Implementing Type-Checking Communication Protocols in the Go Programming Language with Gobble

Aidan Butler (2281611B)

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

7th September, 2017

Abstract

The increasing importance of communication between the elements of concurrent and distributed systems presents software developers with the practical challenge of ensuring that different system elements communicate with one another in accordance with well-formed protocols which will prevent deadlocks from occurring. The Scribble protocol description language and its associated tools allow developers to define and test communication protocols in order to ensure their well-formedness using mathematical analysis techniques based on the π-calculus. The Go programming language has in-built primitives designed to facilitate communication within both concurrent and distributed systems based on the communication sequential process (CSP) calculus. The purpose of this paper is to introduce Gobble, a source-to-source compiler which converts well-formed Scribble protocols into Go code in which protocol adherence is ensured by the use of session types: formal, structured descriptions of a protocol which specify for each message sent a data type, a sequential position and a direction. The implementation makes use of coroutines and channels, two of Go's CSP-inspired features designed to facilitate concurrent computing. The user can decide whether to produce an implementation for use on a single system that defines communication across channels between concurrently running co-routines or one for use on a distributed system which defines communication across a TCP connection. They can also decide whether to use Gobble's automatically generated implementation of the protocol or to write their own using the Gobble API.

Acknowledgements

To Prof. Simon Gay, for setting me on the road to what's turned out to be an immensely pleasurable journey into the world of languages, compilers and communication structures and for helping me along the way so ably.

And above all to Marta, for a year of patience and unfailing support: Sei l'amore della mia vita.

Contents

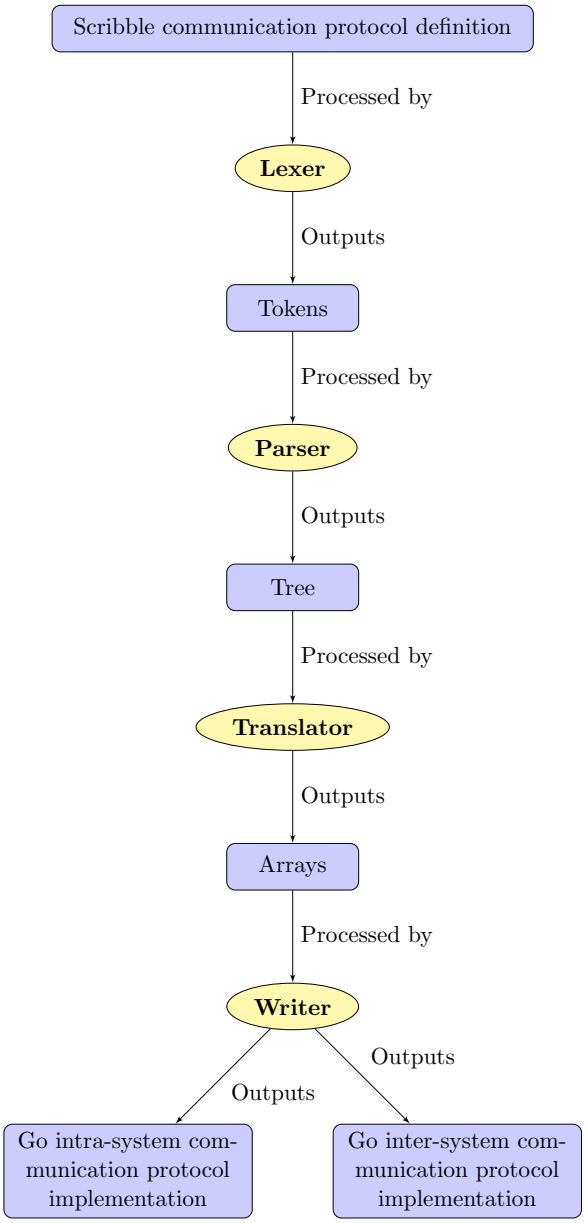


Figure 1: Flowchart representing the work of Gobble, a source-to-source complier written in Go which translates communication protocols defined in Scribble into executable Go code. At each stage of the compilation process the output of the preceding stage is treated as read-only, facilitating concurrent data processing.

Chapter 1

Project Motivation

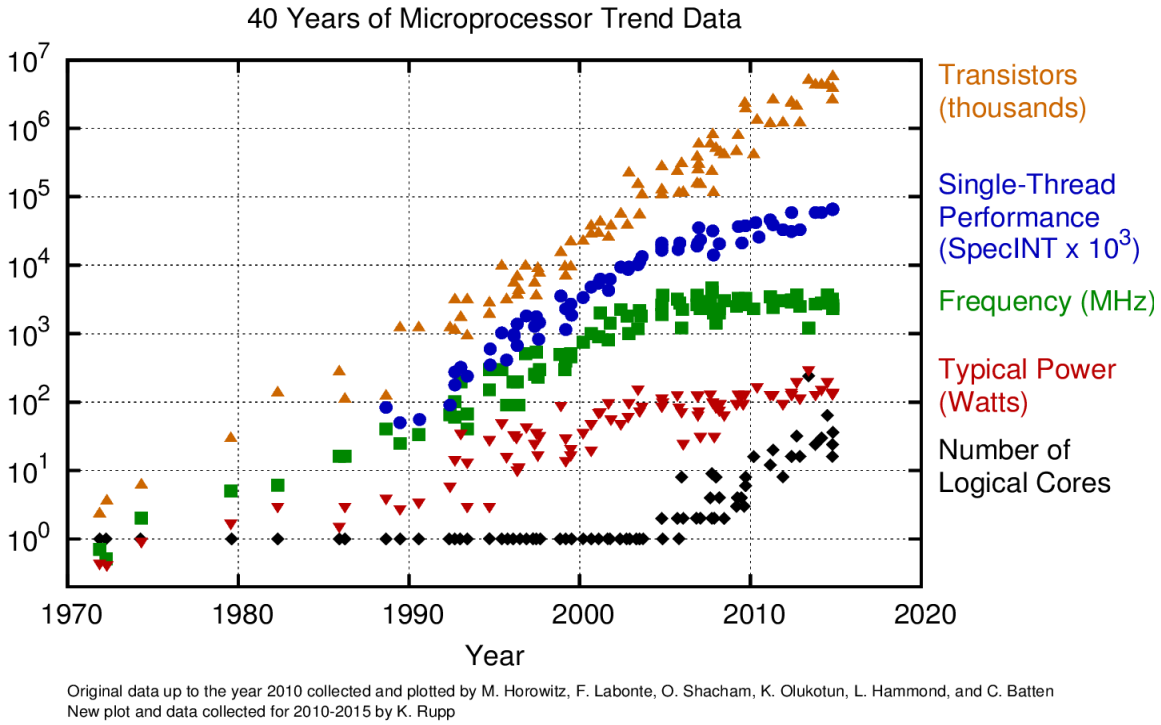
1.1 Historical Background

While early mainframe computers were orders of magnitude quicker and more accurate at carrying out calculations than human workers, their large capital cost, great bulk and relative scarcity meant that their use for external communication as part of distributed systems was not an original priority of those who worked with them. The fact that they were single processor machines initially lacking even basic operating systems that could handle multi-tasking meant that internal communication between concurrent process was not a priority either. Recognition of the importance of data structures to computing led the designers of early high-level languages to incorporate abstractions to deal with them into their designs, principally through the creation of type systems which could differentiate between data types such as booleans, integers and strings[8]. However, the fact that communication between and within computer systems was of secondary importance to those language designers meant that abstractions designed to deal with that issue were absent from early higher-level languages, such as C. Widely-adopted new programming languages have tended to build upon what has gone before, meaning that most widely-used modern languages, such as C++, Java and Python, are well equipped with native abstractions designed to deal with data structures but poorly equipped with ones designed to deal with communication structures.

This is problematic because communication is today of fundamental importance to the design of computing systems. The first reason for this is increasing external communication between the different elements that make up distributed systems, such as the ever-increasing number of devices connected to the Internet. The second reason is increasing internal communication due to the fact that while the number of processors within a typical computerised device is increasing rapidly, the speed which which those processors operate is increasing only slowly; meaning that efficient use of system resources increasingly requires the use of concurrent and parallel data processing techniques[4](see fig. 1.1).

One of the most significant problems faced in communication systems is the possibility of deadlocks occurring. A deadlock is a situation which is defined by four conditions being met amongst a group of two or more communicating parties. In their 1971 paper *System Deadlocks* Coffmann et al. describe what a system deadlock is and set out the four

Figure 1.1: Historical data showing that while Moore's law—the observation that the number of transistors in a dense integrated circuit doubles approximately every two years—continues to hold true, efficient use of computing power increasingly requires the use of parallel processing.[17]



conditions that have to be met in order for one to occur. Firstly, there must be mutual exclusion: the resources involved must be unshareable. Secondly, there must be hold-and-wait resource holding: communicating parties must be able to hold on to one resource while they request another. Thirdly, there must be a lack of pre-emption: resources can only be released voluntarily by the parties holding them. Fourthly, there must be circular waiting: two or more parties must be simultaneously waiting to access one another's resources before relinquishing those which they themselves hold[1]. Deadlocks are a major problem because they can lead to systemic paralysis.

1.2 Scribble + Go = Gobble

In order to design communication systems in such a way as to prevent deadlocks from occurring it is first necessary to be able to be able to formally describe those systems. In computer science such descriptions are known as process calculi. Both Scribble and Go, the Gobble compiler's respective input and output languages, are designed to make practical use of insights gained from process calculi.

Scribble[18] is a language created to be used for the design of well-formed concurrent programming protocols. The language has its theoretical foundation in the π-calculus, a mathematical system designed for reasoning about commination systems and described by its

creators as "a calculus of communicating systems in which one can naturally express processes which have changing structure"[9].

Scribble was designed by π-calculus researchers working on the theoretical problem of how to describe communication systems in such a way that it would be possible to test them to ensure that no deadlocks would occur.[20] The language was designed as a concise way of sketching out how communication systems work, abstracted from any particular implementation. Since the language's creation, tools have been developed which make it possible to check Scribble protocols in order to ensure that they are free of deadlocks and to convert global protocol descriptions into numerous local descriptions that can be implemented for each communicating party[16, 15].

Go[14], despite its being designed by and for engineers primarily as a practical tool, also contains elements based on theoretical process calculus research. In the case of Go, the calculus in question is communicating sequential processes (CSP). CSP is based on an elaboration of the ideas published in Hoare's 1978 paper *Communicating Sequential Processes*. In that paper Hoare "suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method"[5]. He then goes on to describe a theoretical programming language in which parallel processes communicate with one another using input and output channels and goes on to show how such a language could be used to solve a range of concurrent programming problems. Hoare's original idea for a theoretical programming language was subsequently developed by him and others into CSP.

At the time that Hoare wrote his original paper parallel processing was still a primarily theoretical consideration. Mutli-processor computers were rare and those that did exist were typically designed to hide the fact from the user by using each processor for a different task. Despite this reality, Hoare indulges in a bit of science fiction writing at the outset of the paper, stating "developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor"[5]. Thirty years later, Hoare's hypothetical future had become Google's daily reality and his proposed system of concurrent processes communicating through channels had been integrated into a new programming language designed by the company's engineers to meet that reality's challenges.

By the early 21st century system designers at Google found that communication had become one of their principle computing challenges. Dealing with the massive volumes of data generated by Google's search business and the other services it provided required coordinated communication. Code had to be written to enable vast numbers of multi-processor servers to manage their internal parallel processes and talk to one another and to other computers around the world. A group of engineers working at the company—including Rob Pike and Ken Thompson, both of whom had been involved in the creation of the Unix operating system at Bell Labs—came to the conclusion that the programming languages that the company was using to manage those systems could be improved upon and began developing a new alternative, which was to become Go[13].

According to Pike, the main language being used to manage Google's systems up to that time was C++ and Go was equipped with a number of features designed to overcome C++'s perceived shortcomings while retaining its perceived advantages of speed and versatility.

Firstly, the language was to have a concise, simplified C-like syntax in order to reduce programme size and complexity and minimise the time required by developers to learn it. Secondly, the language was to have automatic garbage collection, in order to reduce the problem of memory leaks. Thirdly, the language was to have fast compile times in order to speed up the production cycle. Fourthly, the language was to be designed with in-built communication tools including a standard library equipped to deal with cross network communication and concurrency primitives based on CSP[13].

