination
	|	$\oplus_{i \in I}\ p!l_i(S_i).T_i$	Selection
	|	$\&_{i \in I}\ p?l_i(S_i).T_i$	Branching
	|	t	Type Variable
	|	$\mu t.S$	Recursive Type

Figure 2.8: Syntax of multiparty session types

In binary session types, ensuring that both participants implemented session types which were duals of one another was enough to guarantee the correctness of the protocol. With multiple participants however, that is not the case anymore. Although you can ensure that each pair of participants has dual interactions, in order to guarantee the correctness that alone is not enough. The projection operation ensures that the composition of the different local types produced satisfies the global type. Multiparty session types can therefore provide many guarantees about the communication of well-typed multiparty sessions such as:

- **Progress**: A multiparty session M can either have ended ($M \equiv p :: 0$) or it can continue to execute (there exists M' such that $M \longrightarrow M'$). This means that every sent message will eventually be received and every process waiting for a message eventually receives one. [28, 11]

- **Subject reduction**: If a well-typed session $M : G$ reduces to M', then $M' : G'$ is well typed[28].

- **Type Safety**: If $M : G$ is well-typed then the session will never get stuck - a session where there are processes which have not finished must able to continue executing[28].

- **Protocol fidelity**: All interactions which happen are expressed in the global type of the protocol[11].

- **Communication Safety**: There can never be a mismatch between the types of messages which are sent and which are expected[11].

$$
\begin{aligned}
(p \longrightarrow p' : \{l_i(S_i).G_i\}_{i \in I}) \upharpoonright q &= \begin{cases} \bigoplus_{i \in I}\ p'!l_i(S_i).(G_i \upharpoonright q) & \text{if } q = p \\ \&_{i \in I}\ p?l_i(S_i).(G_i \upharpoonright q) & \text{if } q = p' \\ G_{i_0} \upharpoonright q & \text{where } i_0 \in I, \text{ if } q \notin \{p,\ p'\} \\ & \text{and } \forall i, j \in I.\ G_i \upharpoonright q = G_j \upharpoonright q \end{cases} \\
(\mu t.G) \upharpoonright q &= \begin{cases} \mu t.(G \upharpoonright q) & \text{if } G \upharpoonright q \neq t \\ \texttt{end} & \text{otherwise} \end{cases} \\
t \upharpoonright q &= t \\
\texttt{end} \upharpoonright q &= \texttt{end}
\end{aligned}
$$

Figure 2.9: Projection of global types to local types

2.1.5 Scribble API Endpoint Generation

Scribble[29, 25] is a protocol description language based on multiparty session types. The Scribble framework translates this formal definition of the protocol into an implementation in one of various programming languages, thus applying the theory of MPST in a practical setting. A protocol describes the structure of the message exchanges between different roles, the entities which will participate in the protocol.

We present the syntax of the Scribble language as defined in [25] in Figure 2.10. A Scribble module can contain multiple protocols, with one of them being the designated point of entry for the computation. Protocols can call each other through the `do` construct, but this essentially corresponds to inlining the called protocol. In order for a protocol call to be valid, it must be called with the number of roles specified in its declaration. The remaining syntactic constructs relate closely to the different session types in the theory, described in Section 2.1.4. A message exchange is specified by the label of the message and the type of its payload, `S`, and the sender and receiver roles. Although here we only specify one payload type there could potentially be none or more multiple ones. Some implementations of Scribble also permit the user to optionally specify names for each of the payload fields, but we omit this in our notation as well. The `choice` construct encodes the external choice by a role, with a set of the different continuations which can happen based on the role's choice. The recursive type and type variables are directly encoded by the `rec` and `continue` t constructs, and the `end` session type is represented with the `end` construct.

In a syntactically correct protocol where all the protocol calls are valid, protocol calls can be expanded with the interactions of each called protocol to produce a single large protocol containing all the interactions which must be performed. Scribble then verifies that the protocol is well-formed, which ensures local protocols can be generated for all the roles. Syntactically correct protocols can be ambiguous, for instance, if the first message in two branches of a `choice` have the same signature, as other roles would not be able to tell which branch has been chosen. The

$$
\begin{array}{rcl}
\textit{Module} & ::= & P+ \\
\\
P & ::= & \texttt{global protocol } \textit{pro}(\texttt{role } A_1, \ \ldots\ , \texttt{role } A_n)\ \{\, G \,\} \\
\\
G & ::= & \\
& | & \texttt{choice at } A\ \{\, G_1 \,\}\ \texttt{or}\ \ldots\ \texttt{or}\ \{\, G_n \,\} \\
& | & \texttt{a(S) from } A \texttt{ to } B;\ G \\
& | & \texttt{rec } t\ \{\, G \,\} \\
& | & \texttt{continue } t \\
& | & \texttt{do } \textit{pro}(A_1, \ \ldots\ , A_n);\ G \\
& | & \texttt{end}
\end{array}
$$

Figure 2.10: Syntax of Scribble global protocols

local protocols are derived by projecting the global type onto each of the roles. A communication finite state machine (CFSM) is then generated for each of the local protocols, which expresses the valid transitions between states that a role can carry out during the protocol. These transitions represent a message exchange with a different participant.

An example of a global protocol can be seen in Figure 2.11, which shows a calculator protocol carried out by two roles: a client who enters the operands and the server who carries out the operation. The protocol has a recursive loop in which the client can choose to either send two numbers to multiply to the server, get the result back and start over again or quit the protocol. The resulting CFSM for the server S can be seen in Figure 2.12.

```
1  global protocol Calc(role S , role C ) {
2      rec Loop {
3        choice at C {
4           multiply(int, int) from C to S;
5           result(int) from S to C ;
6           continue Loop ;
7        } or {
8           quit() from C to S ;
9           terminate() from S to C ;
10       }
11     }
12 }
```

Figure 2.11: Calculator Protocol

Using these CFSM representations, Scribble can generate role APIs in different target

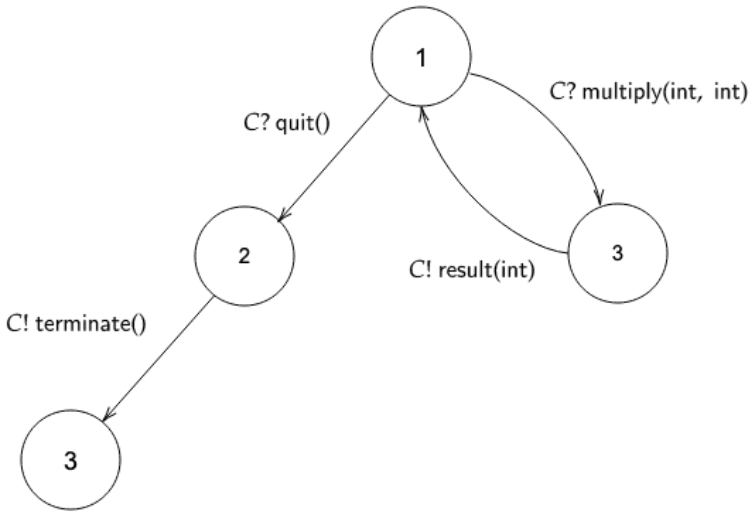

Figure 2.12: CFSM for role S in the Calculator Protocol

programming languages which the developers can use to implement the protocols. In an object-oriented programming language like Java, the code generation scheme converts each state in the CFSM into a different class. Each outgoing transition is implemented as a class methods which carry out the necessary communication operations and returns the next state. Using these classes, the user can implement the protocol by chaining method calls, transitioning only between valid states until the final state is reached, which will have no outgoing transitions. However, to ensure the correctness of the implementation every state must only be used once, and enforcing this may require runtime checks depending on the programming language.

Implementations using the Scribble-generated APIs benefit from the correctness guarantees of MPST, ensuring that there are no communication errors and that the protocol is followed properly.

2.2 Nested Protocols

2.2.1 Motivations

The theory of session types has been extended in different directions in order to be able to become more expressive. Some advances have made it possible to annotate the protocol with logical assertions to define extra properties that the protocol should meet[7]. Other developments have made it possible to express protocols with parametrised roles[9] (expressing protocols where n participants carry out particular roles), or to have greater control over how participants can join or leave a session through a dynamic multiparty session[13].

Demangeon and Honda [12] try to address a different challenge: how to structure and represent and the protocols used in networking and in distributed systems, which are becoming increasingly large. In many cases, these protocols are highly modular, and they often share a simiar structure. In order to better define and manage these protocols, they extend the theory of session types with nested protocols, which make it possible to define complex protocols using a simpler modular structure. Using this approach, protocols which have a similar structure can be grouped together under a single parametrised protocol, and complex protocols which might call other protocols can be expressed by making a call to those subprotocols. Moreover, different calls to the same subprotocol can be made with different parameters to achieve different behaviours. In their theory, subprotocols can also bring in new participants by 'inviting' them to participate. This can help simplify some protocols, as it enables users to specify protocols where participants are brought in/contacted only if required.

2.2.2 Nested Session Calculus

The theory they proposed is based on the idea of nesting protocols, where a parent protocol may define independent subprotocols and call them during its execution. Calls to a protocol can pass in different kinds of values as parameters such as typed values (numbers, strings, etc.), roles from the parent protocol which will participate in the subprotocol and even other protocols which might be used during the call, similar to higher-order functions in functional programming.

Subprotocols are implemented as subsessions. The participant calling the protocol will create a new private session for its execution, and will send *internal invitations* to all the roles in the parent protocol which will participate, and *external invitations* to bring in new agents. In this way, only the roles which have a access to the subsession, so the other roles in the parent session will not be able to interfere with it. This makes it possible to have private interactions between roles in a public session. Using the Scribble language described in Section 2.1.5, it is also possible to define subprotocols, but as opposed to this theory, protocols calls in Scribble are carried out within a single session, because a protocol call corresponds to inlining the interactions of that protocol. This means that the number of roles participating in Scribble protocols have to be known *statically*, whereas in nested protocols they can be dynamically introduced through protocol calls.

Calling subprotocols also abstracts away their actual implementation, which means that the implementation of the subprotocol can be changed (e.g. to change how authentication is carried out or to improve the protocol's security) without changing the implementation of the parent protocol. Subprotocols also give a better separation between the different execution branches of the protocol, as different external participants can be invited only when required, reducing the complexity and utilisation of resources. For instance, in an HTTP-like protocol, when a proxy received a

request from a client, it could have the choice of either returning a cached response directly or to initiate a *Contact* protocol to involve the server and get the response.

$P,\ Q$	::=	Processes
	0	Nil Process
	$\mid\ P \mid Q$	Parallel Composition
	$\mid\ a(x).P$	Receive Ext. Invitation
	$\mid\ \overline{a}\langle s\rangle.P$	Send Ext. Invitation
	$\mid\ P\ +\ P$	Non-determinism
	$\mid\ k?[r,r]_{i\in I}\{l_i(x_i).P_i\}$	Branching
	$\mid\ k![r,r]l\langle v\rangle.P$	Selection
	$\mid\ (\nu\ u)\ P$	Scope Restriction
	$\mid\ \texttt{new}\ s\ \texttt{on}\ k\ \texttt{with}\ (\widetilde{v})\&(\widetilde{a}\ \texttt{as}\ \widetilde{r}).P$	Subprotocol Call
	$\mid\ s\downarrow[r,\ r_1 : r_2](x).P$	Receive Internal Invitation
	$\mid\ s\uparrow[r,\ r_1 : r_2]\langle s\rangle.P$	Send Internal Invitation
	$\mid\ \mu X(x).P\langle v\rangle$	Recursive Process
	$\mid\ X\langle v\rangle$	Recursive Variable
	$s,\ k,\ ...$	Session names
	$a,\ b,\ u,\ ...$	Shared channels
	v	Values
	$r,\ r',\ ...$	Role Identifiers
	$x,\ y,\ z,\ ...$	Variables

Figure 2.13: Processes in Session Calculus for Nested Protocols

Figure 2.13 shows the session calculus presented in the paper, which includes the syntax for invitations and defining protocols. The calculus is based on the π-calculus and shares a lot of the constructs of other existing session calculi[7]. Names are divided into two kinds: session channels, which handle all exchanges within a session, and shared channels, used to send and receive external invitations. The constructs presented is similar to the one described in Section 2.1.4, but it also introduces new some new constructs:

- Parallel composition, scope restriction and the nil process are defined as before.
- Recursion is also defined in a similar way, an additional value can be passed in to the recursive call which may be used when the recursive call is expanded.
- $a(x).P$ and $\overline{a}\langle s\rangle.P$ are used for receiving and sending external invitations over the shared channel a.

- $P + P$ is the non-deterministic process. Execution can continue with either one of the two processes.

- $k?[r_1, r_2]_{i \in I}\{l_i(x_i).P_i\}$ is the branching operation from r_1 to r_2 with continuation P_i in session k. $k![r_1, r_2]l\langle v\rangle.P$ is the selection operation, its dual primitive.

- $s \downarrow [r,\ r_1 : r_2](x).P$ is the action for waiting for an internal invitation sent by r to r_1 in order to play role r_2 in session s.

- $s \uparrow [r,\ r_1 : r_2]\langle s\rangle.P$ is the action of sending an invitation from r to r_1 in order to play role r_2 in session s.

- $\texttt{new}\ s\ \texttt{on}\ k\ \texttt{with}\ (\tilde{v})\&(\tilde{a}\ \texttt{as}\ \tilde{r}).P$ is the operation for calling a subprotocol. It introduces a new subsession s within the parent session k, passing in arguments $\tilde{v}$ and using shared channels $\tilde{a}$ to send the external invitations.

2.2.3 Nested Session Types

The paper also extends the syntax of session types to include the types for defining and calling subprotocols as well as sending and receiving invitations. The extended syntax for local and global types is shown in Figure 2.14. They define kinds (types for types) in order to formalise the definition of protocols. These kinds include $\diamond$, which represents the protocol type, and $\longrightarrow$, which denotes parametrisation. The definition of global types is essentially the same as before:

- Termination, branching and recursion remain mostly unchanged. They also define the type for parallel composition.

- They introduce the construct $G_1 \oplus^r G_2$ to represent located choice, where role r can choose between two different branches.

- $\texttt{let}\ \mathscr{P} = \lambda\tilde{r}^1,\ \tilde{y} \mapsto \texttt{new}\ \tilde{r}^2.G\ \texttt{in}\ G'$ defines a subprotocol $\mathscr{P}$. The vector of roles $\tilde{r}^1$ defines a subset of the roles which will participate in the protocol, which will be carried out by roles from the parent protocol that will be internally invited. On the other hand, vector $\tilde{r}^2$ contains the set of roles which will be externally invited. The protocol also takes in vector of values $\tilde{y}$.

- $r\ \texttt{calls}\ \mathscr{P}\langle\tilde{r},\ \tilde{y}\rangle.G$ is the subprotocol call carried out by role r, internally inviting roles $\tilde{r}$ and passing in the values $\tilde{y}$.

The local session types have the same constructs as before, as well as the new constructs to handle invitations and protocol calls:

- Termination, branching, selection and recursion are almost unchanged, and they introduce a type for parallel composition.

- The construct $T_1 \oplus^r T_2$ represents a located (internal) choice by role r.

- $\texttt{call}\ \mathscr{P} : G\ \texttt{with}\ (\tilde{v}\ \texttt{as}\ \tilde{y} : \tilde{S})\&(\tilde{r}^2).T$ is a call to the subprotocol $\mathscr{P}$, which has global type G, sending a vector of values $\tilde{v}$ as arguments with sorts $\tilde{S}$ and externally inviting the roles in $\tilde{r}^2$ to participate in the subprotocol.

- Sending and accepting invitations for a role r are handled by the $\texttt{req}\ \mathscr{P}[r]\langle\tilde{v}\rangle\ \texttt{to}\ r.T$ and $\texttt{ent}\ \mathscr{P}[r]\langle\tilde{v}\rangle\ \texttt{from}\ r.T$ constructors respectively.

Val	::=	$\texttt{int} \mid \texttt{bool} \mid \texttt{string}$	Sorts
K, S	::=	$Role \mid Val \mid \diamond \mid (K_1 \times \ldots \times K_n) \longrightarrow K$	Kinds

GLOBAL TYPES

G	::=	$\texttt{end}$	Termination
	$\mid$	$\texttt{let}\ \mathscr{P} = \lambda\tilde{r}^1,\ \tilde{y} \mapsto \texttt{new}\ \tilde{r}^2.G\ \texttt{in}\ G'$	Subprotocol Def
	$\mid$	$r\ \texttt{calls}\ \mathscr{P}\langle\tilde{r},\ \tilde{y}\rangle.G$	Subprotocol Call
	$\mid$	$r_1 \longrightarrow r_2 : \sum_{i\in I}\{l_i(S_i).G_i\}$	Branching
	$\mid$	$G_1 \bigoplus^r G_2$	Located Choice
	$\mid$	$G_1 \mid G_2$	Parallel Composition
	$\mid$	t	Type Variable
	$\mid$	$\mu t.G$	Recursive Type

LOCAL TYPES

T	::=	$\texttt{end}$	Termination
	$\mid$	$\texttt{send}[r]!_{i\in I}\{l_i(x_i : S_i).T_i\}$	Selection
	$\mid$	$\texttt{get}[r]?_{i\in I}\{l_i(x_i : S_i).T_i\}$	Branching
	$\mid$	$T_1 \bigoplus T_2$	Located Choice
	$\mid$	$T_1 \mid T_2$	Parallel Composition
	$\mid$	$\texttt{call}\ \mathscr{P} : G\ \texttt{with}\ (\tilde{v}\ \texttt{as}\ \tilde{y} : \tilde{S})\&(\tilde{r}^2).T$	Subprotocol Call
	$\mid$	$\texttt{ent}\ \mathscr{P}[r]\langle\tilde{v}\rangle\ \texttt{from}\ r.T$	Accept Invitation
	$\mid$	$\texttt{req}\ \mathscr{P}[r]\langle\tilde{v}\rangle\ \texttt{to}\ r.T$	Send Invitation
	$\mid$	t	Type Variable
	$\mid$	$\mu t.T$	Recursive Type

Figure 2.14: Syntax of Session Types for Nested Protocols

As before, deriving the local types from the global type is done through the projection operation, although now it becomes necessary to define a protocol environment to hold the types of the nested protocols, which gets updated by the `let in` constructor.

