

Imperial College London
Department of Computing

Session Types Extractor for MPI Programs

by

He Xiao

Supervised by Professor Nobuko Yoshida

Submitted in partial fulfilment of the requirements for
the MSc Degree in Computing Science / Machine Learning of Imperial College London

September 2013

Abstract

The MPI (Message Passing Interface) languages are very popular all over of the world and there are a few theoretical works on data flow analysis in MPI programs [13, 29, 10, 30]. It is both beneficial and risky to use MPI paradigm in parallel computing. The performance of an MPI program can be extremely high if it is used properly; however, considerable amount of energy may be lost due to communication errors in MPI programs. In order to make full use of MPI paradigm and avoid the potential runtime errors, some static type checking is desired. However, to analyse a real MPI source code at compile time is difficult in general because of the lack of easy-to-use analysis tool. A reasonable approach to solve this problem is to develop user friendly analysis tools in the framework of Session Type Theory [18, 34]. Session type theory aims to ensure runtime communication safety through statically analyse the session types of the programs and it also shows interest in analysing MPI programs. There is a previous application [23] which can judge whether an MPI program conforms to a given protocol; however, it suffers from the soundness problem and its underlying basis – the session type theory has evolved. In order to capture the latest parameterised feature [26] of session type theory, this project developed a stand-alone and portable analysis tool for MPI programs. The tool is able to extract parameterised protocol from a C project (which may involve multiple source files), which specifies the communication pattern of the target MPI program. It can not only extract the abstract structure of the source program, but also analyse the basic semantics of MPI operations behind their lexical representations. The semantics analysis is achieved by simulating the execution of target MPI program and it enhances the precision of analysis. Apart from the software application, there are also several theoretical innovations: LFP (least fixed point) of protocols 16, strength of matching 14 and concepts of *ranges* and *conditions* 8. In this report, the design and evaluation of the session type extractor application will be explained in detail.

Contents

1	Introduction	7
1	This project	8
2	Motivation	9
3	Contributions	10
4	Orgnization	10
2	Background	12
1	Session Types	12
1.1	Motivation Of Developing Session Type Theory	12
1.2	Basic Concept Of Session Types	13
1.2.1	basic syntax of session types	13
1.2.2	basic properties of session types	15
1.3	Protocols in Multiparty Session Types	16
1.4	Operational semantics of Session Types	18
1.5	Typing System of Session Types	21
1.6	Evolution of Session Types	23
2	Pabble	25
2.1	Syntax	27
2.2	The advantages of using Pabble	27
3	Message Passing Interface <i>MPI</i>	28
4	static type checking	29
4.1	Motivation of Static Type Checking	29
5	Traditional Analysis Of MPI Programs	29
5.1	MPI-CFG	30
5.2	pCFG	30
6	MPI and Parameterised Session Type Protocols	32
3	Design & Implementation	33
1	Terminologies and Concepts used in design and implementation	33
1.1	Rank Variable	33
1.2	Range and Condition	33
1.2.1	Range	33
1.2.2	Condition	34
1.3	Rank-Related or Non-Rank-Related	34
1.4	Executor and Target of MPI operation	34
1.5	Unilateral MPI operations	35
1.6	Roles	35

1.7	Skeleton of MPI tree	36
2	Toolset	36
2.1	LLVM Infrastructure and Clang Compiler	36
3	Design Goals	37
4	Unsupported cases	37
5	Input and Output of The Program	41
6	Reading Multiple Source Files	42
7	Architecture Of The Session Type Extractor	43
8	In-depth explanation of Range and Condition	45
8.1	AND operation	45
8.1.1	AND operation for <i>Range</i>	45
8.1.2	AND operation for <i>Condition</i>	46
8.2	NEGATION operation	47
8.3	All the other manipulations on <i>Range</i> and <i>Condition</i>	48
9	Data Structures Used In The Application	50
9.1	Abstract Syntax Tree	50
9.1.1	AST Consumer	50
9.1.2	AST Traversal	50
9.2	Communication Tree	51
9.2.1	Construction of <i>CommTree</i>	52
9.3	MPI tree	53
9.4	Optimisation of CommTree	54
10	Simulation	54
10.1	The Design of Simulation	56
10.2	Multiple Roles, Single Tree	56
10.3	Blocking and Deadlock	57
10.4	Unblocking roles and Marking nodes	58
10.5	Optimisation of Simulation	59
11	Extract Condition From Expression	59
11.1	Condition in boolean expression	59
11.2	Condition in Target Expression	60
12	Analysis of Language Constructs In The MPI program	61
12.1	Assignment	61
12.2	Choice	62
12.3	Loop	64
12.3.1	While	64
12.3.2	For	64
12.4	Little summarise of the language constructs	64
13	MPI Primitives	65
13.1	Non-blocking operations	66
14	Matching of MPI operations	67
14.1	Perfect matching and Occasional matching	67
14.2	Techniques for identifying the matching of MPI operations	68
15	Generation of Protocol	70

16	LFP	72
17	The Challenges	73
4	Test & Evaluation	75
1	Compare with the previous project	75
2	Test Cases	80
2.1	Condition Extraction Test	80
2.1.1	Simple <i>conditions</i>	80
2.1.2	Complex <i>conditions</i>	81
2.2	Simple Constructs Test	83
2.3	Unsupported Cases Test	85
2.4	Deadlock tests	87
2.4.1	Classic deadlock	87
2.4.2	Deadlock caused by MPI_Wait	88
2.4.3	Deadlock caused by collective operations	89
2.5	Testing of complex program with multiple source files	91
2.6	LFP examples	92
2.7	Matching examples	92
2.8	Real MPI Program Test: Ring topology	96
5	Conclusion	97
1	Future Work	98

List of Figures

2.1	The structure before delegation	14
2.2	The structure after delegation	15
2.3	Multiparty Session types hierarchical relationships	17
2.4	operational semantics	18
2.5	session initiation: step1	19
2.6	session initiation: step2	19
2.7	session initiation: step3	20
2.8	Some typing system rules	22
2.9	Pabble Syntax	27
2.10	27
2.11	28
2.12	MPI-CFG	30
2.13	constant propagation in pCFG	31
3.1	The skeleton of MPI tree for the sample prgram: We can find that the rank-related choices are ignored and only the most important structures which are visible to every process are stored in the skeleton of MPI tree.	36
3.2	Dataflow of the application	44
3.3	The special range [startIndex..endIndex] is equal to the union of two normal ranges: $\{[0..endIndex],[startIndex..N-1]\}$	46
3.4	CommTree and MPITree in different phases	53
3.5	Pruning of CommTree	54
4.1	The roles generated from code 4.3	81
4.2	The roles generated from code 4.4	82
4.3	Protocol for code 4.5	84
4.4	Output of analysing code 4.6 from cmd	85
4.5	Output of analysing code 4.9	88
4.6	Output of analysing code 4.10	89
4.7	Output of analysing code 4.11	90
4.8	Project structure in test case	91
4.9	Output of analysing code 4.12 using 100 processes	93
4.10	Output of analysing code 4.12 using 5 processes	93
4.11	Output of analysing code 4.13	94
4.12	Output of analysing code 4.14	95
4.13	Output of analysing code 9	95

List of Tables

- 3.1 The basic information of the *CommNodes* used in the application. 52
- 4.1 Characteristics comparison between old and current applications 80

1 Introduction

The computing services have become prevalent in modern life. Every field can benefit from introducing computing devices, and the demands of greater computing power are increasing steadily over time. There are generally two possible ways to improve computing capability: increasing the speed of a single processor or making multiple processors cooperate. Due to the limitation of silicon, the Moore's Law is about to end [22], and it has become increasingly difficult to speed up the hardware with the exponential rate like before. Moreover, the computationally intensive and dynamic tasks such as environmental simulations and simulations in aerospace engineering, which need much more computing power than their counterparts, cannot be performed by a single machine in time so that the results are still valid and useful. Therefore, in the above situations, parallelising the tasks seems to be a rescue. As a result, the parallel computing becomes an increasingly important research area.

In parallel computing, the communication technology is a key component; and therefore it has gained great attentions due to the popularity of parallel computing methodology. There are two classical paradigms for communication: shared address space and message passing. In the former approach, different processors communicate with each other by writing to and reading from the shared memory. In ideal scenario where a CRCW (Concurrent-read, Concurrent-write) PRAM(Parallel random access machine) model is used, the communication cost is negligible but in real life, it may suffer from a variety of overheads issues such as cache thrashing and contention in shared accesses [14, pp. 31,61-62]. More over, due to the uncertainty, it is hard to make an accurate model of the communication costs for shared memory communication. Therefore, the latter message passing approach, which has more predictable costs, gains greater popularity and is massively used in large amounts of parallel applications.

The Message Passing Interface (MPI) is a parallel computing paradigm that adopts the message passing approach. It defines the syntax and semantics of communication functions so that the users can write portable and scalable large-scale parallel programs in C or Fortran 77 [33].

Although there are well-tested and efficient implementations for the MPI library routines. There is no guarantee that the combination of these routines is still safe. So MPI program can have amazing performance if being used properly, but communication error like deadlock and type mismatch may occur if the program is designed carelessly. In communication-oriented applications, the major cost comes from the interactions among processors and communication failure will incur huge costs. Therefore, in order to exploit the benefits provided by MPI thoroughly without worrying about the potential hazards, there is a need to verify the communication safety and reliability.

In contrast, the session type theory and the related implementations not only guarantee the communication safety in concrete applications, but also give a clear representation of the structure of the whole conversation in a highly abstracted level, making it possible to identify and eliminate the

redundant communication in the design phase. Using session type theory, the communication in a program can be abstracted to session types which are written in Scribble syntax [17]; The session types can be analysed separately. If the session type is correct and the program conforms to it, then the communication part of the program will be guaranteed to be correct directly. Furthermore, session-based programming has a comprehensive toolchain to cover every step in product development which facilitates the development of concrete applications.

However, the session-based languages such as SessionJ [21] and SessionC [27] are domain specific languages which introduce new keywords. They are currently available only inside the laboratory; so their user base is greatly smaller than that of MPI. It is not easy to let most of the MPI programmers turn to session-based languages; however, it is quite feasible to use the session type theory to guide the MPI programming so that most of the errors can be detected at compile time. The benefits can be preserved while the defects can be weakened if MPI combines with Session Type Theory properly. This report will explore one of the possible ways to analyse MPI code: extracting the protocol followed by the MPI source code in Scribble syntax, which can be analysed by the Scribble tool later.

1 This project

The aim of this project is the implementation of an independent tool in C++ such that, given a standard MPI program written in C language, it should be able to extract the parameterised session types from the source code and produce the protocol in Scribble syntax. The tool should be able to read the MPI source code from an external file and output the Scribble protocol to an external file. It should be able to analyse the semantics of different language constructs such as choice (E.G. if and switch), recursion (like for and while loop) as well as the basic MPI primitives (will be introduced in detail later in chapter 3, section:13). Some error-prone programming style (ref:chapter 3, section 4) can be detected during the parsing and the warning message will be shown to the console.

2 Motivation

As described in the introduction, both the MPI paradigm and the session type theory have noticeable drawbacks.

The compiler used for compiling MPI C program is mpicc and the static type checking is quite weak. Neither the type mismatch (E.G. process A is sending an integer to process B while B is trying to receive a double) nor the participants missing (E.G. process A is trying to receive an integer from process B while B does not attempt to send any data to A) can be detected. This makes the execution of the MPI program not reliable. When a program is running, we do not know whether it will corrupt in the next moment (it might be the case that the execution is in the erroneous branch but it has not hit the error point). What is worse, even if the program runs perfectly for this execution, we cannot guarantee that it is correct for all the execution path (There might be many execution path, and the error point is not in the current path). The dynamic testing can prove a program is incorrect by giving counter examples, but this approach cannot prove correctness by enumerating because there are infinitely many cases.

For the session type theory, the session based programming languages are mainly designed as extensions of major languages such as Java or C but they do not support all the latest features of the original languages. The programmers need to familiarise themselves with the new syntax and semantics before they can enjoy the benefits of session types; this slows down the growth of session-based languages. The small user base also makes the potential bugs of these domain specific languages hard to detect.

But the above problems might be overcome by combining the advantages of MPI and session type theory. The session type theory and the toolchain can be known by more programmers while the MPI program can gain more safety guarantee. At present, the technique of writing high quality MPI-like code from Session Type protocols (SessionC [27]) has already been developed. However, the technique is quite insufficient for the other way around. There was a project [23] in 2012 which can extract the MPI primitives from the source code; however, the purpose of that extraction is to build rank trees but not to generate protocols; furthermore, the application developed in the previous project does not support parameterised protocol. Therefore, in order to generate the parameterised protocol from the MPI source code, it is necessary to develop a new application.

3 Contributions

The major contribution of this project is the creation of an independent application which can extract the parameterised session type protocol from MPI C programs. It is more scalable than the previous project [23] (more details of the comparison can be found at ref: 1). The extraction of session types is the very first step of the static type checking of MPI programs. And to achieve this overall objective, several milestones are made first:

1. Invent a basic mechanism which supports multiple source files as input.
2. Create a subclass of RecursiveASTVisitor to traverse the AST of the MPI program which is able to find the function recursion.
3. Invent a mechanism for representing and manipulating ranges(intervals) and set of ranges (1.2).
4. Build a mechanism for simulating the execution of MPI programs (10). This mechanism can judge whether a prospective MPI operation can actually happen. It is also able to detect deadlocks caused by type mismatching and absence of interaction participants.
5. Propose the concept of least fixed point for protocols of MPI programs (16).

4 Organization

The remaining report is organised as follows:

- *Chapter 2:* The motivation and basic concept of session type theory are introduced briefly and then the operational semantics and typing system are explained through a brand-new example. The evolutionary history and current work on session types are introduced at the end of this chapter;

- *Chapter 3*: The compiler upon which this application is built is described in detail at first. The required program input and the expected output are introduced and the unsupported cases are listed. Then the architecture of the application is introduced briefly. Then the brand-new representation of interval is exemplified; and how this new design can facilitate the internal representation of processes with specific conditions is explained in detail. The important data structures used by the application are introduced with concrete examples. Then an extensive description of the mechanism used for analysing the language constructs of the source MPI program is given. After that, a brand-new mechanism for simulating the execution of the MPI program is introduced. At last, the problems encountered and the corresponding solutions are also discussed.
- *Chapter 4*: In the test and evaluation chapter, we will first use the toy examples to show the important application features. Then we will test real-world MPI programs. At last, a comparison between the application produced in this project and the application designed in the previous project will be given, showing the aspects at which the application in this project has improved a lot.
- *Chapter 5*: In this last conclusion chapter, the challenges in the static analysis of MPI program will be reviewed and the possible future extensions of the current application is discussed.

2 Background

1 Session Types

1.1 Motivation Of Developing Session Type Theory

As we have mentioned in the introduction part, the communication among processes has become an increasingly important aspect of modern software. As a result, various technologies appear, aiming to increase the reliability of communications while preserving expressive power. However, as some authors suggested in their papers [18, 21], the traditional mechanisms for communications among processes suffer from different problems. For instance, the RMI (Remote Method Invocation) is not very flexible; It can describe a single interaction step well, but it cannot directly abstract the whole scene; To give a concrete example, think about buying some goods in a market. You can ask the seller the price of a specific item and then may or may not bargain. After that, you may or may not decide to buy the item. This process can repeat several times to represent buying multiple goods. There are numerous possible ways to combine the above actions to simulate the buying behaviour of a customer. The RMI API is able to articulate any specific action but not the whole behaviour. The approach of socket programming, on the other hand, is more flexible; However, it often transmits untyped raw byte data, resulting in the lost of communication safety. Socket programming may also use protocols, like the example given by the Oracle's Java tutorial [28]. However, as shown in that tutorial, only server uses a protocol. It makes communication safer in the clients' points of view because server's behaviour is predictable, however, this unilateral conformance to a protocol also makes the server vulnerable to the vicious clients. The lack of compiler support makes the conformance to protocol an option but not a mandatory requirement.

In order to overcome the above-mentioned shortcomings of existing approaches, session type theory incorporates the basic sending/receiving actions with the concepts borrowed from traditional programming language such as, selection/branching, recursion and delegation. Session type is able to represent a series of reciprocal interactions as a single but complete unit. With the typing system and runtime monitor, the session-based programs can be checked both statically and dynamically. It is therefore both sound and expressive.

1.2 Basic Concept Of Session Types

The central idea of session-based programming is the session. A session is a series of interactions between two parties (in classic binary sessions [18]) or among multiple parties (in multi-party sessions [19, 24]). A session type gives a single type to the series of interactions need to be performed in the viewpoint of a participant of the conversation. A session type consists of a sequence of interactions, each of which describes both the direction of the communication (send or receive) and the types of the transmitted data. Any interaction between two parties of a session is conducted over a private channel, preventing the occurrences of interferences. The communication between

two endpoints can happen only if their session types are compatible. Two session types are called compatible with each other if both of them are the co-type of the other [18]. The compatibility of session types among communicating parties can be checked statically to avoid interaction errors at runtime.

1.2.1 basic syntax of session types

Most syntax and semantics of session types correspond naturally to those of object-oriented programming languages. The notations used in this report are borrowed from [18] and [34]. Some basic session-based programming constructs are introduced here, and more complex communication structures can be generated by combining these basic operations.

The session initiation is used to establish a fresh channel over which the subsequent interactions can be performed. It has the form:

request a(k) in P accept a(k) in P (initiation of session)

request a(k) represents the action of requesting the initiation of a new session via public channel ‘a’, a fresh channel *k* will be generated and used by program *P* to perform the later communications. This is similar to ‘new Type()’ in Object-Oriented Programming languages where ‘Type’ is a public name and the expression ‘new Type()’ will return a fresh address in memory through which we can access the object. The difference is that ‘new Type()’ will create a new object which does not exist before; whereas in session based programming, when we call request statement, the participant has already existed, and we only get a new approach to access that participant.

k!(&e); P k?(e) in P (data sending/receiving)

The *k!⟨T⟩* represents to send data of type *T* along channel *k*. The session pattern *k!⟨e₁⟩... k!⟨e_n⟩.k?(e_{n+1})* is similar to the Java’s method invocation ‘*e_{n+1} k.someMethod(e₁, ..., e_n)*’ i.e. first send some data as arguments, and then expect an output.

k ◁ l; P k ▷ {l₁ : P₁ || ... || l_n : P_n} label selection/branching

The label selection and branching are mapped to switch block in programming languages. The selector makes a choice by sending a label to the receiver; The receiver reacts by executing the program which is bound with the received label.

μt...t recursion.

The recursion in session types is just like the recursion in traditional programming languages. Whenever a symbol ‘*t*’ is encountered, the control flow will come to the position labelled by ‘*t*’.

throw k⟨k'⟩ catch k(x) in P channel sending/receiving (delegation)

The channel transfer is similar to transport of value; The difference is that, in a delegation, instead of moving primitive data type like number or String, a communication channel (or more precisely, one end of a communication channel) is moved from one agent to another. E.g.