The philosophy underlying CSP and Go's implementation of concurrency primitives is "Don't communicate by sharing memory, share memory by communicating"[13], meaning that data should be shared between concurrent processes by means of thread-safe communication channels designed specifically for that purpose, rather than by locking and unlocking the different areas of memory in order to ensure that threads do not come into conflict with one another, the main approach used in C++.

Go's concurrency primitives consist of co-routines, known as "goroutines", and channels which are used to communicate between them. Goroutines are implemented as light-weight threads which can be launched simply by placing the keyword `go` before a function call. Channels are typed, thread-safe queues which are defined and generated by a declaration such as `myChannel := make(chan int)` for an integer channel. Goroutines and channels work together synchronously, meaning that communication parties linked by a channel will block whilst waiting for a value to be sent or received using the `<-` operator. For example, in a dialogue between goroutines 1 and 2 goroutine 1 will block on sending a value into a channel (e.g. `myChannel <- sentVal`) until it is received by goroutine 2 (e.g. `receivedVal := <- myChannel`). It is possible to mitigate this blocking by attaching an arbitrarily sized buffer to a channel, but the channel and goroutines will still block if and when that buffer is filled. If all of a process's goroutines block a deadlock occurs and the go runtime will bring the process to a halt (see Figure 1.2). The tendency of such systems to enter into deadlocks makes them an obvious target for implementation of the well-formed communication protocols produced and validated by the Scribble tools.

Gobble is a source-to-source complier which aims to achieve just this. As an input it takes validated Scribble protocols and as an output it produces functioning Go code. A full description of the genesis, aims and development of the Gobble project will form the content of the following chapter.

Figure 1.2: A Go `deadlock` message. Gobble is designed to help programmers avoid these.

```
fatal error: all goroutines are asleep - deadlock!
```

Figure 1.3: An editor's autocomplete functionality using Gobble-generated session types to list the options available at a given step in communication protocol. Gobble is designed to provide programmers with this.

```
Client_rec1_choice1.
                    func Send_Accept() (*Client_rec1_choice1_C2, error)
                    func Send_RejectAndLeave() error
                    func Send_TryAgain() (*Client_rec1_1, error)
```

Chapter 2

Project Aims

The principle aim of the Gobble project is to develop a practical implementation of session types by constructing a system for translating validated Scribble protocols into executable Go code.

2.1 Session Types

A session type is a computing abstraction designed for communication which takes the form of a structured description of a protocol and that specifies for each message sent its data type, its sequential position and the direction in which it is to be sent[8]. Session types are of interest to computer scientists insofar as they allow them to study computing as communication rather than computing as data processing, they are also of interest to software engineers insofar as they make it possible for them to produce deadlock-free communication code.

Scribble is a simple, concise language which is designed to describe communication protocols in order to generate session types. A Scribble protocol consists of a description of the sequential exchange of messages between a number of communicating parties, or `roles`. For example, the Scribble protocol in Listing 2.1 describes two interactions between `Client` and `Server`. First, `Client` sends `Server` a `PaymentInfo` message with a data-type `string`. Second, `Server` sends `Client` a `ConfirmPayment` message with a data-type `bool`.

```
1 global protocol MakePayment(role Client, role Server) {
2     PaymentInfo(string) from Client to Server;
3     ConfirmPayment(bool) from Server to Client;
4 }
```

Listing 2.1: A Scribble global protocol

Once a Scribble protocol has been defined it can be checked using Scribble tools developed based on the theory of multi-party session types[6] in order to ensure that no deadlocks can occur either between communicating pairs or between multiple interacting pairs[16, 15]. Once this has been done, local protocols, such as the one in Listing 2.2 are produced.

```
local protocol MakePayment at Client(role Client, role Server)
    {
    PaymentInfo(string) to Server;
    ConfirmPayment(bool) from Server;
}
```

Listing 2.2: A Scribble local protocol

From a software engineering point of view, the advantages of session types consist in knowing that software correctly implemented using session types will be deadlock-free and in the development time saved once the session types defining a particular protocol have been generated. From the same perspective, the disadvantages of session types lie firstly in the time cost of defining the session types for a given protocol and secondly in the costs in terms of processing power and memory use resulting from the additional code required to define session types. While the latter of these costs would appear to be unavoidable, the former can be vastly reduced by the development of software which will automatically translate a minimal communication protocol description into executable software that makes use of session types. Gobble has been developed in order to achieve this goal.

2.2 Previous Work in the Area

Gobble is inspired by and builds upon a project to translate Scribble protocols into a Java API realised by Raymond Hu and Nobuko Yoshida at Imperial College London, the source code for which is available online[15]. In the paper *Hybrid Session Verification through Endpoint API Generation*, in which they describe their work, Hu and Yoshida state that their aim was to produce "a new hybrid session verification methodology for applying session types directly to mainstream languages, based on generating protocol-specific endpoint APIs from multiparty session types."[7]. Their software's output is local implementations for each role in a multi-party communication session.

The API Hu and Yoshida developed represents session types using objects, Java's principal unit of data abstraction. For example, using their system the first step in the exchange between `Client` and `Server` described in Listing 2.1 and Listing 2.2 could be represented as a `Client_1` object. That object will have zero or more methods attached, each representing an option permitted by the protocol. For example, `Client_1` could have an attached `.SendPaymentInfo()` method. Any data items to be sent at that step in the protocol can be represented as parameters to the methods. For example, such a `Client_1.SendPaymentInfo()` method could take a `myPaymentInfo` string parameter. The return values for the resulting method call will be an object representing the next step in the protocol, if any any. For example, the return value for a call to `Client_1.SendPaymentInfo(myPaymentInfo)` might be `Client_2`.

The Gobble project was also influenced by related work by Dardha et al. on Mungo, a front-end typechecking tool for Java, and by StMungo a programme which automatically translates Scribble protocol descriptions into Mungo specifications, enabling static typechecking of the correctness of an implementation[8].

2.3 Requirements

The motivation for developing a new Scribble API and translation tool targetting another language when one already exists for Java is based on the fact that Java is not a language specifically designed for communication and is therefore lacking in-built primitives to deal with communication easily and efficiently; this applies in particular to tools for concurrency and mutli-threading in Java. The choice of Go as an alternative candidate was based on a number of considerations. Firstly, the fact that the language has in-built primitives—channels and goroutines—which were specifically designed for communication and fit well with the π-calculus and CSP ways of conceptualising communication. Secondly, the choice of Go was motivated by the fact that the language is similar enough to Java, most notably in its allowing one to programme in an Object Oriented style, to ensure that the Hu and Yoshida's approach to representing session types as non-data objects could be replicated. Thirdly, the choice of Go was motivated by the fact that the language's growing popularity[19] increases the chance that others will benefit from and choose to add to the project.

Further to that primary objective, a secondary aim was to develop a system for fully translating Scribble into executable code using the API. The principle motivation here was the automation of the routine and time-consuming task of manual communication protocol implementation, with the aim of making Gobble a more useful tool for programmers by allowing them to gain a level of abstraction from implementing communication protocols.

A third goal was to produce a handwritten parser for use in API generation and code translation, rather than resorting to the use of a parser-generator for the task. This desire was motivated partly by the belief that a handwritten parser could be more easily optimised and debugged than an automatically generated one, partly by a desire to reduce the tool's complexity and dependency load by ensuring that all of its code was written in a single language and partly by the inherent didactic value of learning about parsers by writing one. This approach was made possible by Scribble's relatively small size and simple structure.

A forth goal was to gain a good working knowledge of the Go programming language, one with which the author had never used until the start of the project.

2.4 Use cases

Restated as testable use cases and prioritised using the MoSCoW (Must have, Should have, Could have, Won't have) system, the project requirements can be stated in the form shown in Table 2.1:

The project's aims and requirements having been established, the following chapter will deal with the design and structure of the Gobble compiler.

Priority	**Use case**
Must have	User can automatically generate a Go API output from a Scribble input
Must have	User can automatically generate an API including Scribble's core flow of control structures: recursion and branching
Should have	User can automatically generate a translation of a Scribble protocol: a full, working implementation of an API
Should have	User with knowledge of no languages other than Scribble and Go can read and alter all project code
Should have	User can automatically generate an API of Scribble code using the full range of Version 0.3 of the Scribble language specification [18]
Could have	User can specify types using Scribble's payload type declaration system

Table 2.1: Gobble project use cases and their assigned levels of priority

Chapter 3

Gobble

3.1 The Anatomy of Gobble

As shown in Figure 1 at the beginning of this dissertation, the Gobble compiler consists of four elements: a lexer, a parser, a translator and a writer.

3.1.1 The Lexer

The purpose of the lexer is to read a local Scribble protocol from file and processes it into an array of lexical tokens. These tokens consist of strings which can be either runs of alphanumeric characters such as `protocol` or punctuation characters with specific meanings in Scribble, such as { or ;.

3.1.2 The Parser

Gobble classifies a Scribble protocol into two basic types of structure: "messages" (the axiomatic units of the protocol, consisting of a message `from` one communicating party `to` another) and "conversations" consisting of ordered sequences of such messages and/or other conversations divided into `par` blocks (indicating where multiple conversations may be executed concurrently), `choice` blocks (where multiple options may be selected by one of the conversation's participants) and `rec` blocks (for which looping behaviour is prescribed). For a global Scribble protocol containing all of these elements see Listing 3.1.

```
1 module TravelAgency;
2
3 global protocol BookJourney(role Client, role Aggregator, role
      BrutishAirways, role QueasyJet) {
4     GreetAggregator() from Client to Aggregator;
5     GreetClient() from Aggregator to Client;
6     rec MakeBooking { // Recursive/loop block
7         RequestItinerary(string, int) from Client to Agregator
              ;
```

```
8           par { // Concurrent block
9               CheckAvailabilityAndPrice1(string, int) from
                    Aggregator to BrutishAirways;
10              CofirmAvailabilityAndPrice1(bool, int) from
                    BrutishAirways to Aggregator;
11          } and {
12              CheckAvailabilityAndPrice2(string, int) from
                    Aggregator to QueasyJet;
13              CofirmAvailabilityAndPrice2(bool, int) from
                    QueasyJet to Aggregator;
14          }
15          ProvideFlightInformation(bool, string) from Aggregator
                to Client;
16          choice at Client { // Choice block
17              Accept() from Client to Aggregator;
18              RequestPaymentInfo() from Aggregator to Client;
19              ProvidePaymentInto(string) from Client to
                    Aggregator;
20              ConfirmPayment(bool) from Aggregator to Client;
21          } or {
22              RejectAndLeave() from Client to Aggregator;
23          } or {
24              TryAgain() from Client to Aggregator;
25              continue MakeBooking; // Recur/Iterate instruction
26          }
27      }
28  }
```

Listing 3.1: A global Scribble protocol

The parser reads through this tree recursively, launching a new goroutine to read through each sub-conversation as it goes and saving the information in a tree data structure composed of conversations stored in "nodes", where each node represents a message, a `par` block, a `choice` block or a `rec` block. Each element of this tree is represented by a struct, which consists of collection of fields which may include other structs. During this stage each element of the tree is given a unique identifier for use in subsequent computation.

3.1.3 The Translator

Next, the translator takes the tree generated by the parser and reads through it recursively to extract all of the information required to write the API and the translation. The translator makes two passes through the protocol, one going forwards and the other going backwards. The forward pass collects most of the required sequential and message data, the backward pass adds information indicating when a message at the end of a conversation needs to be followed by a "jump" back up the tree rather than protocol termination. The translator collects all of this information in a structure composed of arrays of structures representing each of the element types that make up the protocol. In order to improve

the efficiency of subsequent calculations, the translator then generates hashmaps to allow constant time access to these structures using their unique identifier strings.

3.1.4 The Writer

Finally, the writer takes in these arrays and reads through them concurrently to generate a working Go API and implementation of the Scribble protocol. If the user has passed Gobble one Scribble protocol to process, Gobble will output an inter-system implementation with code allowing it to communicate with other roles over a TCP/IP connection. If the user has passed Gobble multiple Scribble protocols, Gobble will combine them to form an intra-system implementation in which all roles will be launched by a single process and will communicate directly with one another across channels.