Figure 2.15 shows the definition of projection for the nested protocol definition and nested protocol call. Projection on the remaining constructors is defined in same way as before. The projection of a protocol definition updates the environment with the protocol's global type and signature and continues recursively projecting the type. When projecting a nested protocol call onto a role r^p, there are multiple cases to consider:

- if r^p is the caller but does not participate itself in the subprotocol, the projection must only send out the internal invitations in parallel, as well as initialising the protocol call and executing the projection of the continuation.
- if r^p is the caller as well as a participant, then it must send all the internal invitations to the participating roles, including itself, and it must also accept its own invitation. The role must also initialise the protocol call and proceed to execute the projection of the continuation.
- if r^p did not call the protocol but is a participant of the protocol, then it must simply accept the invitation, with the continuation being the projection of the continuation of the global type.
- otherwise, r^p is neither a participant or the caller, so the projection is the projection of the continuation of the global type.

$$(\texttt{let } \mathcal{P} = \lambda\widetilde{\mathtt{r}}^1.\widetilde{y} \mapsto \texttt{new } \widetilde{\mathtt{r}}^2.G_{\mathcal{P}} \texttt{ in } G') \Downarrow^{\mathbf{Env}}_{\mathtt{r}^p} = G' \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathtt{r}}^1,\widetilde{y};\widetilde{\mathtt{r}}^2;G_{\mathcal{P}})}_{\mathtt{r}^p}$$

$$(\mathtt{r}^A \texttt{ calls } \mathcal{P}(\widetilde{\mathtt{r}}^0,\widetilde{v}).G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathtt{r}}^1,\widetilde{y};\widetilde{\mathtt{r}}^2;G_{\mathcal{P}})}_{\mathtt{r}^p} =$$

$$\begin{cases} \text{if } \mathtt{r}^p = \mathtt{r}^A,\ \mathtt{r}^A \notin \widetilde{\mathtt{r}^0} \\ \quad \texttt{call } \mathcal{P} : G_{\mathcal{P}} \texttt{ with } (\widetilde{v} \texttt{ as } \widetilde{y})\&(\widetilde{\mathtt{r}}^2).[(G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathtt{r}}^1,\widetilde{y};\widetilde{\mathtt{r}}^2;G_{\mathcal{P}})}_{\mathtt{r}^p} \\ \quad \| \texttt{ req } \mathcal{P}[\mathtt{r}^1_0]\langle\widetilde{v}\rangle \texttt{ to } \mathtt{r}^0_0 \| \cdots \| \texttt{ req } \mathcal{P}[\mathtt{r}^1_n]\langle\widetilde{v}\rangle \texttt{ to } \mathtt{r}^0_n] \\ \text{if } \mathtt{r}^p = \mathtt{r}^A \text{ and } \mathtt{r}^A = \mathtt{r}^0_i \\ \quad \texttt{call } \mathcal{P} : G_{\mathcal{P}} \texttt{ with } (\widetilde{v} \texttt{ as } \widetilde{y})\&(\widetilde{\mathtt{r}}^2).[(G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathtt{r}}^1,\widetilde{y};\widetilde{\mathtt{r}}^2,G_{\mathcal{P}})}_{\mathtt{r}^p} \\ \quad \| \texttt{ ent } \mathcal{P}[\mathtt{r}^1_i]\langle\widetilde{v}\rangle \texttt{ from } \mathtt{r}^A \ \| \texttt{ req } \mathcal{P}[\mathtt{r}^1_0]\langle\widetilde{v}\rangle \texttt{ to } \mathtt{r}^0_0 \| \cdots \| \texttt{ req } \mathcal{P}[\mathtt{r}^1_n]\langle\widetilde{v}\rangle \texttt{ to } \mathtt{r}^0_n] \\ \text{if } \mathtt{r}^p \neq \mathtt{r}^A \text{ and } \mathtt{r}^p = \mathtt{r}^0_i \\ \quad \texttt{ent } \mathcal{P}[\mathtt{r}^1_i]\langle\widetilde{v}\rangle \texttt{ from } \mathtt{r}^A.(G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathtt{r}}^1,\widetilde{y};\widetilde{\mathtt{r}}^2;G_{\mathcal{P}})}_{\mathtt{r}^p} \\ \text{Otherwise} \\ \quad (G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathtt{r}}^1,\widetilde{y};\widetilde{\mathtt{r}}^2;G_{\mathcal{P}})}_{\mathtt{r}^p} \end{cases}$$

Figure 2.15: Projection of Global Type in Nested Protocols[12]

2.2.4 Returning Values from Subprotocols

One limitation of the formulation expressed above is the inability of communication from a subsession back to the parent subsession. This means that it is impossible to

express in a session type the relationship between a value calculated during a subsession and a value which a role might send after the nested protocol has ended.

For example, in the Client-Proxy-Server protocol described in Figure 2.16, the value the server returns is *ans*, but that value cannot be seen outside of the *Contact* subprotocol, so the proxy (middle) can only return ans_0 after completing the call to *Contact*. This protocol description does not provide any guarantees that these two values are the same. The paper proposes some further extensions to the syntax of session types so that the protocol ends by returning a value to the role which initiated the call in order to be able to return information from the subprotocol.

Nevertheless, the current theory will suffice in most scenarios, as the kinds of the messages alone restrict what values can be sent, and it is up to the user's implementation to decide which value to send. Any value which satisfies this constraint can be considered a valid implementation of the protocol.

$$
\begin{array}{l}
\texttt{let } Contact = \lambda \text{agent}, req \mapsto \\
\qquad \texttt{new } \text{server}. \\
\qquad \text{agent} \rightarrow \text{server} : \text{request}(req). \\
\qquad \text{server} \rightarrow \text{agent} : \text{answer}(ans). \\
\qquad \texttt{end} \\
\texttt{in} \\
\text{client} \rightarrow \text{middle} : \text{request}(req_0). \\
\qquad (\text{middle} \rightarrow \text{client} : \text{answer}(ans_0).\texttt{end}) \\
\oplus^{\text{middle}} \\
\qquad (\text{middle } \texttt{calls } Contact(\text{middle}, req_0). \\
\qquad \text{middle} \rightarrow \text{client} : \text{answer}(ans_0).\texttt{end})
\end{array}
$$

Figure 2.16: Example of Global Session Type for CPS Protocol[12]

Chapter 3

Extending Scribble with Nested Protocols

In this chapter we extend the Featherweight Scribble language[29] to model nested protocols. We extend the syntax of both global and local types with constructs for representing nested protocols similar to the ones described in [12]. We introduce a precise definition of a protocol's scope - the protocols which are you are able to call from within a protocol, and extend the definition of well-formedness to take our syntax extensions into account. Finally, we extend the projection of Scribble global protocols with the new constructs we introduce.

3.1 Syntax Extensions

In Section 2.1.5 we described how the Scribble syntax already has most of the constructs needed to encode session types, including constructs for encoding labelled message exchanges, recursion and external choice. The session types presented in the nested sessions paper[12], which we describe in Section 2.2.3, also include some constructs which are not currently implemented in Scribble, like the internal choice session type. However, we have decided to focus on implementing the main functionality related to nested protocols, such as making it possible to declare protocols inside other protocols and introducing the use of invitations in order to participate in a nested protocol call, rather than extending Scribble with other non-essential constructs. We will therefore not support internal choice, but this should not be a great limitation, as the external choice construct will suffice for most use cases.

In the nested protocols theory, a nested protocol call can also specify values which should be passed into the subsession, but we will not support this ability to send values when calling a nested protocol explicitly in our protocol declarations. Instead, in our implementation we will provide a means for the user to pass in values to subsessions by initialising the state of the different roles which will participate in the nested protocol. Hence, this omission should not have a great impact in practice.

We introduce a precise definition for the scope of protocols, which the original

paper[12] did not handle explicitly. Through the use of scopes, the users can encapsulate different behaviours as nested protocols which can only be used inside the protocol where they are defined, and these scope restrictions will be verified by the framework.

3.1.1 Global Protocols

We have extended the definition of a Scribble module, which we described in Section 2.1.5, with nested protocols. Previously, a module was a collection of global protocols. With our extension, it can also contain zero or more top-level nested protocols, although it must still contain at least one global protocol (with no dynamic roles) which can be used as the entry point for the computation. As in [12], nested protocols can also be defined inside both nested and global protocols.

Although nested protocols could in theory be defined anywhere within a scope, for ease of parsing in the Scribble implementation and to provide more structured Scribble modules, we restrict the location where nested protocols can be declared. At the top-level, any nested protocols must be defined before the global protocol declarations, and when defined within other protocols, they must be declared before any of the protocol's interactions. These restrictions do not impact the expressiveness of the implementation, as the order of the protocol declarations within a scope does not matter.

Module ::= $N*$ $P+$

PBody ::= $N*$ G

N ::= `nested protocol` *pro*(`role` A_1, ... , `role` A_n; `new role` B_1, ... , `role` B_m) { *PBody* }

P ::= `global protocol` *pro*(`role` A_1, ... , `role` A_n) { *PBody* }

G ::=

- `choice at` A { G_1 } `or` ... `or` { G_n }
- a(S) `from` A `to` B; G
- `rec` t { G }
- `continue` t
- `do` *pro*(A_1, ... , A_n); G
- A `calls` *pro*(A_1, ... , A_n); G
- `end`

Figure 3.1: Syntax for Scribble module with nested protocols

A formal specification of our proposed syntax for Scribble modules with nested protocols is given in Figure 3.1, and we provide some examples of global and nested protocols in Figure 3.2. We introduce several new keywords in order to define nested

protocols and calls to nested protocols, while trying to preserve as much of the original syntax and behaviour as possible.

- The `nested` keyword can be used to define nested protocols with dynamic participants. Dynamic participants are separated from regular participants by a `new` keyword in the protocol declaration, but it is not necessary for nested protocols to have dynamic participants, in which case the latter part of the declaration can be omitted. When a protocol is defined within another protocol, it restricts the scope where it can be used, much like defining a nested function in a programming language. We also support shadowing of protocol names: if you redefine a protocol in an inner scope, then it will override the previous definition in that scope and all its subscopes.
- The `calls` construct is used to introduce a new subsession when a role calls a nested protocol, inviting a set of roles to carry out the interactions of the nested protocol. In a `calls` construct the dynamic participants of the nested protocol are omitted - it is only necessary to specify which existing roles will be participating, as the remaining roles will be dynamically created.
- We modify the semantics of the `do` construct, which previously resulted in expanding the interactions of the called protocol, to instead create a subsession. Its semantics are equivalent to a `calls` construct, where the first participant in the protocol is treated as the caller. Even though they express the similar behaviours, we restrict the use of the `calls` construct to only call nested protocols and the `do` construct to call global protocols.
- The distinction between global and nested protocols is important, and needs to be verified to ensure that any protocol call used the correct construct. Moreover, protocol calls need to be verified to ensure that the call matches the signature of the protocol which is in scope, as protocol declarations with the same protocol name in different scopes may involve a different number of participants.
- As we did in Section 2.1.5, we have simplified the notation for labelled messages. Even though we only write down labelled messages with one payload field, Scribble also supports user-defined labelled messages with zero or more payload fields. In our implementation the user may optionally define names for any of the payload fields as well, as long as there are not any duplicate field names within the message payload.
- As before, the `end` construct can be omitted when writing down protocol declarations.

3.1.2 Scopes

As it was briefly mentioned in Section 3.1, the definition of nested protocols presented in [12] did not handle the scopes in which protocols could be called explicitly. In our Scribble extension we provide a more precise definition of protocol's

```
1
2   nested protocol NestedProtocol(role X; new role Y) {
3       do GlobalProtocol(X, Y);
4   }
5
6   global protocol GlobalProtocol(role A, role B) {
7       B calls NestedProtocol(A);
8   }
```

Figure 3.2: Example protocols showcasing our Scribble syntax extensions

scope and how it affects which protocols can be called from within different protocols in a Scribble module.

In a Scribble module we have the top-level scope, where the user can define global and nested protocols which can be called from any protocol, no matter how deeply nested its declaration is. As we mentioned in Section 3.1.1, each protocol declaration introduces a new scope of its own. Any nested protocols defined within are not visible outside that protocol, but can be used in that protocol's implementation and any nested protocols defined within.

The main restriction we have applied is that within any given scope, there must never be a clash between protocols of the same kind. For simplicity, the naming conflicts only take into account the name of the protocol rather than the full signature. This means that global protocols, which can only be defined at the top level, must always have unique names, and no two nested protocols defined in the same scope can have the same name.

Global vs Nested Protocols

The distinction between the `do` and `calls` constructs for calling global and nested protocols makes it possible to unambiguously define a nested protocol and a global protocol with the same name without one definition shadowing the other. However, in this case the user must be careful when calling the protocol, especially if they have signatures with the same number of non-dynamic participants, as it can be easy to confuse which protocol call was intended. If the signatures are different this should be less of a problem, as trying to call a protocol with the wrong number of participants will produce an error.

As we mentioned in Section 3.1.1, defining a nested protocol with the same name as a nested protocol in an outer scope will override the definition in the current scope and any inner scopes. Global protocol definitions cannot be shadowed, as they can only be defined within the top-level scope. When checking that protocol calls are valid it is important to keep track of the protocols which are currently in scope and their signatures to ensure that the call matches the number of roles in the protocol's

signature.

3.1.3 Local Protocols

After extending the syntax for Scribble global protocols we also had to modify the definition of projection of Scribble global types presented in [25] to include these new constructs, which we describe in Section 3.4. We therefore extend the syntax of Scribble local types with constructs for sending and receiving invitations following the session types presented in [12].

L ::= `local protocol` $A@pro$(`role` A_1, ... , `role` A_n; `new role` B_1, ... , `role` B_m) { T }

RecvMsg ::=
- | a(S) `from` B;
- | `accept` $C@pro$(A_1, ... , A_n; `new` B_1, ... , B_m) `from` A;

T ::=
- | `choice at` A {T_1} `or` ... `or` {T_n}
- | `rec` t { T }
- | `continue` t
- | a(S) `to` B; T
- | *RecvMsg* T
- | `invite`(A_1, ... , A_n) `to` pro; T
- | `create`(`role` B_1, ... , `role` B_m) `in` pro; T
- | `end`

Figure 3.3: Syntax of Scribble local protocols extended with invitations

Our proposed extensions to the syntax of Scribble local protocols are shown in Figure 3.3, and some examples of local protocols, which correspond to the projections of the protocols in Figure 3.2, are can be seen in Figure 3.4. We have kept all the original Scribble constructs and added three new ones for sending invitations, accepting invitations and for initializing dynamic roles:

- The sending of internal invitations is expressed by the `invite` construct, which specifies the roles which are going to participate and the name of the protocol to which they are invited. The syntax for the `invite` construct shown above can be seen as a shorthand for sending a series of individual invitations in parallel (asynchronously) to all the roles who are going to participate in the protocol. Indeed, this idea could be expressed with an alternative notation such as: {`invite`(A_i) `to` pro;}$_{i \in \{1, \ldots, n\}}$; T

- Bringing new participants into the subsession through external invitations is done through the `create` statement, which specifies the dynamic roles of the protocol to be created. Similarly to the `invite` construct, the proposed syntax can be seen as a shorthand for the sending in parallel of external invitations.

- Internal invitations are accepted through the `accept` construct, which contains information about the participant which sent the invitation, the local protocol which the role is going to be carrying out as well as all the other roles which will be participating as well.

```
1
2   local protocol X@NestedProtocol(role X; new role Y) {
3       invite(X, Y) to GlobalProtocol;
4       accept A@GlobalProtocol(X, Y);
5   }
6
7   local protocol Y@NestedProtocol(role X; new role Y) {
8       accept B@GlobalProtocol(X, Y);
9   }
10
11  local protocol A@GlobalProtocol(role A, role B) {
12      accept X@NestedProtocol(A; new Y) from B;
13  }
14
15  local protocol B@GlobalProtocol(role A, role B) {
16      invite(A) to NestedProtocol;
17      create(role Y) in NestedProtocol;
18  }
```

Figure 3.4: Examples of Scribble local protocols showcasing the new syntax constructs

3.2 Well-formedness

In order to be able to generate the implementation of a protocol specified by the user, the Scribble framework must first verify that the protocol is well-formed. According to the definitions in [25, 12], in order for a protocol to be well-formed it must be projectable and all protocol calls must be valid. A protocol is projectable if the projection is defined for all the participants of that protocol, and a protocol call is valid if the called protocol is defined/in scope, it is called with the correct number of roles and all the roles involved are distinct and in scope (participating in the current protocol).

Despite our extensions to the Scribble framework, the well-formedness property remains the same. We must highlight the importance of ensuring that protocol calls are valid after introducing nested scopes and the ability to shadow protocol declarations, which make it possible users to declare protocols with the same name and different declarations in different scopes. We also extend the definition of projection given in [12] by incorporating the full merge operator, which the original paper did not. In order to do this, we extend the definition of the merge operator defined in [25] to incorporate our new constructs for representing protocol calls.