Before the delegation, A and B can communicate with each other via channel *s*, which has two

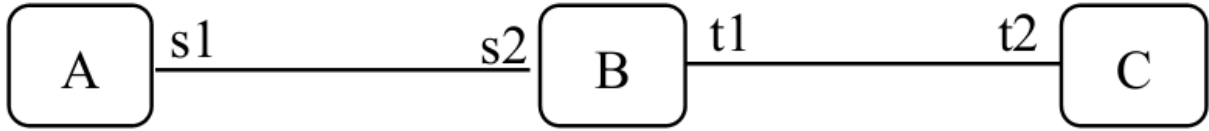


Figure 2.1: The structure before delegation



Figure 2.2: The structure after delegation

ends ($s1$ and $s2$), held by A and B respectively. Similarly, B and C are connected by channel t . During delegation, B performs $s2!(t1)$, while A performs the dual type $s1?(x)[t1/x]$. After the delegation, B granted A to interact with C on his/her behalf. After getting B's port 't1', A can communicate with C as if he were B. C does not need to notice this change.

1.2.2 basic properties of session types

There are several common characteristics followed by all the session types. The properties listed below are summarised by Raymond Hu and Rumyana Neykova in their papers [21], [20] and [24].

1. **Asynchrony:** The asynchrony property is achieved by embedding queues inside the communication channel. The channel is not just a communication medium, it also becomes a buffer with some storage capability. In session initiation, local configurations (the input and output buffers) are also prepared at both endpoints of the channel ([20]). The send operation is therefore non-blocking and the frequency of synchronisation is decreased in this design. One agent can take part in multiple conversations, which might interleave with each other. The transport of session request messages via shared channel is unordered, while the communication in an established session is order preserving due to the FIFO (First in First out) signature of queue ([20]). For example, both Message $m1$ and $m2$ are sent to B by A; $m1$ was sent earlier than $m2$; then it is guaranteed that $m1$ will arrive before $m2$.
2. **Linearity:** According to Hu's definition in his paper [21], linearity means the receiver of a control message should be unique. For example, in the delegation, it is not allowed to send the same channel to multiple destinations. Any kind of concurrent usage of sockets (the endpoints of a channel) are prohibited to avoid the competition for using channels. Think about the situation where both agent D and E hold a socket $k1$ with session type " $!String!int$ ", on the other side of the channel, B has the co-type " $?String?int$ ". D sends a String first and then E also sends a String through socket $k1$; however, B can only receive one String and the next received data is expected to have type int , which finally turns out not to be the case; Therefore, non-linear usage of session sockets will break the integrity of session types and are prohibited by the typing system.
3. **Safety:** Communication safety means that the interactions in a session will never incur any communication errors. For the basic correctness requirement, according to Dezani-Ciancaglini

and DeLiguoro, only data of the expected type is exchanged (2010, cited by Neykova [24]). The communication safety in session types is an extension of this basic requirement and gives more safety guarantees [24].

4. **Liveness:** The liveness property means deadlock-free in communication. Once a session is started, the participants should be able to eventually make progress and finally complete the session without falling into deadlocks. This property can be guaranteed as long as the program passes the type checking [24].
5. **Fidelity:** Session fidelity means the progress of the communications follows the scenario prescribed in the session type[24]. This property indicates the behaviours of an agent in a conversation is predictable.

1.3 Protocols in Multiparty Session Types

In common software development process, the very first step is always producing specifications to capture the user requirements, specify what the systems can do and give clear criteria for evaluating the product. Conforming to the specification does not necessarily mean the product satisfies the user requirements, but violating the specification always signifies failure. Therefore, conforming to predefined specification is a necessary condition for developing a piece of good software. In order to conform to the specification, we need be able to understand the specification thoroughly, which in turn requires the specification is of good quality.

Luckily, all the above-mentioned requirements can be supported by the toolchain of Session Types. In the phase of writing specification, the dedicated protocol description language ‘Scribble’ is used to give clear and verifiable communication specifications for both the system as a whole and each individual agent within the system. The protocol of a session-based programming language is similar to a specification of a traditional software application; The subtle difference is that the former omits the description for local computation due to its communication-oriented nature.

Session Theory uses a global type (protocol) to capture the communication behaviours of agents and their dependencies in an outsider’s viewpoint at the very beginning. The global type is then projected to local types for all members that participate in the conversation. The global type describes the communication behaviours within the whole system whereas the local type concentrates on a specific participant and only depicts the required actions for that particular participant [17]. A toy example from [24] is revised and used to describe the hierarchical relationships between global type, local types and actual programs.

As shown in the figure 2.3, in the global view, agent A needs to pass a String to B, and then agent B needs to transport a String to C. In this simple scenario, after projection, it is clearly shown in the local type T_B that agent B first needs to receive a String from A and then pass another String to C. There are various ways to implement the concrete programs, but whatever the programs look like, they need to conform to their local types. The verification of conformance is performed in the static type checking, which will be explained in detail later. In the example, program B conforms to local type T_B because the String concatenation is still a String. The projection from global type

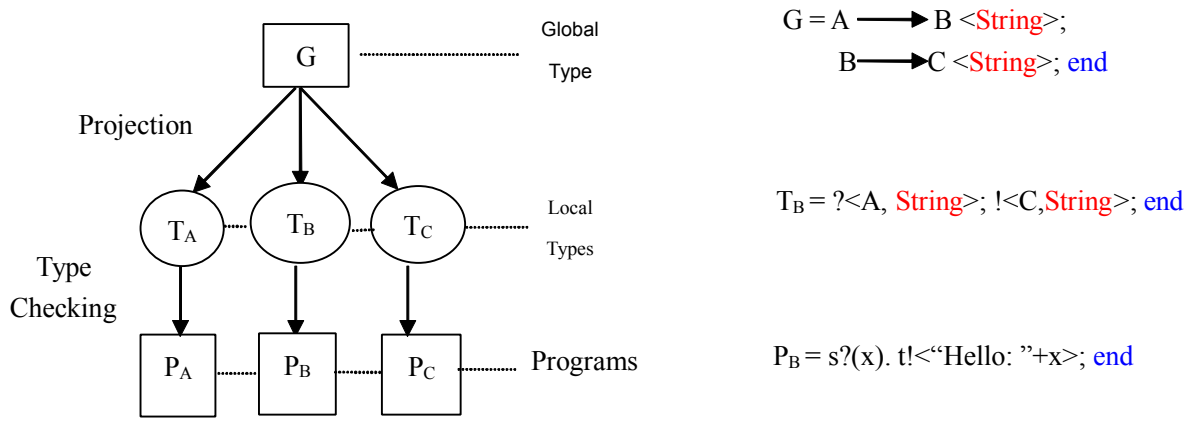


Figure 2.3: Multiparty Session types hierarchical relationships

to local types has such an effect that as long as the programs conform to the local types, they will automatically conform to the global type. Therefore, in type checking, only the local type is needed to be checked against.

The purpose of protocol or global type is to give a high level abstraction so that the overall behaviours of the target system can be understood at a glance. In the situations where every participant plays exactly one role, the terms ‘participant’ and ‘role’ are interchangeable. However, in the dynamic environment, the participants can dynamically join or leave conversations while the roles are still stable and describe predefined behaviours [12]. For example, ‘lecturer-X’ may refer to a fixed role which requires the lecturer to teach students knowledge and answer questions with regard to module X; There might be multiple persons who are competent to do the job and for the same person, there might be multiple modules that he/she can teach. Therefore, it is likely in the real life, one person stops teaching the current module and begins to teach another one and the remaining work required by the old role can be delegated to a candidate. It is also possible for a powerful and energetic person to teach more than more module at the same time. As we can feel in the example, the introduction of role-based session types increases the expressive power and flexibility of session theory dramatically.

1.4 Operational semantics of Session Types

The operational semantics are expressed by the reduction rules listed in figure 2.4. This figure was produced by Hu in paper [20] to illustrate the operational semantics of SessionJ, but the underlying concepts are similar. These operational semantics indicate which operations are allowed to happen and the effects of executing them. They describe the ways in which the system can evolve and therefore can be used to predict the behaviours of the system.

In session initiation phase, the participants involved in the conversation exchange identification information between each other and local IO(input and output) queues are generated by each participant. [Request 1] describes that during the session initiation, a fresh channel with endpoints s and \bar{s} is created. Counterintuitively, both endpoints are created by the requestor in Hu’s work. Because the requestor should not have foreknowledge about the acceptor before the session is completely established, it might be more natural for the session requestor to wait for the reply from the acceptor instead of assuming a predefined port used by the acceptor side. This will also give more freedom to the acceptor with regard to what endpoint to be used.

The details of session initiation are shown by the diagrams step by step.

[Request1]	$\bar{a}(x:S);P \longrightarrow (\nu s)(P\{\bar{s}/x\} \mid \bar{s}[S, \mathbf{i}:\mathcal{E}, \mathbf{o}:\mathcal{E}] \mid \bar{a}\langle s \rangle)$	$(s \notin \text{fn}(P))$
[Request2]	$a[\bar{s}] \mid \bar{a}\langle s \rangle \longrightarrow a[\bar{s}.s]$	
[Accept]	$a(x:S).P \mid a[s.\bar{s}] \longrightarrow P\{s/x\} \mid s[S, \mathbf{i}:\mathcal{E}, \mathbf{o}:\mathcal{E}] \mid a[\bar{s}]$	
[Send]	$s!\langle v \rangle;P \mid s[!(T);S, \mathbf{o}:\vec{h}] \longrightarrow P \mid s[S, \mathbf{o}:\vec{h}.v]$	
[Receive]	$s?(x).P \mid s[?(T);S, \mathbf{i}:v.\vec{h}] \longrightarrow P\{v/x\} \mid s[S, \mathbf{i}:\vec{h}]$	
[Sel], [Bra]	$s\triangleleft l_i;P \mid s[\oplus\{l_i:S_i\}_{i \in I}, \mathbf{o}:\vec{h}] \longrightarrow P_i \mid s[S_i, \mathbf{o}:\vec{h}.l_i]$	$(i \in I)$
	$s\triangleright \{l_j:P_j\}_{j \in J} \mid s[\&\{l_i:S_i\}_{i \in I}, \mathbf{i}:l_i.\vec{h}] \longrightarrow P_i \mid s[S_i, \mathbf{i}:\vec{h}]$	$(i \in I \cap J)$
[Comm]	$s[\mathbf{o}:v.\vec{h}] \mid \bar{s}[\mathbf{i}:\vec{h}'] \longrightarrow s[\mathbf{o}:\vec{h}] \mid \bar{s}[\mathbf{i}:\vec{h}'.v]$	
[Instance]	$\text{def } D \text{ in } (X\langle \vec{v} \rangle \mid Q) \longrightarrow \text{def } D \text{ in } P\{\vec{v}/\vec{x}\} \mid Q$	$X(\vec{x}) = P \in D$
[Arriv-req]	$E[\text{arrived } a] \mid a[\bar{s}] \longrightarrow E[b] \mid a[\bar{s}]$	$(\bar{s} \geq 1) \downarrow b$
[Arriv-msg]	$E[\text{arrived } s h] \mid s[\mathbf{i}:\vec{h}] \longrightarrow E[b] \mid s[\mathbf{i}:\vec{h}]$	$(\vec{h} = h.\vec{h}') \downarrow b$
[Typecase]	$\text{typecase } s \text{ of } \{(x_i:T_i)P_i\}_{i \in I} \mid s[S] \longrightarrow P_i\{s/x_i\} \mid s[S] \quad \exists i \in I. (\forall j < i. T_j \not\leq S \wedge T_i \leq S)$	

Figure 2.4: operational semantics

source: Hu et al. (2010) [20].

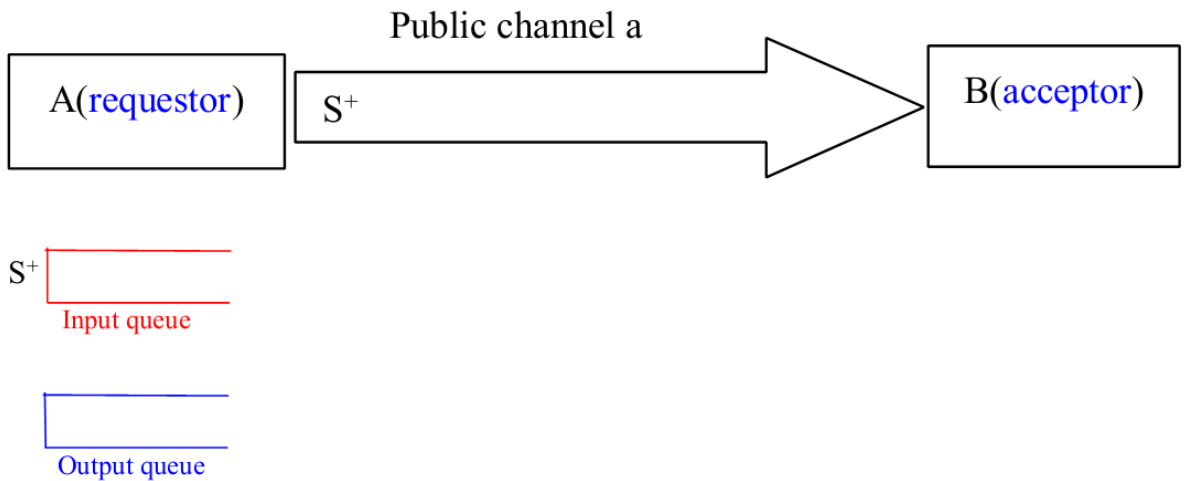


Figure 2.5: session initiation: step1

At first, the requestor A generates its input and output queues and transmits a request message (include the address of its input mailbox and the session type information) to agent B via public channel a . A does not know the address of agent B for this session yet so the output queue is not labelled.

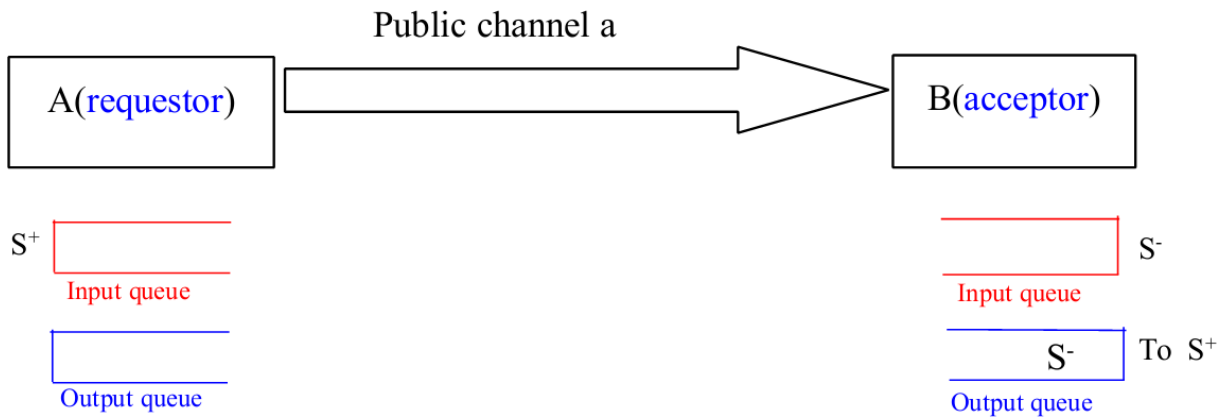


Figure 2.6: session initiation: step2

As shown in figure 2.6, after receiving agent A's session request, agent B accepted and created the local queues. Because the request message contains session type information, agent B is aware of the overall communication structure. Agent B also received A's private socket address, he/she knows where the messages in output queue should go. Agent B then put his/her private address for this session to the output queue and later it will be sent to A.

As shown in figure 2.7, in step 3, the requestor A will get the private address of agent B via its input queue. The address of B will be used when the messages for B need to be sent. After step 3, both agent A and B knew each other and the session has been established.

N.B. S^+ and S^- can be regarded as the addresses of input queues for A and B respectively, while the addresses of output queues are totally private and only visible to the owners.

The communications on the established session are simple. Thanks to the input and output queues, both sending and receiving messages are non-blocking; Each time a new session is established, a new mailbox dedicated for that conversation will be created and the agent can check all his/her mailboxes fairly and regularly. Therefore the same agent can participate in multiple sessions simultaneously without worrying the possibility of deadlocking caused by bad scheduling (Eg. the circular wait problem) (Note that the characteristic of non-blocking receiving is not a nature of session types but just an implementation choice).

The queues or bufferers are used to avoid frequent synchronisation so that the computations in different agents can be performed more independently [16]. In practice, the messages stored in the output buffer are usually sent when the buffer is full or maybe the flush command is executed periodically. In the synchronisation between the sender and the receiver, the message in the front of the sender's output queue will be removed and sent to the receiver; When the message arrives at

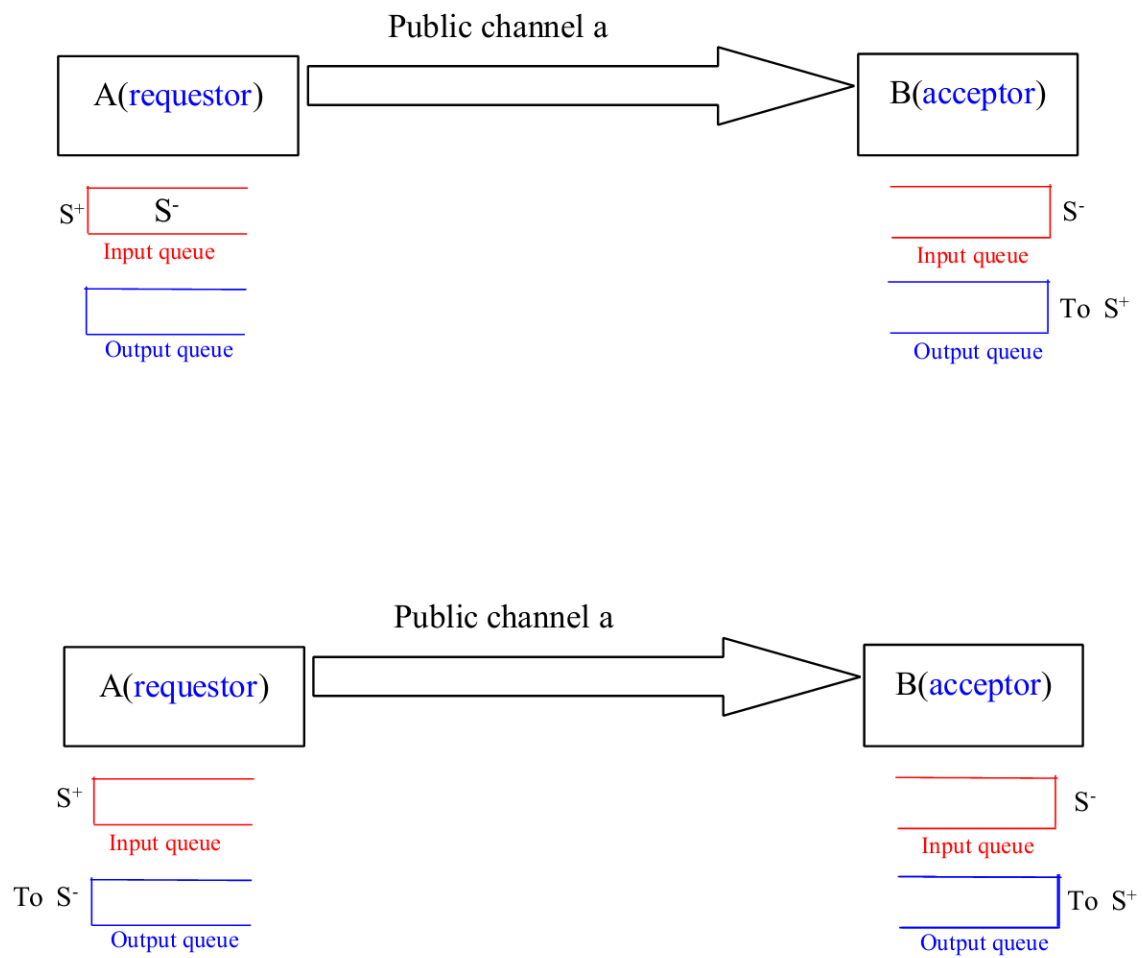


Figure 2.7: session initiation: step3

the destination, it will be appended to the end of the recipient's input queue. Each time a correct action is performed, the corresponding part in the session type is removed to reflect the update.