3.2 The Gobble Development Process

At the time that work on the Gobble compiler began, the exact structure of the desired Go output for a given Scribble input was poorly defined beyond the general requirement that it should "implement a given Scribble protocol in Go using session types". As a consequence of this the project was effectively as much a research project as a software development one. In fitting with this reality, the decision was made to develop Gobble incrementally. The basic development pattern used was the the following:

1. Produce a hand-written prototype of a given element of the desired API and one or more implementations of a protocol using that API.
2. Test, evaluate and debug that hand-written element, reworking it until satisfied.
3. Produce a minimal viable version of the Gobble lexer, parser, translator and writer necessary to produce that output, testing and debugging until satisfied with the results.
4. Refactor code to reduce coupling and increase cohesion.
5. Add another item of functionality and repeat.

The main downside of this approach was the need to regularly refactor code that had already been written in order to incorporate new feature, e.g. to create a generalised function to implement what originally been conceived of as a "one off" section of code in an earlier iteration of the compiler. The upsides consisted in the fact that it made it possible to experiment with and adapt the project as time went on and the problem-space was better understood and to automate much of the process of prototype generation, which was no small consideration, given that the code for a single protocol API and its implementation for one participant can easily run to hundreds of lines of code.

A Note on Gobble's Design

Gobble is designed to take advantage of the Go's easy-to-use, safe and efficient concurrency tools. In order to make this possible, at each stage of the translation process the compiler treats the output of the preceding stage as read only, a pre-requisite for safe concurrent data processing.

In accordance with Phil Karlton's maxim that "There are only two hard things in Computer Science: cache invalidation and naming things"[2] Gobble takes names seriously. In particular, care has been taken to ensure that (where possible) each function in the Gobble package does only one thing and has a name that clearly describes what it does. The result is functions that sometimes have rather long names, but whose purpose can generally be clearly understood without additional clarificatory comments. Accordingly, the Gobble source code is sparely commented, aside from a brief explanatory note at the top of each file and occasional notes explaining the behaviour of particularly long functions or ones with unusual behaviour.

3.3 Files Submitted with this Disertation

The files accompanying this dissertation are:

- Instructions on how to create build of the Gobble source files and run them: readme.txt.
- Gobble source files: main.go, lexer.go, parser.go, translator.go, writer_methods.go, writer_main.go, writer_network.go and writer_functions.go.
- Gobble test source files: test1_client.scr, test1_server.scr, test2_client.scr, test2_server.scr, test3_A.scr, test3_B.scr, test3_C.scr, test4_A.scr, test4_B.scr. test5_A.scr, test5_B.scr, test6_A.scr, test6_B.scr, test7_C.scr, test7_P.scr, test8_A.scr, test8_B.scr, test8_C.scr.
- Case study source files: aggregatorLocal.scr, brutishAirwaysLocal.scr, clientLocal.scr and queasyJetLocal.scr.
- Case study inter-system communication output: client_main.go, client_api.go and client_network.go.
- Case study intra-system communication output: aggregator_api.go, aggregator_main.go and aggregator_network.go.
- Speed benchmark test files: speedTest_Client.scr, speedTest_Server.scr, speedTests-Main.go, speedTestsMethods.go, speedTestsChannels.go, speedTestsStructs.go.

Chapter 4

Case Study: An Online Flight Aggregator Booking Protocol

This chapter will present a case study of the Go output produced by Gobble from a given Scribble input.

A Note on Syntax

The discussion below presupposes a basic knowledge of one or more C-like object oriented languages, such as Java, Python or C++, but neither assumes nor requires a knowledge of Scribble or Go. Where a detailed knowledge of some unusual Go feature is required, the relevant section will be highlighted and a detailed explanation will be provided. That said, the reader unacquainted with Go may find it useful to refer to the following table to understand some of the features of Go syntax which are absent from some of the above-mentioned languages:

Symbol	Meaning	Example
`:=`	Short declaration	`var myString string; string = "dog"` is equivalent to `myString := "dog"`
`<-`	Send to or receive from channel	Sending: `myString = <- myChannel`; receiving `myChannel <- myString`
`*`	Pointer	Type `*myType` is a pointer to an instance of type `myType`
`&`	Generate pointer	`&myStruct` generates a pointer to `myStruct`

Table 4.1: Go syntax, a brief guide for the perplexed

4.1 Input

The Scribble input for this case study is a hypothetical booking protocol for an online flight aggregator, a web application which allows customers to compare and purchase flights

tickets for a given itinerary provided by multiple airlines. The protocol is designed to be as simple as possible while still including all of the different Scribble structures that Gobble can process.

Listing 3.1, in the preceding chapter, showed a global version of the protocol we will be examining, while Listing 4.1 shows the local implementation of the protocol from the point of view of the `Aggregator`. Existing Scribble tools written in Java[15] and Python[16] are designed to take in Scribble global protocols (which show the objective interactions between all of the participants in a communication protocol from a neutral perspective), check that they will not produce deadlocks, and output local protocols (designed to show a subjective view of the protocol from the point of view of one of the participants).

```
1  module TravelAgency;
2
3  local protocol BookJourney at Aggregator (role Client, role
       Aggregator, role BrutishAirways, role QueasyJet) {
4      Initiate() from Client;
5      Acknowledge() to Client;
6      rec MakeBooking { // Recursive/loop block
7          RequestItinerary(string, int) from Client;
8          par { // Concurrent block
9              Check1(string, int) to BrutishAirways;
10             Confirm1(bool, int) from BrutishAirways;
11         } and {
12             Check2(string, int) to QueasyJet;
13             Confirm2(bool, int) from QueasyJet;
14         }
15         Inform(bool, string) to Client;
16         choice at Client { // Choice block
17             Accept() from Client;
18             RequestPayment() to Client;
19             ProvidePayment(string) from Client;
20             ConfirmPayment(bool) to Client;
21         } or {
22             Reject() from Client;
23         } or {
24             TryAgain() from Client;
25             continue MakeBooking; // Recur/Iterate instruction
26         }
27     }
28 }
```

Listing 4.1: A local Scribble protocol

Translated into English prose, the protocol shown in Listing 4.1 describes the following series of interactions:

1. A `Client` initiates a dialogue with the `Aggregator`.
2. The `Aggregator` acknowledges the `Client`.

3. The `Client` then sends a request to the `Aggregator` for a given itinerary (e.g. a flight from Glasgow to Paris on November 12, 2017).

4. The `Aggregator` then forwards the `Client`'s request to multiple airlines it deals with, in this case `BrutishAirways` and `QueasyJet`. The airlines respond to the `Aggregator` detailing their availability and prices: these interactions between the `Aggregator` and the airlines are defined as taking place concurrently, meaning that they can happen either in parallel or in an arbitrary sequence, but in any case must both be completed before moving on to the next step in the protocol.

5. The `Aggregator` informs the `Client` of the options available.

6. The `Client` now has a choice as to how to proceed:

   (a) They can choose to accept the offer, in which case an exchange of payment information will take place and the dialogue will terminate;

   (b) they can chose to reject the offer, in which case the dialogue between the parties will come to an end; or

   (c) they can choose to repeat the process, in which case the dialogue will be able to submit a request for another itinerary, returning to step 3.

4.2 API Output

A Note on Object Orientation in Go

The core part of Gobble's output is its API, which represents the stages of a Scribble protocol as data structures. The central challenge for designing the programme's output was therefore to find a way of representing sequential steps of communication using Go's data structures. The approach to this challenge taken by Hu and Yoshida in their Java API was to representing each step of communication as a Java class, an approach they had proven could work[7]. At first glance, attempting to replicate this approach in Go faced a seemingly insurmountable challenge: Go does not have classes. Happily, this turned out not to be the case.

Java is designed to have what might be termed "heavy-weight object orientation": aside from a few basic types, such as strings, integers and booleans, everything the language contains is a subtype of an inheritance tree descending from the base "object" class.

Go, in contrast, has what might be termed "light-weight object orientation". This is despite the fact that it, in addition to lacking classes, it has some other features that make it look very unlike a classic "object oriented" language: for example, functions are first class in Go, meaning that it is possible for them to stand independently instead of being methods to a Java-style class, and inheritance is not permitted, meaning that the construction of something similar to Java's class tree is not possible. Go's form of object orientation stems from the fact that it is possible to transform a function, which has a header of the form:

```
1 func <function_name>(<parameter_name> <parameter_type>) (<
    return_type>) {
```

Listing 4.2: The syntax of a Go function header

into a method on any locally defined type by adding an associated type declaration:

```
func (<associated_type_instance_name> <associated_type>) <function_name
    >(<parameter_name> <parameter_type>) (<return_type>) {
```

Listing 4.3: The syntax of a Go method header

As a consequence of this design feature it is possible to associate methods with structs in order to form entities which behave like Object Oriented Programming (OOP) objects, with their associated methods and instance variables. Polymorphism, another classical OOP attribute, is also made possible by the fact that Go has interfaces.

Gobble represents session types as entities which are functionally indistinguishable from OOP objects and are composed of structs and methods. Accordingly, the rest of this paper will sometimes refer to such abstract representations of protocol steps as Gobble "objects" despite the fact that Go does not explicitly contain any abstraction of that name.

4.2.1 Structs

The first half of the objects which make up the Gobble API consist of structs. These fall into three types: the sequential structs used to represent each step of the communication protocol; the data types used to represent the data sent between the communicating parties during each step of the communication protocol; and the channels over which the said data is transmitted.

Sequential Step Representation

```
type Client_rec1_1 struct {
        Channels *Channels
        Used     bool
}
```

Listing 4.4: A Gobble struct representation of a step in a locally defined communication protocol

Each step in a local protocol is represented by a separate struct. Each such struct is given a name indicating its position within the protocol. For example, the `Client_rec1_1` name shown in Listing 4.4 can be read (backwards) to indicate that it represents "the first message in the first `rec` block in the Client local protocol". This system of naming has been used in order to ensure that each struct will have a unique name which is meaningful and readable both for computers and humans. The struct's two fields are a pointer to a `Channels` object and a `Used` boolean variable whose value is set to `false` when the struct is created and which is subsequently set to `true` as soon any of the struct's associated methods are used, thus ensuring adherence to the protocol by preventing the same step being made repeatedly or multiple options being selected when the protocol allows only one choice to be made.

Sent Data Representation

```
type request_from_client_to_aggregator_string_int struct {
        Param1 string
        Param2 int
}
```

Listing 4.5: A Gobble struct representation of the data to be sent during one step in a locally defined communication protocol

The data to be sent between the communicating parties at each stage of the protocol is also represented as a distinct struct, and is given a name reflecting that messages name, the type of data being sent, its sender and its recipient, in order to individuate it from the other message data-types. In this case the struct type is `request_from_client_to_aggregator_string_int`.

Channels

```
type Channels struct {
        RequestFromClientToAggregator_string_int  chan
           request_from_client_to_aggregator_string_int
        ...additional channels...
}
```

Listing 4.6: A Gobble struct containing the channels used by one participant in a multi-party communicaiton protocol

Each struct representation of a single step in the communication protocol contains a pointer to a struct containing the channels across which communication will take place. There is a unique channel sending a unique data type for each step of the communication protocol, to ensure that deviations from the correctly defined protocol will be detected and flagged by Go's type checking system. Shown is the `RequestFromClientToAggregator_string_int` channel is used to send the `request_from_client_to_aggregator_string_int` data type.