3.3 Renaming Protocols

The representation of Scribble nested protocols we have defined in Section 3.1.1 is similar to the representation given in [12], but the distinction between global and nested protocols and protocol calls in Scribble makes it more complicated and verbose to define the projection of protocols. Name clashes between protocols in different scopes will also need to be resolved at some point before code generation to be able to generate a correct implementation of the protocols, so we introduce an intermediate step before projection in order to solve these issues.

We propose introducing an extra preprocessing step on the protocols before projecting them, once the global and nested protocols have been validated to ensure they are syntactically correct and that all protocol calls are valid. The objective in this step is to generate a simpler, flattened representation of the Scribble module, where all the protocols are stored in a single set. In order to achieve this, name clashes need to be resolved to ensure that all protocol names are unique, and all references to the protocols in protocol calls must be renamed to the new unique names. Ensuring that all protocol names are globally unique has the added advantage of enabling us to remove the distinction between nested and global protocols and treat protocol calls to both global and nested protocols uniformly, as there can be no ambiguity with regards to which protocol is going to be called.

The result of this renaming process is a set of protocols which do not contain any nested protocols within, where both global and nested protocols have been converted into this intermediate representation. We define a different syntax for this representation, as shown in Figure 3.5, which is essentially the same as a nested protocol declaration without any nested protocols inside.

$$\texttt{protocol}\ P(\texttt{role}\ A_1, \ldots, \texttt{role}\ A_n; \texttt{new}\ \texttt{role}\ B_1, \ldots, \texttt{role}\ B_m)\ \{G\}$$

Figure 3.5: Intermediate representation of protocols after renaming

We present a renaming algorithm which uses two environments. The first one maps the original protocol names that were accessible in the current scope to the unique names of the protocols they refer to, while the second environment holds the set of all the unique protocol names which have been generated. The scope environment

can be used to update the protocol calls in the interactions of a protocol, and the protocol name environment is necessary to generate globally unique protocol names. We define the algorithm in pseudocode in Figures 3.6 and 3.7.

3.3.1 Renaming Algorithm

The first step in the algorithm is to aggregate the information from the protocol declarations in the **top-level scope** into both environments. When resolving clashes between top-level nested and global protocols we have decided to always change the nested protocol names and keep the global protocol names without changing them. With this approach we are able to only keep a single environment for the nested protocols which are in the current scope, as global protocol names will not change, so there is no need to track them in a separate environment. We add the unique names of both the top-level global and nested protocols to the protocol name environment, and store a mapping from the nested protocol names to their new unique names in the scope environment. As well as updating the environments, we also change the protocol names in the nested protocol declarations. This process can be seen in lines 15-25 of the `rename_module` function in Figure 3.7.

Once we have all the information about the top-level scope, we can proceed to **recursively rename** the interactions of all the protocols, aggregating all the resulting protocols in a single set. In order to update the interactions of a protocol, we must first update both the scope and protocol name environments with the nested protocol declarations inside the current protocol. The procedure for how this can be done is shown in the function `rename_nested_protocols` in Figure 3.6. We first generate a new unique name for each of the protocols and add it to the protocol name environment. The scope environment is updated by creating/modifying the mapping for the protocol's old name so that it refers to the new name. We also update the declarations of the nested protocols with their new unique names here. Once both environments have been updated, we recursively rename the interactions of each of the nested protocols and store them in the result set. Finally, we update the protocol names in the current protocol's interactions using the scope environment and add the protocol with its new interactions to the result set.

The procedure we have just described is implemented in the `rename_and_flatten_protocols` function in Figure 3.7. We need to return the updated protocol name environment as well as the set of renamed protocols to ensure that the protocol names we generate are unique. Otherwise, clashes could occur between protocols defined inside different nested protocols, but we can discard the updated scope environment once the protocol's interactions have been updated, as it does not contain any useful information outside the current scope.

3.3.2 Renaming Protocol Interactions

Renaming a protocol's interactions simply requires recursively traversing the global type, as shown in the `rename_protocol_calls` function in Figure 3.6. The only interactions which change are the global protocol calls and nested protocol calls. In order to treat both types of calls in the same way, global protocol calls are converted explicitly to the `calls` construct with the first participant initiating the call. As global protocol names were not changed, the protocol name in the call remains the same. On the other hand, updating nested protocol calls requires looking up the unique protocol name corresponding to the protocol being called in the scope environment and building a new `calls` construct with it.

```
1   def rename_protocol_calls(G, ScopeEnv):
2       match G with:
3           | A calls P(A_1, ... , A_n); G_1 ->
4               newP = ScopeEnv[P]
5               newG_1 = rename_protocol_calls(G_1, ScopeEnv)
6               return A calls newP(A_1, ... , A_n); newG_1
7           | do P(A_1, ... , A_n); G_1 ->
8               newG_1 = rename_protocol_calls(G_1, ScopeEnv)
9               return A_1 calls P(A_1, ... , A_n); newG_1
10          | choice at A {G_i}_{i∈I} ->
11              return choice at A
                    { rename_protocol_calls(G_i, ScopeEnv) }_{i∈I}
12          | rec t {G_1} ->
13              newG_1 = rename_protocol_calls(G_1, ScopeEnv)
14              return rec t {newG_1}
15          | a(S) from A to B; G_1 ->
16              newG_1 = rename_protocol_calls(G_1, ScopeEnv)
17              return a(S) from A to B; newG_1
18          | continue t ->
19              return continue t
20          | end ->
21              return end
22
23  def rename_nested_protocols(nested_protocols, ScopeEnv,
        ProtocolNames):
24      renamed_protocols = ∅
25      for (nested protocol P(role A_1, ... , role A_n;
            new role B_1, ... , role B_m) {nested_protos; G}) in
            nested_protocols:
26          newP = UNIQUE_NAME(P, ProtocolNames)
27          ScopeEnv = ScopeEnv[P -> newP]
28          ProtocolNames = ProtocolNames ⋃ {newP}
29          renamed_protocols = renamed_protocols ⋃ {nested protocol
                newP(role A_1, ... , role A_n; new role B_1, ... , role B_m)
                {nested_protos; G}}
30      return (renamed_protocols, ScopeEnv, ProtocolNames)
```

Figure 3.6: Algorithm for renaming protocol interactions

```
def rename_and_flatten_protocols(nestedProtocol, ScopeEnv,
    ProtocolNames, AllProtocols):
    match nestedProtocol with:
      | nested protocol P(role A_1, ... , role A_n; new
          role B_1, ... , role B_m) {nested_protocols; G} ->
        (ScopeEnv, ProtocolNames, nestedProtocols) =
            rename_nested_protocols(nestedProtocols, ScopeEnv,
              ProtocolNames)
        for nestedProtocol in nestedProtocols:
          ProtocolNames, AllProtocols =
              rename_and_flatten_protocols(nestedProtocol,
                ScopeEnv, ProtocolNames, AllProtocols)

        newG = rename_protocol_calls(G, ScopeEnv)
        AllProtocols = AllProtocols ⋃ {protocol
            P(role A_1, ... , role A_n; new role B_1, ... , role B_m)
            {newG}}
        return ProtocolNames, AllProtocols

def rename_module(nested_protocols, global_protocols):
    ProtocolNames = ∅
    ScopeEnv = {}
    TopLevelProtocols = ∅
    AllProtocols = ∅
    for (global protocol P(role A_1, ... , role A_n) {nested_protos;
        G}) in global_protocols:
      ProtocolNames = ProtocolNames ⋃ P
      TopLevelProtocols = TopLevelProtocols ⋃ {nested protocol
          P(role A_1, ... , role A_n; ∅) {nested_protos; G}}

    renamed_nested_protocols, ScopeEnv, ProtocolNames =
        rename_nested_protocols(nested_protocols, ScopeEnv,
          ProtocolNames)
    TopLevelProtocols = TopLevelProtocols ⋃
        renamed_nested_protocols
    for top_level_protocol in TopLevelProtocols:
        ProtocolNames, AllProtocols =
            rename_and_flatten_protocols(top_level_protocol,
            SetupEnv, ProtocolNames, AllProtocols)

    return AllProtocols
```

Figure 3.7: Algorithm for renaming protocols in a Scribble module

3.4 Projection

Like the definition of projection presented in [12], projecting a global or nested protocol requires having an environment which keeps track of all the protocols which are in scope. Our definition of projection is defined on the flattened representation of the Scribble module we described in Section 3.3, where all the protocols are stored in one large set and the distinction between global and nested protocols has been removed. Moreover, in order to successfully rename all the protocols the original Scribble module must have been checked to ensure all protocol calls are valid. From this flattened representation, building the projection environment is a trivial process, where the protocol declarations in the set can be aggregated to create a mapping from protocol names to their role signatures ($P \mapsto \{A_1, \ldots, A_n; B_1, \ldots, B_m\}$).

It is not a problem to store the signatures of all the protocols in the environment, because in the validation step and the renaming process we will have already verified that the scope restriction is not violated, so the protocols will only ever need to look up protocols which are defined in their scope in the projection environment. The definition of projection is undefined for protocol calls to protocols which are not in the environment, but this case should never arise if the protocols have passed the previous validation steps we have described.

3.4.1 Projection of Global Protocols

In order to generate all the local protocols in a Scribble module, we must project every protocol onto every role taking part in it, which will generate a much larger set of local protocols. We define the projection of a Scribble protocol P onto a role A in Figure 3.8.

$$(\texttt{protocol}\ P(\texttt{role}\ A_1, \ldots, \texttt{role}\ A_n; \texttt{new role}\ B_1, \ldots, \texttt{role}\ B_m)\ \{G\})\downarrow_A^{Env} =$$

$$\begin{cases} \textit{If}\ A \in \{A_1, \ldots, A_n, B_1, \ldots, B_m\}: \\ \texttt{local protocol}\ A@P(\texttt{role}\ A_1, \ldots, \texttt{role}\ A_n; \texttt{new role}\ B_1, \ldots, \texttt{role}\ B_m)\ \{G\downarrow_A^{Env}\} \\ \\ \textit{Otherwise}: \quad \textit{undefined} \end{cases}$$

Figure 3.8: Projection of a Scribble protocol

The projection operation on global protocols remains the same as defined in [25] for all constructs except the `choice`. Figure 3.9 shows these definitions, which we have taken from the paper. We denote $\mathscr{P}(G)$ as the set of roles participating in protocol G.

Previously, the `do` construct would be expanded before projection, which we do not do. However, after removing the distinction between global and nested protocols, all calls to global protocols using the `do` construct will have been converted to `calls`, so we also do not need to define a rule for projecting `do`.

$$(\texttt{a(S) from } B \texttt{ to } C;\ G') \downarrow_A^{Env} =$$
$$\begin{cases} \texttt{a(S) to } C;\ (G' \downarrow_A^{Env}) & \textit{if } A = B \\ \texttt{a(S) from } B;\ (G' \downarrow_A^{Env}) & \textit{if } A = C \\ G' \downarrow_A^{Env} & \textit{otherwise} \end{cases}$$

$$(\texttt{rec } t\ \{G'\}) \downarrow_A^{Env} =$$
$$\begin{cases} \texttt{rec } t\ \{G' \downarrow_A^{Env}\} & \textit{if } A \in \mathscr{P}(G') \\ \texttt{end} & \textit{otherwise} \end{cases}$$

$$(\texttt{continue } t) \downarrow_A^{Env} = \texttt{continue } t$$

$$(\texttt{end}) \downarrow_A^{Env} = \texttt{end}$$

Figure 3.9: Projection of constructs taken from Featherweight Scribble[25]

Projection of Protocol Calls

The projection of the `calls` construct is based on the projection of the protocol call which was defined in [12], and it considers the same four cases. When the projected role is making a call, it must send all the invitations to the other participants, potentially including itself. It must also send external invitations to the new dynamic participants which will take part in the subsession, if any. If there are no dynamic participants, then the `create` construct can be omitted. If the projected role is taking part in the subsession, then it must also accept the invitation to the appropriate role in the nested protocol from the caller. If the projected role is neither the caller or a participant in the protocol call, then the protocol call can be ignored. The formal definition of the projection of the `calls` construct is shown in Figure 3.10

3.4.2 Projection of Choice

The Scribble `choice` construct does not have a single well-defined semantics and projection definition. Different implementations of Scribble may have marginally different definitions about what constitutes a well-formed choice. We base our definition in the Multiparty Session Types[11] and the definition of the Scribble `choice` construct presented in [25]. We define the projection of the `choice` construct in Figure 3.12.

Before taking nested protocols into account, we define a `choice` construct as a directed choice from the participant making the choice, e.g. `B`, to a single participant, which receives a distinct first message from `B` in each choice branch. Through this first message, the receiving role will be able to identify which choice `B` makes. With the addition of nested protocols and invitations, labelled messages `a(S)` are not the

$$(C\ \texttt{calls}\ P(A_1, \ldots, A_n);\ G') \downarrow_A^{Env,\ P \mapsto \{D_1, \ldots, D_n;\ B_1, \ldots, B_m\}} =$$

$$\begin{cases} \texttt{invite}(A_1, \ldots, A_n)\ \texttt{to}\ P; & \text{if } A = C,\ \exists i.C = A_i \\ \texttt{create}(\texttt{role}\ B_1, \ldots, \texttt{role}\ B_m)\ \texttt{in}\ P; & \\ \texttt{accept}\ D_i@P(A_1, \ldots, A_n;\ \texttt{new}\ B_1, \ldots, B_m)\ \texttt{from}\ C; & \\ (G' \downarrow_A^{Env,\ P \mapsto \{D_1, \ldots, D_n;\ B_1, \ldots, B_m\}}) & \\ \\ \texttt{invite}(A_1, \ldots, A_n)\ \texttt{to}\ P; & \text{if } A = C,\ C \notin \{A_1, \ldots, A_n\} \\ \texttt{create}(\texttt{role}\ B_1, \ldots, \texttt{role}\ B_m)\ \texttt{in}\ P; & \\ (G' \downarrow_A^{Env,\ P \mapsto \{D_1, \ldots, D_n;\ B_1, \ldots, B_m\}}) & \\ \\ \texttt{accept}\ D_i@P(A_1, \ldots, A_n;\ \texttt{new}\ B_1, \ldots, B_m)\ \texttt{from}\ C; & \text{if } A \neq C,\ \exists i.C = A_i \\ (G' \downarrow_A^{Env,\ P \mapsto \{D_1, \ldots, D_n;\ B_1, \ldots, B_m\}}) & \\ \\ (G' \downarrow_A^{Env,\ P \mapsto \{D_1, \ldots, D_n;\ B_1, \ldots, B_m\}}) & \text{otherwise} \end{cases}$$

Figure 3.10: Projection of the `calls` construct

$$R_1 = R_2 \implies \texttt{IS\_MSG\_FROM}(R_2,\ \texttt{a(S)}\ \texttt{from}\ R_1\ \texttt{to}\ C;\ G)$$

$$R_1 = R_2 \implies \texttt{IS\_MSG\_FROM}(R_2,\ R_1\ \texttt{calls}\ P(A_1, \ldots, A_n);\ G)$$

$$R_1 = R_2 \implies \texttt{IS\_RECV\_FROM}(R_2,\ \texttt{a(S)}\ \texttt{from}\ R_1;\ )$$

$$R_1 = R_2 \implies \texttt{IS\_RECV\_FROM}(R_2,\ \texttt{accept}\ D@P(A_1, \ldots, A_n;\ \texttt{new}\ B_1, \ldots, B_m)\ \texttt{from}\ R_1;\ )$$

$$\texttt{FIRST\_RECEIVERS}(G) = \begin{cases} \{B\} & \text{if } G = \texttt{a(S)}\ \texttt{from}\ A\ \texttt{to}\ B;\ G' \\ \{A_1, \ldots, A_n\} & \text{if } G = A\ \texttt{calls}\ P(A_1, \ldots, A_n);\ G' \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 3.11: Auxiliary definitions for projection of `choice` and merge operator

only type of messages which can be sent. A protocol call involves sending invitations to different roles, and receiving different invitations will also enable participants to discern which choice `B` makes. Therefore, it is valid to have a combination of protocol calls and message exchanges initiated by B as the first interactions in the `choice` branches.

In order to enforce this restriction, we define an auxiliary predicate `IS_MSG_FROM(R, G)`, which can be seen in Figure 3.11. We define two rules from which this predicate can be derived, one for a protocol call and one for a labelled message exchange. The rules specify that in order to derive the predicate, the role which is the first parame-

$$(\texttt{choice at } B\ \{G_i\}_{i\in I})\downarrow_A^{Env} =$$

$$\begin{cases} \texttt{choice at } B\ \{(G_i\downarrow_A^{Env})\}_{i\in I} & \text{if } A = B \text{ or } A \in \bigcap_{i\in I} \texttt{FIRST\_RECEIVERS}(G_i) \\ \bigsqcup_{i\in I} (G_i\downarrow_A^{Env}) & \text{otherwise} \end{cases}$$

if $\forall i \in I.\texttt{IS\_MSG\_FROM}(B, G_i)$

Otherwise it is undefined

Figure 3.12: Projection of `choice` extended with invitations

ter in the predicate must also be the sender of the message in the first interaction of the global type. Then, in the definition of the projection of `choice` we require that the global type of each branch satisfies the predicate $\texttt{IS\_MSG\_FROM}(B, G_i)$, where B is the role making the choice.

On top of this restriction, we still require that each first message is unique in each branch. We consider that two protocol calls are equal only if the protocol name is the same and the same participants are invited to carry out the same roles in the called protocol, and two labelled messages are the same if both the labels and all the payloads are the same.

Taking invitations into account makes it possible for a `choice` to not have a single first receiver, because during the setup of a protocol call multiple invitations are sent out in parallel. Any of the roles which receive one of these invitations could potentially be considered as the first receiver in the `choice`. We therefore build the set of possible receivers as the intersection of the first receivers in each branch. This intersection must not be empty for the choice to be valid. As before, sending a labelled message restricts the possible candidates to the receiver of the message. We extract this information from the global types of each branch with the function $\texttt{FIRST\_RECEIVERS}(G)$.