The transition rule *Sel* and *Bra* are similar to basic data sending and receiving. But instead of transmitting primitive value, a special label is transferred from the selector to the options provider. According to the label being transferred, the proceeding actions are decided by each party independently. For instance, look at the *Sel* rule in diagram 2.4. After selecting label L_i , the label L_i is added to the end of output queue and the remaining session type associated with the channel becomes S_i . After the selection, the selector will continue by executing the program P_i which was associated with the label L_i . For the agent who offers the branching, in receiving a label, the actions opposite to those of the selector are performed, which can be summarised as co-type of selector's session type.

1.5 Typing System of Session Types

The typing system is used for checking the well-formedness of a program. It can report syntax errors as well as enforce the types of expressions are valid and compatible with the expectations according to the definitions in the program. The stronger a typing system is, the more errors can be found in compile time and the fewer errors will be encountered at runtime. Session-based language has a typing system such that the runtime properties such as communication safety, progress and fidelity can be guaranteed through static type checking.

Take the definition in [18], the major typing system is of the form:

$$\theta; \Gamma \vdash P \triangleright \Delta$$

The above formula means the process P conforms to typing Δ under the environment $\theta; \Gamma$. "Sorting Γ specifies the protocols at the free names of P , while typing Δ specifies P 's behaviours at its free channels", the explanation given in [18] clearly shows how to reduce the typing rules.

$$\begin{array}{cc}
\text{[ACC]} \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma, a : \langle \alpha, \bar{\alpha} \rangle \vdash \text{accept } a(k) \text{ in } P \triangleright \Delta} & \text{[REQ]} \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \bar{\alpha}}{\Theta; \Gamma, a : \langle \alpha, \bar{\alpha} \rangle \vdash \text{request } a(k) \text{ in } P \triangleright \Delta} \\
\text{[SEND]} \frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \quad \Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k![\tilde{e}]; P \triangleright \Delta \cdot k : \uparrow[\tilde{S}]; \alpha} & \text{[RCV]} \frac{\Theta; \Gamma \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k?(\tilde{x}) \text{ in } P \triangleright \Delta \cdot k : \downarrow[\tilde{S}]; \alpha}
\end{array}$$

Figure 2.8: Some typing system rules

source: Honda et al. (1998) [18]

In the rules [ACC] and [REQ], the environment specifies the protocol at the free name a is α and $\bar{\alpha}$ respectively. Because free channel k has been bound to a session type α or $\bar{\alpha}$ in the session initiation, there is no free channel k in the program; Δ only concerns the behaviours of P at its free channels, therefore, the entry k should be removed from *Delta*.

In the [SEND] rule, k is a free channel. Therefore, according to the order of executing program, a list of data with types \tilde{S} is output to channel k , followed by the session type α which describes the actions performed by P at channel k . [RCV] rule is similar.

The binary session typing system ensures that in any conversation, the two participants have compatible session types. Informally, two session types are compatible if they are co-type to each other. We use the type with an overline to denote the co-type. E.g. $!\langle T \rangle; \overline{S}$ (sending some data of type T and then co-type of S) is the co-type of $?\langle T \rangle; S$ (receiving some data of type T and then type S); $\&\{L_i : T_i\}$ (branching) is the co-type of $\oplus\{L_i : \overline{T_i}\}$ (selecting).

There are also sub-typing system integrated in the system to support subtypes so that anywhere a super type is expected, the corresponding sub-types can appear. Generally speaking, due to multiple paths may be available at runtime (E.g if-else blocks), and we require the execution preserves the typing; Therefore, we may need a session type to be a sub-type of its updated version (i.e. after an execution, the session type is updated and the new type will become a super type of the old type).

1.6 Evolution of Session Types

Session types theory has existed for almost two decades now. Since the first paper on this theory published, many subsequent extensions were developed, making session types theory more expressive and more robust. In this section, some milestones of session theory will be briefly introduced.

Binary Session Types The papers [31] and [18] established the theoretical basis of session types. The language primitives and basic typing system was introduced in [31]. The paper [18] enhanced the expressive power by introducing the concept of delegation, making the first step towards higher-order session types. However, the syntax of the rule [PASS] in the [18] is too conservative; In some cases, even if the delegation is valid, it just cannot happen. The paper [34] overcame this limitation by introducing the concept of polarised channels. In the updated theory, the two ends of a channel should be explicitly distinguished. Channels become runtime entities created by rule [LINK] and should not appear in the static programs. For example, after some channel k is created in the session initiation phase, the ends k_+ and k_- will be allocated to the two participants connected by k respectively. Any communication happens on the channel k in the later stage should specify which end of the channel is used.

Multiparty Session Types Although many communication patterns can be captured by combinations of binary session types, in many circumstances, it is more elegant and suitable to allow more than two participants communicate in one session. Therefore, in 2008, multiparty asynchronous session types theory was introduced in [19]. A new concept called global types is introduced to describe the shared protocol among all the participants in a multi-party session. The actions required by each participant are described by local types which can be derived by projection of global type. However, the progress property can only be guaranteed within a single session in the system proposed by paper [19]. In a protocol where several sessions interleaved with one another, the global progress cannot be guaranteed. To overcome this limitation, a novel static session type system was introduced in paper [9] to ensure the global progress property in the dynamically interleaved multi-party asynchronous sessions; In the system proposed by authors of [9], after session initiation, every participant will share the private session name and get a channel indexed by its role in the session. When a participant with role p output a message m via its channel, the message m will be labelled by the sender's role p and then appended to the end of the queue of the session. Receive

operation is similar, the role of the receiver will be used to check whether the head of the queue is a message whose recipients list contains the receive operation’s requestor.

Parameterised Session Types The traditional Scribble protocols describe the interactions among different roles explicitly. However, this hard coding mechanism is verbose and inflexible. For example, in a ring topology, to capture a scenario where every process sends an integer to its neighbour in turn, the traditional protocol needs to specify 100 individual interactions if there are 100 processes involved: i.e. we need to explicitly state that process 1 sends an int to process 2, process 2 sends an int to process 3, ..., process 99 sends an int to process 100, process 100 sends an int to process 1. What’s worse, if the number of processes changes, then both the global and local session types need to be modified to reflect the changing environment. The parameterised session type theory overcomes this limitation by introducing the parameterised protocol concept and add that concept into session-based languages. The paper [26] proposes the toolchain for parameterised session C. A variant of Scribble named “Pabble” is developed to write parameterised protocols. The corresponding projection algorithm is also provided to transform the parameterised global protocol into parameterised local protocols. Different processes are classified into a number of logical groups according to their communication patterns. The processes within the same logical group perform the same role and are distinguished by indices. The number of required local protocols is reduced dramatically and the same parameterised protocol can be reused by instantiated with a new number if the number of participants changes. In the earlier example, the scenario can be described by just two lines: process i sends an integer to process $i + 1$ if i is between 1 and $N - 1$, and process N sends an integer to process 1, where N is the number of processes that participate in the conversation.

Multi-role Session Types In order to enforce safe communications, the traditional multiparty session types require the number of participants is fixed when the session starts. In actual applications, any number of participants may dynamically join or leave a session. The new multi-role extension presented in paper [12] makes session types applicable in such dynamic environments. In that paper, the concept of ‘role’ is defined as classes of local behaviours. Each session is equipped with a registration process which maintains a mapping between roles and participants (for each role, there is a corresponding participants set, the members of which all play that role). When an agent asks to join a session via a public channel, its identification will be added to a participants set according to the role it intended to play. The multi-role extension increases the flexibility of session types greatly.

Nested Protocol The paper [11] uses nested protocol to describe sub-sessions. A direct benefit of using nested protocol is to initiate a session only when necessary, and this lazy initiation reduces complexity and resource usage of the program. Nested protocol also allows one to call multiple versions of the same protocol with different arguments, which increases its modularity and flexibility [11].

Reconnection-Based Delegation It is well known that in a delegation, the passive party does not have to notice the change of his/her partner. But in fact, that phenomenon only happens in systems where the indefinite redirection mechanism is used to implement delegation. In the indefinite redirection mechanism, after a delegation, even if the session sender has already delegated the remaining tasks to the session receiver and logically finished, it still needs to redirect the

messages so that the communications between the passive party and session receiver can continue. There is no extra things needed at the passive party side but the overheads at the session sender side are increased. The paper [21] compares and contrasts the indefinite redirection strategy and reconnection based strategy, and proposes two reconnection-based delegation implementations to reduce the delegation overheads.

Toolchain

A set of comprehensive tools is developed to support the applications of session types theory. The toolchain is still developing but has already covers every stage of software development. The protocol description language ‘Scribble’ [17] can facilitate the design of protocols. The latest parametrised protocol allows the designers to get rid of the tedious repetition and customise the protocol conveniently. After writing global protocols, the corresponding local types can be derived automatically by algorithms. There are many examples of using protocols in [16].

There are also many language extensions for session types which can be used in the implementation stage. These include Session Java [21, 20], Session C [27], Session Python [25]. These language extensions have extended syntax and typing system to make session constructs expressible and compilable. These session-based languages are always evolving to reflect the latest achievements in session types theory.

2 Pabble

Pabble (Parameterised Scribble) is a variant of the traditional Scribble language. The original Scribble is not suitable for describing MPI programs due to the issues listed below.

There is only one keyword *recur* to denote loops in the current Scribble language and it cannot represent the number of iterations explicitly. This might be fine in certain situations because in a classic object-oriented programming model, an object is just a passive service provider. An object will perform some actions only if it was asked to do so. For example, in the travel agency scenario, the agency is like an object and it only replies to the customer if the customer made a request at an earlier point. Therefore, it is the customer who dominates the conversation. Therefore, the number of iterations do not need to be represented explicitly. The dominating party tells the dominated party what to do by sending an appropriate label. Therefore the party that dominates the conversation can make a decision to terminate the loop of communications unilaterally.

However, it is not the case for MPI programs. MPI programs typically use “single program multiple data” paradigm. The processes that run the MPI program are team mates but not master and workers. Different processes execute the program independently, and therefore, most of the decisions can be made on its own without consulting others. So it is possible that process 0 decides to send an integer to process 1 three times via a loop while the process 1 is waiting to receive an integer for five times. The number of iterations is significant in this situation however it is not expressible by the current Scribble syntax.

This problem can be solved by using the *Pabble*, which offers the keyword *foreach* to specify the number of iterations. The unpublished paper [26] written by Nick Ng proposed the language *Pabble* and gave an in-depth explanation. The basic concepts will be briefly explained here.

2.1 Syntax

Global Pabble

`global protocol str(role R, ...) { G }`

Global protocol

`G ::= l(T) from R to R;`
`| choice at R { G } or ... or { G }`
`| rec l { G }`
`| continue l;`
`| foreach (b) { G }`

Expression

`e ::= e+e | e-e | e*e | e/e | e%e | -e`
`| e<<e | e>>e | num | i, j, k, ... | N`

Role

`R ::= str | str[h]...[h] h ::= b | e`

Local Pabble

`local protocol str at R(role R, ...) { L }`

Local protocol

`L ::= l(T) from R C; | l(T) to R C;`
`| choice at R { L } or ... or { L }`
`| rec l { L }`
`| continue l;`
`| foreach (b) { L }`

Range expression

`r ::= i : e..e | e..e`

Condition

`C ::= if R | ε`

Type

`T ::= int | float | ...`

Figure 2.9: Pabble Syntax

source: Nicholas Ng and Nobuko Yoshida [26]

The syntax is quite similar to that of the traditional Scribble except the introduction of indices notation and the keyword *foreach*.

2.2 The advantages of using Pabble

To capture the communication scenario using the traditional Scribble protocol is tedious in some certain cases. For example, to describe the communication pattern of circulating information repeatedly in a ring topology, the traditional Scribble protocol will be something like this:

One role is needed for each individual participant. The protocol will be quite long if the number of participants is large. What is worse, the whole protocol needs to be recreated if the number of participants changes. Therefore, the traditional Scribble protocol is not very flexible when facing the changing requirements. But if we observe the traditional Scribble protocol carefully, it is easy to notice that the interactions can be classified into two logical

```

1  global protocol Ring(role Worker1,
2     role Worker2, role Worker3,
3     role Worker4) {
4     rec LOOP {
5         Data(int) from Worker1 to Worker2;
6         Data(int) from Worker2 to Worker3;
7         Data(int) from Worker3 to Worker4;
8         Data(int) from Worker4 to Worker1;
9         continue LOOP; }}

```

Figure 2.10

source: Nicholas Ng and Nobuko Yoshida [26]

groups; in the first group, the $Worker_{i+1}$ receives a message from $Worker_i$; and in the second group, the Worker with largest index sends a message to the $Worker_1$.

After analysing the two logical groups, we can find that every roles in the same group share the same communication pattern. The whole scenario of this ring topology can therefore be represented by five line of Pabble protocol (shown on the left hand side); and amazingly, the contents of the

protocol does not need to be changed when the number of workers changes; the only thing needed to do is to instantiate the protocol with the desired number. Through the explanation of this example, it is quite clear that Pabble is more flexible, compact and reusable. As a result, the Session Type Protocol extractor produced in this project chooses Pabble protocol as its output.

3 Message Passing Interface *MPI*

```

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; }

```

Figure 2.11

source: Nicholas Ng and Nobuko Yoshida [26]

Message Passing Interface (MPI) is a widely used message-passing programming paradigm used for programming in parallel computers [14]. It is designed by a group of researchers from academia and industry [33]. Since the first release in June 1994 [33], it has evolved a lot. At

present, the standard has several popular versions: MPI-1, which emphasizes message passing and has a static runtime environment, and MPI-2, which includes new features such as dynamic process management [33].

An MPI program can either be written in asynchronous or loosely synchronous paradigms [14]. All concurrent tasks can be executed asynchronously in the former; but that also makes the reasoning of programs harder and result in non-deterministic behaviour [14]. In contrast, the latter is a good compromise between the parallelism and predictability. The program written in loosely synchronous paradigm only synchronise when perform interactions, and the tasks can be executed independently between the interactions [14].

Most MPI programs are written using the *Single Program Multiple Data (SPMD)* approach [14]. In this approach, all the processes execute the same source code. They may behave differently because of the different ranks they have. In the **SPMD** approach, there may exist special conditionals (the choices related to the rank variable, like “if(rank==0)”), specifying the process-specific task (E.G. only process with rank 0 can execute the statements in the previous if block,). The application in this project assumes the MPI programs are written in **SPMD** approach.

4 static type checking

4.1 Motivation of Static Type Checking

There exist two major approaches to reason a program: static analysis and dynamic monitoring. The advantage of static type checking is based on its low cost; The more errors can be detected at compile time, the fewer will be encountered at runtime. The runtime monitoring might be easier but once the error happens, it causes loss; the loss caused by communication error can be extremely huge if thousands of or even millions of cores are involved.

However, MPI applications are more difficult to analyse statically because [13, 29] :

1. number of MPI processes are unknown at compile time.

2. the existence of conditional statement enables a segment of the program only being executed by a subset of the processes, while the code that resides outside of the special conditionals should be executed by all the processes.
3. applications use complex arithmetic expressions to define the processes.
4. The meaning of ranks may change according to the MPI communicators that the MPI calls use;
5. MPI provides several non-deterministic primitives such as `MPI_ANY_SOURCE` and `MPI_Waitsome` which makes static mappings between send and receive operations very difficult.

5 Traditional Analysis Of MPI Programs

The authors of [10, 13] suggested a compiler analysis framework to perform the analysis task. In the next two sections, the basic concepts of MPI-CFG (MPI control flow graph) and pCFG (parallel control flow graph) will be briefly introduced.

5.1 MPI-CFG

```
begin program (0)
x=0 (1)
z=2 (2)
b=7 (3)
if (rank == 0) then (4)
x=x+1 (5)
b=x*3 (6)
send(x) (7)
else (8)
receive(y) (9)
z=b*y (10)
endif (11)
f = reduce(SUM,z) (12)
end program (13)
```

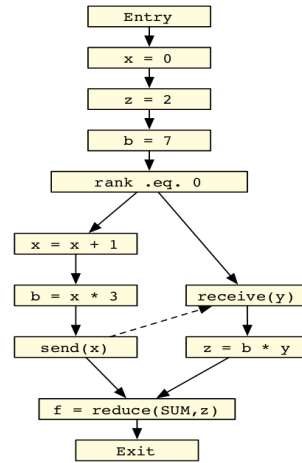


Figure 2.12: MPI-CFG

source: Strout et al. (2006) [30]

The above pseudo code program and the corresponding MPI-CFG (MPI Control Flow Graph) come from paper [30]. The MPI-CFG extends the traditional CFG by adding communication edges, which are represented by dashed arrows. The communication edge comes from the sender and points to the recipient. And the nodes connected by the communication edges are called communication nodes. The ordinary nodes in the MPI-CFG are blocks of local computations which can be done independently while in communication nodes, processes need to communicate with others in order to make progress.

Some interesting analysis can be performed via this MPI-CFG such as the constant propagation. That is a test for evaluating whether a variable can hold some constant value after some execution of the program; It uses forward slicing techniques to test the scope which will be affected by a particular assignment.