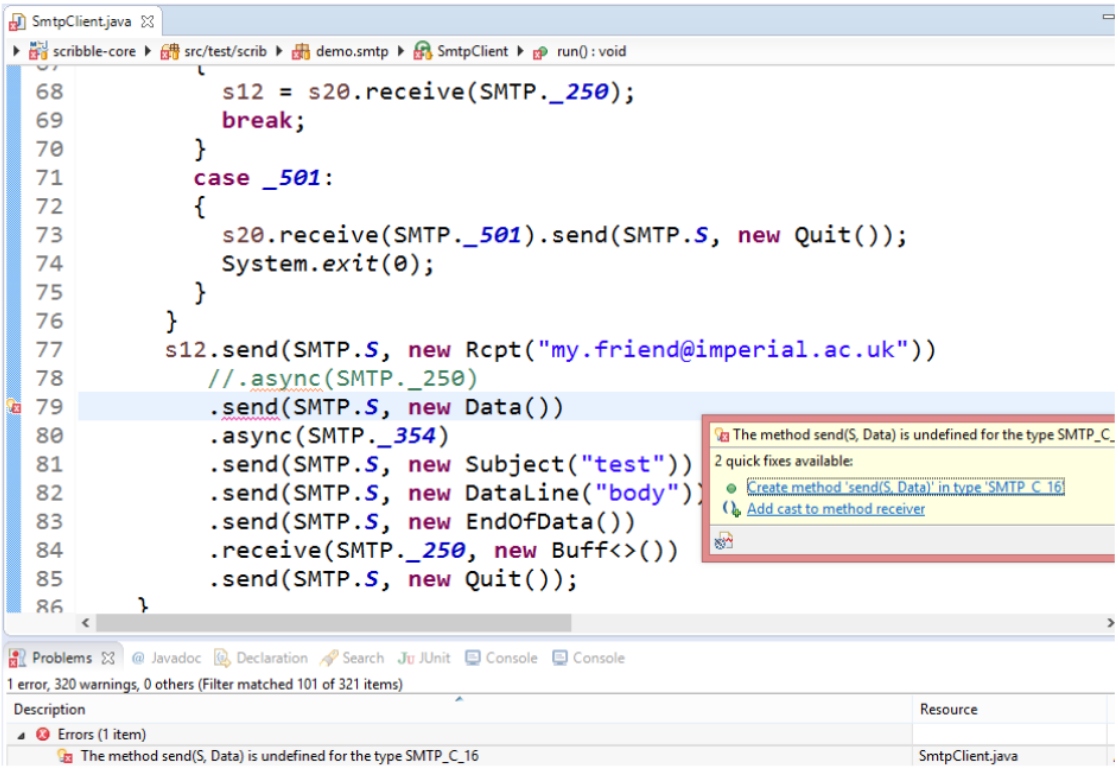
4.2.2 Methods

The second half of the objects which make up the Gobble API consists of the methods which carry out the actions required at each step of the communication protocol. These are divided into two kinds: sending and receiving. The structs representing each step of the communication protocol may have either a single associated method (in the case of the struct representing part of a linear path of progression through the protocol) or multiple methods (in the case of the struct representing a step in the communication protocol in which multiple choices are open to one of the participants).

A Note on Multiple Return Values in Go

Java, the language in which Hu and Yoshida's Scribble API was implemented, is a single return value language. This means that a given Java method can only return a value of a single type. It also means that when one wishes to create a Java function that returns more than one value, that function must combine those multiple values into a single data structure, such as an array or an object. This limitation has also meant that Java has developed a system of throwing exceptions for dealing both with unexpected programme errors (such as a null pointer being encountered) and with expected programming events which cannot easily be encapsulated in a single return value (such as the end of a file being reached)[11]. One consequence of Java's being a single return value language is that it was possible for Hu and Yoshida to develop their Java API to send messages using a "dot" notation, (see Figure 4.1).

Figure 4.1: An image of Java "dot" notation as used by Hu and Yoshida in their implementation of a Scribble API, taken from their paper *Hybrid Session Verification through Endpoint API Generation*. Adherence with Go's idiomatc error handling system means that the use of this notation is not possible using the Gobble API.[7]



Go, in contrast, is a multiple return value language which allows a given function or method to return an arbitrary number of values. This means that it is possible to write functions which return multiple values without those values needing to be combined into a single data structure. It also means that it has been possible for Go's designers to take an alternative approach to error handling in which the final return value of a function or method is an error, a type of interface which represents any value that can describe itself as a string. Errors are typically dealt with using a `if`-statements which check whether they have a

value of `nil` and take error handling action if they do not[3], as shown in Listing 4.7:

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
// do something with the open *File f
```

Listing 4.7: Idiomatic error handling in Go

The Go standard library is written using this error handling system and writing idiomatic Go code essentially requires it. Accordingly the choice was made to abandon Hu and Yoshida's dot notation in Gobble and to use multiple return values. Instead, each Gobble API method has the pattern of return values illustrated in Listing 4.8 (below), consisting of:

1. One or more structs representing subsequent steps in the protocol, separated by commas;
2. Zero or more received values, separated by commas; and
3. A single error value.

```
        return retVal, in_1_string, in_2_int, nil
```

Listing 4.8: A Gobble method return line

Sending

```
func (self *Client_rec1_1) Send_Request_string_int(
    param1 string, param2 int) (*Client_rec1_2, error) {
        defer func() { self.Used = true }()
        sendVal :=
           request_from_client_to_aggregator_string_int{param1
           : param1, param2: param2}
        retVal := &Client_rec1_2{Channels: self.Channels}
        if self.Used {
                return retVal, errors.New("Dynamic session
                   type checking error: attempted repeat
                   method call to one
                   request_from_client_to_aggregator_string_int
                   ()")
        }
        self.Channels.RequestFromClientToAggregator_string_int
            <- sendVal
        return retVal, nil
}
```

Listing 4.9: A Gobble send method

Listing 4.9 shows `Send_Request_string_int()`, a data sending method. Highlighted in yellow, the most significant parts of the method are:

1. In line 1, the `self` identifier representing a pointer to the `Client_rec1_1` struct with which the method is associated.
2. Also in line 1, the `param1 string` and `param2 int` parameters received by the method which contain the data to be sent at that current step of the communication protocol.
3. In lines 2 and 5, the deferred anonymous function which sets `Client_rec1_1`'s `Used` value to `true` once the method has completed and the check which returns an error message if `self.Used`'s value is `true`. This dynamic check ensures that a runtime error will be raised in the case of an attempt to execute a method associated with a given struct more than once, in violation of the communication protocol.
4. In line 4, the creation of the `Client_rec1_2` object representing the subsequent step in the communication protocol, a pointer to which is returned as `retVal`.
5. In line 8, the `<-` indicating the data given as the method's parameters is being sent into a communication channel to be received by `Aggregator`.

Receiving

The previously examined send method's corresponding receive method which is created as part of `Aggregator`'s local protocol is essentially the same in structure as the send method, except for a couple of features:

1. In line 1, the method's associated struct is `Aggregator_rec1_1` rather than `Client_rec1_1`.
2. In line 12, the method's return values include the values of type `string` and `int` which were received by `Aggregator_rec1_1` from `Client_rec1_1`.

```
1 func (self *Aggregator_rec1_1) Receive_Request_string_int() (*
    Aggregator_rec1_par1_start, string, int, error) {
2       defer func() { self.Used = true }()
3       var in_1_string string
4       var in_2_int int
5       retVal := &Aggregator_rec1_par1_start{Channels: self.
          Channels}
6       if self.Used {
7               return retVal, in_1_string, in_2_int, errors.
                  New("Dynamic session type checking error:
                  attempted repeat method call to one or more
                   methods of a given session type struct.
                  Struct: Aggregator_rec1_1; method:
                  Receive_Request_string_int()")
8       }
```

```
9           in := <-self.Channels.
               RequestFromClientToAggregator_string_int
10          in_1_string = in.param1
11          in_2_int = in.param2
12          return retVal, in_1_string, in_2_int, nil
13  }
```

Listing 4.10: A Gobble receive method

4.3 Inter-System Communication

The Gobble API's representation of intra-system communication, as described above, essentially consists of multiple goroutines communicating with one another across channels. The API's approach to inter-system communication makes no changes to this system, but rather adds another layer to it. For inter-system communication, instead of goroutines communicating with one another directly, they communicate with other dedicated goroutines which handle the sending and receiving of messages across a TCP connection.

```
1   func SendToAggregator(chans *Channels, trans *Transmitter) {
2           ...setting up TCP connection...
3           for {
4                   select {
5                   case out := <-chans.
                       RequestFromClientToAggregator_string_int:
6                           g := AggregatorClientGob{Name: "
                               Request_from_Client_to_Aggregator_
                               string_int",
                               Request_from_Client_to_Aggregator_
                               string_int: out}
7                           err := trans.Encoder.Encode(g)
8                           if err != nil {
9                                   log.Fatal(err)
10                          }
11                  ...other cases...
12                  default:
13                          // Keep looping
14  }   }   }
```

Listing 4.11: A Gobble function to send messages received from a channel onward across a TCP connection

Listing 4.11 illustrates how this is implemented in practice. The `SendToAggregator()` function runs an infinite loop on a dedicated goroutine in which it waits to be sent messages from the channels linking `Client` and `Aggregator`. In the example shown, if the goroutine receives data from the `RequestFromClientToAggregator_string_int` channel it then adds this to the struct `g` and encodes `g` to be sent across a TCP/IP connection.

```
1   func ReceiveFromClient(chans *Channels, trans *Transmitter) {
```

```
        var in ClientServerGob
        for {
                err := trans.Decoder.Decode(&in)
                if err != nil {
                        log.Fatal(err)
                }
                if in.Name == "
                   Request_from_Client_to_Aggregator_string_int
                   " {
                        chans.RequestFromClientToAggregator_
                           string_int
                           <- in.Request_from_Client_to_Aggregator
                           _string_int
                } else if
                        ...other message types...
                } else {
                        log.Fatal("ReceiveFromServer()
                           received unknown gob: ", in)
}       }       }
```

Listing 4.12: A Gobble function to receive messages sent across a TCP connection

At the other end of the TCP connection, as illustrated in Listing 4.12, there is another dedicated goroutine running a function waiting to receive the data stream originating from `Client`. That goroutine receives the value, assigns it to a pointer to `in` and then sends the data into the local `Request_from_Client_to_Aggregator_string_int` channel.

Data sent across the TCP connection is encoded in the form of Gob streams. Gob streams are data streams specifically designed for transmitting Go data structures[12]. The decision to use Gob streams for data transmission in Gobble was based on two factors. Firstly, Gobs are specifically designed to encode and decode efficiently into typed Go data structures, meaning that they offer ease of use and fit well with the system of session types that Go uses. Secondly, Gobs send data in binary form, meaning that they offer a more efficient means of communication than would text-based alternatives such as JSON or XML.

The main advantage of implementing Gobble's inter-system communication ability in this way is that it ensures that Gobble's core API and inter-system communication code are encapsulated from one another, meaning that one can altered without affecting the other. For example, if one desired to alter the inter-system communication code to send data in JSON rather than Gob format so as to enable communication with non-Go systems one could do so without making any alteration to the main Gobble API.

As a default Gobble sets the connection type to `tcp`, the network address to `localhost` and the port to `:8080`, values which can be altered by editing the output's `main()` method. In order to ensure that each communicating pair specified in the communication protocol have one defined server and one defined client and that both sides agree on which that is, Gobble automatically sets the one whose name comes first alphabetically as the client and the other as the server, e.g. `Aggregator` is automatically set to be the server for messages sent between it and `Client` (the mnemonic here being that "client" comes before "server"

alphabetically). In order to ensure flexibility this can, however, also easily be altered by the programmer. The code for both parties contains the requisite methods for them to act as either client or server to one another and changing the relationship is a simple matter of a programmer's uncommenting or commenting out a few lines or code which are indicated by comments in Gobble's output.

4.4 Translation Output

In addition to producing the above-described API, Gobble also produces full translations of Scribble protocols into executable Go code. The aim of the translation is to use the information contained in the Scribble source to produce all of the predictable "boiler-plate" code required to implement a given protocol in an efficient, safe and consistent form, leaving the programmer free to focus on adding application-specific code that will set the values of data to be transmitted and use tests to set the flow of control.

When executed, the automatically generated code initiates all of the variables representing the data types to be sent to their Go "zero" values (e.g. `0` in the case of integers and an empty string in the case of strings[14]) and prints out statements to the console informing the user that the values have been received by their intended recipients.