Moreover, because the sending of invitations is asynchronous and a role can send an invitation to itself, it is valid to have a `choice` where the role making the choice is also a potential first receiver. This can occur if each branch of the `choice` starts with a protocol call in which that role also participates, but if any branch starts with a labelled message exchange, the first receiver can never be the role making the choice, as it is invalid for a role to send a message to itself.

We extend the definition of projection for `choice` for the case where the projected role is not the role making the choice or one of the roles receiving the first message. The definition proposed in the nested protocols paper only applied a simple merge over all the branches, where the continuation of the role in each of the branches must be exactly the same. Instead, we try to merge the projections of the different

branches using a variation of the full merge operator, which we define in Figure 3.13.

3.4.3 The Full Merge Operator

The definition of the full merge operator we present in Figure 3.13 is a variation of the one defined in [25], which we extend to take into account both invitations and labelled messages.

$$T_1 \sqcup T_2 = T_1 \quad \textit{if } T_1 = T_2$$

$$\texttt{choice at } B\ \{RecvMsg_i\ \ T_i\}_{i \in I} \sqcup \texttt{choice at } B\ \{RecvMsg_j\ \ T'_j\}_{j \in J} =$$
$$\texttt{choice at } B\ \{RecvMsg_i\ \ T_i\}_{i \in I \setminus J} \cup \{RecvMsg_j\ \ T'_j\}_{j \in J \setminus I} \cup$$
$$\{RecvMsg_k\ \ T_k \sqcup T'_k\}_{k \in I \cap J}$$
$$\textit{if } \forall k \in I \cup J.\, \texttt{IS\_RECV\_FROM}(B,\ RecvMsg_k)$$

$$T_1 \sqcup T_2 \quad \textit{is undefined otherwise}$$

Figure 3.13: Definition of the full merge operator

Two local types can always be merged if they are the same type. Two `choice` constructs can be merged if the same role is making the choice, and the first interaction in every branch is the receiving of a different message (invitation or labelled message) from that role. We express this by ensuring that the local type of each branch starts with a *RecvMsg* (defined in Section 3.3), which can either be the receiving of a labelled message or an `accept` construct. Moreover, we enforce that the sender of the first message in all branches must be the role making the choice by requiring that the local type of every branch satisfies the predicate $\texttt{IS\_RECV\_FROM}(B, RecvMsg_k)$, where $RecvMsg_k$ is the first interaction in each branch. We define two rules to derive this predicate in Figure 3.11, where the predicate can only be derived if the role passed in as a parameter is the one sending the message in the local type.

The resulting merged type in this case is a `choice` which combines the branches from both types. The branches which do not overlap between the two `choice` constructs can remain the same, but the branches where the first received message is common to both of them must be recursively merged to produce a new type. To reduce the number of cases to consider in the definition of the merge operator we implicitly use the equivalence between a `choice` with a single branch and the interactions of that branch without the `choice`: $\texttt{choice at } R\ \{T\} \equiv T$.

Chapter 4

Design of Code Generation

In this chapter we describe the different factors and limitations we have had to take into account when designing the structure of the implementation we generate for a protocol. We describe from a high-level how the implementation works and how we divide it across different packages.

4.1 Code Generation Approach

Our code generation approach is different from the one normally used in Scribble, which we described in Section 2.1.5. We do not generate a communication finite state machine (CFSM) from the local protocol of each of the roles, but rather generate code directly from the local type. The main obstacle in creating a CFSM is that it is not possible to encode properly nested sessions using a CFSM. Implementing nested protocols using this approach would require creating a Communicating Stack Machine which could keep track of the stack of protocol calls so it would know which state to return to after the protocol call had finished. Compared to this, generating code directly from the local type is simpler, and it also enables us to design a scheme which directly encodes the behaviour of the local type into the Go implementation. We will describe this scheme in Chapter 6.

4.2 Implementation Design

Here we provide a high-level description of the main components which we have used in our implementation of nested protocols. A more detailed breakdown of each component and how we have structured them is given in Chapter 5.

When coming up with the design of the implementation of nested protocols, we wanted the approach to be as simple and intuitive to use as possible. Because our target language was Go, we wanted to take advantage of the in-built concurrency primitives of the language: *shared memory channels* and *goroutines*.

4.2.1 Implementation of Roles

In our implementation, participants execute in parallel as different goroutines, and the dynamic participants in nested protocols are created as new goroutines every time a protocol call is made. All communications between participants are asynchronous, and they carried out over **buffered channels**. This means that sends will only block once the buffer is full, and a receive will only block when the buffer is empty. We create structs for the different kinds of messages in a protocol, and store all the different channels needed by a role for all its message exchanges in a struct. Each role also has a struct containing all the channels it needs to send and receive invitations. Our invitations consist of two structs, the first one containing the channels that a participant will need to carry out the interactions of the role they will be undertaking, and the second one containing the channels that they will need in order to initiate and participate any protocol calls as their new role.

Each channel that a role has will only be used once, regardless of whether it is used for sending or receiving a labelled message or an invitation. This means that a role will have as many channels as the number of messages it sends and receives.

4.2.2 Callbacks

When generating the implementation of the role, we also generate an interface which contains the signatures of different methods which are called from the role's implementation. This makes it possible for the user to define the protocol's behaviour without modifying the implementation of the protocol which is automatically generated. An instance of the interface is used to maintain the role's state throughout the protocol by calling the interface's methods. These **callbacks** can provide new information which was received, such as a received labelled message, or enable the user to provide input needed for the execution, such as producing the message which will be sent or deciding which branch of a `choice` to follow. Similar approaches to this one using callbacks have been used in other implementations of Scribble, such as [26].

4.2.3 Nested Protocol Calls

The implementation of a role is generated as a **self-contained function**, which only contains the interactions defined in the local protocol. Calls to nested protocols involve an initial setup phase which is initiated by the role which is the caller of the protocol. During this setup, the invitations for all the roles are created: the new channels needed by all the roles are created and aggregated to create the channel and invitation structs for all of the roles. These structs are sent to all non-dynamic participants through their invitation channels. Dynamic participants also receive their channels, but they receive them directly when they are spawned as new goroutines which execute the functions corresponding to their role's implementation. Once the invitations are sent, a role participating in the protocol call simply needs to

accept their invitation, create the callbacks environment for their new role and then call the function which implements the behaviour of that role.

4.2.4 Returning Results from Protocols

In our implementation, we also attempt to tackle the problem of returning information from a protocol call, which was outlined in [12] and we describe in Section 2.2.4. To solve it, we define **result structs** for every non-dynamic role of each protocol. The function implementing the behaviour of these roles will return its corresponding result struct, which can be aggregated into the state of the caller through a callback. The result struct can be generated from a role's state, and it is returned by the last callback that is called in the implementation: `Done()`, which signifies that the role has finished executing all its interactions. The user can fill the structs with whatever information it wants the role to bring out of the subsession.

Although this return value is only the partial state of each role, not a return value for the protocol as a whole, it is a simple mechanism through which roles can retain part of their state after executing in a nested session, and it does not introduce any additional messages in the protocol implementation. This is already a massive improvement over not being able to retain any information at all. However, this solution is not fully satisfactory for the entry-point protocol, which should only be called once to setup all the initial roles and produce a single result. Our proposal to solve this problem is to aggregate the results of all the roles into a single state which the user can access outside of the protocol. We create an interface which accepts the result of each of the initial roles in the entry-point protocol. The user can then define an implementation of the interface which aggregates all these return values in a useful way. Nevertheless, this solution is not ideal, and we will continue to discuss this part of the design and its limitations in Section 5.8.1.

4.2.5 Entry-Point Protocol Setup

When setting up the entry-point protocol, the process is essentially the same as the setup for a protocol call. All the channels for the roles must be created, but instead of receiving their channels as invitations, the roles execute as new goroutines which receive the channels as parameters directly. In order to synchronise the ending of the protocol, we use one of Go's synchronization mechanisms: `sync.WaitGroup`. A wait group has an internal counter which specifies the number of pending jobs which should finish before resuming execution. The main thread waits on the wait group until all the goroutines of the initial participants finish executing. Every time a dynamic participant is created, the wait group's counter must be increased to ensure that execution does not continue before all the participants, including the dynamic participants, have finished executing.

4.3 Implementation Restrictions

In Chapter 3 we have described the Scribble syntax for defining nested protocols. Using our Scribble extensions the user can define complex dependency graphs between protocols with mutual recursion. Moreover, the framework places almost no restrictions with regards to how things should be named: nested protocol definitions can be shadowed by definitions of nested protocols with the same name in different scopes, and there is no fixed conventions for naming roles, protocols or message labels.

Translating the declaration of a protocol defined in a language which gives the user so much freedom about how things are named and structured to a programming language with stricter restrictions like Go is a great challenge. We have tried to come up with a design for the implementation which is simple and easy to use, but in order to do so we have had to place a set of small restrictions on the user when defining the protocols in order to be able to generate their implementation. These restrictions only need to be enforced when generating the implementation of the protocol, so the framework will only make check them before the start of the code generation process. These checks will not be made, for instance, if the user only wants to display the projection of a protocol.

4.3.1 Code Organisation in Go

Go programs are organised into packages, which are a collection of source files which are compiled together. Functions, types, structs and constants defined in a source file are visible to all the other source files in the same package[1]. This means that generating two of these constructs with the same name in the same package will produce an error, even if the declarations are in different source files. Moreover, Go uses capitalisation in order to determine the visibility of the constructs declared within a package - if they start with a capital letter then they can be accessed outside the package, otherwise they can only be accessed within the package.

It is invalid to have cyclic import dependencies between different packages in a codebase, but there are different strategies for solving this issue: from placing the files which depend on one another in the same package to declaring interfaces to remove the coupling between the implementation of different packages. In our design, we opt for the first alternative.

It is not possible to define two packages or source files with the same name in the same directory. By convention, both packages and files tend to have short, single-word names which are all lowercased. For multiword names, the convention is to use camel case rather than underscores: `camelCase` or `CamelCase` rather than `snake_case`.

4.3.2 Naming Restrictions

The lack of any enforceable naming conventions in Scribble makes it difficult come up with a consistent naming scheme for the different parts of the implementation. We aim to adhere to the existing naming conventions in Go wherever possible.

One of the main concerns while generating names for types and functions is to ensure there are no name clashes within the same package. In order to simplify the code generation process and produce meaningful names for the variables and constructs, we place the following constraints on the names of the Scribble protocols, labelled messages and role names:

- **Global protocol names** must be unique when they are lowercased, because they are used to name some packages in the implementation, and they are lowercased for that purpose, following the Go naming convention. The uniqueness of protocol names we currently guarantee through the renaming process described in Section 3.3 is case sensitive, which means two protocols `"P1"` and `"p1"` are considered to be different. This stronger uniqueness property is only required for code generation, so it is not necessary to enforce it at an earlier stage.

- **Message labels** must be unique when capitalised within any protocol. We generate a new struct for every kind of labelled message using the message's label, and in order for the structs to be visible outside the package, they must be capitalised, which makes enforcing this restriction necessary. We also require labelled messages to have a consistent set of fields throughout their uses within a protocol, because we only define one struct per label name. It would be possible to generate different structs for different uses of the same label, but this might not be as clear to the user. We have decided instead to push the responsibility of disambiguating labelled messages to the user, but it should not be hard for them to work around this restriction.

- Naming all of the fields in a message's payload is optional for the user, but if they do name the **payload fields** then the provided names must be unique when capitalised. The payload field names of a labelled message are used as the names of the message struct fields which are generated. In order for them to be visible outside of the package where are defined they must be capitalised in the implementation, so we must enforce this restriction to avoid generating two fields with the same name. Unnamed fields are given an automatically generated unique name based on their type.

- **Role names** within a protocol must also be unique when case is ignored. This restriction is necessary, because we use role names to generate the names for many different structs and functions, and it makes it easier to generate unique names.

While these restrictions will not enforce that our implementation follows the naming conventions in Go, they are useful to ensure that the constructs we generate are

unique within a package. Moreover, the naming scheme we have chosen aims to produce meaningful names which closely resemble the declarations in the Scribble module so that the user can easily understand the generated codebase.

4.3.3 Package Organisation Restrictions

One of the factors which conditioned our design was cyclic imports. Ideally, we would have wanted to achieve a highly-modular design with the implementation of every protocol within its own module. After all, *modularity* is one of the main features that nested protocols can provide. However, the ability to define mutually recursive protocols in Scribble means that any two protocols can potentially depend on one another, whether directly or indirectly. This means that their implementations will also be linked in some way, and therefore, it is not possible to define the whole implementation of each protocol within a different package, as the dependencies between protocols would cause cyclic imports.

Our design therefore tries to split up the implementation of each protocol into different components, and keeps the source files for each of the components in different packages wherever possible. We introduce the components that we generate and their structure in Section 4.4.

4.4 Implementation Structure

The directory structure of the implementation of a protocol is shown in Figure 4.1. We use the `filename.go` / `pkgname` style when referring to the names of packages and files in our implementation. The code we generate is all contained within a single package, named after the entry-point protocol. The different components which we described in Section 4.1 are split across seven packages. For some of these we were able to completely separate the logic of each protocol into a different subpackage, each named after the protocol which they implement. In the directory tree, we add another directory inside these packages to show this difference in their structure, like in the case of the `messages` package. It was not possible to do this for all of them, as we described in Section 4.3.3, so for the remaining packages the implementation of each protocol is defined across one or more different source files stored directly within the package.

Some of the packages we have defined only define structs which are needed for the implementation, while others hold more logic. We describe the purpose of the seven packages below and provide a high-level description of what each one contains:

- `messages/protocol_pkg/`: Within each of these packages we define the structs for the different labelled messages which are exchanged in each protocol.
- `channels/protocol_pkg/`: Within each of these packages we define the structs which hold all the channels that each role in the protocol will need during its execution to communicate with the other roles.

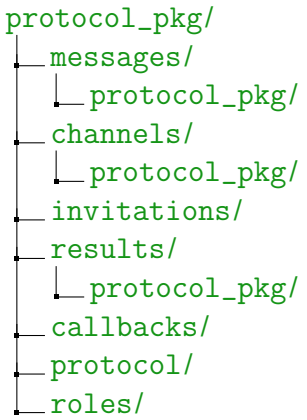

Figure 4.1: Package structure for the implementation of a protocol

- `invitations/`: Each source file within this package defines the invitation structs for all the roles of a different protocol. These invitation structs contain the channels needed for the roles to send and receive invitations during their execution. We also define two special structs which are used to group the channels used to send the invitations during the setup of a protocol call.
- `results/protocol_pkg/`: Within each of these packages we generate empty structs which represent the return values of each of the roles of the protocol.
- `callbacks/`: This package stores the definitions of the callback interfaces of all the roles of all the protocols. The files for dynamic roles also define a function which the user should implement to return an instance of the interface for that role. The callbacks file for a role may contain enums whose values represent the possible alternatives for a choice the role makes.
- `protocol/`: This package contains the logic for the initial setup for the entry-point protocol. It defines an interface which the user can implement to provide an initial state for each of the roles in the protocol, as well as to aggregate the results of each of the roles when they finish executing. It also defines several functions to create all the channels, start the execution of all the roles and synchronise the ending of the protocol.
- `roles/`: This package contains the implementation of all the roles across all the protocols. We also define in this package the code for setting up a call to each protocol.

Chapter 5

Project Implementation

In this chapter we provide a detailed breakdown of the structure of the packages we described in Section 4.4, showing how the different parts of the implementation are generated from the definition of the Scribble protocols, and describe how the different elements of the implementation work together to produce the correct behaviour of the protocol. The code generation scheme for generating a role's implementation from their Scribble protocol specification will be described in Chapter 6.

5.1 Naming and Notation

In Section 4.1 we described from a high-level point of view how our generated implementation works. Before explaining how we generate all the different parts of our implementation, we must first introduce some notation that we will use to name the different components in our code generation scheme, from function names to the names of structs, interfaces and variables:

- `var_name` - variable names
- `"str"` - string literal
- `type_name` - Go primitive types
- `struct_name` - Struct names
- `struct_field` - Names of struct fields
- *`func_name`* - Function names
- *`interface_name`* - Names of Go interfaces
- *`enum_type`* - Names of Go enum types
- `enum_value` - Names of the different values an enum type can take

In our implementation, all generated file and package names are lowercased, following the Go convention. Variable names will always start with a lower case, and the names of all the other language constructs we define: functions, interfaces, structs, etc. will be capitalised so that they can be visible outside the package in which they are defined.

In our code generation definitions, whenever we introduce a new variable name we will assume that the variable name has not been used previously in the same scope. This simplifies the code generation, as we do not have to worry about which variable names have been used and if the previous declaration of that variable has the type we need.

The names we generate use different parts of the Scribble definition in order to clearly show how each component in the implementation relates to the protocol declaration. In order to illustrate how we have done this, we will embed certain reserved keywords in the names we use when we define our code generation scheme. These keywords are placeholders which represent the names that the user would have provided when defining the protocols:

- *protocol*: This denotes the unique global protocol name which was generated after the renaming phase described in Section 3.3. The only exception is when we refer to the package `protocol` in our implementation, which is one of the top-level packages we generate in our implementation. To disambiguate this package from a package named after a user-defined protocol, we will always refer to the latter as `protocol_pkg`.

- *role*: This denotes the name of a role participating in a protocol.

- *role@protocol*: This denotes the unique name corresponding to the local protocol which is the result of projecting the protocol *protocol* onto *role*.