The technique of constant propagation may be used to extract the programs executed by a particular process from a standard MPI program. After a rank is being assumed, every occurrence of the variable that stores the rank can be replaced by the constant. If there is any if statement whose condition is comparing the rank of the process with a value, then the evaluation can be performed and the block of programs which is irrelevant to the process can be eliminated. This can be a good start for the later static type checking.

5.2 pCFG

Parallel Control Flow Graph (pCFG) is another extension of traditional CFG for analysing MPI programs.

The pCFG extends the traditional CFG to parallel application and it still uses the corresponding CFG in the dataflow analysis. Each pCFG node is a collection of process sets; and each process set within a pCFG node is associated with a CFG node which means all the processes in that set are currently executing at that CFG node.

In pCFG, all the processes are in the same group at the beginning of the program. Later, when the discrepancies between different processes appear, the processes will be divided into a finite number of groups according to their behaviours. The groups are dynamically updated during the running of the program. A group of processes will be divided into several subgroups in the next step if these processes need to perform different actions.

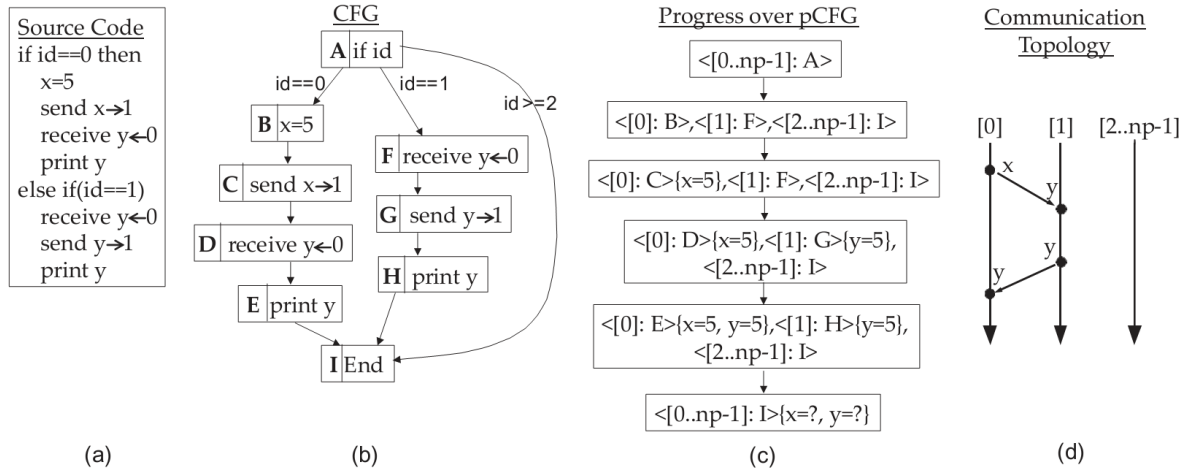


Figure 2.13: constant propagation in pCFG

source: Greg Bronevetsky (2009) [10]

There is a typo in this diagram, after sending the value of x to process 1, process 0 will receive a value from process 1 and assigns that value to its local variable y . The statement **receive y ← 0** should be **receive y ← 1**. The other cascading errors should also be corrected to make it consistent. Despite this small typo, this diagram from [10] illustrates the idea of pCFG clearly.

Graph(a) is the source code while (b) is its control flow graph. Due to statement A is a branching with regard to process id and therefore, process 0 will go to state B while process 1 will further to F, and all the remaining processes will go to the end state I. This splits the original single set into three different subsets. Note that, after process 1 entered the state F, it blocked until process 0 reached state C, at which point, they can communicate and both evolve to the next states. This idea can be useful in verifying progress property of the MPI program.

In this simple example, the progress over pCFG is linear because all the involved processes have a deterministic timeline. This is not always true because in some situations, a process may have different paths to follow in the CFG, resulting in the corresponding pCFG also having multiple outgoing edges. This may be represented as branching logic but due to the space limitation, it will not be explained here.

6 MPI and Parameterised Session Type Protocols

The above two approaches use CFG like technologies to analyse the data flow and control flow of MPI programs. However, there is no clear specification for the output. The output of those algorithms is just some internal CFG representation; therefore it is hard for other programs to make use of the analysis results.

In contrast, the Scribble protocol from Session Type Theory has syntax defined in BNF notation. After the application generate the Scribble protocol for the MPI program, that protocol can be used by other Scribble tools later. This seamless connection increases the re-usability of the output. After getting a correct protocol, the programmer can freely optimise the implementation. As long as the generated protocol is still the same as the original correct one, there is no semantic change in the communication aspect.

Another important reason for choosing Scribble protocol as the analysis out is: there is natural correspondence between MPI program and the Scribble protocol. Both the MPI program and the Scribble global protocol describe the overall scenario for the whole communication. Well, more precisely, both of them contain the overall communication scenario; the MPI program is not just about communication. Such similarity can still be found even if we switch to the finer-grained view. The global protocol can be projected to local protocol which describes what the endpoint agent needs to do, while the actual program executed by each process can also be extracted from the original MPI program.

To sum up, there are strong relationships between MPI program and Scribble Protocol, which makes the combination of them meaningful.

3 Design & Implementation

1 Terminologies and Concepts used in design and implementation

1.1 Rank Variable

The *rank variable* means the variable which stores the rank number for the processes. The name of the *rank variable* can be obtained by analysing the MPI function “*int MPI_Comm_rank(MPI_Comm comm, int *rank)*”. For example, in the code:

```
1 MPI_Comm_rank(MPLCOMM_WORLD, &rank);
```

The rank variable is *rank*.

1.2 Range and Condition

1.2.1 Range

The concept “*Range*” specifies a wraparound interval within which the process rank can be. The rank can be any integer between 0 and N-1 where N is the number of processes. The notation used to represent range is [startIndex..endIndex], which is borrowed from [26]. The notation is wraparound because the start index might be greater than the end index. For example [5..2] denotes all the ranks that are either greater than 4 or less than 3; The two intervals ([5..N-1] and [0..2]) are adjacent in the context of MPI program, therefore it is more compact to shrink them to the single range [5..2]. The wraparound *range* is just an internal representation of the scope of processes for computation convenience, in the output protocol, it is represented as relative index. For example, the interaction *Data(MPI_INT) from MPI_COMM_WORLD[0..N-1] to MPI_COMM_WORLD[1..0]* will be represented as *Data(MPI_INT) from MPI_COMM_WORLD[rank:0..N-1] to MPI_COMM_WORLD[rank+1]* if *rank* is the rank variable.

The notation *range* can also be used to represent the range of any variable with consecutive values. A variable will have consecutive range of values in the conditional block. For example, in the code:

```
1 if (i>1 && i<6){  
2 MPI_Bcast(buf0, buf_size, MPI_INT, i, MPLCOMM_WORLD);  
3 }
```

the variable *i* has range [2..5].

1.2.2 Condition

A *Condition* represents a collection of *Ranges*. It describes the set of all the possible values a process rank can be. For instance, if there is a choice in the program like this:

```
1 if(rank > 2 && rank < 6 || rank > 9){
2 MPI_Send (buf0, buf_size, MPLINT, (rank+1) % numOfProcs, 0, MPLCOMM_WORLD);
3 }
```

Then the condition for the processes that execute the code will be `MPI_COMM_WORLD`{[3..5],[10..N-1]} (suppose the rank variable is associated with the MPI world group). The target of the MPI send operation (the recipients) can be represented by the condition `MPI_COMM_WORLD`{[4..6],[11..0]}

The *Range* and *Condition* are two low level data structures which have been equipped with all the necessary operations such as *AND*, *OR* and *NEGATION*, etc. The mechanism for manipulating these data structures will be detailed in section 8.

1.3 Rank-Related or Non-Rank-Related

The terms *rank-related* and *non-rank-related* will appear frequently in this report. They describe the characteristics of the *conditions* owned by variables or *CommNodes* 9.2. If a *condition* contains every number in *range* [0..N-1], then it is *non-rank-related* because it is related to all the processes regardless of their rank numbers; otherwise, the *condition* is *rank-related* because it only refers a subset of the processes.

1.4 Executor and Target of MPI operation

Most of MPI operations (except some Collective operations) have clear two parties: the party which sends data and the party which receives data. The *executor* of an MPI operation is the party which performs the operation while the *target* is the opposite party of the communication. As shown in the last example, in a sending operation like `MPI_Send`, the executor is the sender and the target is the receiver. For a receiving operation, it is the opposite; the recipient is the executor while the sender is the target. We can also find such relationship in some simple collective operations; for example, the `MPI_Bcast` is just a special form of sending operation where the executor is the broadcaster and the target is all the other processes in the communicator group. Both the executor and the target of an MPI operation can be represented by a *Condition* which has already been introduced in the previous paragraph.

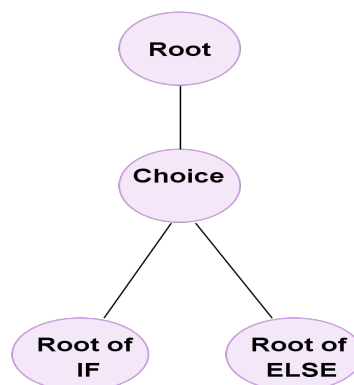
1.5 Unilateral MPI operations

All the basic MPI interactions involve two parties and the collective operations require all the processes of the communicator group to participate. However, for a basic MPI operation, MPI programs usually specify the behaviours of the two parties separately. In the previous program snippet 1.2.2, the `MPI_Send` statement only specify that the executor of that operation needs to send data to the target; it is only an instruction for the senders in this case. However, if there is no corresponding recipients receiving data, the interaction will not happen. In this report, all the prospective MPI interactions are called *unilateral* operations before the communications actually happen.

1.6 Roles

The concept *Role* in the application is a bridge between the participant of MPI operation and the *role* concept *role* in session type protocol. A *Role* in the application has a field of *Range* so that it can represent the executor (or partial executor) of an MPI operation. A *Role* is extracted from the boolean expression of a conditional block. Thanks to the low level operations have been defined well for *Range* and *Condition*, the *Role* can be easily created and split into several smaller *Roles*. As a participant of the protocol, in the simulation phase (details in section 10), the *Role* will traverse the *CommTree* and pick up the relevant MPI operations to execute.

Figure 3.1: The skeleton of MPI tree for the sample program: We can find that the rank-related choices are ignored and only the most important structures which are visible to every process are stored in the skeleton of MPI tree.



1.7 Skeleton of MPI tree

The skeleton of a MPI tree only contains the basic non-rank-related language constructs. It is just like a *CommTree* (details in section 9.2) which has all the rank-related sub-trees eliminated. For example, in the code below:

```

1  if (test==0){
2    if (rank==1)
3      MPI_Send (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD);
4
5    if (rank==0)
6      MPI_Recv (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD, &status);
7  }
8
9  else{
10   if (rank==0)
11     MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
12
13   if (rank==1)
14     MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD, &status);
15 }

```

The skeleton of the MPI tree has the structure shown in the diagram 3.1.

2 Toolset

2.1 LLVM Infrastructure and Clang Compiler

The application developed in this project is an MPI source code analysis tool based on LLVM infrastructure and Clang compiler. *LLVM* is not an acronym but the full name of a project which produces a collection of reusable compiler and toolchain technologies [7]. At first, LLVM was just a research project at the University of Illinois, which aims to provide a modern, SSA-based (SSA: static single assignment) compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages; now, it has included many sub-projects on a wide range, such as code optimisation and code generation [7].

Clang is one of the sub-projects of LLVM. It is a C/C++/ObjC front-end for LLVM compiler

which offers fast compiles and low memory usage [3]. It is suitable for different kinds of clients. If a user wants to debug a program written in C language family, then Clang can offer very expressive diagnostic information and accurate locations of the errors. If the client wants to refactor or perform an in-depth semantic analysis, then she needs to get highly detailed information about the source code; in this case, Clang compiler can generate a complete AST (Abstract Syntax Tree) for the original program [2].

In the application produced by this project, thanks to the golden combination of LLVM back-end and Clang front-end, the lexical analysis and AST generation can be performed by invoking just several library functions. This greatly reduces the amount of low level tedious work, and as a result, the main focus can be put on the high level analysis.

3 Design Goals

As mentioned at the beginning, the final delivered software is expected to extract the parameterised session types from the standard MPI program and produce the protocol in Scribble syntax. It should be able to trace the control flow of program, starting from the first line of the main function of the program being analysed.

After the implementation, all the basic requirements have been satisfied. However, the control flow analysis is constrained by the information available at compile time. Due to the fact that the IF statement is heavily used to distinguish the work of different processes, the analysis of IF statements becomes quite difficult. Due to these difficulties and most importantly, the limited time available, some rare cases are ignored so that the major efforts can be made to improve the quality of the software in its supported area. The unsupported scenarios are listed in the section 4.

4 Unsupported cases

In order to make the program robust and generate the correct outcome, some premises are introduced. The program will check whether the required preconditions are satisfied, if not, then the program will throw an exception and stop immediately. The details of these preconditions will be discussed.

- *Nesting non-rank related choice within rank-specific block when there are MPI operations within the non-rank related choice.*

There are two kinds of choice in MPI program: the rank-related choice and non-rank-related choice. The rank-related choice is the choice whose condition involves rank variable. For example, the expression “*if(rank==0)*” indicates a rank-related choice if the variable *rank* stores the rank number of the process which executes the program. Only process with rank 0 can execute the *then* block of “*if(rank==0)*”; while the non-rank-related choice is the one that has impact on the behaviour of all the processes and does not relate to partial processes directly. For instance, “*if(test==1)*” is the head of a non-rank-related choice given the variable *test* is not relevant to the rank number. If the variable *test* is evaluated to 1, then all the processes will enter the body of the *then* block; otherwise, everyone will go to the *else* block. Now consider the program below:

```

1  if(rank==0){
2      bool nonRank0=false;
3      /*some calculation here*/
4      if(nonRank0){
5          MPI_Recv(buf0, buf_size, MPLDOUBLE, 1, 0, MPLCOMMWORLD, &status);}
6
7      else{
8          MPI_Send(buf0, buf_size, MPLINT, 1, 0, MPLCOMMWORLD);
9      }}
10
11 if(rank==1){
12     bool nonRank1=false;
13     /*some calculation here*/
14     if(nonRank1){
15         MPI_Recv(buf0, buf_size, MPLDOUBLE, 0, 0, MPLCOMMWORLD, &status);}
16
17     else{
18         MPI_Send(buf0, buf_size, MPLINT, 0, 0, MPLCOMMWORLD);
19     }}

```

It is quite clear that the behaviour of this MPI program is non-predictable because the effects of the calculations are not deterministic. If process 0 and process 1 both enter *if* block or they both enter *else* block, i.e. the boolean variables *test0* and *test1* are set to both true or both false, then there will be a deadlock. The program can execute without deadlock in other possible paths. This non-deterministic execution will make the static analysis extremely difficult. And using the conservative analysis, this case will be regarded as deadlock.

In contrast, if the program can be refactored to the code below, then a perfect analysis result can be obtained. Even though the communication pattern is still unknown at compile time, at least we are confident that there will not be any deadlock.

```

1  bool nonRank=false;
2  /*some non-rank related calculation here*/
3  if(nonRank){
4      if(rank==0){
5          MPI_Send(buf0, buf_size, MPLINT, 1, 0, MPLCOMMWORLD);
6      }
7      if(rank==1){
8          MPI_Recv(buf0, buf_size, MPLINT, 0, 0, MPLCOMMWORLD,&status);
9      }
10 } else{
11     if(rank==0){
12         MPI_Recv(buf0, buf_size, MPLINT, 1, 0, MPLCOMMWORLD,&status);
13     }
14     if(rank==1){
15         MPI_Send(buf0, buf_size, MPLINT, 0, 0, MPLCOMMWORLD);
16     }
17 }

```

- *Mixture of rank-related and non-rank-related conditions in boolean expression.*

In complex boolean expressions it is normal to have several smaller expressions connected by logical conjunction (&&) or logical disjunction (||) operator. It is easier to analyse single item. For example, if we see the boolean expression “*if(rank==0)*”, we know its enclosed block can only be executed by the process with rank number zero; for the expression “*if(rank>1)*”, it is clear that it is related to the processes with rank number greater than one. We can even assume that a block with non-rank related condition (like “*if(nonRank==0)*”) is accessible by any process, because in SPMD, every process has exactly the same values for the same non-rank variable.

The author of the previous MPI type checking report [23] designed very complex mechanism for handling the if statement. He enumerated many different combinations of boolean conditions. He distinguished the rank-related conditions from the non-rank-related conditions. For example, according to his report, if a *if* block has condition *if(rank1&&(nonRank&&nonRank))*, then only processes whose rank numbers satisfy the rank-related variable *rank1* can access the block. This mechanism looks rational at first because every process may satisfy the non-rank-related condition while the rank-related condition can only be satisfied by certain processes. However, the author of the previous project only considers the situation in if block but little about the else block. If we consider the negation of the previous condition, which is *!(rank1&&(nonRank&&nonRank))*; we can find that this negation can be transformed to *!rank1||!nonRank||!nonRank*, which can be further deduced to all rank condition (every process might find the negation of a non-rank variable true). And here is the problem, for the processes specified by the rank-related condition *rank1*, they may execute either if or else block; and the issues caused by non-determinism have already been discussed in the first unsupported scenario. In order to remove this non-determinism, the behaviour of mixing rank-related and non-rank-related conditions is forbidden. Although some expressive power is losing, the predictability and accuracy have been enhanced a lot.

- *Multiple possible values for executor or target of MPI operation.*

```

1  if ( test1==0){
2    x=0;
3  } else {
4    x=1;
5  }
6
7  MPI_Bcast( buf0 , buf_size , MPI_INT , x ,MPLCOMMLWORLD) ;
8  MPI_Gather( sendarray , 100 , MPI_INT , rbuf , 100 , MPI_INT , x , MPLCOMMLWORLD) ;
9

```

After the execution of the non-rank related block, there are two possible values for variable *x*, 0 or 1. This makes the output protocol non-deterministic; either process 0 performs a broadcast followed by a gather, or process 1 does the same task. Unfortunately, this scenario cannot be described by simply making a sequence of choices. For example, if we write the scenario like this: *process 0 broadcasts or process 1 broadcasts; process 0 gathers or process 1 gathers* , then the semantic meaning of the original program will be changed because process

1 broadcasts followed by process 0 gathers is not permitted in the original program. Even if we design sophisticated mechanism to capture the semantics of this kind of program correctly, we might face the risk of state explosion (each time a non-rank-related choice is encountered, the possible execution paths will be doubled). Therefore, to make the program manageable, once the executor or target of an MPI operation is found to have multiple possible values, the program will throw an exception and terminate.

5 Input and Output of The Program

The user can offer the program starting arguments in two approaches: through console or via configuration file. The detailed instruction for running the application can be found in the user manual. In this section, only a brief introduction will be given.