```
1  func runAggregator(Aggregator1 *Aggregator1) (interface{},
       error) {
2          Aggregator2, err1 := Aggregator1.Receive_Initiate()
3          if err1 != nil {
4                  log.Fatal(err1)
5          }
6          Aggregator_rec1_1, err2 := Aggregator2.
               Send_Acknowledge()
7          if err2 != nil {
8                  log.Fatal(err2)
9          }
10         retVal, err3 := LoopAggregator_rec1(Aggregator_rec1_1)
11         if err3 != nil {
12                 log.Fatal(err3)
13         }
14         return retVal, nil
15 }
```

Listing 4.13: A Function from a Gobble translation

The automatic translation produced by Gobble consists of function such as the one show in Listing 4.13 which can contain both calls to methods representing messages to be sent to other communicating parties (as in the case of the call to `Aggregator1.Receive_Initiate()` in line 3) and calls to other functions (as in the case of the call to `LoopAggregator_rec1-(Aggregator_rec1_1)`) in line 11. All such functions have two return values, the first of which has type `interface{}` and the second of which is of type `error`.

A Note on Go Interfaces

The `interface{}` return type represents Go's "empty interface". Go has a system of implicit interfaces in which an interface can be defined to apply to any type that has associated methods with the same name, parameter and return value types as those defined in the interface[14]. For example, any type which has associated `Lock()` and `Unlock()` methods with no parameters and no return types will can be treated as an instance of the `Locker` interface shown in Listing 4.14:

```
1 type Locker interface {
2         Lock()
3         Unlock()
4 }
```

Listing 4.14: A Go interface

Accordingly, all types can be treated as instances of the empty interface because all types fulfil the requirement of having at least zero associated methods. The blank interface is used by Gobble here because it provides a simple, concise way of passing pointers to session type objects up the programme's tree of execution until they reach the point where they are required. For example, it allows a pointer to a struct to be passed up the tree to a recursive block where it can be tested to see if it matches the first value in the recursive block: if the types do match, looping can continue; if they do not, then the pointer can be passed up another level and programme execution can continue (see subsection 4.4.1 below for an example).

Similarly, the reason that all Gobble functions and methods return an `error` value is to make it easy to customise code to pass error messages up the chain of execution to the point where one wishes to deal with them.

4.4.1 Recursion Blocks

Scribble defines looping behaviour by means of a `rec` keyword. The Gobble API allows for multiple possible implementations of these `rec` blocks. One of these is the recursive implementation shown in Listing 4.15 where `runAggregator_rec1_choice1_A()` makes a recursive call to `LoopAggregator_rec1()` in line 16:

```
1 func LoopAggregator_rec1(Aggregator_rec1_1 *Aggregator_rec1_1) (
     interface{}, error) {
2         retVal, err := runAggregator_rec1(Aggregator_rec1_1)
3         if err != nil {
4                 log.Fatal(err)
5         }
6         return retVal, nil
7 }
8         ...intermediary functions...
9 func runAggregator_rec1_choice1_A(Aggregator_rec1_choice1 *
     Aggregator_rec1_choice1) (interface{}, error) {
```

```
        Aggregator_rec1_1_new, err1 := Aggregator_rec1_choice1
            .Receive_TryAgain()
        if err1 != nil {
                log.Fatal(err1)
        }
        return LoopAggregator_rec1(Aggregator_rec1_1_new)
}
```

Listing 4.15: A recursive implementation of a Scribble `rec` block using the Gobble API

There is, however, a problem with such recursive implementations: Go does not optimise for tail call recursion. This means that making recursive calls in Go results in unsustainable growth in the size of the programmes stack. Testing shows that even very simple Scribble protocols can result in a Go stack overflow occurring after around one million recursive calls. This is a serious problem for communication protocols containing looping structures which, by their nature, are designed to be used repeatedly.

Accordingly, the decision was made to implement Scribble `rec` blocks in Gobble using iteration rather than recursion when producing automatic translations, as is shown in Listing 4.16 below:

```
func LoopAggregator_rec1(Aggregator_rec1_1_old *
    Aggregator_rec1_1) (interface{}, error) {
        var retVal interface{}
        looping := true
        for looping {
                retVal, err := runAggregator_rec1(
                    Aggregator_rec1_1_old)
                if err != nil {
                        log.Fatal(err)
                }
                switch t := retVal.(type) {
                case *Aggregator_rec1_1:
                        Aggregator_rec1_1_old = t
                default:
                        looping = false
                }
        }
        return retVal, nil
}
        ...intermediary functions...
func runAggregator_rec1_choice1_A(Aggregator_rec1_choice1 *
    Aggregator_rec1_choice1) (interface{}, error) {
        Aggregator_rec1_1_new, err1 := Aggregator_rec1_choice1
            .Receive_TryAgain()
        if err1 != nil {
                log.Fatal(err1)
        }
        return Aggregator_rec1_1_new, nil
```

```
25 }
```

Listing 4.16: An iterative implementation of a Scribble `rec` block using the Gobble API produced as an automatic translation

In this implementation iterative looping is achieve using the system of passing pointers to structs up the tree of execution as interfaces and testing them for type to see where they are needed described above in section 4.4. On line 24 `runAggregator_rec1_choice1_A` returns a pointer to an `Aggregator_rec1_choice1` struct as an instance of an empty interface. When the pointer reaches `LoopAggregator_rec1` it is tested in the type switch at line 9. If the received value is of type `*Aggregator_rec1_choice1` looping continues, otherwise the loop breaks and the return value continues on its way up the path of execution.

4.4.2 Choice Blocks

Looping is one of the centrally important structures in computation and the one that makes it possible for computers to deal with large amounts of data. However, in order for looping to be able to fulfil useful tasks it needs to be combined with tests of some kind. In Scribble, while `rec` blocks play the role of C's `for` and `while`, `choice` play is the role of C's `if` and `else`.

Scribble `choice` blocks are characterized by one communicating party (the "chooser") sending a message to one or more other communicating parties (the "chosen") indicating to them which among a set possible paths through the protocol should be taken.

The Gobble API represents `choice`s as structs with multiple associated methods. For example, at one point in the flight aggregator protocol, `Client_rec1_choice1` can choose between executing `.Accept()`, `.Reject()` or `.TryAgain()` methods. As part of Gobble's session-type-safety system, if an attempt is made to execute more than one such method pertaining to a given structure or to execute one such method repeatedly the user (as shown in Listing 4.17) will produce an error message (such as the one shown in Figure 4.2)

```
1 retVal, err := runClient_rec1_choice1_A(Client_rec1_choice1)
2 retVal, err = runClient_rec1_choice1_B(Client_rec1_choice1)
```

Listing 4.17: Erroneous code attempting to call the both methods `A` and `B` associated with the `Client_rec1_choice1` struct: which will trigger the error message shown in Figure 4.2

Figure 4.2: An error message resulting from Gobble dynamic-session-type-checking error produce by the code in Listing 4.17

```
2017/09/06 11:17:32 Dynamic session type checking error: attempted repeat method
 call to one or more methods of a given session type struct. Struct: Client_rec1
_choice1; method: Send_RejectAndLeave()
exit status 1
```

The "chosen" communicating parties must be prepared to deal with any choice that the "chooser" might decide to make. Accordingly, protocol implementations for the chosen parties must include `select` blocks similar to those shown below in Listing 4.18:

```
1  func Make_Aggregator_rec1_choice1_Choices(
       Aggregator_rec1_choice1 *Aggregator_rec1_choice1) (
       interface{}, error) {
2          var recVal interface{}
3          var err error
4          select {
5          case received_A := <- Aggregator_rec1_choice1.Channels.
               TryAgainFromClientToAggregator_Empty:
6                  Aggregator_rec1_choice1.Channels.
                       TryAgainFromAggregatorToAggregator_Empty <-
                       received_A
7                  recVal, err = runAggregator_rec1_choice1_A(
                       Aggregator_rec1_choice1)
8                  if err != nil {
9                          log.Fatal(err)
10                 }
11         ...other cases...
12         }
13         return recVal, nil
14 }
```

Listing 4.18: A function containing a Gobble choice `select` block

This `select` blocks controls the flow of the programme by transferring the received data type from `TryAgainFromClientToAggregator_Empty` (a `Client` to `Aggregator` channel) to `TryAgainFromAggregatorToAggregator_Empty`, an internal, buffered channel used by `Aggregator` to send the value on to the method where it will be needed.

4.4.3 Parallel Blocks

The Scribble language specification states that a `par` block is used to specify "a set of interactions that can happen concurrently", meaning that those interactions may occur simultaneously or in an arbitrary order[18] . The Gobble API allows for either or those possibilities.

The Gobble API represents the `par` block in the Flight Aggregator protocol using two dedicated objects, `Aggregator_rec1_par1_start` and `Aggregator_rec1_par1_end`. The first of these, `Aggregator_rec1_par1_start`, returns the two Gobble objects representing the first steps in both of the concurrent protocol paths: `Aggregator_rec1_par1_A1` and `Aggregator_rec1_par1_B1`. Once those paths have been completed messages are sent to the `done_rec1_par1_A` and `done_rec1_par1_B` channels.

```
1  func (self *Aggregator_rec1_par1_end) EndPar() (*
       Aggregator_rec1_2, error) {
2          ...set-up...
3          par_A_done := false
4          select {
5          case <-self.Channels.done_rec1_par1_A:
```

```
6                       par_A_done = true
7               default:
8                       // Continue execution
9               }
10              ...checking case B...
11              if !par_A_done {
12                      return retVal, errors.New("Dynamic session
                           type checking error: attempted call to
                           Seller_rec1_par1_end.EndPar() prior to
                           completion of parallel process
                           Seller_rec1_par1_A")
13              }
14              ...error message for case B...
15              return retVal, nil
16      }
```

Listing 4.19: A Gobble `.EndPar` method

As Listing 4.19 shows, `Aggregator_rec1_par1_end.EndPar()` checks to see that process completion messages have been sent by each of the concurrent processes in the `par` block. If all such messages have been received, `Aggregator_rec1_par1_end.EndPar()` will return the subsequent object in the protocol. In contrast, if only one of the paths through the parallel block has been executed prior to `.EndPar()` being called (as in the code shown in Listing 4.20) an error message (such as the one shown in Figure 4.3) will result.

```
1  runAggregator_rec1_par1_A(Aggregator_rec1_par1_A1)
2  Aggregator_rec12, err_end := Aggregator_rec1_par1_end.EndPar()
```

Listing 4.20: Erroneous code attempting to call an `.EndPar()` method before all paths through a `par` block have been executed: which will trigger the error message shown in Figure 4.2

Figure 4.3: An error message resulting from Gobble dynamic-session-type-checking error produce by the code in Listing 4.20

```
2017/09/06 12:33:49 Dynamic session type checking error: attempted call to Selle
r_rec1_par1_end.EndPar() prior to completion of parallel process Seller_rec1_par
1_B
exit status 1
```

```
1  func runAggregator_rec1_par1(Aggregator_rec1_par1_start *
       Aggregator_rec1_par1_start) (*Aggregator_rec1_2, error) {
2               Aggregator_rec1_par1_end, Aggregator_rec1_par1_A1,
                   Aggregator_rec1_par1_B1, err :=
                   Aggregator_rec1_par1_start.StartPar()
3               if err != nil {
4                       log.Fatal(err)
5               }
6               var newWg sync.WaitGroup
7               newWg.Add(1)
```

```
8           go runAggregator_rec1_par1_A(Aggregator_rec1_par1_A1,
               &newWg)
9           newWg.Add(1)
10          go runAggregator_rec1_par1_B(Aggregator_rec1_par1_B1,
               &newWg)
11          newWg.Wait()
12          Aggregator_rec12, err_end := Aggregator_rec1_par1_end.
               EndPar()
13          if err_end != nil {
14                log.Fatal(err_end)
15          }
16          return Aggregator_rec1, nil
17  }
```

Listing 4.21: An automatically generated Gobble implementation of a `par` block

Gobble's automatic translations of Scribble `par` blocks, such as the one shown in Listing 4.21, are designed to take advantage of the language's easy-to-use and efficient concurrency support. In line 2 the `Aggregator_rec1_par1_start.StartPar()` method is called, returning the `Aggregator_rec1_par1_end`, `Aggregator_rec1_par1_A1` and `Aggregator_rec1_par1_B1` objects. In line 6 a new `sync.WaitGroup` object is created, an object which waits for a collection of goroutines to finish. In line 7 a new item is added to the `WaitGroup` to represent the `A` path through the concurrent section of the protocol. In line 8 a goroutine to execute the `runAggregator_rec1_par1_A()` function is launched and is passed a pointer to the `WaitGroup` as a parameter. In lines 9 and 10 the same process is repeated for path `B`. In line 11 the `WaitGroup`'s `.Wait()` method is called to cause the main goroutine to pause until the `A` and `B` goroutines signal that they have finished execution. Once the waiting period has come to an end, the `.EndPar()` method will be executed and—provided that no error message has been returned—programme execution will continue.

An alternative, sequential implementation of the concurrent section (lines 6-11) of the automatically generated function shown in Listing 4.21 can be seen in Listing 4.22 below.