  In the current implementation, our naming scheme would generate the local protocol name for the local protocol *role@protocol* as *protocol_role*. This scheme on its own is not guaranteed to be unique even if protocol names are globally unique and the roles participating in a protocol are also unique, so we have to do extra work to ensure that there are no name clashes. In our definitions we will use this notation to hide away these complexities, which should be handled during the implementation of the code generation.

- *msg*: This denotes the label of a labelled message which is exchanged by two roles in the protocol.

- *payload_field*: This denotes the name of field inside the payload of a labelled message.

- *payload_type*: This denotes type of a field inside the payload of a labelled message.

- `roleChannels`, `inviteChannels`, `wg` and `env`: These are special variable names which are used as parameters in the functions we generate to implement the behaviour of the roles.

The names we generate for our implemetations do not follow Go's camel case naming convention. They are generated from the names provided in the Scribble protocol declaration, and we use underscores to separate the parts we take from the protocol declaration from the other parts of the name instead of using mixed caps. This is the simplest way to ensure that the names produced are readable, especially if the final name we produce places user-defined names side by side.

5.2 Imports

In the description of our code generation scheme we will omit the import statements which would need to be generated in each source file. However, it should be clear from the names of the packages we access which imports would be needed. It is important to note that import clashes may arise in the implementation. Because some of the packages store the implementation of each protocol inside a package with the same name, if a source file tried to import two of those packages directly at the same time it would produce an error. For instance, a source file would not be able to directly import both the message and the channel structs for protocol *proto1*, as the desired implementations would be inside two packages named `proto1`.

To resolve this ambiguity when defining or code generation scheme, we introduce a new naming convention to refer to the package where the implementation of a protocol is defined - in such cases we will write: `protocol_package`. For instance, we would write `proto1_messages` to refer to package `proto1` inside package `messages`. We use a similar solution in our Go implementation, where we create import aliases to refer to both packages with different names.

5.3 Package `messages`

As we mentioned in Section 4.4, this package contains the structs for the different labelled messages which are exchanged in the different protocols of the Scribble module, each defined within a different subpackage. The package for each protocol contains a single source file, `messages.go` with the declarations of all the message structs.

We generate one struct per message label, with every payload field in the protocol message translating to a field inside the struct, and we only generate one struct per message label even if the label is used multiple times in the protocol. As we mentioned in Section 3.1.1, the user can optionally define the names for the payload fields as long as they are unique (within the same message). When generating code,

the framework will automatically generate unique names for any fields which the user does not name. Therefore, every payload field in all labelled messages will have a name which can be used to generate a field name in the message struct. For simplicity, we currently assume that the payload types which the user gives are valid Go primitive types, so we use them directly in the implementation as the types of the message struct fields.

$[\![\texttt{msg}(\texttt{payload\_field}_1 : \texttt{payload\_type}_1, \ldots, \texttt{payload\_field}_n : \texttt{payload\_type}_n)]\!] =$

```
type msg struct {
    payload_field_1  payload_type_1
    ...
    payload_field_n  payload_type_n
}
```

Figure 5.1: Generation of message struct from labelled message

Figure 5.1 illustrates this code generation process, showing how the different parts of a labelled message declaration are used to build the definition of the message struct. Although the message label name and payload fields need not be capitalised in general, in our code generation scheme definitions we assume that they are so we can use the message label and payload field names directly in the struct declaration.

5.4 Package `channels`

As we mentioned in Section 4.4, this package contains the declarations of the structs which contain the channels needed for the roles of a protocol to send and receive labelled messages. The structure of this package is similar to the `messages` package, where each protocol has its own subpackage with a source file named `channels.go` containing the channel structs for the roles of that protocol.

Every labelled message exchange in a protocol is carried out over a separate channel. In order to generate the channel struct for a role, we go through its local types and create a struct field for every labelled message which it sends or receives. Each of these fields holds a channel of the message struct corresponding to the label message in the exchange. A different channel struct is therefore generated from the local type of each role.

In Figure 5.2 we show what the struct generated by the scheme we have described would look like. Every field in the struct is generated from a labelled message exchange in the protocol's local type. The fields inside the struct that we generate must have unique names, but in our definition we assume that this is achieved in the implementation.

```
type role_Chan struct {
    role1_msg1  chan protocol_messages.msg1
    ...
    rolen_msgn  chan protocol_messages.msgn
}
```

Figure 5.2: Channel struct for role *role* in protocol *protocol*

5.5 Package `invitations`

In this package we define the invitation structs for the roles of all the protocols defined in the Scribble module. All the invitation structs for the roles of a protocol are stored in one same file named after the protocol they belong to: `protocol.go`. These files are all stored directly inside the package to avoid cyclic dependencies.

5.5.1 Invitation Structs

As we discussed in Section 4.2, in our implementation invitations consist of two structs: the channel struct and the invitation struct for the new role that a participant is going to carry out during a protocol call. These structs contain fresh channels that the role can use to communicate with the other participants in the subsession. For this reason, in order to send or receive an invitation two different channels are required.

In our design we use different channels each time we send or receive an invitation. For every protocol call that a role initiates, we add two channels to the struct to send the invitations for each of the participants in the call. If the protocol only participates in the protocol call, then we add two channels to receive the invitations. If the role is the one initiating that call, but also participates in the protocol, then it will be sending the invitation to itself over the channels created for sending the invitation, so there is no need to create any other channels.

We illustrate the process of how we add fields to a role's invitation struct using it's local type in Figure 5.3. We name the structs using the local protocol they are generated from in order to be able to clearly identify them whenever they are used, as all the invitation structs are defined within the same package. The channel fields inside the struct are named differently depending on whether the channel is used for sending or receiving invitations. In the case where a role is sending an invitation to itself, only one set of channels is created, following the naming scheme for channels used to send invitations.

The channel used to send the channel struct for the new role needs to refer to the struct defined inside the `channels` package. On the other hand, the invitation struct will always be defined within the `invitations` package itself, so it is possible to

$Env = Env',\ protocol' \mapsto \{role'_1, \ldots, role'_n;\ role''_1, \ldots, role''_m\}$

(`invite`$(role_1, \ldots, role_n)$ `to` $protocol'$;) $\implies$

```
type role@protocol_InviteChan struct {
    ...
    Invite_role_1_To_role'_1@protocol'              chan protocol'_channels.role_1'_Chan
    Invite_role_n_To_role'_n@protocol'_InviteChan   chan role_n'@protocol'_InviteChan
    ...
    Invite_role_n_To_role'_n@protocol'              chan protocol'_channels.role_n'_Chan
    Invite_role_n_To_role'_n@protocol'_InviteChan   chan role_n'@protocol'_InviteChan
    ...
}
```

if $role \neq role_0$ then :

(`accept` $role'_i@protocol'(role_1, \ldots, role_n;$ `new` $role''_1, \ldots, role''_m)$ `from` $role_0$;) $\implies$

```
type role@protocol_InviteChan struct {
    ...
    role_0_Invite_To_role'_i@protocol'              chan protocol'_channels.role'_i_Chan
    role_0_Invite_To_role'_i@protocol'_InviteChan   chan role'_i@protocol'_InviteChan
    ...
}
```

otherwise no new fields are added

Figure 5.3: Process for populating the invitation struct of role *role* in protocol *protocol*

access them directly without any explicit package access. It would not be possible to move the declarations of the invitation structs of each protocol into a different package, because it is possible for protocols to be mutually recursive, and trying to import the invitations structs could cause cyclic imports.

5.5.2 Protocol Setup Structs

We also define two more structs in each file, which are used to send the invitations during the setup of a protocol call. One of them will contain the channels over which to send the role channel structs to the participants, one for each non-dynamic participant in the protocol, and a similar struct which will contain the channels over which to send the invitation structs to the participants. The structure of these structs can be seen in Figure 5.4, and more details on how they are used will be given in Section 5.9.1.

```
type protocol_RoleSetupChan struct {
    role1_Chan  chan protocol_channels.role1_Chan
    ...
    rolen_Chan  chan protocol_channels.rolen_Chan
}

type protocol_InviteSetupChan struct {
    role1_InviteChan  chan role1@protocol_InviteChan
    ...
    rolen_InviteChan  chan rolen@protocol_InviteChan
}
```

Figure 5.4: Channel structs used during the setup of a protocol call

5.6 Package `results`

This package has a structure which is almost identical to the `messages` and `channels` packages. Each protocol has its own subpackage with a source file named `results.go` containing the result structs for the roles of that protocol.

We generate an empty struct for each non-dynamic role in the protocol. These structs are returned by the function implementing the behaviour of these roles. The user can fill the structs with fields to store whatever useful information the role can bring out of the session. This struct is returned by the *Done*() callback, which signifies that the role has finished executing. Therefore, the user can use all of the information stored in the state of the role to build this result struct. We show what the struct generated for a role would look like in Figure 5.5.

```
type role_Result struct {
}
```

Figure 5.5: Result struct for role *role* in protocol *protocol*

5.7 Package `callbacks`

In this package we define the interfaces for the callbacks environment of all the roles in all the protocols. This environment contains the signatures of all the functions which are called from the protocol's implementation. By providing an implementation of the interface, the user can define the behaviour of the protocol without modifying the functions which implement the behaviour of the different roles.

Each interface is defined in a file named after one of the local protocols which was generated by the projection of the protocols in the Scribble module. By doing so, we guarantee that the file names will be unique, and also make it easy to identify where the implementation of each role is defined. The file may also contain definitions for enums which represent the different branches that the role can choose to follow in the choices it makes during its execution. We illustrate what the definition of the interface looks like in Figure 5.6.

```
type role@protocol_Env interface {
    callback_1
    ...
    callback_n
}
```

Figure 5.6: Callbacks interface for role *role* in protocol *protocol*

5.7.1 Callback Generation

We generate callbacks based on the interactions that a role carries out in the nested protocol. Like we mentioned in Section 4.2, these callbacks will enable the user to update the state of a role based on the messages that it receives from other roles or the results of a protocol call it takes in. They will also allow them to provide inputs to guide the execution of the protocol, like deciding which branch to take in a `choice` or building the message struct to send to another role. In order to generate these structs we recursively traverse the local protocol of a role and add callbacks to the interface based on the interactions it carries out.

The correspondence between the local protocol constructs and the callbacks we generated is shown if Figure 5.7. Note that the definitions we provide here are just meant to illustrate how we generate callbacks from each interaction, they do not express the recursive traversal of the local protocol.

When a role receives a message, we add a callback which has the received message as an argument so that the user can update the state of the environment as needed. On the other hand, when sending a message, we add a callback which returns message struct which needs to be sent.

When accepting an invitation to participate in a protocol call, we generate two callbacks, one of which is called before the protocol call is carried out and the other one after the role has finished executing its interactions in the called protocol. The first callback returns the initial state for the role in the called protocol, which enables the user to use the role's current state when creating the new initial state. The second

$Env = Env',\ protocol' \mapsto \{role'_1, \ldots, role'_n;\ role''_1, \ldots, role''_m\}$

(msg(payload) from $role'$;) $\Longrightarrow$

msg_From_role'(msg protocol_messages.msg)

(msg(payload) to $role'$;) $\Longrightarrow$

msg_To_role'() protocol_messages.msg

(accept $role'_i$@$protocol'$($role_1, \ldots, role_n$; new $role''_1, \ldots, role''_m$) from $role_0$;) $\Longrightarrow$

$To\_role'_i@protocol'\_Env$() $role'_i@protocol'\_Env$

$ResultFrom\_role'_i@protocol'$(result protocol'_results.role'_i_Result)

(invite($role_1, \ldots, role_n$) to $protocol'$;
create(role $role''_1, \ldots,$ role $role''_m$) in $protocol'$;) $\Longrightarrow$

$protocol'\_Setup$()

(choice at $role'$ { T_1 } or ... or { T_n }) $\Longrightarrow$

if $role' = role$ then:

$role\_Choice$() $role@protocol\_Choice$

otherwise no new callbacks are generated

(end) $\Longrightarrow$

if $role \notin$ dynamic_participants($protocol$) then:

$Done$() protocol_results.role_Result

otherwise:

$Done$()

Figure 5.7: Process for generating callbacks from role *role*'s local protocol

one takes in the result returned after executing the called protocol as a parameter, which can be used to incorporate the information returned from the protocol call into the role's state. The implementation of our setup for a protocol call combines the sending of invitations to existing roles and the creation of the dynamic participants, so we produce a single callback for both local protocol constructs.

Choice Callback

When a role makes a choice, it must decide which branch of execution to follow. Each branch starts by sending a different message to a non-empty set of roles. When

executing the `choice`, we first need the user to decide which branch to follow, and we define an enum type with one value for each branch in the `choice` to encode this decision. We introduce a callback which will return one of these enum values so that the user can decide which execution path to follow.

$$\texttt{LABEL}(T) = \begin{cases} \texttt{msg} & \text{if } T = \texttt{msg(payload) to } role;\ T' \\ \texttt{protocol} & \text{if } T = \texttt{invite}(role_1, \dots, role_n) \texttt{ to } protocol;\ T' \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$(\texttt{choice at } role\ \{T_i\}_{i \in \{1, \dots, n\}}) \implies$$

```
type role@protocol_Choice int
const (
    role@protocol_LABEL(T1) role@protocol_Choice = iota
    role@protocol_LABEL(T2)
    ...
    role@protocol_LABEL(Tn)
)
```

Figure 5.8: Scheme to generate enums for each of the branches in a `choice` made by role *role*

The enum types and values we generate must be unique in the whole package, and we ensure that this is the case in our implementation. We incorporate the local protocol in the names we generate to show which role is making the choice. The enum values are named after the label of the first interaction in every branch of the `choice`. We show how these enums are generated in Figure 5.8. Enums in Go are usually defined as constants whose type is a type alias of `int`, and the use of the special `iota` keyword auto-increments the value that each successive constant receives.

If the role making the `choice` is not the role in the local protocol, then it is not necessary to generate any further callbacks, as the callbacks generated for the first interactions in each branch will be enough to distinguish which branch of execution was selected.

Done Callback

Finally, as we mentioned in Section 4.2, whenever a branch of the execution in the protocol finishes executing, which is encoded through the `end` local type, a call to a special callback will be made. This callback, which we name *Done*(), is used to clean up and finalise the state of the role. In the case of non-dynamic participants, this callback will also return a result struct containing the useful information that the user wants to return from the role's final state. In this way, the useful computation

carried out during a protocol can be returned outside of the subsession. We only add a single *Done*() callback to the environment, because it is always meant to be the last callback to be called after a protocol finishes executing, no matter how many different execution paths there are.

5.7.2 Initial State of Dynamic Participants

As we have seen, when a role accepts an invitation to a protocol call, the new environment they use while they act as that role is created from their current state. However, for dynamic participants this is not possible, because dynamic participants are goroutines which are created every time a protocol call is made. They don't have any prior state, so in order to create their initial state we generate function declarations whose return type is the role's environment for the user to implement. These functions are stateless, so the user will not be able to provide any input when generating their state. This should not be a massive obstacle in practice, as the user could choose to explicitly build up their state through labelled message exchanges in the protocol. We show what a function declaration would look like in Figure 5.9.

```
func New_role@protocol_State() role@protocol_Env {
    panic("TODO:Implement")
}
```

Figure 5.9: Declaration of function to create the initial state of a dynamic role *role*

5.7.3 Package Design Restrictions

The reason why the definition of all the callback interfaces needs to be placed in the same packages is, as we have mentioned numerous times, because protocol calls can be mutually recursive. The callbacks we generate for setting up the environment of the new role which a participant is going to carry out depends on the interface of that role. If the interfaces were defined in different packages, importing them could cause cyclic imports. We therefore chose to ensure that they are all directly visible to one another by placing them within the same package.

5.8 Package `protocol`

This package contains all the logic for initialising the roles for the entry-point protocol as well as aggregating the results from each of the roles. Inside the package we generate a single file named after the entry-point protocol, `protocol.go`, where all the implementation is defined. Inside this file we define three things: an interface which we use to initialise the state of the roles in the protocol and aggregate the

results they compute, a function where we create and assign the channels used by the roles to communicate and a set of functions for initialising each of the roles.

5.8.1 Protocol Setup Environment

As we mentioned in Section 4.2, in order to generate an implementation of the protocol which can be easily parametrised by the user, we define an interface which returns the initial state of every role in the protocol. In order to initialise the protocol, we only require the user to provide an instance of the interface with the information needed to generate the initial states of all the roles. We also use the same interface to aggregate the results returned by all the roles once they have finished executing. The user will still hold a reference to the instance of the interface which was used during the initialisation, so after aggregating the results produced by the roles in the interface, the user will have access to the protocol's "result". We show what the definition of this interface would look like in Figure 5.10.

This solution is not ideal, as it is easy for the user to introduce race conditions in the implementation when trying to aggregate the results of different roles. Because the roles execute in parallel, they will all finish at different points in time. If a role finishes executing shortly after another one and they both attempt to write their result into the shared state, race conditions could occur if they both modify the same parts of the state, unless the user used a synchronization mechanism. Nevertheless, we consider it to be the user's responsibility to ensure that the logic for aggregating the state is correct.

```
type protocol_Env interface {
    New_role_1_Env() callbacks.role_1@protocol_Env
    ...
    New_role_n_Env() callbacks.role_n@protocol_Env

    role_1_Result(result protocol_results.role_1_Result)
    ...
    role_1_Result(result protocol_results.role_n_Result)
}
```

Figure 5.10: Entry-point protocol setup interface declaration

5.8.2 Protocol Setup Function

The main function we generate is the one which holds all the logic to start the execution of all the roles. The function receives an instance of the protocol setup interface we have just described as a parameter and does the following:

- Create all the channels required by all the roles in the protocol to send labelled messages.
- Create all the channels that the roles will need to send and receive invitations.
- Create the channel and invitation structs for all the roles in the protocol, filling in all the fields with the channels it created.