There are four arguments can be identified: the MPI source files list, the library including path, the number of processes, and the warning level. The first argument is compulsory while the others are optional. The MPI source file list is the collection of all the MPI source files which are needed for the analysis. If there are multiple files, then the paths need to be separated by “;”. The first path must point to the file which contains the main function. The library including path is the path of the library used in the application. These paths are used to initialise the header search paths of Clang. The user needs to specify the path of MPI’s *include* folder and the OS’s default *include* folder, either by command arguments or configuration file (will be introduced in the user manual).

The number of processes can be any positive number and the presence of an exact number facilitates the simulation of the MPI program. If the user does not specify the number of processes, then a default value 100 is used. The number of processes is necessary because for most of the MPI programs, it is not possible to find a generic protocol capable of describing all the possible executions with different number of processes. As we know, the user can specify the number of processes when executing an MPI program. For example, in the code below:

```
1 if(rank==5){
2   MPI_Send(buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD);
3 }
4
5 if(rank==6){
6   MPI_Recv(buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD, &status);
7 }
```

- If the user set the number of processes to a number less than 6 (the largest possible rank is at most 4), then the program terminates normally, though there is no communication at all.
- If the number of processes is equal to 6, then there will be a process with rank 5 which can execute the MPI Send operation; however, the program will end with a runtime error due to there is no process with rank number 6.
- If the number of processes is greater than 6, then the communication happens successfully and the generated protocol is stable (it will not change due to the increase of the number of processes).

There will be more discussion about the impact of the number of processes on the generated protocol in the section 16 of the current chapter.

The warning level argument is used to adjust the strictness level. It is indicated by the boolean variable *STRICT* in the application code. If the variable *STRICT* is set to false, then the matching of MPI operations is performed more loosely; the matching of MPI operations will not consider the types of the transmitted data; this makes the matching successful even if the data type is not matched (example of data type mismatch: process A is sending integer to B but process B is receiving a double value from A). If the warning is set to strict level, then the data type mismatch will be detected and the matching will be more conservative.

6 Reading Multiple Source Files

It is easy to compile a C project with multiple source files. In the command line, simple type:

```
gcc -o executable sourcefile_1.c sourcefile_2.c ... sourcefile_n.c
```

and then an executable will be produced from compiling all the input c files [4]. We can achieve the same effect using Clang compiler, but when Clang is used to analyse code, the AST will only be created for the main file even if multiple source files are relevant.

Although the Clang compiler only constructs a single AST for each translation unit, it will include the files specified by *#include* statements in the analysed unit. The problem is that, in most cases, instead of directly including other implementation files, only a header file containing function skeletons is used. This is a good programming habit which reduces the degree of coupling; however, when we analyse the main source code, we want to get the bodies of the functions when they are invoked; but for the functions whose definitions are given in other files, we can only get the function specifications from the headers, which prevents the control flow to be traced.

The author of [23] considered to get ASTs for each single file and then combine them. However, it is too difficult to do that in a short time. Therefore, the author of this report invented a simple but working method: when the application starts, the user needs to specify all the arguments, among which is the source files list; the paths of all the relevant implementation files can be extracted from the source file list; subsequently, a copy of the main-file (which contains main function) will be created (with a suffix), and for each implementation file “**somePath/x.c**”, there will be a statement “*#include somePath/x.c*” inserted to the beginning of the main-file’s copy. After the insertions complete, instead of the original main-file, the main-file’s copy which contains the extra *include* statements will be sent to Clang for AST generation. The temporary copy of the main-file will be deleted before the application terminates and the original main-file will not be modified.

7 Architecture Of The Session Type Extractor

The data flow diagram 3.2 depicts the high level structure of the project. At first, the MPI source code is parsed by the Clang compiler instance and the corresponding abstract syntax tree is generated. Then the MPI type checking consumer will traverse the abstract syntax tree, starting

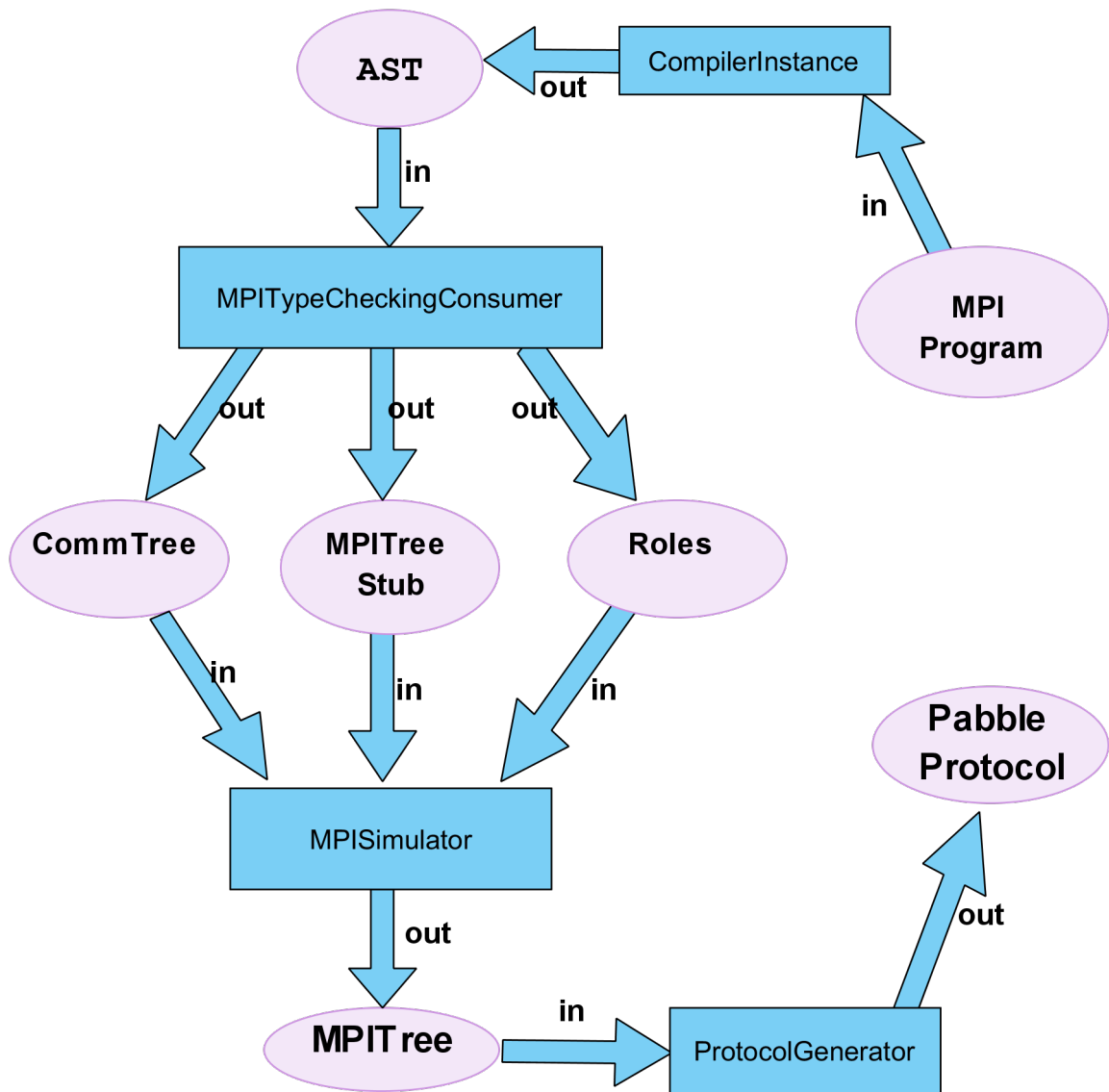


Figure 3.2: Dataflow of the application

from the main function and tracing the basic control flow. During the traversal, the *CommTree*(see section 9.2) containing the unilateral MPI operations and the stub of *MPITree*(see section 9.3) will be constructed. The roles corresponding to the MPI processes will be created from analysing the choice statement and recursion statement. After all the statements in main function has been traversed successfully, the complete *CommTree* and skeleton of *MPITree* have been built. And the *MPI Simulator* will let all the roles created in the AST traversal phase traverse the *CommTree* from root node. The roles will try to match MPI operations and once a successful matching is found, the actually happened operation will be inserted to the *MPITree* skeleton. When the simulation finishes, the construction of *MPITree* will have been completed. Finally, the protocol generator can traverse the *MPITree* to produce the parameterised Scribble Protocol (Pabble).

In the traversal of AST, the MPI type checking consumer will analyse the assignment statement and perform a simple constant propagation in order to increase the accuracy of the generated protocol (Some MPI operation has its target or executor stores in a variable). The details of analysing values of variables and the simple constant propagation will be explained in the section 12.1. In the traversal, different rules are defined to handle different language constructs, the details will be described in section 12.

8 In-depth explanation of Range and Condition

In the terminology introduction part, a high level description of *Range* and *Condition* has already been given. In this section, an extensive description of the basic operations on them will be given in order to understand the remaining part of the report better.

8.1 AND operation

8.1.1 AND operation for *Range*

The *AND* operation for *Range* is defined to model the intersection of two *ranges*. This operation can reflect the semantics of logical operator “&&” as well. For example, in the code below:

```
1 if (rank>2 && rank<7) { ... }
```

Two boolean expressions are combined by the logical operator &&. These two expressions represent range $[3..\infty]$ and $[-\infty..6]$ respectively; and the intersection of these two ranges is $[3..6]$, which is exactly the range denoted by the whole boolean expression ‘*rank*>2 && *rank*<7’. In fact, every boolean expression which involves rank variable needs to perform a *AND* operation with the complete range of processes: $[0..N-1]$ to ensure that the *Condition* of a rank-related variable is always falling into the legal range $[0..N-1]$.

There are several helper methods are defined to facilitate the computation of *AND*.

- *bool Range::isThisNumInside(int num)* is implemented to test whether a given number is inside the current range.
- *bool Range::hasIntersectionWith(Range other)* is implemented to test whether the current range has intersection with the given range.

The implementation is quite simple, if either range has whichever ending point falling inside the other range, then return true, otherwise return false.

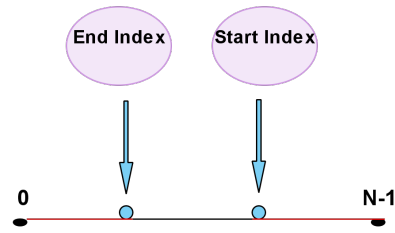
There are several cases in handling the *AND* operation of two ranges. Whichever case it is, an intersection testing will be performed first, if there is no intersection between the two ranges, then the resulting intersection is definitely empty set.

Implementation of AND operation on *Range*: Given $\text{range1} = [s0..e0]$, $\text{range2} = [s1..e1]$, where $s0, e0, s1, e1$ are all integers within $[0..N-1]$ (N is the number of processes) and range1 has intersection with range2 .

- Case 1: If the two operands are both normal ranges (end index \geq start index) or both special ranges (end index $<$ start index), then the intersection of range1 and range2 is $[\max(s0,s1)..min(e0,e1)]$.
- Case 2: One of the two operands is special range and the other is normal range. As shown in the diagram 3.3:

Assume range1 is the special range, then range2 must be a normal range. From the figure 3.3, it is clear that range1 is equal to $\{[0..e0], [s0..N-1]\}$. Then the final intersection of range1 and range2 will be the union of $[s1..min(e0,e1)]$ and $[\max(s0,s1)..e1]$.

Figure 3.3: The special range $[startIndex..endIndex]$ is equal to the union of two normal ranges: $\{[0..endIndex],[startIndex..N-1]\}$.



8.1.2 AND operation for *Condition*

Sometimes, it is not enough to compute a single range. For example, in the code:

```
1 if ((rank<=2 || rank>=5 && rank<=7) &&
2    (rank>=1 && rank<=3 || rank>=4 && rank<=6)) { ... }
```

The processes described by the boolean expression is the intersection of *Conditions* $\{[0..2],[5..7]\}$ and $\{[1..3],[4..6]\}$. The *AND* operation operating on *Conditions* is based on the *AND* operation for *Range*.

Implementation of AND operation on *Condition*:

Algorithm 1 Conjunction of two *Conditions*

```
1: Input Data: Condition  $A:\{ranA_0,\dots,ranA_m\}$ ,  $B:\{ranB_0,\dots,ranB_n\}$ 
2: Result:  $A \cap B$ 

3: function AND( $A$ ,  $B$ )
4:   Condition  $C = \{\}$ ; ▷  $C$  is used to hold the final result
5:   for all  $(ran1, ran2) \in A \times B$  do
6:      $C = C \cup (ran1 \text{ AND } ran2)$  ▷ AND operation for ranges was defined earlier
7:   end for
8:   return  $C$ 
9: end function
```

In the algorithm 1, for each pair $(ran1, ran2)$ in Cartesian Product of Condition A and B , the intermediate intersection $ran1 \text{ AND } ran2$ is computed. The union of all these intermediate intersections is the final intersection of Condition A and Condition B .

8.2 NEGATION operation

The *range* and *condition* concepts are mainly used to describe the intervals of processes. Therefore, the negation of a *range* is designed to be its complementary set in the universal set $[0..N-1]$. The negation of a *condition* 'c' can be obtained by using a recursive function to intersect the negations of all the member ranges in *condition* 'c'.

Algorithm 2 Negation of *Range* and *Condition*

```
1: Input Data: Range ran: [s..e],  $s, e \in \mathbb{N}$ 
2: Result: Condition c which is the negation of ran

3: function NEGATE(ran)
4:   if ran covers all numbers in  $[0..N-1]$  then
5:     return {}
6:   else if  $s==0$  then
7:     return  $\{[e+1..N-1]\}$ 
8:   else
9:     if  $s > e$  then
10:      return  $\{[e+1..s-1]\}$ 
11:    else
12:      return  $\{[0..s-1],[e+1..N-1]\}$ 
13:    end if
14:  end if
15: end function

1: Input Data: Condition A:{ranA0,...,ranAm}
2: Result: Condition X which is the negation of A

3: function NEGATE(A)
4:   if  $A == \{\}$  then
5:     return  $\{[0..N-1]\}$ 
6:   else if  $A == \{ran\}$  then ▷ condition A contains a single range 'ran'
7:     return Negate(ran); ▷ calculate the negation of ran
8:   else
9:     return AND(Negate(ranA0), Negate( $\{ranA1, \dots, ranAm\}$ ))
10:  end if
11: end function
```

8.3 All the other manipulations on *Range* and *Condition*

Similar to the logical reasoning, all the operations on *Range* and *Condition* can also be defined on top of the basic *NEGATION* and *AND* operations. In the implementation of *Diff(Condition other)*, which computes all the processes which are inside the current condition but not present in the condition *other*, this idea is used. As a result, the implementation is simply one line of code: *this-¿AND(Condition::negateCondition(other));*

There are many helper methods being defined to facilitate the manipulation of *ranges* and *conditions*, due to the space limitation, they will not be covered in detail here.

9 Data Structures Used In The Application

9.1 Abstract Syntax Tree

In this subsection, we will briefly introduce the AST consumer and AST traversal algorithm used in this project.

9.1.1 AST Consumer

In order to gather useful information from the AST of the MPI program, an AST consumer has to be implemented. The AST consumer used in the Session Type Extractor is based on two interfaces from Clang's AST library: the *RecursiveASTVisitor* and the *ASTConsumer*; the former specifies the order of traversing the AST nodes while the latter defines the way of extracting information from the nodes. Because the traversing and extracting are highly interleaved, a single class named *MPITypeCheckingConsumer* is chosen to implement both interfaces.

9.1.2 AST Traversal

The recursive traversal mechanism has already been defined in Clang's AST library and all the *Traverse*Stmt* have already been implemented by Clang. Therefore, in most cases, only the corresponding *Visit*Stmt* function needs to be implemented.

When the *HandleTopLevelDecl(DeclGroupRef d)* method is invoked, all the top level declarations are iterated. If the function declaration *main* is found, it will be recorded by the variable *mainFunc*; after all the top level declarations are visited, the method *HandleTranslationUnit(ASTContext &Ctx)* will be called. If there are no compilation errors found, the main function will be visited formally by calling *VisitDecl(mainFunc)*; Then the recursive traversal will be started from the main function.

The high level control flow tracing is achieved by two methods: *VisitFunctionDecl(FunctionDecl *funcDecl)* and *VisitCallExpr(CallExpr *E)*. During the traversal of statements in the main function, if a function invocation is found by the method *VisitCallExpr*, then the basic information of the function will be extracted; if the function has a body defined, then the control flow will be switched to the function being called by executing the code *VisitFunctionDecl*. If the function being called does not have a body, then the function name will be checked; if the function is a supported MPI operation, it will be caught by an appropriate *if* block and some further manipulation can be conducted. The current application can only support invoking user-defined function with return type *void*. Due to the limitation of available time, the mechanism of passing arguments has not been implemented, therefore for the function calls with arguments, the accuracy of analysis cannot be guaranteed.

The analysis of the program is in fact a process of decomposition. The body of any function is just a compound statement, the traversal mechanism implemented by Clang's *RecursiveASTVisitor* can guarantee all the sub-statements in the compound statement being traversed properly, regardless of what statement types the enclosed statements may have.

```
1 int main(int argc, char *argv[])
2 {
3     int a=2;
```

```

4
5  if (a==1) {...}
6  else {...}
7
8  for (int i=0; i<5; i++){...}
9  while (a>0) {...}
10
11 return 0;
12 }

```

In the code snippet shown in 9.1.2, there are totally five statements in the compound statement (the body of main function). The *TraverseStmt* method inherited from *RecursiveASTVisitor* guarantees that each of five enclosed statements can be traversed recursively in turn when we traverse the compound statement.

The statement types of these five enclosed statements are *DeclStmt*, *IfStmt*, *ForStmt*, *WhileStmt* and *ReturnStmt*. When the recursion statement like *ForStmt* and *WhileStmt* are traversed, the conditions are evaluated first and then the body will be traversed by invoking method *TraverseStmt*. For the *IfStmt*, after the boolean condition has been analysed, the body in the *then* block and the body in the *else* block will be traversed in turn.

Each time a user-defined function is called, its name will be put into a function stack. Each time a *ReturnStmt* is encountered, one item will be popped from the stack. When a function name is being pushed onto the stack, a checking will be performed. If the function name going to be inserted can be found on the top of the stack, then a function recursion is found; and an exception will be thrown because the current application does not support function recursion. A special mechanism will be applied to make sure the *void* function without *ReturnStmt* can be handled correctly.

9.2 Communication Tree

The *CommTree* is a tree which contains all the basic language constructs and *unilateral MPI operations* (1.5). If we regard the unilateral MPI operations as unfinished tasks, then the *CommTree* will be a tree of tasks. The language constructs are intermediate nodes and the MPI operations are the leaves. Each node in the tree has a *condition* which specifies what kind of *role* is allowed to visit it. In the simulation, different roles traverse the same *CommTree* from the root node. Every role needs to visit the non-rank-related nodes but the traversal path may vary when the rank-related nodes are encountered. The details of the simulation will be explained in section 10.