```
1           runAggregator_rec1_par1_A(Aggregator_rec1_par1_A1)
2           runAggregator_rec1_par1_B(Aggregator_rec1_par1_B1)
3           Aggregator_rec12, err_end := Aggregator_rec1_par1_end.
               EndPar()
```

Listing 4.22: One alternative implementation of line 6-11 of the function shown in Listing 4.21

Chapter 5

Project evaluation

This project evaluation will consist of five sections: a description of the tests Gobble was subjected to during its development; a report on the responses to an expert evaluation of the Gobble API; a benchmark speed test analysis; an analysis of Gobble's met and unmet project requirements; and a suggested list of areas for future work.

5.1 Testing

In order to ensure a level of objectivity in the testing of Gobble, the test protocols used to evaluate the compiler's performance were not developed by the author but were instead sourced externally, from the test cases used to developed for the Scribble Java tool developed by Hu and Yoshida and available at the project's online repository at github.com/scribble/scribble-java/tree/master/scribble-demos. The test cases chosen were selected to cover the full range of the Scribble protocols that Gobble can deal with (consisting of a range of combinations of simple messages, `rec` blocks, `choice` blocks and `par` blocks) and were altered to have their specified Java data types replaced with inbuilt Go data-types. Given that Gobble is designed to process only Scribble local protocols which have already been checked for correctness by other means (such as the tool available at github.com/scribble/scribble-python) the test cases do not include examples of non-valid Scribble protocols. These Scribble source protocols can be found in Appendix A of this report and in the files accompanying this document.

The protocols were used to test Gobble to ensure that the compiler could:

1. Combine all of the protocol roles to form a single Go source package in which each role was assigned to a separate goroutine for intra-system communication.
2. Generate separate Go source packages for each role including the ability to connect with other roles over a TCP/IP connection for inter-system communication.

The Go source code produced was then tested to ensure that:

1. The code compiled without errors.
2. The code executed correctly along all possible choice paths.

The results of these tests can be seen in Table 5.1.

Test	**Scribble source processed by Gobble without errors**	**Generated file compiled without errors**	**All choice paths executed without errors**
Case study	✓	✓	✓
Test 1	✓	✓	✓
Test 2	✓	✓	✓
Test 3	✓	✓	✓
Test 4	✓	✓	✓
Test 5	✓	✓	✓
Test 6	✓	✓	✓
Test 7	✓	✓	✓
Test 8	✓	✓	✓

Table 5.1: Gobble test results

5.2 Expert Evaluation

The purpose of Gobble is to provide Go programmers with a tool to enable them to carry out the complex task of implementing a communication protocol using a session-type-safe API. Consequently, a meaningful evaluation can only be carried out by an experienced Go programmer who has a basic understanding of what session types are and who has completed a programming task using the Gobble API. Given the restrictive nature of these criteria and the time and effort required to complete such an evaluation, the decision was made to obtain a detailed, qualitative evaluation of the API from two expert Go programmers rather than attempting to gain a greater amount of more superficial and less meaningful quantitative information from a greater number of users.

5.2.1 Response details

The expert evaluators read through an extensive briefing on the purpose and theoretical foundations of the Gobble project, which can be found at goo.gl/forms/XfFrqSKvtZuS9qvm2. They subsequently carried out a programming task which can be found at play.golang.org/p/6kdTrw51jB and responded to a questionnaire asking for them to evaluate the Gobble API. The following is a summary of their responses:

- The expert evaluators confirmed that they had produced commercial production Go code.
- The expert evaluators confirmed that they had encountered deadlocks when writing Go code and that the main tool they currently used for preventing them was Go's `sync.WaitGroup` tool.
- They confirmed that they had both attempted and completed the programming challenge. They stated that the most difficult part of the challenge consisted in dealing with Gobble's system of naming things.
- In response to the question "How user-friendly would you say the Gobble API is, compared to writing out code to check the sequential correctness of messages manually?" they both gave the answer "5/5: Far more user-friendly" explaining that it made the programme flow more obvious and that "the strict alternation between sender and receiver makes this sort of communication very clear".
- In response to the question "If you were required to produce Go code implementing a communication protocol, how likely would you be to make use of an API such as Gobble to aid in that process?" they gave the answers "4/5: More likely" (no explanation given) and "3/5: Neither likely nor unlikely" (could not envisage a use-case).
- In response to the question "If you were required to produce Go code implementing a communication protocol, how likely would you be to make use of an automatic code generation tool such as Gobble to write the basic implementation for you, in addition to the API?" they both gave the answer "5/5: Far more likely", explaining that they viewed hand-implemented code as "brittle" and stating that a "higher abstraction helps with clarity in protocol declaration" and that "If the library is well maintained and well written it removes technical debt from your own project and makes it easier to reason about".
- In response to the request "Please suggest one or more improvements that could be made to the Gobble API that would make you more likely to use it." They stated that:
  - The Gobble API could be improved by an option to use "semantically named steps rather than numbers like Client1"
  - They would be more likely to use Gobble if they "knew Scribble already".
  - That Gobble's output would be more useful if it could deal with protocols containing arbitrary numbers of roles of a certain type, such as `Client` or `Worker`.

5.2.2 Conclusions Drawn from Responses

This evaluation suffers from the obvious disadvantage that it has a sample size of two, meaning it is important not to attempt to draw sweeping conclusions from the evaluator's responses. However, assuming that the evaluators' views are representative of a those of significant subset of Go programmers, it does indicate that the following should be taken into consideration in forthcoming iterations of Gobble:

- That Gobble's target problem space—the prevention of communication deadlocks—is one that affects Go programmers and one which they already make use of tools to address.
- That Go programmers find using the Gobble API significantly more user-friendly than they would implementing session types by hand.
- That Gobble's output could be improved in terms of usability and readability if Gobble object names could be defined semantically rather than numerically.
- That Go programmers find the level of abstraction gained from Gobble's automatic Scribble-to-Go translations useful.
- That a lack of familiarity with Scribble—a simple but far from widely-used language—is a factor which would discourage Go programmers from using Gobble.

5.3 Speed Testing

While Gobble can greatly reduce the time cost of producing communication protocol code it can do little to reduce the time cost of executing that code, which must therefore be taken into consideration when deciding whether to use Gobble session types or not. The exact level of those costs will obviously differ from protocol to protocol and from implementation to implementation, but in order to gain a rough idea of their level it was decided to develop a benchmark to examine Gobble's performance. The benchmark test chosen consists of two implementations of a simple Scribble protocol, one created using Gobble generated code and the other using a simple, hand-written implementation without any of Gobble's session-type-safety or communication abstraction. The full code for both versions can be found in Appendix B and in the files accompanying this dissertation:

The protocol, which describes a simple exchange of integers between `Client` and `Server` roles, is shown below in Listing 5.1:

```
1 module SpeedTest;
2
3 global protocol SpeedTest(role Client, role Server) {
4     rec loop {
5         firstStep(int) from Client to Server;
6         secondStep(int) from Server to Client;
7         continue loop;
8 }     }
```

Listing 5.1: The Gobble speed test protocol

The tests consisted of running both versions of the code in iterations of powers of ten from 10^0 iterations up to 10^6 iterations. The results of those tests are shown in Table 5.3:

Iterations	**Time Unit**	**Gobble**	**Non-Gobble**	**Performance Ratio**
10^0	μs	10	11	**1.1**
10^1	μs	23	17	**0.74**
10^2	μs	135	69	**0.51**
10^3	μs	1200	511	**0.42**
10^4	ms	13.2	5.4	**0.40**
10^5	ms	131	52	**0.40**
10^6	s	1.27	0.53	**0.41**

Table 5.2: Gobble speed test results

The tests indicate that while there is little initial difference between the performance of the Gobble and non-Gobble implementations of the protocol at 1 iteration, as the number or iterations increases the non-Gobble implementation performs more quickly than the Gobble one. By the time 1,000 iterations are reached, the Gobble implementation is only around 40% as fast as the non-Gobble one, and this ratio remains unchanged up to 1,000,000 iterations.

If the time performance of a section of communication code is of particular importance to the viability of an application, this means that there is an obvious argument against using Gobble session types. However, it is also worth considering three other factors when deciding whether to use Gobble or not:

1. Session types improve code safety by ensuring that it will be deadlock free.
2. If a protocol is non-recursive, then other factors may dominate execution time, meaning that there will be no significant additional time cost associated with using Gobble-generated code.
3. In the case of inter-system communication, it is likely that the time cost of communication entropy will dominate, meaning that the additional costs of using Gobble session types may be acceptable.

5.4 Meeting Requirements

Table 5.3 shows which of Gobble's project requirements were met during the development process.

Priority	Use case	Result
Must have	User can automatically generate a Go API output from a Scribble input	✓
Must have	User can automatically generate an API including Scribble's core flow of control structures: looping and branching	✓
Should have	User can automatically generate a translation of a Scribble protocol: a full, working implementation of an API	✓
Should have	User with knowledge of no languages other than Scribble and Go can read and alter all project code	✓
Should have	User can automatically generate an API of Scribble code using the full range of Version 0.3 of the Scribble language specification [18]	✗
Could have	User can specify types using Scribble's payload type declaration system	✗

Table 5.3: Gobble project use cases and their levels of priority, with results

5.4.1 Met Requirements

Having achieved both of its "must have" requirements, the Gobble project can be said to have been a success.

Gobble delivers on its first "must have" requirement: it can be used to generate a Go API-output that makes use of session types to prevent communication deadlocks from a given Scribble input.

Gobble also delivers on its second "must have" requirement: it can deal with the core flow of control structures that make it possible for non-linear protocols to be implemented. Gobble can implement looping behaviour as prescribed by Scribble's `rec` blocks and branching behaviour as prescribed by Scribble's `choice` blocks.

The Gobble project also managed to achieve two of its three "should have" requirements.

Gobble can automatically generate full Go translations of the APIs it produces. The decision was made to prioritise this use case because of its importance for increasing Gobble's usability. The principal disadvantage that using session types poses for a programmer is the amount of time-consuming and routine "boilerplate" code that they require. Gobble's usefulness as a programming tool is greatly increased by its automating the production of such code.

Gobble has also been successfully implemented entirely in Go, by hand. The decision was made to prioritise this for a number of reasons:

1. Complexity reduction: all things being equal, a programme written in one language will be simpler than one written in multiple languages.

2. Usability: the target user group for Gobble is Go programmers, so it makes sense for it to be written in that language so that they can understand and alter the code as required.

3. Pedagogical reasons: the author had never before written a parser or translator and had little knowledge of Go and saw writing as a parser and translator in Go as a good way of learning about those topics.

5.4.2 Unmet Requirements

The Gobble project has, however, been unable to achieve one of its "should have" requirement and its only "could have" requirement. The principle reason for this shortfall was a lack of the time required to write, test and debug the code necessary to deliver all of Gobble's desired functionality. This necessitated the prioritisation of certain elements of that functionality.

The unmet "should have" requirement consists in the fact that Gobble cannot deal with the full range of the Scribble Version 0.3 language specification[18]. In particular, Gobble cannot deal with Scribble protocols containing Scribble's `do` and `interruptible` blocks. The principle reasons for giving a lower level of priority to implementing the full scope of the Scribble language were:

1. Redundancy: Scribble's `do` keyword acts as kind of function call or `goto` command used to call one protocol from within itself or another protocol, meaning that its functionality can be replicated either by extending a single protocol or using a `rec` block; while Scribble's `interruptible` acts as a kind of branching instruction, meaning that its functionality can be replicated by the use of a `choice` block.

2. Safety: Go does not have tail recursion optimisation. Gobble's automatically generated Go code implements looping behaviour specified in `rec` blocks as iterative `for` loops which will not cause uncontrolled stack growth and potential overflow. It would be difficult to ensure that this was also the case for protocols that prescribed looping behaviour using the `do` keyword, particularly in the case of complex recursive calls made between multiple Scribble protocols. Therefore, forcing users to describe loops using `rec` instead of `do` leads to greater code safety. Similarly, Scribble's authors state that the use of interrupts "can cause various communication race conditions to arise" in the implementation of a protocol, meaning that it may be preferable, for example, to implement a cancellation procedure in a protocol using a `choice` block rather than an `interruptible` one[20].

Despite these reservations, it is hoped that forthcoming iterations of Gobble will be able to deal with the full set of the Scribble 0.3 language specification.

Gobble's unmet "could have" requirement consists in the fact that Gobble users cannot use Scribble's payload type declarations to specify the types of data being sent across a protocol connection.

The Scribble Version 0.3 language specification states that a Scribble protocol should contain payload type declarations such as the one show below in Listing 5.2:

```
type <java> "java.lang.Integer" from "rt.jar" as Int;
```

Listing 5.2: A Scribble payload type declaration

Each payload type declaration should include a schema type identifier (`java` in the case of Listing 5.2), the name of the type defined in an external message schema file (`java.lang.Integer` in the case of Listing 5.2) and the filename of the schema source file that provides the definition of the payload type (`rt.jar` in the case of Listing 5.2).

Gobble currently ignores such declarations. In the case of the built-in types, such as `string`, `bool` or `int64`, Gobble deals with Go data types automatically and no action is required by the user. In the case of other Go data types Gobble requires the programmer to deal with them as they would in other Go code, either by declaring them locally within the package or by importing them from another package.

The main motive for including these payload type declarations in Scribble protocols is that they make it possible, in principle, for those protocols to fulfil their aim of abstracting a communication protocol description from an implementation written in any particular language. In practice, the usefulness of such declarations is limited by the lack of a functioning, generalised system for sharing data types across languages. The development of such a system would certainly lie beyond the scope of this project. It is for this reason that dealing with other use cases has been given priority over dealing with payload type declarations when developing Gobble.

That said, it is hoped that future iterations of Gobble will provide data type declarations for the following three reasons:

1. To comply with the Scribble Version 0.3 language specification
2. To enable the automated generation of import declarations in Gobble output code
3. To make it possible to specify a URL from which data type declaration can be downloaded

5.5 Areas for Future Work

As stated above in subsection 5.4.2, the main aim for the future development of Gobble is to continue with further iterations of the programme to give it the ability to deal with protocols written using the full range of the specifications contained in the Scribble Version 0.3 language specification, in particular `interruptible` blocks, `do` instructions and payload type declarations.

Beyond this, three other areas of expansion for the project seem particularly worthy of consideration:

1. The expansion of Gobble to:

(a) Check the validity of a given Scribble global protocol; and

(b) Translate valid global protocols into local protocols

While tools to achieve these aims have already been written in Python and Java, [15, 16] adding this functionality to Gobble would increase decrease the number of dependencies required to produce its Go output and increase the user-friendliness of Gobble.

2. The creation of improved educational materials (such as tutorials or videos) to explain to Go programmers what Gobble is for and how to use it. In light of Gobble's expert evaluation, it would seem that such tools should focus on:

   (a) Explaining what session types are;

   (b) Explaining the Scribble protocol language; and

   (c) Explaining Gobble's object naming syntax;

3. The expansion of Gobble to deal with Pabble ("Parameterized Scribble") protocols. Pabble is an expanded version of Scribble which can be used to describe communication protocols featuring arbitrary numbers of roles of a particular type and ensure that such protocols will be deadlock-free [10]. This would fit with a recommendation made by one of the expert reviewers. Such functionality would be useful because many communication-based processes are structured in this way, e.g. a server dealing with an arbitrary number of clients or a main process creating an arbitrary number of worker processes to deal with a data-set. Figure 5.1 shows a Pabble specification for a communication protocol involving a multi-worker processes. A Pabble description of the Flight Aggregator case study described in Chapter 3 would, for example, allow the `Aggregator` to request flight details from an arbitrary number of airlines without needing to specify each one explicitly in the Scribble protocol source.

Figure 5.1: A Pabble protocol describing interactions between an arbitrary number of `Workers`[10].

```
1  global protocol Ring(role Worker[1..N]) {
2    rec LOOP {
3      Data(int) from Worker[i:1..N-1] to Worker[i+1];
4      Data(int) from Worker[N] to Worker[1];
5      continue LOOP; }}
```

Chapter 6

Conclusion

The Gobble project has managed to successfully deliver its key goal: the creation of a source-to-source compiler that translates a given Scribble communication protocol into executable Go code that implements an intra- or inter-system communication which will be guaranteed to be deadlock free as a result of using multi-party session types. In doing so, Gobble makes use of goroutines and channels, Go's built-in primitives designed to ensure memory-safe communication in concurrently operating systems. Gobble is able to process Scribble protocols containing linear sequences of messages exchanged between parties, concurrently defined sub-sections of protocols, recursively defined sub-sections of protocols and branching protocols. The structure of the Gobble compiler has been subjected to an analysis and its output has been examined in a detailed case study. It has been shown that Gobble has successfully passed a series of tests designed to ensure that it works as described in this document. Gobble's speed performance has been measured using a benchmark tests. Gobble's output has been subjected to an expert evaluation to determine its use to working software engineers. The success of the Gobble project has also been subjected to an evaluation and suggestions have been made as to areas for future development. Once the Master's project evaluation process has been completed, the author intends to make Gobble available online as an opensource tool for Go programmers and those with an interest in session types to make use of and contribute to.

Appendix A

Scribble Source Code for Test Cases

```
// Protocol 1_client

module Request;

local protocol Request at Client(role Client, role Server) {
        choice at Client {
                REQUESTL() to Server;
                rec X {
                        choice at Client {
                                HOST() to Server;
                                continue X;
                        } or {
                                USERA() to Server;
                                continue X;
                        } or {
                                ACCEPT() to Server;
                                continue X;
                        } or {
                                ACCEPTL() to Server;
                                continue X;
                        } or {
                                ACCEPTE() to Server;
                                continue X;
                        } or {
                                DNT() to Server;
                                continue X;
                        } or {
                                CONNECTION() to Server;
                                continue X;
                        } or {
                                UPGRADEIR() to Server;
                                continue X;
```

```
                        } or {
                                BODY() to Server;
                        }
                }
        }
}
```

```
// Protocol 1_server

module Request;

local protocol Request at Server(role Client, role Server) {
        choice at Client {
                REQUESTL() from Client;
                rec X {
                        choice at Client {
                                HOST() from Client;
                                continue X;
                        } or {
                                USERA() from Client;
                                continue X;
                        } or {
                                ACCEPT() from Client;
                                continue X;
                        } or {
                                ACCEPTL() from Client;
                                continue X;
                        } or {
                                ACCEPTE() from Client;
                                continue X;
                        } or {
                                DNT() from Client;
                                continue X;
                        } or {
                                CONNECTION() from Client;
                                continue X;
                        } or {
                                UPGRADEIR() from Client;
                                continue X;
                        } or {
                                BODY() from Client;
                        }
                }
        }
}
```

```
// test2_server

module Response;
```

```
4
5  local protocol Response at Server(role Client, role Server) {
6          HTTPV() to Client;
7          choice at Server {
8                  twoHundred_resp() to Client;
9          } or {
10                 fourOfour_resp() to Client;
11         }
12         rec Y {
13                 choice at Server {
14                         DATE() to Client;
15                         continue Y;
16                 } or {
17                         SERVER() to Client;
18                         continue Y;
19                 } or {
20                         STRICTTS() to Client;
21                         continue Y;
22                 } or {
23                         LASTM() to Client;
24                         continue Y;
25                 } or {
26                         ETAG() to Client;
27                         continue Y;
28                 } or {
29                         ACCEPTR() to Client;
30                         continue Y;
31                 } or {
32                         CONTENTL() to Client;
33                         continue Y;
34                 } or {
35                         VARY() to Client;
36                         continue Y;
37                 } or {
38                         CONTENTT() to Client;
39                         continue Y;
40                 } or {
41                         VIA() to Client;
42                         continue Y;
43                 } or {
44                         BODY() to Client;
45                 }
46         }
47 }
```

```
1  // test2_client
2
3  module Response;
4
```

```
local protocol Response at Client(role Client, role Server) {
        HTTPV() from Server;
        choice at Server {
                twoHundred_resp() from Server;
        } or {
                fourOfour_resp() from Server;
        }
        rec Y {
                choice at Server {
                        DATE() from Server;
                        continue Y;
                } or {
                        SERVER() from Server;
                        continue Y;
                } or {
                        STRICTTS() from Server;
                        continue Y;
                } or {
                        LASTM() from Server;
                        continue Y;
                } or {
                        ETAG() from Server;
                        continue Y;
                } or {
                        ACCEPTR() from Server;
                        continue Y;
                } or {
                        CONTENTL() from Server;
                        continue Y;
                } or {
                        VARY() from Server;
                        continue Y;
                } or {
                        CONTENTT() from Server;
                        continue Y;
                } or {
                        VIA() from Server;
                        continue Y;
                } or {
                        BODY() from Server;
                }
        }
}
```

```
// test3_A

module Proto1;

local protocol Proto1 at A(role A, role B, role C) {
```

```
        choice at A {
                one() to B;
                three() from C;
        } or {
                four() to B;
                six() from C;
        }
}
```

```
// test3_B

module Proto1;

local protocol Proto1 at B(role A, role B, role C) {
        choice at A {
                one() from A;
                two() to C;
        } or {
                four() from A;
                five() to C;
        }
}
```

```
// test3_C

module Proto1;

local protocol Proto1 at C(role A, role B, role C) {
        choice at A {
                two() from B;
                three() to A;
        } or {
                five() from B;
                six() to A;
        }
}
```

```
// test4_A

module Proto2;

local protocol Proto2 at A(role A, role B) {
        oneTwoThree(int, string) to B;
}
```

```
// test4_B

module Proto2;
```

```

local protocol Proto2 at B(role A, role B) {
        oneTwoThree(int, string) from A;
}
```

```
// test5_B

module Proto3;

local protocol Proto3 at B(role A, role B) {
        rec X {
                choice at A {
                        one() from A;
                        two() from A;
                        continue X;
                } or {
                        three() from A;
                }
                four() from A;
        }
        five() from A;
}
```

```
// test5_A

module Proto3;

local protocol Proto3 at A(role A, role B) {
        rec X {
                choice at A {
                        one() to B;
                        two() to B;
                        continue X;
                } or {
                        three() to B;
                }
                four() to B;
        }
        five() to B;
}
```

```
// test6_A

module Fibonacci;

local protocol Fibonacci at A(role A, role B) {
        rec Fib {
                choice at A {
                        fibonacci(int64) to B;
```

```
                        fibonacci(int64) from B;
                        continue Fib;
                } or {
                        stop() to B;
                }
        }
}
```

```
// test6_B

module Fibonacci;

local protocol Fibonacci at B(role A, role B) {
        rec Fib {
                choice at A {
                        fibonacci(int64) from A;
                        fibonacci(int64) to A;
                        continue Fib;
                } or {
                        stop() from A;
                }
        }
}
```

```
// test7_P

module Negotiate;

local protocol Negotiate at P(role C, role P) {
        propose(string) from C;
        rec X {
                choice at P {
                        accept() to C;
                        confirm() from C;
                } or {
                        reject() to C;
                } or {
                        propose(string) to C;
                        choice at C {
                                accept() from C;
                                confirm() to C;
                        } or {
                                reject() from C;
                        } or {
                                propose(int) from C;
                                continue X;
                        }
                }
        }
```