  Every channel will be used exactly twice, once in the struct of the sender role and once in the struct of the receiver role. The channels which are used by roles to send invitations to themselves are the only exception, as they will only be used once to fill the corresponding fields in the invitation struct of that role.
- Create a `sync.WaitGroup` variable and increment its counter by the number of roles in the protocol.

  As we described in Section 4.2, we use a wait group variable, `wg`, to synchronise the ending of the protocol. All the roles will execute as different goroutines. We use a wait group to block the main thread's execution until all the roles have finished executing. Whenever one of the roles finishes executing, it will call the `wg.`*Done*`()` method on the wait group, decrementing its internal counter, and the main thread's execution will only proceed once the counter drops to zero.
- Use the protocol setup environment to create the initial state of all the roles.
- Start the execution of each role by creating a different goroutine for each one, providing the setup interface, the wait group, the role's channel struct and invitation struct and the role's callback interface as parameters.

5.8.3 Initial Roles Setup

The functions we create to start the execution of each role in the entry-point protocol for the first time are special wrappers around the ones which implement the behaviour of the roles. We show how they are implemented in Figure 5.11. Their body consists of three statements: the `defer` statement ensures that no matter what, once current function finishes executing, either by returning a value or raising an exception, the function call given will be executed. In this case, we defer the execution of the call to decrement the wait group's counter. We then call the function implementing the role's behaviour, passing the wait group variable, the channel and invitation structs and the role's state as parameters. The role's result which is returned is then aggregated into the protocol result by passing it as a parameter to the role's callback in the protocol environment.

Decrementing the counter cannot be done inside the function which implements the role's behaviour, as that function can be called as a result of a protocol call, and the role's execution could continue after the function returned. If the counter is decremented too many times, it will drop to zero before all the goroutines finish

```
func Start_role@protocol(protocolEnv protocol_Env, wg *sync.WaitGroup,
        roleChannels protocol_channels.role_Chan,
        inviteChannels invitations.role@protocol_InviteChan,
        env callbacks.role@protocol_Env) {
    defer wg.Done()
    result := roles.role@protocol(wg, roleChannels, inviteChannels, env)
    protocolEnv.role_Result(result)
}
```

Figure 5.11: Role start function for role *role* in protocol *protocol*

executing. This would mean that the main thread would be able to continue executing before all the roles had aggregated their results, so the result for the protocol returned to the user would be incomplete/erroneous. It would even possible for the main thread to completely finish executing before the goroutines were able to finish, forcefully terminating them.

5.9 Package `roles`

This package contains the implementation for all the local protocols generated from the Scribble module as well as the logic for setting up a call to any of the protocols. The role implementations are defined in files named after the local protocols, `role@protocol.go`, which contain a single function with all the logic. Similarly, the setup for each protocol is defined in a file named `protocol_setup.go`, where all the setup logic is contained within a single function.

5.9.1 Protocol Setup Functions

As we discussed in Section 5.5, our invitations consist of two structs: the channel and the invitation struct that the roles will need to communicate with the other roles in during the protocol call. The purpose of the setup function is to create the channel and invitation structs that all the roles participating in the protocol call will need, and making them available to the correct participants. This process involves creating the channels for the dynamic participants as well, because non-dynamic roles can communicate with dynamic ones and we have to ensure that *all* the roles have access to the channels they need to communicate. The process for setting up protocol calls is similar to the one we described for initialising the different roles for the entry-point protocol in Section 5.8.2:

- Create the channels that the roles will need for their labelled message exchanges.
- Create the channels that the roles will need to send and receive invitations.

- Create the structs for the channels and invitations of each role.
- Send the invitations to the non-dynamic participants in the protocol.
- If there are any dynamic participants in the protocol, then create a new goroutine for each one to execute the function implementing their behaviour.

  In order for the dynamic participants to be created they require an initial state. Because dynamic roles are always created from scratch every time a protocol call is made, we make them stateless. The user will be able to create the constant initial state of a dynamic role and return it using the extra function we generate in the `callbacks` package, which we described in Section 5.7.2.

  Because we are creating new goroutines, we must increment the counter of the wait group variable that keeps track of the number of roles still executing. Otherwise, the main thread could resume before all the roles had finished executing.

In order to send the invitations to the correct participants, the function must have access to the channels over which the participants are expecting the invitations to be sent. In order to do this, we use the two additional structs defined in the `invitations` package that we introduced in Section 5.5: `protocol_RoleSetupChan` and `protocol_InviteSetupChan`. These structs contain the channels over which each role will be waiting to receive their invitations. Therefore, when sending the channel and invitation struct to each role we simply select the channels corresponding to that role in the struct. These two structs will be built by the caller of the protocol, who will have access to all the necessary channels, and passed in as parameters to the setup function.

```
func protocol_SendCommChannels(wg *sync.WaitGroup,
        roleChannels invitations.protocol_RoleSetupChan,
        inviteChannels invitations.protocol_InviteSetupChan) {
    ...
}
```

Figure 5.12: Declaration of protocol call setup function for protocol *protocol*

We mentioned in our description of the setup function how we need to increment the wait group by the number of dynamic participants in the protocol, but we also need to decrement the counter once a dynamic role finishes executing. We do this by adding a `defer wg.Done()` call at the beginning of the function of a dynamic participant, as we show in Figure 5.13. Because we need to increment and decrement wait group's counter in order to synchronise the termination of the protocol correctly, we pass it in as a parameter to our function. We show the signature of a setup function in Figure 5.12.

By using this design, we are able to define all the logic for the protocol setup in a single function and call it from any protocol, as opposed to having to create a different implementation of the setup process every time a protocol call was made.

5.9.2 Role Implementation Functions

Our implementation of a role's behaviour requires 4 different components:

- A reference to the wait group which is used to monitor whether all roles have finished executing.
- The struct containing the channels used by the role to exchange labelled messages with other roles in the protocol.
- The struct containing the channels to send and receive invitations to nested protocol calls.
- A variable containing the role's state, which implements the interface *role@protocol_Env*, which gets updated as the role interacts with the other participants.

These four variables are passed in as parameters to the function implementing the role's behaviour. The signature of the functions we generate can be seen in Figure 5.13. There are two differences between the functions of a dynamic and non-dynamic role. Firstly, a dynamic role does not produce a result after it finishes executing. The purpose of introducing return values is so that the role which participated in a protocol call can bring any results computed outside of the protocol. However, dynamic participants are newly created every time a protocol call is made, so when they finish executing there will be no-one to process their return value. For the same reason, we can carry out the `defer wg.Done()` call inside the function, since the role finishes executing after the function returns. This was not possible for the entry-point function, were we had to create wrappers for initialising the roles of the entry-point protocol for the first time, as there is no guarantee that the role will finish executing after the function exits.

5.9.3 Package Design Restrictions

Like many of the packages we have described, the files containing the implementation for the roles in each protocol cannot be separated into different packages. This is due to the fact that protocols can be mutually recursive, which means that the implementation of roles in different protocols can call each other, which would cause cyclic dependencies if the files were stored in different packages. The files containing the setup for each protocol also need to be defined within the same package as the role implementations to avoid cyclic imports. If a protocol is recursive, then the implementation of the role which carries out the recursive call will depend on the setup function, but the setup function depends on the implementation of the dynamic participants, as the dynamic participants are created within the setup function. Therefore, it would not be possible to define the implementation of the roles

```
⟦local protocol role_i@protocol(role role_1, ... , role role_n;
                                new role role'_1, ... , role role'_m) { T }⟧ =

func role_i@protocol(wg *sync.WaitGroup,
        roleChannels protocol_channels.role_i_Chan,
        inviteChannels invitations.role_i@protocol_InviteChan,
        env callbacks.role_i@protocol_Env) protocol_results.role_i_Result {
    ⟦T⟧
}


⟦local protocol role'_i@protocol(role role_1, ... , role role_n;
                                new role role'_1, ... , role role'_m) { T }⟧ =

func role'_i@protocol(wg *sync.WaitGroup,
        roleChannels protocol_channels.role'_i_Chan,
        inviteChannels invitations.role'_i@protocol_InviteChan,
        env callbacks.role'_i@protocol_Env) protocol_results.role'_i_Result {
    defer wg.Done()
    ⟦T⟧
}
```

Figure 5.13: Generation of role *role*'s implementation function from its local protocol

in a different package from the protocol setup logic.

Chapter 6

Implementation of Local Protocols

In this Chapter we will define our scheme for implementing the behaviour of a Scribble local type in Go. As we mentioned in Section 5.9, a role's implementation uses 4 parameters: `wg`, `roleChannels`,`inviteChannels` and `env`, which are used to access the channels used to communicate with other roles, to setup nested protocol calls and to maintain the role's state. A role's implementation mainly consists of the different interactions which it carries out with the other roles in the protocol, and the actual protocol logic is implemented through the callbacks which are interleaved between the interactions to update the role's state and guide the execution of the protocol.

Our code generation scheme recursively traverses the protocol specification for each role in order to generate its corresponding Go implementation. In the definitions of our code generation scheme, we assume that all the structs, functions and enums defined in the other packages have all been generated. We also assume that the signature of the role implementation functions for all the roles across all the protocols are known.

6.1 Message Exchanges

In order to receive a labelled message from a different role, we first assign the result of receiving the message struct over the corresponding role channel to a variable. This message struct is then used to update the role's state by sending it as a parameter to the *msg_From_role'* callback. Sending a labelled message involves a similar process: we generate the user-defined message struct using the *msg_To_role'* callback and send it over the corresponding message channel. The Go code we generate can be seen in Figure 6.1.

```
⟦msg(payload) from role′; T⟧ =
    msg := <-roleChannels.role′_msg
    env.msg_From_role′(msg)
    ⟦T⟧

⟦msg(payload) to role′;⟧ =
    msg := env.msg_To_role′()
    roleChannels.role′_msg <- msg
    ⟦T⟧
```

Figure 6.1: Code generation scheme for message exchanges carried out by *role*

6.2 Protocol Calls

The code generation process for protocol calls consists of two different parts: setting up a protocol call, given by `invite` and `create`, and accepting an invitation to participate in a nested protocol, which is encoded by `accept`. We show the code generation scheme for these constructs in Figure 6.2.

First, the role calls the *protocol_Setup* callback to update its state before the protocol setup. Setting up a protocol call involves generating and sending all the invitations to the participants as well as creating new goroutines for the dynamic participants. As we described in Section 5.9, this logic is implemented in the *protocol_SendCommChannels* function. Before calling it, a *role* must first create the two protocol setup structs which the function takes in as parameters. The fields of the role and invitation setup structs must be set to the channels that the participating roles will use to receive their invitations, matching each participant to the role they will carry out. As we described in Section 5.5, these invitation channels will be stored in the role's `inviteChannels` struct. Once these structs are created, the role can call the setup function for that protocol.

The code generation for the `accept` construct not only generates the code for accepting the invitation and participating in the protocol, but also collects the results from the protocol call and updates the role state with it. The definition we show assumes that the role which initiates the protocol call is not the same as the current role. As we explained in Section 5.5 when a role accepts an invitation it sends to itself, the channels used are the same ones which were used to send the invitation.

The role must first receive the invitation that the caller will have sent in order to have the channels it will need for communicating with the other participants. After

the invitation has been received, the role needs to generate the environment which will maintain its state during the protocol call. As we described in Section 5.7, the *To_role$'_i$@protocol$'$_Env* callback will return its new state. The user can decide how this state is generated, possibly using information from the role's current state to create the new one. Once the role has received the message channels, invitation channels and its new environment, it can participate in the protocol call as the new role by calling that role's implementation function. Because the role it will carry out in the protocol call cannot be a dynamic participant, that role's implementation will return a result. This result can then be used to update the role's current state by passing it as a parameter to the *ResultFrom_role$'_i$@protocol$'$* callback.

$Env = Env',\ protocol' \mapsto \{role'_1, \ldots, role'_n; role''_1, \ldots, role''_m\}$

$[\![$ `invite`$(role_1, \ldots, role_n)$ `to` $protocol'$;
`create(role` $role''_1, \ldots,$ `role` $role''_m)$ `in` $protocol'$; $T]\!] =$

```
env.protocol_Setup()
protocol'_rolechan := invitations.protocol'_RoleSetupChan {
    role'_1_Chan:   inviteChannels.Invite_role_1_To_role'_1@protocol'
    ...
    role'_n_Chan:   inviteChannels.Invite_role_n_To_role'_n@protocol'
}
protocol'_invitechan := invitations.protocol'_RoleSetupChan {
    role'_1_InviteChan:   inviteChannels.Invite_role_1_To_role'_1@protocol'_InviteChan
    ...
    role'_n_InviteChan:   inviteChannels.Invite_role_n_To_role'_n@protocol'_InviteChan
}
protocol_SendCommChannels(wg, protocol'_rolechan, ,protocol'_invitechan)
[[T]]
```

$[\![$ `accept` $role'_i@protocol'(role_1, \ldots, role_n;$ `new` $role''_1, \ldots, role''_m)$ `from` $role_0;\ T]\!] =$

```
role'_i@protocol_chan := <-inviteChannels.role_0_Invite_To_role'_i@protocol'
role'_i@protocol_invitechan :=
          <-inviteChannels.role_0_Invite_To_role'_i@protocol'_InviteChan
role'_i@protocol_env := env.To_role'_i@protocol'_Env()
role'_i@protocol_result := role'_i@protocol(wg,
                                           role'_i@protocol_chan,
                                           role'_i@protocol_inviteChan,
                                           role'_i@protocol_env)
env.ResultFrom_role'_i@protocol'(role'_i@protocol_result)
[[T]]
```

Figure 6.2: Code generation scheme setting up a protocol call and accepting invitations

6.3 Choice

The implementation of `choice` that we generate changes based on whether the role we are implementing is making the choice or not.

If the role is making the choice, then in order for the implementation to know which branch of execution to follow we need input from the user. For this purpose we use the *role_Choice* callback and the *role@protocol_Choice* enum we defined in Section 5.7. The enum value returned by the user encodes which branch has been chosen. In our implementation, this choice operation is carried out using a `switch` statement, which will match the enum value and execute the implementation which is generated for that branch of the protocol. We show the definition of this code generation scheme in Figure 6.3. We use the same `LABEL()` function we defined in Section 5.7 to refer to the `choice` enum values.

$[\![$ `choice at` *role* $\{T_i\}_{i\in\{1\ldots n\}})]\!] =$

```
role_choice := env.Role_Choice()
switch role_Choice {
case callbacks.role@protocol_LABEL(T1):
  [[T1]]
...
case callbacks.role@protocol_LABEL(Tn):
  [[Tn]]
default:
  panic("InvalidChoice")
}
```

Figure 6.3: Code generation scheme for a `choice` made by *role*

On the other hand, if the role is not making the `choice`, the role will only be able to determine which branch to execute after receiving the first message. As we defined in Section 3.4.2, the first interaction in every branch of a well-formed `choice` will always be either the accepting an invitation or the receiving a labelled message.

Go's `select` statement can be used to make a goroutine wait on multiple communication operations in parallel[4], and execute the body of the first `case` which becomes available. If multiple statements are available at the same time, a random one will be chosen between them. Our implementation uses a `select` statement where every case contains the implementation of one of the `choice` branches. As we showed in Sections 6.1 and 6.2, the first statement in the implementation of both accepting an invitation and receiving a labelled message is a receive on one of the role's chan-

nels. This first receive statement from the implementation of every branch is used in each `case` statement to determine which execution path to follow once a message is received on any of the channels. This code generation scheme is shown in Figure 6.4.

$$\texttt{GENERATE\_CASE}(\mathit{Impl}) = \begin{cases} \texttt{case label\_var := <-ch:} \\ \quad \mathit{Impl}' & \text{if } \mathit{Impl} = [\texttt{label\_var := <-ch;}\ \mathit{Impl}'] \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$[\![\texttt{choice at } \mathit{role}'\ \{T_i\}_{i\in\{1\ldots n\}})]\!] =$$

```
select {
GENERATE_CASE([[T_1]])
...
GENERATE_CASE([[T_n]])
}
```

Figure 6.4: Code generation scheme for a `choice` which is made by a different role

6.4 Recursion

Unfortunately, the code generation scheme we have defined does not support the `rec` construct in Scribble protocols. The implementation we generate relies on the fact that all the channels that a role needs for communication will be created before the protocol starts and stored in the `roleChannels` and `inviteChannels` structs, and that a role will use every channel for a single message exchange within the protocol. This becomes an issue when trying to implement `rec`, because it is not possible to know how many channels will be needed to execute a protocol, as it could potentially unfold infinitely.

6.4.1 Choice and Recursion

Our initial approach for generating the implementation of recursion permitted reusing channels between different recursion unfoldings, but this lead to race conditions when recursion was combined with `choice`. We showcase a simple example where this issue would arise in Figure 6.5. This protocol models a loop where a sender keeps sending values to a receiver until it decides to stop, sending a message to end the communication.

Let us assume that the sender sends one message before the termination message, and that the same two channels are used for the `choice` both times. Because channels are asynchronous, after sending the first message the sender would be able to continue with the second `choice` without waiting for the receiver to receive it. It is not guaranteed that the first message will have been received by the time the sender sends the termination message. This means, that it is possible for the receiver to be able to choose to receive the termination message before the first message which was sent, because the channels used in both choices are the same and a `select` statement will choose one of the available options at random.

In this example, the issue could be resolved by making the channels synchronous, which would force the sender to wait for the receiver to receive the first message before sending the termination message, but this solution would not be enough for our implementation. We have highlighted the problem with a protocol which carries out labelled message exchanges, but the same issue would arise if the choice involved calling two different protocols. If the role making the `choice` did not participate in the call, it could send out the invitations one after another, leading to the same situation. Moreover, in the case of invitations, we require that they are sent over **asynchronous** channels because a role needs to be able to send an invitation to itself.