9.2.1 Construction of *CommTree*

The *CommTree* is made up of *CommNode*. There are different types of *CommNode*, corresponding to various programming constructs. A complete description of *CommNode* is listed in table 3.1.

The *CommTree* is gradually constructed during the traversal of *AST* of the MPI program. There is a temporary node called *curNode* which references the node that we currently working on. At first, *curNode* references the root node which represents the compound statement in the *main function*. Each time a construct is recognised, the corresponding type of *CommNode* will be generated and inserted to the current node. Each time an intermediate node is inserted to the tree, it will

Node Type	Corresponding Language Constructs	Is Intermediate Node ?
ROOT	Compound Statement	yes
CHOICE	If Statement and Switch Statement	yes
RECUR	While loop and For loop with indefinite iteration number	yes
FOREACH	For loop with fixed iteration number	yes
CONTINUE	Continue Statement	no
WAIT	MPI.Wait operation	no
BARRIER	MPI.Barrier operation	no
SEND	MPI operations related to sending data	no
RECV	MPI operations related to receiving data	no

Table 3.1: The basic information of the *CommNodes* used in the application.

become the new *curNode*. Therefore, the statements surrounded by language constructs can have correct parents. Furthermore, when all the inner statements of a construct have been traversed, the *curNode* will go up one level to avoid the later parallel statements being captured unexpectedly. Therefore, the hierarchy of the *CommTree* can be correctly constructed. To facilitate the traversal of the tree, every node keeps a reference to its right sibling; each time a new node is going to be inserted to *curNode*, a reference of the new node will be passed to the last child of the *curNode* for updating its right sibling reference.

9.3 MPI tree

While *CommTree* is used for holding the tasks, the *MPITree* is created for storing the program simulation result. It can collate the available information so that a better output can be produced by traversing its nodes. The skeleton of *MPITree* is created when the program's *AST* is traversed. During the simulation, if a successful matching of MPI operations is found, then the actually happened MPI operation will be inserted to a proper node of the *MPITree*. The whole process will be depicted in a use case below.

MPI program:

```

1 for(int i=0; i<5; i++){
2   if(rank==0)
3     MPI_Send(buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
4
5   if(rank==1)
6     MPI_Recv(buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD, &status);
7 }
```

Corresponding *CommTree* and *MPITrees* in different phases are shown in figure 3.4. As shown in the diagram, the *CommTree* is constructed via a depth first pre-order traversal. Each time a new node is inserted to the *CommTree*, the index number will be increased by one and stored by the new node. If the node is non-rank-related, then it will be used to generate an MPI node. Assume the node has index x and the generated MPI node is *MPINode* $*mpiNode$, then the tuple $(x, mpiNode)$ will be inserted to the map "*IndexAndMPINodeMapping*". These data structures are very useful in the simulation, which will be introduced in the next section.

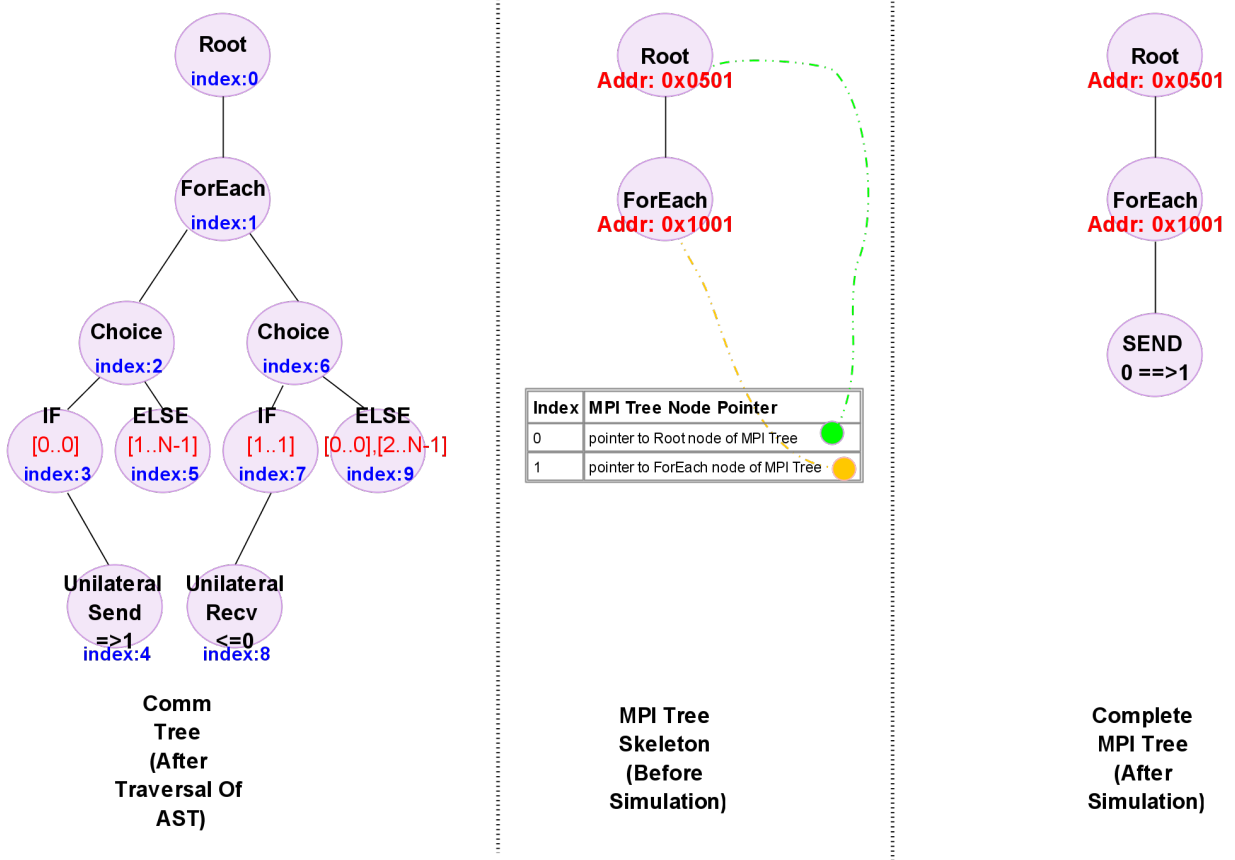
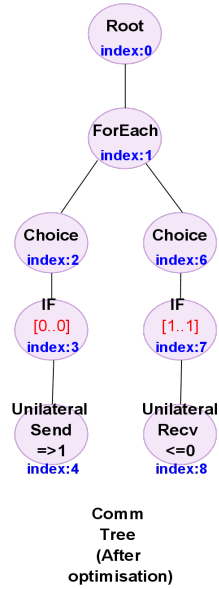


Figure 3.4: CommTree and MPITree in different phases

9.4 Optimisation of CommTree



Sometimes, after the *CommTree* is constructed, the *condition* for some node is *false*, i.e. no processes can visit the node. Therefore, the *CommTree* will be pruned before the simulation starts. This can speed up the traversal of the tree a little. After this optimisation, the *CommTree* in figure 3.4 will become the tree shown in the figure 3.5.

Figure 3.5: Pruning of CommTree

10 Simulation

Recall the analogy between *unilateral MPI operations* and tasks. Only two complementary unilateral MPI operations can match and make the interaction actually happen; this is similar to dividing a task into two parts; only both parts of the task are performed, can the overall task be accomplished. In the example described by figure 3.4, there are two tasks posted on node 4 and node 8 respectively. When a legitimate role visits the task node, it will simulate the process of performing the task by reporting it to the *MPISimulator*. The simulator is responsible for checking whether the newly reported task is complementary with any previous unmatched tasks. If there is no matching, then the task will be inserted to the pending list and becomes one of the unmatched tasks. In the other case, a successful matching is found, then the communication happens and the MPI operation representing this interaction will be inserted to the right *MPINode* in the *MPITree*. The simulator is able to find the accurate *MPINode* because of the data structures which are mentioned in last section. The simulator starts from the node where the communication happens and queries its ancestor nodes recursively until a non-rank-related node is found; then, by looking up the map “*IndexAndMPINodeMapping*”, the desired *MPINode* can be found. After a successful interaction, both tasks participating the interaction will be consumed.

When the simulation starts, every *role* will start visiting the root node. Each time a new node

is encountered, the *role* will compare its own *condition* and the *condition* of the node. There are several possibilities here:

- If the conditions match perfectly, then the *role* may go deeper to visit the first child of the node if the node is an intermediate node, or perform the corresponding task if the visited node is an MPI operation node;
- If these two conditions do not have intersection, then the *role* will go to the right sibling of the current visited node;
- If these two conditions have overlap but not the same, then
 - If the current visited node is a task node, then the task on the node cannot be reported completely because of absence of partial executors. For example, in the code:

```

1 if (rank>1 && rank<6)
2   MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
```

Suppose the *role* with condition $\{[2..3]\}$ visits the *Send* node. However, the whole task requires a role with *condition* $\{[2..5]\}$ to perform. In this case, the old task will be divided into two pieces, the piece with condition $\{[2..3]\}$ will be reported to the simulator while the other piece with condition $\{[4..5]\}$ will preserve the unreported status. The visitors can only perform the unreported tasks in order to avoid the same task to be performed multiple times.

- If the current visited node is an intermediate node, then the *role* (the visitor) will be divided to two new *roles* according to the intersection of the conditions of the *role* and the node. For example, in the previous code snippet, assume the visitor *role* has condition $\{[4..9]\}$. When the *role* visits the root node of the if statement, the *role* will be replaced by two new *roles*: one with *condition* $\{[4..5]\}$ and the other with *condition* $\{[6..9]\}$. The *role* with *condition* $\{[4..5]\}$ will go deeper to visit the children of the current node while the *role* with *condition* $\{[6..9]\}$ will go to visit the right sibling of the current node.

10.1 The Design of Simulation

- The simulation cannot be performed easily by *roles* only due to their short-sightedness. When a *role* finds a unilateral MPI operation, it is not able to decide whether there exists a complementary operation without looking ahead enough nodes. This looking ahead is tedious and involves a lot of repetitions.
- The simulation cannot be accomplished by *MPISimulator* alone. Some MPI operation is blocking and the executor cannot continue until a successful matching is found. Therefore, when a blocking MPI operation is encountered, we cannot simply report it and continue to next node. If we record the conditions for the blocked processes and does not report the tasks related to them until they are unblocked, then there will be many nodes left unmarked even if they have been visited. The order of traversal will be very unpredictable if only an *MPISimulator* is used.

Therefore, the design of simulation in this project separates the responsibilities of *roles* and *simulator* clearly. The *roles* are only responsible for traversing the *CommTree* and report the doable

unilateral MPI operation, leaving the job of judging whether the operation can actually happen to the *MPI Simulator*.

10.2 Multiple Roles, Single Tree

In the phase of *CommTree* construction, a single *CommTree* and probably multiple *roles* are created. All the *roles* share the same *CommTree* so that any update of the tree will take effect without any delay. In the example shown by diagram 3.4, there will be four *roles* generated: $role\{[0..0]\}$, $role\{[1..N-1]\}$, $role\{[1..1]\}$ and $role\{[0..0],[2..N-1]\}$. At first, all these four *roles* will be initialised in such a way that all of them point to the root node with index 0.

The simulation is performed in a while loop and the simulation will not terminate unless a deadlock is found or the root node of *CommTree* has been marked. In each iteration, all the *roles* in the role list will traverse the *CommTree* in turn. In each traversal, the *role* will not return unless it is split up into several new roles or it encounters an operation.

In the example, $role\{[0..0]\}$ will traverse the *CommTree* first. It will continuously go deeper until it reaches the node with index 4 (the *SEND* node). There will be an MPI send operation in the pending list and the node with index 4 will be marked after this traversal of $role\{[0..0]\}$.

Then $role\{[1..N-1]\}$ will perform its first traversal. When it reaches the choice node, it will choose the *ELSE* branch to visit; and it will directly mark the root of *ELSE* branch because that node does not have any children. After several steps, $role\{[1..N-1]\}$ reaches the node 7, which is the root of *IF* branch of the second choice. $Role\{[1..N-1]\}$ will be divided into two new *roles*: $role\{[1..1]\}$ and $role\{[2..N-1]\}$. Note that we have already had a $role\{[1..1]\}$ in our *role list*; however, the previous $role\{[1..1]\}$ will be overwritten by the new one because the newly created *role* is visiting a node whose index is larger (closer to the end of the tree).

When $role\{[1..1]\}$ traverses the tree, it will start from node 7 and find the unilateral receive operation quickly. After that *receive* operation is reported by $role\{[1..1]\}$, $role\{[1..1]\}$ will return and be blocked. But the simulator will conclude that this *receive* operation matches the *send* operation in the pending list and the actually happened operation (process 0 sends an integer to process 1) will be inserted to the *MPI Tree*. As a result of successful matching, the $role\{[1..1]\}$ will be unblocked and the *RECV* node will be marked.

All the other traversals are trivial and will be skipped here. An intermediate node will be marked if all of its children have been marked. Therefore, the root node will finally be marked and the simulation will terminate.

10.3 Blocking and Deadlock

A *role* may become blocked if it encounters a blocking operation. For example, a blocking receive operation or a synchronisation operation like *MPI.Wait* or *MPI.Barrier*. If all the active *roles* (the *roles* which have not finished the traversal of the tree) are blocked, then it is not possible to make any progress; As a result, a deadlock happens and the program terminates with warning message. There is a dedicated class called “*CollectiveOPManager*” to check whether there are deadlocks in collective operations. Because collective operations are blocking, the process participating a

blocking operation cannot continue before finishing; therefore the collective operations can only happen one by one; as a result, the *CollectiveOPManager* will only record at most one collective operation at any time. After the current monitored collective operation is finished, the reference to it will be set to *null*.

Recall that during the simulation, the *roles* will traverse *CommTree* to search MPI operations. If a collective operation is reported by a *role* and the collective operation monitored by the *CollectiveOPManager* is null, then the newly reported operation will be monitored by the *CollectiveOPManager*. If there is a second collective operation found by any *role* before the current monitored collective operation is finished, then there is some process who has not finished the first collective operation but waiting to execute some other collective operation; as a result, neither operation can finish normally and a deadlock occurs.

10.4 Unblocking roles and Marking nodes

Unblocking roles The blocking is not achieved by directly changing the status of the *role* because quite frequently, not all the processes in the blocking *role* can be unblocked after an interaction happens. For example:

```

1 if (rank==0)
2   MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
3
4 if (rank==1 || rank==2)
5   MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD, &status);

```

In the previous code snippet, we can find that the *role*{[1..2]} is the executor of the *receive* operation. However, after the interaction (process 0 sends an integer to process 1) happens, only partial processes are unblocked, therefore, we cannot directly unblock the whole *role*; instead, we create a new *role*{[1..1]} to reflect the fact that process 1 is able to continue.

Marking nodes We have known that an intermediate node will be marked if all of its children have been marked. We need to define the base case to make the mechanism works.

For a task node, whether it should be marked depends on whether all the tasks on it have been done. Each time an interaction happens, the executor conditions of both involving operations will be updated accordingly. If the executor of an operation can be ignored, then the task has been accomplished and the node it inhabits will be marked. In the example code shown in paragraph of *Unblocking roles*, if there is another statement at the end:

```

1 if (rank==0)
2   MPI_Send (buf0, buf_size, MPI_INT, 2, 0, MPLCOMM_WORLD);

```

then the task of receiving integer from process 0 will be done (the executor processes of the task are process 1 and process 2 and both of them have accomplished the task). So finally the *MPI_Recv* node will be marked.

10.5 Optimisation of Simulation

The simulation can be optimised by emptying the role list in synchronisation point (like *MPI_Barrier* node or other collective operation nodes). Because a collective operation node cannot be marked

unless it has been visited by all the processes, we can just put a single role with *range* [0..N-1] to the *role* list and continue simulation.

11 Extract Condition From Expression

There are different kinds of expressions, and those having numeric values or range of numeric values can be represented by *conditions*. In this section, the technique of extracting *conditions* from expressions will be introduced. The most basic cases will be introduced first so that the more complex structures can be analysed by method of *divide and conquer*. The analysis of MPI operations in complex language constructs can be greatly simplified after the conditions related to the constructs are extracted.

11.1 Condition in boolean expression

- ***Evaluable expressions***

If the boolean expression can be evaluated as an integer or boolean value by *Clang*, then let *Clang* evaluate the expression. If the expression is a non-zero constant or boolean *true*, then just return the full *range*:{[0..N-1]}. If the expression is evaluated to zero or *false*, then return empty condition {}.

- ***Non-rank-related Variables***

If the expression is a non-rank-related variables, then the value of it is non-deterministic and it is likely to be true, therefore a complete *range* {[0..N-1]} will be returned.

- ***The expression with parenthesis***

The *condition* of its sub-expression (the inner expression without parenthesis) will be returned.

- ***The negation expression***

The *condition* for the sub-expression without negation operator will be computed first and then use the negation function of *condition* to get the resulting condition.

- ***Binary expressions***

1. ***operator is '&&'***

Get the *conditions* from its left hand side (LHS) and right hand side (RHS) expressions. Return the intersection of them.

2. ***operator is '||'***

Get the *conditions* from its LHS and RHS expressions. Return the union of them.

3. ***both LHS and RHS are variables***

Return complete *range*{[0..N-1]}.

4. ***both LHS and RHS are not relevant to rank variables***

Return complete *range*{[0..N-1]}.

5. *One rank-related variable and one number*

There is a dedicated method defined to handle the operation between a rank-related variable and a number. For example, the expression $rank \geq 2$ will produce a condition $\{[3..N-1]\}$. Here N can be arbitrary instantiated number (the number of processes has been given as a program argument).

6. *all the other cases*

To predict conservatively, every uncovered cases are likely to be true, so return the complete $range\{[0..N-1]\}$.

11.2 Condition in Target Expression

The target expression refers to the expressions representing targets of MPI operations. For example, in code snippet:

```
1 if (rank>0 && rank<6)
2   MPI_Send (buf0, buf_size, MPI_INT, rank+1, 0, MPL_COMM_WORLD);
```

The target of this MPI_Send operation is $rank+1$, which is an expression which cannot be directly obtained. However, in this context, the rank related variable $rank$ has already been bound to $\{[1..5]\}$, therefore, $rank+1$ can be obtained by simple arithmetic calculation.

To extract *conditions* from target expression is similar to that from boolean expression. However, the comparison expressions and pure boolean values are not allowed.

The main possibilities are listed here:

- *Constant number*

If the expression can be evaluated as a constant number x , then the condition returned will be $\{[x..x]\}$

- *Variable*

If the variable has been associated with a *condition*, then just return that *condition*.