```
}
```

```
// test7_C

module Negotiate;

local protocol Negotiate at C(role C, role P) {
        propose(string) to P;
        rec X {
                choice at P {
                        accept() from P;
                        confirm() to P;
                } or {
                        reject() from P;
                } or {
                        propose(string) from P;
                        choice at C {
                                accept() to P;
                                confirm() from P;
                        } or {
                                reject() to P;
                        } or {
                                propose(int) to P;
                                continue X;
                        }
                }
        }
}
```

```
// test8_A

module ParallelNotLinear;

local protocol ParallelNotLinear at A(role A, role B, role C)
   {
        par {
                one() to B;
        } and {
                three() to B;
        }
}
```

```
// test8_A

module ParallelNotLinear;

local protocol ParallelNotLinear at B(role A, role B, role C)
   {
        par {
```

```
                one() from A;
                two() to C;
        } and {
                three() from A;
                four() to C;
        }
}
```

```
// test8_C

module ParallelNotLinear;

local protocol ParallelNotLinear at C(role A, role B, role C)
   {
        par {
                two() from B;
        } and {
                four() from B;
        }
}
```

Appendix B

Go Source Code for Benchmark Tests

B.1 Gobble-generated benchmark test code

```
1  package main
2
3  import (
4          "fmt"
5          "log"
6          "sync"
7          "time"
8  )
9
10 func runServer_rec1(Server_rec1_1 *Server_rec1_1) (interface
       {}, error) {
11         Server_rec1_2, _, err1 := Server_rec1_1.
               Receive_firstStep_int()
12         if err1 != nil {
13                 log.Fatal(err1)
14         }
15         var sending_2_1_int int
16         Server_rec1_1_new, err2 := Server_rec1_2.
               Send_secondStep_int(sending_2_1_int)
17         if err2 != nil {
18                 log.Fatal(err2)
19         }
20         return Server_rec1_1_new, nil
21 }
22
23 func runServer(wg *sync.WaitGroup, Server_rec1_1 *
       Server_rec1_1) (interface{}, error) {
24         defer wg.Done()
25         retVal, err1 := LoopServer_rec1(Server_rec1_1)
```

```
26          if err1 != nil {
27                  log.Fatal(err1)
28          }
29          return retVal, nil
30  }
31
32  func StartServer(chans *Channels) *Server_rec1_1 {
33          start := &Server_rec1_1{Channels: chans}
34          return start
35  }
36
37  func LoopServer_rec1(Server_rec1_1_old *Server_rec1_1) (
        interface{}, error) {
38          var retVal interface{}
39          var err error
40          looping := true
41          for looping {
42                  retVal, err = runServer_rec1(Server_rec1_1_old
                        )
43                  if err != nil {
44                          log.Fatal(err)
45                  }
46                  switch t := retVal.(type) {
47                  case *Server_rec1_1:
48                          Server_rec1_1_old = t
49                  default:
50                          looping = false
51                  }
52          }
53          return retVal, nil
54  }
55
56  func runClient_rec1(Client_rec1_1 *Client_rec1_1) (interface
        {}, error) {
57          var sending_1_1_int int
58          Client_rec1_2, err1 := Client_rec1_1.
                Send_firstStep_int(sending_1_1_int)
59          if err1 != nil {
60                  log.Fatal(err1)
61          }
62          Client_rec1_1_new, _, err2 := Client_rec1_2.
                Receive_secondStep_int()
63          if err2 != nil {
64                  log.Fatal(err2)
65          }
66          return Client_rec1_1_new, nil
67  }
68
```

```
69 func runClient(wg *sync.WaitGroup, Client_rec1_1 *
      Client_rec1_1) (interface{}, error) {
70         defer wg.Done()
71         retVal, err1 := LoopClient_rec1(Client_rec1_1)
72         if err1 != nil {
73                 log.Fatal(err1)
74         }
75         return retVal, nil
76 }
77
78 func StartClient(chans *Channels) *Client_rec1_1 {
79         start := &Client_rec1_1{Channels: chans}
80         return start
81 }
82
83 func LoopClient_rec1(Client_rec1_1_old *Client_rec1_1) (
      interface{}, error) {
84         var retVal interface{}
85         var err error
86         var counter int
87         looping := true
88         for looping {
89                 if counter == 1000000 {
90                         return "", nil
91                 } else {
92                         counter++
93                 }
94                 retVal, err = runClient_rec1(Client_rec1_1_old
                      )
95                 if err != nil {
96                         log.Fatal(err)
97                 }
98                 switch t := retVal.(type) {
99                 case *Client_rec1_1:
100                        Client_rec1_1_old = t
101                default:
102                        looping = false
103                }
104        }
105        return retVal, nil
106 }
107
108 func main() {
109        startTime := time.Now()
110        chans := NewChannels()
111        startStructServer := StartServer(chans)
112        startStructClient := StartClient(chans)
113        var newWg sync.WaitGroup
114        go runServer(&newWg, startStructServer)
```

```
        newWg.Add(1)
        go runClient(&newWg, startStructClient)
        newWg.Wait()
        endTime := time.Now()
        totalTime := endTime.Sub(startTime)
        fmt.Println("Execution time = ", totalTime)
}
```

```
package main

import (
        "errors"
)

func (self *Server_rec1_1) Receive_firstStep_int() (*
   Server_rec1_2, int, error) {
        defer func() { self.Used = true }()
        var in_1_int int
        retVal := &Server_rec1_2{Channels: self.Channels}
        if self.Used {
                return retVal, in_1_int, errors.New("Dynamic
                   session type checking error: attempted
                   repeat method call to one or more methods
                   of a given session type struct. Struct:
                   Server_rec1_1; method: Receive_int()")
        }
        in := <-self.Channels.firstStepFromClientToServer_int
        in_1_int = in.Param1
        return retVal, in_1_int, nil
}

func (self *Server_rec1_2) Send_secondStep_int(param1 int) (*
   Server_rec1_1, error) {
        defer func() { self.Used = true }()
        sendVal := SecondStep_from_Server_to_Client_int{Param1
           : param1}
        retVal := &Server_rec1_1{Channels: self.Channels}
        if self.Used {
                return retVal, errors.New("Dynamic session
                   type checking error: attempted repeat
                   method call to one or more methods of a
                   given session type struct. Struct:
                   Server_rec1_2; method: Send_int()")
        }
        self.Channels.secondStepFromServerToClient_int <-
           sendVal
        return retVal, nil
}

```

```
func (self *Client_rec1_1) Send_firstStep_int(param1 int) (*
   Client_rec1_2, error) {
        defer func() { self.Used = true }()
        sendVal := FirstStep_from_Client_to_Server_int{Param1:
            param1}
        retVal := &Client_rec1_2{Channels: self.Channels}
        if self.Used {
                return retVal, errors.New("Dynamic session
                   type checking error: attempted repeat
                   method call to one or more methods of a
                   given session type struct. Struct:
                   Client_rec1_1; method: Send_int()")
        }
        self.Channels.firstStepFromClientToServer_int <-
           sendVal
        return retVal, nil
}

func (self *Client_rec1_2) Receive_secondStep_int() (*
   Client_rec1_1, int, error) {
        defer func() { self.Used = true }()
        var in_1_int int
        retVal := &Client_rec1_1{Channels: self.Channels}
        if self.Used {
                return retVal, in_1_int, errors.New("Dynamic
                   session type checking error: attempted
                   repeat method call to one or more methods
                   of a given session type struct. Struct:
                   Client_rec1_2; method: Receive_int()")
        }
        in := <-self.Channels.secondStepFromServerToClient_int
        in_1_int = in.Param1
        return retVal, in_1_int, nil
}
```

```
package main

// Structs

type Client_rec1_1 struct {
        Channels *Channels
        Used bool
}

type Client_rec1_2 struct {
        Channels *Channels
        Used bool
}

```

```
type FirstStep_from_Client_to_Server_int struct {
        Param1 int
}

type SecondStep_from_Server_to_Client_int struct {
        Param1 int
}

type Server_rec1_1 struct {
        Channels *Channels
        Used bool
}

type Server_rec1_2 struct {
        Channels *Channels
        Used bool
}
```

```
package main

// Channels

type Channels struct {
        doneCommunicatingWithClient chan bool
        doneCommunicatingWithServer chan bool
        firstStepFromClientToServer_int chan
           FirstStep_from_Client_to_Server_int
        secondStepFromServerToClient_int chan
           SecondStep_from_Server_to_Client_int
}

func NewChannels() *Channels {
        c := new(Channels)
        c.doneCommunicatingWithClient = make(chan bool)
        c.doneCommunicatingWithServer = make(chan bool)
        c.firstStepFromClientToServer_int = make(chan
           FirstStep_from_Client_to_Server_int)
        c.secondStepFromServerToClient_int = make(chan
           SecondStep_from_Server_to_Client_int)
        return c
}
```

B.2 Hand-written benchmark test code

```
package main

import (
```

```
        "fmt"
        "sync"
        "time"
)

func serve(first chan int, second chan int) {
        for {
                <-first
                var out int
                second <- out
        }
}

func client(first chan int, second chan int, wg *sync.
   WaitGroup) {
        defer wg.Done()
        var counter int
        max := 1000000
        for counter <= max {
                var out int
                first <- out
                <-second
                counter++
        }
}

func main() {
        start := time.Now()
        first := make(chan int)
        second := make(chan int)
        var wg sync.WaitGroup
        go serve(first, second)
        wg.Add(1)
        go client(first, second, &wg)
        wg.Wait()
        end := time.Now()
        elapsed := end.Sub(start)
        fmt.Println("Elapsed time = ", elapsed)
}
```

Bibliography

[1] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971. URL: http://doi.acm.org/10.1145/356586.356588, doi:10.1145/356586.356588.

[2] Martin Fowler. Twohardthings, 2009. URL: https://martinfowler.com/bliki/TwoHardThings.html.

[3] Andrew Gerrand. Error handling in go, 2011. URL: https://blog.golang.org/error-handling-and-go.

[4] William Harrod. A journey to exascale computing, 2012. URL: https://science.energy.gov/~/media/ascr/ascac/pdf/reports/2013/SC12_Harrod.pdf.

[5] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. URL: http://doi.acm.org/10.1145/359576.359585, doi:10.1145/359576.359585.

[6] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, January 2008. URL: http://doi.acm.org/10.1145/1328897.1328472, doi:10.1145/1328897.1328472.

[7] Raymond Hu and Nobuko Yoshida. *Hybrid Session Verification Through Endpoint API Generation*, pages 401–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. URL: https://doi.org/10.1007/978-3-662-49665-7_24, doi:10.1007/978-3-662-49665-7_24.

[8] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 146–159, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2967973.2968595, doi:10.1145/2967973.2968595.

[9] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992. URL: http://dx.doi.org/10.1016/0890-5401(92)90008-4, doi:10.1016/0890-5401(92)90008-4.

[10] Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised Scribble for Parallel Programming. In *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 707–714. IEEE Computer Society, 2014. doi:10.1109/PDP.2014.20.

[11] Oracle. Class exception, 2016. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html.

[12] Rob Pike. Gobs of data, 2011. URL: https://blog.golang.org/gobs-of-data.

[13] Rob Pike. Go at google, 2012. URL: https://talks.golang.org/2012/splash.article/.

[14] The Go Project. The go programming language specification, 2016. URL: https://golang.org/ref/spec.

[15] The Scribble Project. Scribble-java, 2013. URL: https://github.com/scribble/scribble-java.

[16] The Scribble Project. Scribble-python, 2013. URL: https://github.com/scribble/scribble-python.

[17] Karl Rupp. 40 years of microprocessor trend data, 2015. URL: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/.

[18] The Scribble team. Scribble language reference: Version 0.3, 2013. URL: https://www.doc.ic.ac.uk/~rhu/scribble/langref.html.

[19] TIOBE. Tiobe index for august 2017, 2017. URL: https://www.tiobe.com/tiobe-index/.

[20] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *8th International Symposium on Trustworthy Global Computing - Volume 8358*, TGC 2013, pages 22–41, New York, NY, USA, 2014. Springer-Verlag New York, Inc. URL: https://doi.org/10.1007/978-3-319-05119-2_3, doi:10.1007/978-3-319-05119-2_3.