For this reason, making the channels synchronous is not enough to solve our issue, and because it is not possible to create a struct with an infinite number of channel fields, we have concluded that it is impossible to directly encode recursion with our current design.

```
1
2   global protocol SendLoop(role Sender, role Recv) {
3       rec LOOP {
4           choice at Sender {
5               Msg() from Sender to Recv;
6               continue LOOP;
7           } or {
8               End() from Sender to Recv;
9           }
10      }
11  }
```

Figure 6.5: Protocol combining `rec` and `choice`

6.4.2 Recursion as Nested Protocols

Even though we cannot directly generate code for the `rec` construct, it is still possible to achieve almost the same behaviour by modifying the protocol definition.

The main problem with our previous approach was that we were trying to create the channels for all the recursion unfoldings at the beginning, which was not possible. Instead, if we dynamically create a new set of channels for each unfolding of the protocols and share them across the roles which participate in the body of the recursion we could achieve the same result.

This is exactly what we are already doing for setting up protocol calls to nested protocols, where we create the channels each role will use for communicating with the other participants. Therefore, by extracting the body of a `rec` construct into a new nested protocol and replacing the occurrences of recursion variables in the protocol's specification with calls to the new nested protocol we can achieve almost the same behaviour as before. The main difference is that this approach introduces a point of synchronization for all the roles at the beginning of each execution of the recursion which the `rec` construct does not have, but this should not affect the expressiveness of our implementation.

Following this approach, we had to introduce an extra step in the pipeline before the projection, where we recursively traverse the protocols defined in the Scribble module and create new nested protocols for each recursion block we encounter. This additional step can be seen in Figure 1.1.

The signature of each of the new protocols we introduce must include only the roles from the original protocol which participate in the body of the recursion block, and their body is the same as the body of the recursion block, where every instance of `continue` *t* is replaced by a call to the new nested protocol.

6.5 End

The end local type signifies that the protocol has finished, and the code we generate shares the same semantics. To provide the user with an opportunity to perform whatever cleanup on the role's state, we first call the *Done* callback. As we described in Section 5.7, in the case of dynamic participants, this callback will not return anything, whereas for non-dynamic participants it will return a user-generated result for the role's execution of the protocol. After calling the callback, we return from the function, returning the role's result for non-dynamic participants. This process can be seen in Figure 6.6.

$$
[\![\texttt{end}]\!] = \begin{cases} \texttt{env.}\mathit{Done}\texttt{()} & \text{if } \mathit{role} \in \texttt{dynamic\_participants}(\mathit{protocol}) \\ \texttt{return} & \\ & \\ \texttt{return env.}\mathit{Done}\texttt{()} & \text{otherwise} \end{cases}
$$

Figure 6.6: Code generation scheme for `end` inside protocol *role@protocol*

Chapter 7

Evaluation

The main objective of our project is to develop the first practical implementation of the multiparty session types (MPST) theory of nested protocols. Our work aims to increase the expressiveness of Scribble[29], an MPST-based framework, to be able to model a large subset of real-world message passing APIs which could not be specified with the current theory. We will present different approaches which can be used to express common distributed computation patterns in our extended framework, and evaluate the strengths and limitations of our work through case studies which are implemented using these patterns.

7.1 Distributed Computation Patterns

The major limitation for existing MPST-based frameworks is that they cannot specify protocols where the number of participants is not known at the beginning of the session[9]. Many communication protocols used in practice do not have this information available statically. For instance, in a routing protocol, when a client wants to send a message to a different host over the network it won't necessarily know how many intermediate hops the message will have to go through before reaching the intended receiver. These kinds of protocols cannot be expressed by existing MPST frameworks, but nested protocols are able to do it, as new participants can be dynamically introduced through nested protocol calls.

To demonstrate the increased expressiveness provided by nested protocols, we define three common communication patterns which are used in distributed computation using nested protocols.

7.1.1 Ring Protocol

We show how nested protocols can be used to define a dynamic ring protocol, where the number of participants of the ring can grow as the protocol executes. In this setting, participants can only communicate with participants which are 'adjacent' to them in the ring. A role initiates the communication by sending a message to the

next participant, which gets forwarded all around the ring until it comes back to the first role.

```
1
2   nested protocol Forward(role S, role E; new role RingNode) {
3       msg(msg:string) from S to RingNode;
4       choice at RingNode {
5           RingNode calls Forward(RingNode, E);
6       } or {
7           msg(msg:string) from RingNode to E;
8       }
9   }
10
11  global protocol Ring(role Start, role End) {
12      choice at Start {
13          Start calls Forward(Start, End);
14          msg(msg:string) from End to Start;
15      } or {
16          msg(msg:string) from Start to End;
17          msg(msg:string) from End to Start;
18      }
19  }
```

Figure 7.1: Dynamic Ring Protocol

This behaviour can be encoded with nested protocols by using two protocols: `Ring` and `Forward`, as shown in Figure 7.1. Protocol `Ring` introduces the first and last roles in the ring, where role `Start` will send the first message around the ring, and role `End` will forward the message back to `Start`. If the ring only has two participants, they can just send the message to each other, but for larger rings, new participants must be introduced.

Protocol `Forward` defines the behaviour for increasing the ring size, which involves forwarding a message from a start role `S` to a new intermediate role `RingNode`. `RingNode` can then decide whether to close the ring by forwarding the message back to role `E`, which represents the last node in the ring, or to introduce a new intermediate node by calling the `Forward` protocol again. Once the last role receives the message, all that is left is for the `End` role to forward it back to role `Start` in the `Ring` protocol.

7.1.2 Pipeline Protocol

Another common pattern used to carry out tasks in a distributed system is a pipeline, which is essentially a ring with the last link missing. A task is forwarded through a

sequence of roles, each connected only to the next role in the pipeline, ending with the last role receiving the message/result.

```
1
2   nested protocol Forward(role S, role E; new role Node) {
3       msg(msg:string) from S to Node;
4       choice at Node {
5           Node calls Forward(Node, E);
6       } or {
7           msg(msg:string) from Node to E;
8       }
9   }
10
11  global protocol Pipeline(role Start, role End) {
12      choice at Start {
13          Start calls Forward(Start, End);
14      } or {
15          msg(msg:string) from Start to End;
16      }
17  }
```

Figure 7.2: Dynamic Pipeline Protocol

Our approach for defining this protocol, shown in Figure 7.2, is the same as the one we used to define a dynamic ring in Section 7.1.1. The protocol for forwarding the message to a new intermediate node remains the same as before, but we remove the last interaction to send the message back to the initial role in the `Pipeline` protocol.

7.1.3 Fork-Join Protocol

Distributed computation tasks often adopt a divide and conquer approach, where a large task is divided into smaller ones which are executed in parallel before aggregating all of the results from the subtasks together. This approach can be encoded in the fork-join model, which involves two stages: the **fork** and the **join** phase. During the fork phase, the main thread assigns each subtask to a different thread of execution without waiting for the result, and in the join phase the main thread waits to receive the results of all the subtasks which were spawned in the initial phase.

Nested protocols can be used to model this approach, even when the number of tasks is only known at runtime. We define two protocols in Figure 7.3 to do this. Both protocols involve two roles, one which is in charge of assigning the tasks, and a second role which carries out the task and returns the result.

In the entry-point protocol, `ForkJoin`, role `Master` can decide whether the subtask can be carried out by role `Worker` alone or if it needs to be split further amongst

```
1
2   nested protocol Fork(role M; new role W) {
3       choice at Master {
4           Task() from M to W;
5           M calls Fork(M);
6           Result() from W to M;
7       } or {
8           End() from M to W;
9       }
10  }
11
12  global protocol ForkJoin(role Master, role Worker) {
13      choice at Master {
14          Task() from Master to Worker;
15          Master calls Fork(Master);
16          Result() from Worker to Master;
17      } or {
18          SingleTask() from Master to Worker;
19          Result() from Worker to Master;
20      }
21  }
```

Figure 7.3: Dynamic Fork-Join Protocol

other participants. `Master` can delegate another subtask to a new role by calling the `Fork` protocol, where the master role `M` can choose to assign a new task to the role or send it a message to indicate that the fork stage of the protocol has finished. Once a call to the `Fork` protocol has finished, the master role can receive the result computed by the worker role.

The key insight for implementing the Fork-Join model is that nested protocols produce a stack of states every time a protocol call is made, much like the function call stack of a computer program. After the protocol call finishes, the roles resume their execution from the state they were in before the call. Our protocol specification takes advantage of this to achieve the execution of each subtask in parallel. After assigning a task to a worker, the master proceeds to assign the next one by making a recursive call to the `Fork` protocol without waiting for a result. This enables the worker role to compute its result while the remaining subtasks are being assigned. Once all the tasks are assigned, the results are aggregated in reverse order by traversing the protocol call stack until a single final result is produced.

Furthermore, this approach can be extended to define protocols which implement Fork-Joins with various levels of recursion. In such a case, the subtasks assigned to the workers may still be too complex for them to produce a result alone, so the workers can enact the `Master` role in a new `Fork` protocol call in order to further

split the subtask amongst other participants. This will enable them to aggregate the results of the subtasks before returning the result to the master role which assigned the task to them in the first place.

We have therefore described how we can model a parallel divide and conquer paradigm using nested protocols, which can be used to define protocols that carry out arbitrarily large tasks by breaking them down into smaller independent jobs that are carried out in parallel.

7.2 Case Studies

Using the patterns that we have described, we will demonstrate how nested protocols can be used to define protocols which could not be expressed with the previous theory.

7.2.1 Fibonacci

Calculating the nth Fibonacci number is a problem where the computation of the each successive number is the result of adding the previous two numbers in the sequence. This procedure could already be modeled with the existing theory[9], but we use it to showcase how we can apply the **ring** protocol design to practical use case.

```
1
2   nested protocol Fib(role Res, role F1, role F2; new role F3) {
3       Fib1(ubound:int, idx:int, val:int) from F1 to F3;
4       Fib2(idx:int, val:int) from F2 to F3;
5       choice at F3 {
6           F3 calls Fib(Res, F2, F3);
7       } or {
8           Result(fib:int) from F3 to Res;
9           End() from F3 to F2;
10      }
11  }
12
13  global protocol Fibonacci(role Start, role F1, role F2) {
14      StartFib1(n:int, val:int) from Start to F1;
15      StartFib2(n:int, val:int) from Start to F2;
16      Start calls Fib(Start, F1, F2);
17  }
```

Figure 7.4: Nth Fibonacci Number Protocol

The protocol definition in Figure 7.4 shows our proposed specification for the Fibonacci protocol. Our approach follows a variant of the ring design where each role

can communicate with the two following roles in the ring. When protocol `Fibonacci` starts, role `Start` sends the first two numbers in the sequence to `F1` and `F2` before making the first call to the `Fib` protocol. This protocol introduces a new participant which will calculate the next Fibonacci number after receiving the previous numbers from roles `F1` and `F2`. It will then decide whether to send it back as the final result to role `Res` or to make a recursive protocol call to calculate the next number in the sequence. In our implementation, we send the indices of the Fibonacci numbers as well as their values in our messages, and we also send the index of the Fibonacci number we want to compute so that we can identify when the protocol must terminate.

The formulation of the Fibonacci protocol we have presented can be easily modified in order to express the calculation of the infinite Fibonacci sequence, not just the nth term, something which the previous theory could not express. In fact, doing so actually simplifies the protocol, as we no longer have to keep track of the upper bound where we want to stop or how many numbers we have calculated. All that is required is to replace the `choice` in `Fib` with a `Result` message from `F3` to `Res` followed by a recursive call to `Fib`.

7.2.2 Fannkuch-redux

The Fannkuch-redux protocol was extracted from an implementation of the Fannkuch-redux benchmark presented in [15]. The task consists on finding out the maximum number of flips that need to be carried out on any permutation of the numbers from 1 to n. Each flip reverses the order of the first k numbers in the permutation, where k is the first number in the permutation. When the number 1 arrives at the first position, the process finishes, as any further flipping operations will not cause any changes.

This is a computationally-expensive task, as it requires calculating the number of flips for every one of the $n!$ permutations of length n. Moreover, the conjecture is that the maximum count is approximated by $nlog(n)$, which means that each individual task by itself is not trivial.

The implementation we have adapted carries out the computation by dividing all the different permutations into different groups, and the number of flips needed for the permutations in each group are calculated by a different goroutine, which sends the maximum number of flips found back to the first role, which aggregates the results. The main thread only assigns the first task to a worker, and then the worker will decide whether it needs to split any of the remaining permutations with a different role or not.

We have implemented this behaviour in Figure 7.5. This protocol specification follows the same principle as a Fork-Join protocol, where a single role triggers the start of the computation and aggregates the results produced by all the roles, but in this

```
1  nested protocol FannkuchRecursive(role Source, role Worker; new
       role NewWorker) {
2      Task(IdxMin:int, Chunksz:int, Fact:[]int, N:int) from Worker
           to NewWorker;
3      choice at NewWorker {
4          NewWorker calls FannkuchRecursive(Source, NewWorker);
5          Result(MaxFlips:int, Checksum:int) from NewWorker to Source;
6      } or {
7          Result(MaxFlips:int, Checksum:int) from NewWorker to Source;
8      }
9  }
10
11 global protocol Fannkuch(role Main, role Worker) {
12     Task(IdxMin:int, Chunksz:int, Fact:[]int, N:int) from Main to
           Worker;
13     choice at Worker {
14         Worker calls FannkuchRecursive(Main, Worker);
15         Result(MaxFlips:int, Checksum:int) from Worker to Main;
16     } or {
17         Result(MaxFlips:int, Checksum:int) from Worker to Main;
18     }
19 }
```

Figure 7.5: Fannkuch-redux Protocol

case the first role only assigns a task to **one** worker, and it is the workers themselves who introduce new participants into the protocol and assign them their tasks.

It would be possible produce a specification for this program without using nested protocols as long as you first calculate how many participants will be created in the process. Even so, by using nested protocols we can define this behaviour in a less restrictive manner which is closer to the program's original implementation.

7.2.3 Bounded Prime Sieve

The aim of a bounded prime sieve program is to produce all the primes from 2 until a specified upper bound, which can be done by incrementally filtering out all the numbers which are multiples of the primes you have found already. If a number remains after carrying out the sieve with all the numbers less than itself, then that number is a prime. We can express this behaviour using nested protocols by using a similar approach to the one used in Section 7.2.2 to implement the Fannkuch-redux protocol.

The implementation of the bounded prime sieve protocol is split across two proto-

```
1  nested protocol Sieve(role M, role W1; new role W2) {
2      nested protocol SendNums(role S, role R) {
3          rec SEND {
4              choice at S {
5                  Num(n:int) from S to R;
6                  continue SEND;
7              } or {
8                  End() from S to R;
9              }
10         }
11     }
12
13     FilterPrime(int) from W1 to W2;
14     W1 calls SendNums(W1, W2);
15
16     choice at W2 {
17         Prime(n:int) from W2 to M;
18         W2 calls Sieve(M, W2);
19     } or {
20         Finish() from W2 to M;
21     }
22 }
23
24 global protocol PrimeSieve(role Master, role Worker) {
25     FirstPrime(prime:int) from Master to Worker;
26     UBound(n:int) from Master to Worker;
27     choice at Worker {
28         Prime(n:int) from Worker to Master;
29         Worker calls Sieve(Master, Worker);
30     } or {
31         Finish() from Worker to Master;
32     }
33 }
```

Figure 7.6: Bounded Prime Sieve Protocol

cols, as shown in Figure 7.6. The entry-point protocol involves only two roles: `Main` and `Worker`. `Main` is the role which starts the sieve, sending the first prime, 2, and the upper bound of the sieve to `Worker`. With this information, `Worker` can filter out the non-prime numbers between 2 and n, and either communicate to `Main` that there are no numbers left or send the new prime to `Main` and start the sieve process with the next prime and the remaining numbers.

Setting up a successive sieve is always done by calling the `Sieve` protocol, which involves three roles: `M`, `W1` and a new participant, `W2`. `M`'s only purpose is to receive the

primes generated in the sieve, while `W1` sends the new participant the information needed to perform the next sieve: the prime candidates which have not been filtered out yet and the prime number to use during the sieve. These possible primes are sent through the `SendNums` protocol, where `Sender` repeatedly sends messages to a receiver, `Recv`, until it decides it will send no more values. If the sieve discovers a new prime, `W2` will send it to `M` and start another sieve, otherwise the sieve is finished.

It would not be possible to express a bounded prime sieve without nested protocols, as that would require knowing how many primes are in the range that you want to calculate, and that can only be found by carrying out the sieve.

7.3 Performance

We evaluate the run-time performance of our framework using the case studies we have described. We measured the execution times on a machine with an Intel i7-6700 processor and 16GB RAM, running Debian 10 and Go version go1.13.5.

We defined our own functions for benchmarking our protocol implementations. We compared the execution times of the implementations generated for the three case studies we have described against a hand-written implementation without nested protocols to evaluate the overheads introduced by our implementation. Each of the case studies was run for one thousand iterations before taking the mean execution time. We ran the benchmarks using different parameters which determined which Fibonacci number to calculate, which value to pass in to the Fannkuch-redux protocol and the upper bound for the prime sieve.

We measure the execution time in microseconds, and we compute the ratio $\frac{t_{\text{go}}}{t_{api}}$ between the execution time of the hand-written and our generated implementations. We show plots of our results in Figure 7.7. Each graph shows the relative ratio between the implementations against the baseline of 1, which represents the ideal behaviour where the generated code is as fast as the hand-written implementation.