- *binary operators*

If the target expression are combined by a binary operator, then use the following rules:

1. *+ or - operators*

Get the *conditions* for LHS and RHS. If neither LHS nor RHS is a number, then throws an unsupported operation exception. Otherwise, get the *condition* for the non-number expression and do the corresponding arithmetic manipulation on the derived *condition*, using the number-expression as the operand. To add a number on a *condition* is easy: for each of the *ranges* in the *condition*, add the number on its *starting position* and *ending position*.

2. *% operator*

Currently, the % operation is only supported when the right operand is a number equivalent to N. The calculation is similar to + and - operations.

3. / operator

Currently, the division operation is only supported when both LHS and RHS are numbers. After the quotient q is derived, the condition $\{[q..q]\}$ will be returned.

4. All the other cases

For all the other situations, an exception will be thrown because the targets of MPI operations needs to be clear (The `MPI_ANY_SOURCE` is not supported in the current application).

12 Analysis of Language Constructs In The MPI program

12.1 Assignment

Assignment is frequently used in any programming language. It is also vital in the analysis of MPI programs; the target of MPI operation is often not a simple constant. For example, in a code snippet like this:

```
1 left_neighbor = (rank-1 + N)%N;
2 right_neighbor = (rank+1)%N;
3
4 MPI_Isend(&my_rank,1,MPLINT,left_neighbor,tag,MPLCOMM_WORLD,&reqSend);
5 MPI_Irecv(&data_received,1,MPLINT,right_neighbor,tag,MPLCOMM_WORLD,&reqRecv);
```

The targets of `MPI_Isend` and `MPI_Irecv` are both variables. However, the values of the variables are deterministic at the time when the operations happen: both *rank* and *N* have deterministic values, and the neighbour variables are only dependent on *rank* and *N*. Therefore, the assignment to variables is also analysed to prepare for the potential usage.

It is a trade-off between the accuracy and performance. It is hard to estimate whether a variable will be used as target of an MPI operation. It seems that any integer variable can be used as target. But luckily, the assignment only needs to be analysed once even if it is in a loop.

Sometimes, we may encounter such a situation:

```
1 left_neighbor = (rank-1 + N)%N;
2 right_neighbor = (rank+1)%N;
3
4 if(rank==1)
5     MPI_Isend(&my_rank,1,MPLINT,left_neighbor,tag,MPLCOMM_WORLD,&reqSend);
6
7 if(rank==0)
8     MPI_Irecv(&data_received,1,MPLINT,right_neighbor,tag,MPLCOMM_WORLD,&reqRecv);
```

When we evaluate the *condition* of neighbours, the *condition* of *rank* is $\{[0..N-1]\}$. However, when these variables are used, the context changes and the *conditions* of *rank* shrink; as a result, the previous values of neighbour variables are not valid any more. To solve this problem, an expression string is associated with the *variable* if its *condition* depends on some changeable variable. Whenever the actual condition of a *variable* is going to be used, this expression string will be checked; if the string is not empty, then the value of the *condition* will be recomputed from the expression string (the function of extracting *condition* from expressions have been introduced in section 11.2).

12.2 Choice

The choice construct is both interesting and complex. Some structures of choice statement are syntactically correct but error-prone. To ensure the analysis is stable and accurate, these cases are not supported, which has been explained in section 4.

However, with the help of the concepts *range* and *condition*, the analysis of choices becomes extremely easy for most of common cases.

- **Arbitrary number of combinations of sub-conditions**

The *condition* for a compound boolean expression composed of arbitrary number of sub-expressions with the same nature (rank-related or non-rank-related) can be computed recursively: the *condition* of the simplest expression can be computed; and combining boolean expressions via `&&` and `||` can be simulated by the $AND(Condition\ cond1, Condition\ cond2)$ and $OR(Condition\ cond1, Condition\ cond2)$ respectively.

An example of this case:

```
1 if(rank==0 || rank>7 && (rank<N-5 || rank==N-2)) { ... }
```

- **Arbitrary number of nesting**

In a nesting IF statement, both inner and outer *if* statements are related to *condition*. If a process can enter the inner *if* block, it must first enter the outer one. Therefore, the inner *if* must first obey the *condition* of the outer one. This can be achieved easily by using the *AND* operation on *conditions*. For example:

```
1 if(rank < 7) {  
2   if(rank==2 || rank>4) { ... }  
3 }
```

The inner *if* has the *condition* described by expression “ $rank < 7 \ \&\& \ (rank == 2 \ || \ rank > 4)$ ”, which is $\{[2..2],[5..6]\}$.

To make the mechanism more generally, in any nesting environment, the inner constructs will inherit the condition from its intermediate wrapper constructs. To implement this mechanism, whenever a child node is inserted to the *CommTree*, its *condition* will be updated by intersecting with the *condition* of its parent node.

- **Condition in ELSE block** The *condition* in *else* part is just the negation of that of *if* part, therefore, it can be computed easily via invoking the $Negate(Condition\ cond)$ function.

12.3 Loop

12.3.1 While

The *while* statement is the most basic recursion construct. After the *condition* of *while* statement is extracted, a *CommNode* with type “RECUR” and extracted *condition* will be inserted to the *CommTree*. Then the body of the *while* statement will be traversed.

12.3.2 For

The *for* statement is a special form of *while* statement which may have clear iteration number. The structure of a *for* statement is like this:

```
1 for(initialising statement; conditional expression; updating expression)
```

We can get the starting position and ending position candidates from analysing the initialising statement and conditional expression. If the *condition* in the conditional expression does not contain the initial value of the target variable, then the condition of the *for* statement cannot be satisfied even in the first iteration; as a result, the whole *for* statement will be ignored and is not inserted to the *CommTree*. The updating expression decides whether the *for* loop has a definite number of iterations. For example, in the code:

```
1 for(int i=0; i<5; i--){...}
```

this might be an infinite loop and does not have a definite number of iterations. In contrast, after analysing the code below, it can be known that the *condition* of variable *i* in the *for* loop is $\{[0..4]\}$ and the number of iterations is 5.

```
1 for(int i=0; i<5; i++){...}
```

12.4 Little summarise of the language constructs

The rules for analysing different language constructs are defined separately and invoked during the *AST* traversal in order to construct *CommTree* and skeleton of *MPITree*. Some variable and *condition* mappings are only valid inside certain area; for example, the variable *condition* mapping generated from boolean expressions of *if*, *for* and *while* statements are only valid inside the body of these constructs. The temporary mappings will be removed after leaving the valid area. This removal is achieved by using two data structures, a *variable-condition* map and a *temporary-variable-condition* map. Each time a *if*, *for* or *while* statement is encountered, the newly defined variables can be found through analysing the headers. Then the old values for these variables will be backed up; and then the entries in the formal mapping can be updated. After the body of the construct is traversed, the formal mapping will be restored to the state before visiting the construct by reading the backups.

13 MPI Primitives

There are more than three hundred MPI functions being defined in MPI-V2.2 [8]. Due to the limited time, the session type extractor application can only analyse several most important routines. Among the supported functions, the API of some typical operations are listed here:

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
- `int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
- `int MPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

The arguments in the function specifications have these meaning:

1. **buf, buffer, sendbuf, recvbuf:** these are the buffer addresses used in the procedures. For example, in `MPI_Recv`, it is the receive buffer.
2. **count:** the number of elements in the buffer.
3. **datatype:** the type of the transmitted data.
4. **root, dest, source:** *root* is the executor of the collective operations like `MPI_Bcast` and `MPI_Reduce`, while *dest* and *source* are the target processes of the corresponding operations.
5. **op:** *op* is the MPI operator.
6. **request:** the request object which represents the process which needs to wait until the non-blocking operation finishes; **status:** the state of whether the pending operation has finished.
7. **comm:** *comm* is the communicator group. The supported communicator group in the application is `MPI_COMM_WORLD`.
8. **tag:** *tag* is the message tag.

13.1 Non-blocking operations

`MPI_Isend` and `MPI_Irecv` are two non-blocking operations. The calling of a non-blocking operation may return even if it is not semantically safe to do so [14]. It has less overhead than blocking operation but in order to ensure the safety of sensitive data, it is usually used together with check-status operations or `MPI_Wait` operations [14].

There is a dedicated mechanism for handling MPI non-blocking operations. The non-blocking operations will not be blocked on the operations but may be blocked by synchronisation nodes like `MPI_Barrier` or the `MPI_Wait` if the request object matches. In order to simulate this process, the semantics are mimicked. When a `MPI_Wait` function is encountered, the request variable name will be extracted and then the non-blocking operation with that request object will be searched from the *CommTree*: the search will start from the most recently inserted node and go back step

by step; if an MPI.Wait operation who has the same request object is found, then the search will terminate and the *MPI.Wait node* will not be inserted to the tree (if the process can pass the previous wait operation, it definitely can pass this one); if the MPI operation with the same request object is found, then the MPI.Wait node will be inserted to *CommTree*, and the operation will also record the address of the *wait node*. Therefore, in the simulation, when a non-blocking operation visits a *wait node* with the same request name, the operation will be blocked. If the non-blocking operation actually happens, then the simulator will inform the *wait node* so that the *role* (the processes performing the non-blocking operation) waiting in the *wait node* can be unblocked. If the non-blocking operation finishes before the *role* visits the wait node, the *wait node* will still be informed so that it will not block the *role*.

14 Matching of MPI operations

14.1 Perfect matching and Occasional matching

A perfect matching is a stable matching. Once the matching can be found at a configuration where the number of processes is N , then the matching will always be found for all the configurations where the number of processes is greater than N .

Listing 3.1 : Example Of Perfect Matchings and Occasional Matchings

```

1 //perfect matching examples:
2
3 //sender and receiver are both bounded variables
4 //the conditions represented by the expressions are also bounded
5 if(rank==5){
6     MPI_Send (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD);
7 }
8
9 if(rank==6){
10    MPI_Recv (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&status);
11 }
12
13 if(rank== N/2){
14    MPI_Send (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD);
15 }
16
17 if(rank== N/2+1){
18    MPI_Recv (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&status);
19 }
20
21 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
22 //the ranks of sender and receiver are both represented by constant number
23 if(rank==0){
24    MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
25 }
26
27 if(rank==1){
28    MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD,&status);

```

```

29 }
30
31 ///////////////////////////////////////////////////////////////////
32 ///////////////////////////////////////////////////////////////////
33 //only matched if N is equal to 28
34
35 if(rank==N/4){
36     MPI_Send (buf0 , buf_size , MPI_INT, rank+1, 0, MPLCOMM_WORLD);
37 }
38
39 if(rank==8){
40     MPI_Recv (buf0 , buf_size , MPI_INT, 7, 0, MPLCOMM_WORLD,&status);
41 }

```

In the code snippet 3.1, all the examples except the last one are perfect matchings and will be finally stable with the increment of process quantity. In contrast, in the last example, the matching only happens when the number of processes is 28.

14.2 Techniques for identifying the matching of MPI operations

The MPI operations can be classified into two categories: collective operations and non-collective operations. No matter which category an operation belongs to, there exists a generic mechanism for judging whether it can match with another operation. The mechanism is shown in algorithm 3.

Algorithm 3 Algorithm for checking whether two MPI operations are complementary

```
1: Input Data: MPIOperation op1, op2
2: Result: boolean value indicating whether op1 is complementary operation of op2

3: function COMPLEMENTARY(op1, op2)
4:   if op1 and op2 have different categories then
5:     return false
6:   else if op1 and op2 are in different branches then
7:     return false;
8:   else if data types of op1 and op2 are different then
9:     return false;
10:  else if op1 is collective operation then
11:    if function names of op1 and op2 are different then
12:      return false;
13:    else if op1 is reduce operation && reduce operators of op1 and op2 are different then
14:      return false;
15:    else if executor conditions of op1 and op2 are different then
16:      return false;
17:    else
18:      return true;
19:    end if
20:  else
21:    if op1 and op2 are both sending operation or both receiving operation then
22:      return false;
23:    else
24:      return true;
25:    end if
26:  end if
27: end function
```

In the algorithm, the phrase “*in different branches*” means in different paths of a non-rank-related choice. For example, in the following code snippet 14.2, the process with rank number 0 is not possible to communicate with process 1: for the non-rank-related choice, both processes will enter the same branch but in any branch, there is only a single unilateral operation, which cannot happen anyway.

```
1 if (nonRankVar==2){
2   if (rank==0)
3     MPI.Send (buf0 , buf_size , MPLINT , 1 , 0 , MPLCOMMWORLD);
4 } else {
5   if (rank==1)
6     MPI.Recv (buf0 , buf_size , MPLINT , 0 , 0 , MPLCOMMWORLD,&status);
7 }
```

15 Generation of Protocol

The syntax and semantics of *Pabble* protocol is very similar to those of traditional *Scribble* protocol. Therefore, in order to generate a valid protocol which is compatible with the current tool-chain, the implementation of the protocol generating method is heavily based on the BNF specification of *Scribble* language [32]. There will be adequate examples of generated protocols in the *Test & E*

Evaluation chapter, so only some high level optimisation of the output protocols will be discussed here.

In the simulation, in order to ensure the accuracy, the traversal of the *CommTree* and matching of MPI operations are quite strict. As a result, a single interaction in the source code might be split up to several pieces. For example, in code snippet 15:

```
1 if (rank>=5 && rank<=7){
2   MPI_Recv (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD,&status);
3 }
4
5 if (rank>=6 && rank<=8){
6   MPI_Send (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD);
7 }
```

Processes with *range* [5..7] will be blocked on the MPI_Recv node. Only Process 8 can escape and go to visit the MPI_Send node. After process 8 sends the data to process 7, process 7 can be unblocked and subsequently it will go to unblock process 6... After several iterations, the overall interaction can be accomplished. However, the interactions extracted by the *simulator* is a set of small interactions: {8 → 7, 7 → 6, 6 → 5} but not a single unit {[6..8] → [5..7]}. If the raw debug output is directly used to produce the protocol, then with the number of involved processes increases, the contents of the protocol will be occupied by repetitions of the same communication; the advantages of using parameterised protocol will be buried in that situation.

In order to make the generated protocol concise and compact, there is an optimisation performed when the actually happened MPI operations are inserted to the *MPI Tree*. The operations with the same pattern will be combined together during the optimisation and the methods used are introduced in the algorithm 4.

We have known from the section *Matching of MPI operations* that occasional matching is not stable; therefore the interaction generated from occasional matching of operations is not allowed to be combined with other interactions, even if they satisfy the normal requirements. For example, there are two interactions in the code snippet 15 if the number of processes N equals 28; however, one of the interactions will not happen if $N \neq 28$; therefore, it is not appropriate to combine these two interactions.

```
1 if (rank==N/4){
2   MPI_Send (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD);
3 }
4
5 if (rank==8){
6   MPI_Recv (buf0, buf_size, MPI_INT, 7, 0, MPLCOMM_WORLD,&status);
7   MPI_Send (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD);
8 }
9
10 if (rank==9){
11   MPI_Recv (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&status);
12 }
```

The non-collective operations are divided into three groups: unicast, multicast and gather, according to their executors and targets. A unicast operation has same number of executors and targets.

Algorithm 4 Algorithm for combining two MPI operations with the same pattern

```
1: Input Data: MPIOperation op1, op2
2: Result: null pointer or the combination of op1 and op2

3: function COMBINE(op1, op2)
4:   if op1 or op2 is created from an occasional matching of unilateral MPI operations then
5:     return null pointer;
6:   end if
7:   Condition op1Exec= the executor of op1
8:   Condition op2Exec= the executor of op2
9:   Condition op1Tar= the target of op1
10:  Condition op2Tar= the target of op2
11:  MPIOperation *result;
12:  if both op1 and op2 are unicast operations with the same span then
13:    if op1Exec is adjacent to op2Exec && op1Tar is adjacent to op2Tar then
14:      result->executor= the union of op1Exec and op2Exec
15:      result->target= the union of op1Tar and op2Tar
16:      return result;
17:    else if op1Exec equals op2Exec && op1Tar is adjacent to op2Tar then
18:      result->executor= op1Exec
19:      result->target= the union of op1Tar and op2Tar
20:      return result;
21:    else if op1Tar equals op2Tar && op1Exec is adjacent to op2Exec then
22:      result->executor= the union of op1Exec and op2Exec
23:      result->target= op1Tar
24:      return result;
25:    else
26:      return null pointer;
27:    end if
28:  else if both op1 and op2 are multicast operations or gather operations then
29:    if op1Exec equals op2Exec && op1Tar is adjacent to op2Tar then
30:      result->executor= op1Exec
31:      result->target= the union of op1Tar and op2Tar
32:      return result;
33:    else if op1Tar equals op2Tar && op1Exec is adjacent to op2Exec then
34:      result->executor= the union of op1Exec and op2Exec
35:      result->target= op1Tar
36:      return result;
37:    else
38:      return null pointer;
39:    end if
40:  else
41:    return null pointer;
42:  end if
43: end function
```

For example, [2..5] \rightarrow [3..6] is a unicast; A multicast is an operation where a single executor sends data to multiple targets. For example, [1..1] \rightarrow [3..6] is a multicast; A gather operation is the inverse of multicast, it refers to an operation where a single process receives data from multiple other processes. For instance, [2] \leftarrow [3..6] is a gather operation.

The term *span* refers to the range between the executor and target of a unicast. It is a vector whose magnitude is the distance between the executor and target. For example, in the operation [2..5] \rightarrow [3..6], the span is 1; while in the operation [3..6] \rightarrow [2..5], the span is -1.

16 LFP

The concept *LFP* (Least Fixed Point) is borrowed from the modal logic [15]. In this application, it refers to a point at which the protocol becomes fixed. The purpose of introducing the concept *LFP* is to find a valid interval for the generated protocol. In the section 5, We have shown by example that for some programs, it is impossible to write a generic protocol applicable to all the configurations. The changing of number of processes may have a major impact on the behaviour of the program. Even though there is no silver bullet, we need to make our solution as generic as possible, and that is also the most important objective of the parameterised protocol. In an ideal situation, we can make the parameterised protocol applicable to a new use case by changing its parameters appropriately. The concept *LFP* is an attempt to make the protocol generated from actual program as generic as possible, in order to capture the characteristics of the program as comprehensive as possible.