The graphs show that our generated implementation is significantly slower than the one which was handwritten for the Fibonacci and Prime sieve protocols. The Fibonacci implementation is about five times slower, whereas for the prime sieve protocol the performance becomes twenty times slower. On the other hand, even though the performance of the generated Fannkuch implementation is initially much slower, the difference becomes negligible as the parameter value increases.

These results clearly indicate that our implementation of nested protocol calls incurs a high performance penalty for communication-oriented protocols, where the roles spend most of their time communicating with each other and participating in other nested protocols. Our code generation approach is not optimised for this kind of tasks, as we only focused on ensuring the correctness of the implementation. Our

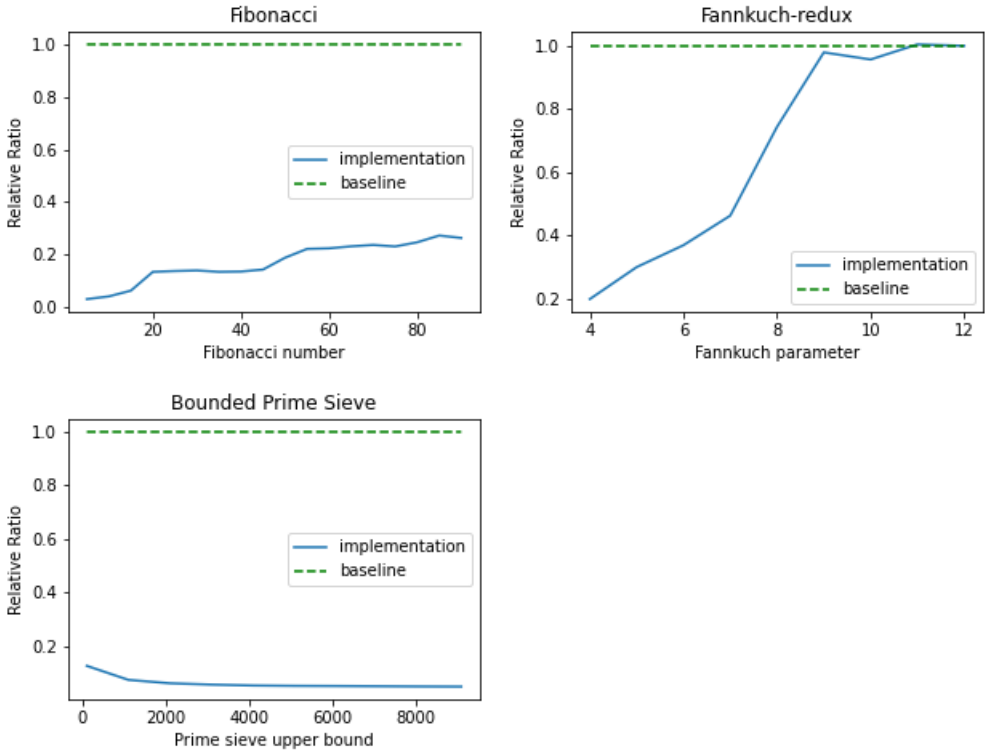

Figure 7.7: Performance comparison: **Scribble-Go** vs. **Go base case** implementations

use of channels is highly inefficient because we only use each channel in one interaction, which will cause higher memory allocation overheads, especially when a lot of nested protocol calls are made.

In our hand-written implementation for the prime sieve protocol, the logic for sending the prime candidates to the next role reused the channels used for sending the numbers, whereas in our generated code every number would be sent over a different channel in a different protocol call. Therefore, to send a single prime candidate we would first have to allocate the channels for the protocol call and send them as invitations, which causes massive allocation and communication overheads, explaining the difference in performance.

The performance drop in the Fibonacci protocol is not as pronounced as the one in the prime sieve protocol, as there are no recursive protocols which cause a lot of additional protocol calls to be made. We still expect a worse performance due to the additional memory allocations for channels and structs, the increased number of message exchanges for invitations and other factors in our code generation scheme, such as passing structs by value instead of by reference to functions, so this performance difference is within reason for a protocol which is mainly communication-oriented.

By contrast, the implementation of the different roles in the Fannkuch-redux pro-

tocol is much more computationally intensive than the other two protocols, and its complexity grows exponentially as the parameter increases. After a certain point, the overheads caused by the implementation of the role itself become more significant than the communication overheads introduced by the protocol implementation. This could explain why for parameter values ≥ 9 the performance difference almost disappears.

7.4 Expressiveness and Limitations

We have demonstrated the increased expressiveness of our extension to the Scribble framework by describing how nested protocols can be used to specify a wide range of protocols which employ different communication patterns. Nevertheless, there are still limitations to which message-passing APIs can be expressed using nested protocols.

Even though with nested protocols it becomes possible to dynamically introduce new participants into a session, the number of a participants in a session is **finite** and **cannot change**. Nested protocols can therefore express processes which can have an arbitrary sequence of steps as long as every step of the computation only requires a fixed number of participants.

An example which nested protocols cannot express is the unbounded prime sieve. This process involves building an infinite chain of participants which incrementally sieve the natural numbers to find the prime numbers much like the bounded prime sieve we described in Section 7.2.3. However, here the process does not stop at an upper bound, it continues forever. A sieve chain between roles which grows infinitely as new primes are discovered cannot be expressed with the current theory of nested protocols for the reasons we have outlined above. This shows that even though nested protocols can be used to describe a large number of message-passing APIs, there are still behaviours which cannot be expressed.

When comparing the expressiveness of our extended framework against a previous implementation[9], we can see that our extension allows us to specify many more protocols. In Table 7.1 we enumerate the protocols that we have already discussed, and whether they can be represented using both frameworks. Both frameworks are unable to express the infinite pipeline of roles in the unbounded prime sieve protocol, but the previous framework is only able to specify two of the remaining protocols which our extended framework can express. The nested protocols theory allows us to describe processes where the number of participants is not known at the start of a session, which overcomes a major obstacle towards describing real-world message-passing APIs that the previous framework faced.

Even though our framework can generate correct implementations for a large number of protocols across many practical applications, our analysis highlights the fact that the performance of our implementations would not be suitable for many of

Protocol	Nested Protocols	Previous theory[9]
`Dynamic Ring` (§7.1.1)	✓	✗
`Dynamic Pipeline` (§7.1.2)	✓	✗
`Dynamic Fork-Join` (§7.1.3)	✓	✗
`Recursive Fork-Join` (§7.1.3)	✓	✗
`Fibonacci` [9](§7.2.1)	✓	✓
`Unbounded Fibonacci sequence` (§7.2.1)	✓	✗
`Fannkuch-redux` [15](§7.2.2)	✓	✓
`Bounded Prime Sieve` (§7.2.3)	✓	✗
`Unbounded Prime Sieve` [21](§7.2.3)	✗	✗

Table 7.1: Comparison of the protocols which can be expressed with/without nested protocols

these use cases. The way we have implemented nested protocol calls, which are the essence of the theory we are implementing, is not optimised, causing protocols that carry out a lot of nested protocol calls to suffer a large performance penalty. Nevertheless, this is the first version we have implemented, so there are lot of areas which can be refined in order to reduce the overhead caused by nested protocol calls.

In the future, the framework could potentially be used in many practical applications, but in its current state the main tasks where it could be useful are tasks which prioritize strong correctness guarantees over efficiency, and in cases where the tasks carried out by the protocols are more computationally expensive. In such cases, the overhead introduced by our design will be less significant compared to the overall execution time of the protocol implementation.

Chapter 8

Conclusion

8.1 Contributions

The aim of our project was to create a framework to statically verify the specification of nested protocols. We have extended the Scribble protocol description language[29] with the multiparty session types theory (MPST) of nested protocols presented in [12]. We have also incorporated the full merge operator into the definition of projection presented in the theory to increase the number of protocols which can be expressed. The Scribble toolchain can now check that a protocol specification using nested protocols provided by the user is valid and obtain the projections for the different roles participating in the protocols.

We develop the first practical application of the nested protocols theory. In Chapter 6, we define a scheme to translate the specification of a role's behaviour, given by the projection Scribble generates, into an API implementation for the role in Go. The APIs generated with our approach ensure that a role will never perform any illegal I/O actions. Our implementation targets Go, one of the most well-established programming languages for developing distributed systems in industry[2], and we leverage Go's concurrency primitives for local concurrency: *goroutines* and *channels*, which had not been treated before in any MPST frameworks for Go.

Our project has focused on implementing the core parts of the nested protocols theory presented in [12]: defining nested protocols within other protocols, calling nested protocols within a parent protocol and making it possible for protocols to have dynamic participants that are only brought in when a protocol is called. These extensions enable us to model many real-world message-passing APIs where the number of participants is not known at the beginning of the session, which is something that could not be done with previous MPST-based frameworks.

We have come up with a possible approach to solve the problem of returning a result from a protocol, as the original theory does not define a way to bringing out the results of the computation carried out during a protocol out of the session. This feature is vital for the usefulness of the implementation, because otherwise the results

of the computation would be lost once a protocol had finished executing.

In Chapter 7 we discuss how our extensions have increased the expressiveness of the Scribble language, demonstrating different patterns which can be used to express concurrent computation using nested protocols and showcasing how these patterns can be applied to implement different examples. We also compare the performance of the implementations we generate to ones which have been implemented manually in order to show which kinds of protocols are most suited to be implemented with our framework.

Our extensions to the Scribble framework have been implemented on top of an existing implementation of Scribble: `nuscr`[1]. At the time this has been written, our contributions have not been merged into the official implementation, but they are publicly available in a fork of the repository[2] under the same GNU General Public License as the original repository.

Overall, our project shows the potential of using nested protocols to describe concurrent computation, and our approach lays a solid foundation for further development in both the theory and practical applications of nested protocols.

8.2 Future Work

Our project has opened up many different interesting avenues for further development in the theory and implementation of nested protocols. We mention a few of the ideas which we would like to highlight:

- **Proving the correctness** of our implementation: From a theoretical standpoint, we have not offered a formal proof of the correctness of our implementation. We have just given justifications as to why we believe our approach is correct. We would like to formalise our work to show the correspondence between our approach and the MPST theory, in the same way that the correctness of Scribble is formalised in [25].
- **Guaranteeing deadlock freedom**: Like the original nested protocol theory presented in [12], our implementation cannot guarantee *deadlock freedom* and the *termination* of a protocol which calls other nested protocols. The reason for this is that when a protocol call is made, it is not always possible to identify whether the protocol call will terminate, or if some/all the participants in the call will be stuck in a recursive loop. Therefore, any interactions which happen after the protocol call may be stuck if one or more of the roles involved were participants in the protocol call which are stuck in an infinite loop. A great extension to this project could involve developing both the theory and implementation to explore what stronger termination properties can proven for nested protocols.

[1] https://github.com/nuscr/nuscr
[2] https://github.com/becharrens/nuscr

- Implementing nested protocols as **Communication Finite State Machines (CFSM)**: As we described in Section 2.1.5, the standard approach for generating role APIs in Scribble is to first create a CFSM which is then used to generate the implementation. However, the current CFSM theory is not able to express nested protocol calls, which is why we generate code directly from the protocol specification for a role that is produced by the Scribble toolchain. It would be interesting to extend the theory of CFSMs in order to overcome this limitation and find out how a CFSM-based implementation of nested protocols compares with our current approach.

- Implementing nested protocols in a **distributed setting**: The current implementation of nested protocols is only designed for a local system, where all the computation is carried out in goroutines executing on a single machine. Using this approach, we have been able to take advantage of Go's concurrency primitives in our initial design, but moving forward, in order for nested protocols to be used in real-world applications they must be able to run in a distributed system. As we already have a working local version, moving to a distributed setting should be less of a challenge, although certain parts of the design would have to change in order to make this transition.

- **Reduce overheads of setting up protocol calls**: As we discussed in Section 7.3, despite the fact that our Scribble framework extension can describe many more practical message-passing APIs than previous implementations, the low performance of our generated protocol implementations limits the possible applications where they can be used. In order for this framework to be applied in a wider range of practical settings we will need to modify our design to reduce the overheads associated with protocol calls.

- **Refine our implementation**: We would also like to improve several other parts of our approach which we did not have the opportunity to work on due to the time constraints. For instance, as we described in Section 6.4, our current design has some limitations for defining protocols with the recursion construct. This issue does not have a great impact on the expressiveness of our implementation, and should be straightforward to fix given the time. We also make some assumptions about the protocol declarations provided by the user, like the payload types of the messages being valid Go types, which need to be validated by the framework.

- Implement **higher-order protocols**: In the nested protocols theory[12], protocol definitions can receive different values as parameters during a protocol call. These parameters can even be other protocols as long as the signature of the protocol argument matches that of the parameter. These protocol parameters can then be used in protocol calls during the execution of the protocol. The definition of higher-order protocols is not supported in our current implementation, but it would be a very interesting extension to pursue in the future.

Bibliography

[1] Documentation - The Go Programming Language. https://golang.org/doc/. Accessed on 2020-01-23.

[2] GoUsers - Companies currently using Go throughout the world. https://github.com/golang/go/wiki/GoUsers. Accessed on 2020-06-13.

[3] Jlint - Find Bugs in Java Programs. http://jlint.sourceforge.net/. Accessed on 2020-01-23.

[4] Tour of Go - Select. https://tour.golang.org/concurrency/5. Accessed on 2020-06-13.

[5] FindBugs - Find Bugs in Java Programs. http://findbugs.sourceforge.net/, 2015. Accessed on 2020-01-23.

[6] ADVE, S. V., AND GHARACHORLOO, K. Shared Memory Consistency Models: A Tutorial. Tech. rep., 1995.

[7] BOCCHI, L., HONDA, K., TUOSTO, E., AND YOSHIDA, N. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR 2010 - Concurrency Theory* (Berlin, Heidelberg, 2010), P. Gastin and F. Laroussinie, Eds., Springer Berlin Heidelberg, pp. 162–176.

[8] BOUDOL, G. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992.

[9] CASTRO, D., HU, R., JONGMANS, S.-S., NG, N., AND YOSHIDA, N. Distributed Programming Using Role Parametric Session Types in Go. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages* (2019), vol. 3, ACM, pp. 29:1—-29:30.

[10] CHEN, T.-C., DEZANI-CIANCAGLINI, M., AND YOSHIDA, N. On the Preciseness of Subtyping in Session Types. In *16th International Symposium on Principles and Practice of Declarative Programming* (2014), ACM, pp. 135–146.

[11] COPPO, M., DEZANI-CIANCAGLINI, M., PADOVANI, L., AND YOSHIDA, N. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming* (2015), vol. 9104 of *LNCS*, Springer, pp. 146–178.

[12] DEMANGEON, R., AND HONDA, K. Nested Protocols in Session Types. In *CONCUR 2012 – Concurrency Theory* (Berlin, Heidelberg, 2012), M. Koutny and I. Ulidowski, Eds., Springer Berlin Heidelberg, pp. 272–286.

[13] DENIÉLOU, P.-M., AND YOSHIDA, N. Dynamic multirole session types. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2011), ACM, pp. 435–446.

[14] FOWLER, S. An Erlang Implementation of Multiparty Session Actors. In *Proceedings 9th Interaction and Concurrency Experience, {ICE} 2016, Heraklion, Greece, 8-9 June 2016* (2016), pp. 36–50.

[15] GUOY, I. Computer Language Benchmarks Game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`, 2017. Accessed on 2020/01/23.

[16] HONDA, K., VASCONCELOS, V. T., AND KUBO, M. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems* (Berlin, Heidelberg, 1998), C. Hankin, Ed., Springer Berlin Heidelberg, pp. 122–138.

[17] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2008), POPL '08, Association for Computing Machinery, pp. 273–284.

[18] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty Asynchronous Session Types. *J. ACM 63*, 1 (mar 2016).

[19] HU, R., AND YOSHIDA, N. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering* (2016), vol. 9633 of *LNCS*, Springer, pp. 401–418.

[20] 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.

[21] LANGE, J., NG, N., TONINHO, B., AND YOSHIDA, N. Fencing off Go: Liveness and Safety for Channel-based Programming. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages* (2017), ACM, pp. 748–761.

[22] MILNER, R. Functions as processes. In *Automata, Languages and Programming* (Berlin, Heidelberg, 1990), M. S. Paterson, Ed., Springer Berlin Heidelberg, pp. 167–180.

[23] MILNER, R. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[24] NEYKOVA, R. Session Types Go Dynamic or How to Verify Your Python Conversations. In *5th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software* (2013), vol. 137 of *EPTCS*, Open Publishing Association, pp. 95–102.

[25] NEYKOVA, R., AND YOSHIDA, N. Featherweight Scribble. *Models, Languages, and Tools for Concurrent and Distributed Programming 11665* (2019), 236–259.

[26] NG, N., COUTINHO, J. G. F., AND YOSHIDA, N. Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction* (2015), vol. 9031 of *LNCS*, Springer, pp. 212–232.

[27] NG, N., YOSHIDA, N., AND HONDA, K. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *Objects, Models, Components, Patterns* (Berlin, Heidelberg, 2012), C. A. Furia and S. Nanz, Eds., Springer Berlin Heidelberg, pp. 202–218.

[28] YOSHIDA, N., AND GHERI, L. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology* (Cham, 2020), D. V. Hung and M. D\textasciiacuteSouza, Eds., Springer International Publishing, pp. 73–93.

[29] YOSHIDA, N., HU, R., NEYKOVA, R., AND NG, N. The Scribble Protocol Language. In *8th International Symposium on Trustworthy Global Computing* (2013), vol. 8358 of *LNCS*, Springer, pp. 22–41.

[30] YOSHIDA, N., ZHOU, F., AND FERREIRA, F. C406 Concurrent Processes Course Materials, Imperial College London, 2019.