17 The Challenges

- **Interprocedural Analysis** The interprocedural analysis is a particular difficult part due to the function recursion and parameter passings. The body of a recursive function may be executed multiple times but it is hard to analyse the exact number of iterations. As a result, it is not easy to analyse the interactions inside the body of a recursive function. Although the author of this report has found a high level similarity between the recursive functions and *do while* loops, due to the time limitation, the transformation from recursive functions to *do while* loops cannot be implemented in time.
- **Analysis of non-deterministic operations** There are several non-deterministic MPI operations, such as receive from `MPI_ANY_SOURCE` and `MPI_WAITANY`. Their runtime behaviours are not predictable and as a result, it is difficult to analyse their semantics and simulate their executions. However, if we do not care the accurate order of the actually happened interactions, their behaviours can still be estimated statically. For example, in the code snippet 17:

```

1  if(rank==0){
2  MPI_Recv (buf1, buf_size, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
3
4  MPI_Recv (buf1, buf_size, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
5  }
6
7  if(rank==1){

```



```
8 MPI_Send (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
9 }
10
11 if (rank==2){
12 MPI_Send (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
13 }
```

It is clear that although the source process in a receive operation can be arbitrary, the destination of a sending operation is deterministic after the program starts. Therefore, if we do not care whether it is process 1 or process 2 sends data to process 0 first, then this pattern can be analysed without problem. However, the difficulty of analysing this pattern will be increased dramatically if the receiver tries to receive data from multiple arbitrary processes via a loop. Again, due to the available time is limit, the analysis of these non-predictable operations are not implemented.

- **Communications in other communication groups** The application developed in this project only supports the single global group: `MPI_COMM_WORLD`. This is because the creation of new groups involves the indices re-computations and it is hard to find the rank mappings between the `WORLD` group and the newly created groups.

4 Test & Evaluation

Evaluation of Application Features

- **Portability:** the application is portable because the paths of machine-dependent resources can be specified by the users when program starts. It has been compiled for both Windows and linux like systems. In the testing, the executables work well on Windows 7 64 bit, Windows 8 64 bit and Ubuntu 12.04 32 bit.
- **Multiple source files:** the application is able to analyse a project which has single main function and multiple implementation files.
- **Simple Analysis of Variables:** If a variable stores a constant value in some point, then the value of that constant can be computed. This might be helpful to determine the target *condition* of an MPI operation if that target is represented by a variable.

1 Compare with the previous project

There was a project which is able to extract MPI primitives and basic program constructs for each rank and judge whether each end-point program conforms to the corresponding end-point protocol [23]. In the remaining part of this section, a compare and contrast between the current project and the previous project will be given and the improvements on the previous application will be highlighted.

- *Portability*

The previous MPI type checker is heavily dependent on Session-C and can only be executed in Unix-like system. The user has to build Session-C before the type checker can be installed. What is worse, if the dependencies changed, the previous application may not function well. The author of this report spent a whole afternoon to try to build the previous application and remove the compiling errors caused by API changes of Clang; In contrast, the Session Type Extractor developed in this project is a stand-alone application. On windows OS, as long as the user has installed the “*Microsoft Visual C++ 2012 Redistributable Package*”, then the *exe* program can be executed easily on command line. On Unix-like system, it is also very easy to execute the binary program, given the paths to the libraries have been specified correctly.

- *Scalability*

The previous project also gave an implementation of extracting the MPI primitives and language constructs; but that approach is not scalable because for every individual process it will construct a rank tree to hold the structure information. That works for small number of processes. However, what if the number of processes is, say, 100,000?

In contrast, the application designed by the author of this report is more scalable because it only generates two extra trees: *CommTree* and *MPITree*, besides the default abstract syntax tree offered by Clang. The major computation time is spent on two areas: the *CommTree* construction and the simulation of MPI program execution. The time spent on *CommTree* construction is only relevant to the complexity of program source code while the maximum time spent on simulation depends on two factors: the number of communications and the number of rank-related choices. For example, in the code below:

```
1 if (rank>2 && rank<6){/*Some MPI primitives*/}
```

There are three *ranges* of processes: [0..2], [3..5] and [6..N-1]. No matter how large the number N is, like 100 or 100,000, as long as $N \geq 6$, the whole processes can be represented by these three *ranges*, or more accurately, six numbers.

- *Analysis of Programming Constructs*

The previous project used very complex mechanism for handling IF statements. However, the analysis is only based on the enumeration of possible cases, which cannot cover every possibilities. Furthermore, the correctness is doubted by the author of this report. As stated in section 4 of design chapter, there will be non-determinism in the *IF statement* if its conditional expression mixes non-rank-related *condition* and rank-related *condition*. In that mechanism which allows the mixture of conditions, it might be the case that, both the MPI primitives occur in if and else block are eligible to be inserted to the same rank tree. For example, in the code snippet 4.1:

Listing 4.1 : Mix rank and non-rank conditions in boolean expression

```
1 if (rank==0 && nonRank==1){/*Some MPI primitives*/}
2
3 else {/*Some other MPI primitives*/}
```

Process 0 might enter the *then block* of this IF statement because the non-rank variable can be 1. However, the negation of the IF condition is “ $rank \neq 0 \parallel nonRank \neq 1$ ”. Any value can be held in the non-rank variable, so “ $nonRank \neq 1$ ” is also satisfiable and the whole expression will be evaluated to the *condition* {0..N-1}; as a result, there will be an unlimited access to the *else block* for every process. The problem is where to insert these two nodes. They should not be siblings because they cannot happen together in any scenario. If divide them into different branches, the complexity of analysis will be increased dramatically because of the non-determinism issues.

In contrast, the analysis of IF statement is greatly simplified by the author of this report. Using the concepts of *range* and *condition* created for this project, the rules for extracting *conditions* from the simplest conditional expression are defined. The *conditions* of arbitrary complex conditional expression can be obtained by decomposing the complex expression and invoking the extraction method recursively. Finally, the *conditions* for every sub-expression

can be easily combined to get the overall *condition* for the original complex expression, with the help of the basic operations which can manipulate the *range* and *condition*.

- *Soundness and Completeness*

In my ISO report, the concept of Soundness and Completeness of an MPI program has been defined. According to my previous proposed definition, a program is said to be sound over a protocol if all the communications in that program have been described by the protocol, i.e. $\text{Comm}(\text{Program}) \subseteq \text{Comm}(\text{Protocol})$. A program is said to be complete over a protocol if all the communications described by the protocol have been performed by the program, i.e. $\text{Comm}(\text{Protocol}) \subseteq \text{Comm}(\text{Program})$. Using these definitions, the application in previous project can be classified as complete but not sound, which means it performs all the required actions but may also do something not prescribed by the protocol. In fact, the situation is worse, the previous application simply ignores the processes which are not monitored by protocols. As a result, a lot of deadlocks caused by the unmonitored processes will not be detected. For example, in the test case 4.2, the program performs all the communications required by the protocol but deadlock still occurs.

Listing 4.2 : Example of Complete But Not Sound Program

```

1 //Program code snippet:
2 if(rank==0){
3 MPI_Recv (buf0, buf_size, MPLINT, 1, 0, MPLCOMMWORLD,&status);
4 MPI_Send (buf0, buf_size, MPLINT, 1, 0, MPLCOMMWORLD);
5 }
6
7 if(rank==1){
8 MPI_Send (buf0, buf_size, MPLINT, 0, 0, MPLCOMMWORLD);
9 MPI_Recv (buf0, buf_size, MPLINT, 0, 0, MPLCOMMWORLD,&status);
10 }
11
12 if(rank==2)
13 MPI_Recv (buf0, buf_size, MPLINT, 1, 0, MPLCOMMWORLD,&status);
14
15 ////////////////////////////////////////////////////
16 //Scribble Protocol:
17 global protocol
18     P
19 (role R0, role R1)
20 {
21 (MPLINT) from R1 to R0;
22 (MPLINT) from R0 to R1;
23 }

```

In contrast, the current application developed in this project not only checks the static structure, but also simulate the executions. The deadlock in this example can be easily detected because there is no matching operation for the *receive* operation performed by process 2.

- *Semantics Analysis*

	Previous Application	This Application
Reading Multiple Source Files	×	✓
Stand-alone Application	×	✓
Need to know Number of Processes	✓	✓
Performance Degraded Due To Number of Processes Increases	✓	×
Checking Conformance of Program Against Protocol	✓	×

Table 4.1: Characteristics comparison between old and current applications

The previous application focused on the static program structure analysis and the support for semantics analysis is poor. In MPI program, the non-blocking operations can be paired with the blocking operations. For example, a blocking operation like *MPI_Send* can match a non-blocking operation like *MPI_Irecv*. However, the previous application only recognise operations by their names but not their semantic meaning; as a result, the previous application may report deadlocks when there is not any. In contrast, the application developed in this project overcome this problem by classifying the MPI operations according to their properties. Any operation which sends data to destinations is regarded as a sending operation. Similarly, any operation that receives data from others is a receiving operation. Any sending operation can be matched with any receiving operation if they satisfy the preconditions which are specified in algorithm 3.

- Variable Analysis

There is only a simple mechanism for analysing variables in the previous application. For example, after evaluating the variable declaration “*int to=2*”, the previous application will associate the variable *to* with the constant *2*. After the mapping established, it will not be updated by later statements such as *to=to + 1*;. To summarise, the previous application does not support constant propagation. In this application, a simple constant propagation is performed to make the variable analysis more accurate. Each time a variable declaration statement or assignment statement is encountered, the right hand side value will be evaluated recursively to get an accurate result; then the left hand side variable can be associated with that result. Each time a mapping is established or changed, the relevant entries in the data structure will be updated immediately to reflect the changes.

- Deadlock Detection

The previous application only focused on the static structures of MPI sending, receiving operations but many deadlocks are caused by synchronisation operations like *MPI_Barrier* or *MPI_Wait*. While the previous application simply ignore these operations, the application developed in this project models the *MPI_Barrier* and *MPI_Wait* explicitly. In the simulation, a barrier node will block every visitor until it is unblocked; and it cannot be unblocked unless it has been visited by all the processes. An *MPI_Wait* node will block the processes which have unfinished non-blocking operations if the request variable matches. To sum up, the two synchronisation operations have been simulated successfully in the current application, and any deadlock caused by these two operations can be detected.

In table 4.1, a clear comparison between the current application and the previous application is listed in terms of the program features.

To summarise, the type checker developed in [23] is suitable to be used in developing new programs, where the correct Scribble Protocol has already been written at the beginning. For the large scale legacy code which does not have a corresponding protocol in the first place, it is inapplicable; In contrast, the session type extractor developed in this project is applicable for both new programs and legacy code. It is also more scalable than the previous application when number of processes increases.

2 Test Cases

2.1 *Condition* Extraction Test

2.1.1 Simple *conditions*

Listing 4.3 : Extract Simple *Conditions*

```
1 #include "mpi.h"
2 int main (int argc, char **argv){
3     int rank=-1;
4     int nprocs = -1;
5
6     MPI_Init (&argc, &argv);
7     MPI_Comm_size ( MPLCOMM_WORLD, &nprocs);
8     MPI_Comm_rank ( MPLCOMM_WORLD, &rank);
9
10    if(rank==0){} else {}
11
12    if(rank==1 && rank==2){}
13
14    if(rank==1 || rank==5){}
15
16    if(rank>2 && rank <7 || rank>10){}
17
18    MPI_Barrier (MPLCOMM_WORLD);
19    MPI_Finalize ();
20 }
```

```

Ready to simulate the execution of the MPI program now!
There are 1 communicator groups involved
The param role name is MPI_COMM_WORLD
The actual roles for this param role are:
The role 0 is MPI_COMM_WORLD[0..N-1]
The role 1 is MPI_COMM_WORLD[0..0]
The role 2 is MPI_COMM_WORLD[1..N-1]
The role 3 is MPI_COMM_WORLD[1..1]
The role 4 is MPI_COMM_WORLD[5..5]
The role 5 is MPI_COMM_WORLD[6..0]
The role 6 is MPI_COMM_WORLD[2..4]
The role 7 is MPI_COMM_WORLD[3..6]
The role 8 is MPI_COMM_WORLD[11..N-1]
The role 9 is MPI_COMM_WORLD[0..2]
The role 10 is MPI_COMM_WORLD[7..10]

```

Figure 4.1: The roles generated from code 4.3

As we can see from the figure 4.1, there are ten roles with different *ranges* generated from the programming constructs of the source code. These *roles* are created just for unit test and demonstration purpose. Only the *role* with *range* $\{[0..N-1]\}$ will participate the simulation at the beginning.

In the first IF statement, *range* $\{[0..0]\}$ for *if* block can be identified from the conditional expression ‘*if(rank==0)*’; and *range* $\{[1..N-1]\}$ for *else* block can be deduced by performing a negation operation on the range $\{[0..0]\}$. The *condition* for the second *if* statement is empty, as a result, a full *range* $[0..N-1]$ will be generated for the implicit *else* part. The third IF statement will produce two *ranges* $[1..1]$, $[5..5]$ for the *if* block and the corresponding *else* block will be $\{[0..0],[2..4],[6..N-1]\}$ (the *range* $[6..N-1]$ is adjacent to $[0..0]$, so $[6..0]$ will be generated). For the last IF statement, *range* $[3..6]$ and $[11..N-1]$ will be created for the *if* block and their complementary part $\{[0..2],[7..N-1]\}$ will be created for the *else* block. According to the output, it is clear that all the *conditions* have been extracted successfully.

2.1.2 Complex conditions

Listing 4.4 : Extract Complex Conditions

```

1 #include "mpi.h"
2 int main (int argc , char **argv){
3     int rank=-1;
4     int nprocs = -1;
5     int test=0;
6
7     MPI_Init (&argc , &argv);
8     MPI_Comm_size ( MPLCOMM_WORLD , &nprocs);
9     MPI_Comm_rank ( MPLCOMM_WORLD , &rank);
10
11     if(rank>2 && rank <7){
12         if(rank==1 || rank==5){}
13     }

```

```

14
15  if (test==2){
16  for (int i=0; i<10; i++){
17      if(rank==7){}
18  }
19  } else{
20      if(rank==22){}
21  }
22
23  MPI_Barrier (MPLCOMM_WORLD);
24  MPI_Finalize ();
25 }

```

```

Ready to simulate the execution of the MPI program now!
There are 1 communicator groups involved
The param role name is MPI_COMM_WORLD
The actual roles for this param role are:
The role 0 is MPI_COMM_WORLD[0..N-1]
The role 1 is MPI_COMM_WORLD[3..6]
The role 2 is MPI_COMM_WORLD[5..5]
The role 3 is MPI_COMM_WORLD[6..6]
The role 4 is MPI_COMM_WORLD[3..4]
The role 5 is MPI_COMM_WORLD[0..2]
The role 6 is MPI_COMM_WORLD[7..N-1]
The role 7 is MPI_COMM_WORLD[7..7]
The role 8 is MPI_COMM_WORLD[0..6]
The role 9 is MPI_COMM_WORLD[8..N-1]
The role 10 is MPI_COMM_WORLD[22..22]
The role 11 is MPI_COMM_WORLD[0..21]
The role 12 is MPI_COMM_WORLD[23..N-1]

```

Figure 4.2: The roles generated from code 4.4

There are thirteen *ranges* recognised during analysing the source code 4.4. The *range* [0..N-1] will be created for any source code. All the other *ranges* are related to the conditionals of the source code. Usually, for each IF conditional expression, three *ranges* will be extracted: one for *then* part and two for the *else* part. The *range* for the *then* part of the first IF statement is [3..6]. There is an inner IF statement and the inner IF will inherit the properties from its parent construct. Therefore, instead of producing *ranges* {[1..1],[5..5]} for *then* block and {[0..0],[2..4],[6..N-1]} for *else* block, the inner IF statement generates {[5..5]} for *then* block and {[3..4],[6..6]} for *then* block. After constructing *conditions* for the *then* block of the first IF, its *then* block will be analysed. The complementary set of [3..6] is {[0..2], [7..N-1]}. The *conditions* in non-rank related IF and FOR statements are {[0..N-1]} and the else part of a non-rank-related choice also has *range* [0..N-1]. That is why both [7..7] and [22..22] can be generated from the send IF statement.

2.2 Simple Constructs Test

Listing 4.5 : Simple Program and corresponding protocol

```
1 if(rank>=0 && rank<=7){
2   MPI_Irecv (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD, &req);
3   MPI_Isend (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD, &req2);
4   MPI_Recv (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD,&status);
5
6   MPI_Wait (&req, &status);
7   MPI_Wait (&req2, &status);
8 }
9
10 if(rank>=1 && rank<=8){
11 MPI_Irecv (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&req);
12 MPI_Isend (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&req2);
13 MPI_Send (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD);
14
15 MPI_Wait (&req2, &status);
16 MPI_Wait (&req, &status);
17 }
18
19 int root=5;
20 MPI_Comm comm=MPLCOMM_WORLD;
21 if(rank==0 || rank==2){
22 MPI_Gather( sendarray, 100, MPI_INT, buf0, 1, MPI_INT, 0, comm);
23 MPI_Reduce( buf0, buf1, 100, MPI_INT, MPLSUM, root, comm );
24 }
25
26 else if(rank==1 || rank>2){
27 MPI_Gather( sendarray, 100, MPI_INT, buf0, 1, MPI_INT, 0, comm);
28 MPI_Reduce( buf0, buf1, 100, MPI_INT, MPLSUM, root, comm );
29 }
30
31 MPI_Bcast(buf0, buf_size, MPI_INT, 0, MPLCOMM_WORLD);
```

```

/*The protocol generated is stable!*/

const N= 10..Inf

global protocol simple_ProToCoL (role MPI_COMM_WORLD[0..N-1])
{
Data(MPI_INT) from MPI_COMM_WORLD[rank:0..7] to
MPI_COMM_WORLD[rank+1];

Data(MPI_INT) from MPI_COMM_WORLD[rank:1..8] to
MPI_COMM_WORLD[rank-1];

Data(MPI_INT) from MPI_COMM_WORLD[rank:1..8] to
MPI_COMM_WORLD[rank-1];

Data(MPI_INT) from MPI_COMM_WORLD[0..N-1] to MPI_COMM_WORLD[0]

Data(MPI_INT) from MPI_COMM_WORLD[0..N-1] to MPI_COMM_WORLD[5]

Data(MPI_INT) from MPI_COMM_WORLD[0] to MPI_COMM_WORLD[0..N-1]
}

```

Figure 4.3: Protocol for code 4.5

As shown in the program 4.5, there are three pairs of basic sending/receiving operations at the beginning. The first two operations are non-blocking operations, therefore, no deadlock will occur even if the receiving operations are placed before sending for both parties. The `MPI_Wait` operation is a blocking operation, it forces the processes to finish the relevant unfinished non-blocking operations. Deadlock can occur if such kind of blocking operations are not used properly. Later, an example of deadlock caused by misusing `MPI_Wait` will be introduced. Well, let's go back to this program. After the basic operations, there are several collective operations. The first two collective operations are split into two parts. A collective op cannot finish until all the processes participate the operation. In this example, the processes in `IF` and `ELSE` block can be combined to the complete *range* and the orders of the operations are also correct, therefore these collective operations can happen without problem.

2.3 Unsupported Cases Test

This application is not perfect, but it tries to identify all the unsupported cases and perform correct analysis in its supported area. There are several unsupported cases listed below to show this application is capable of identifying what it cannot handle.

Listing 4.6 : Mixture of Rank And NonRank Conditions

```

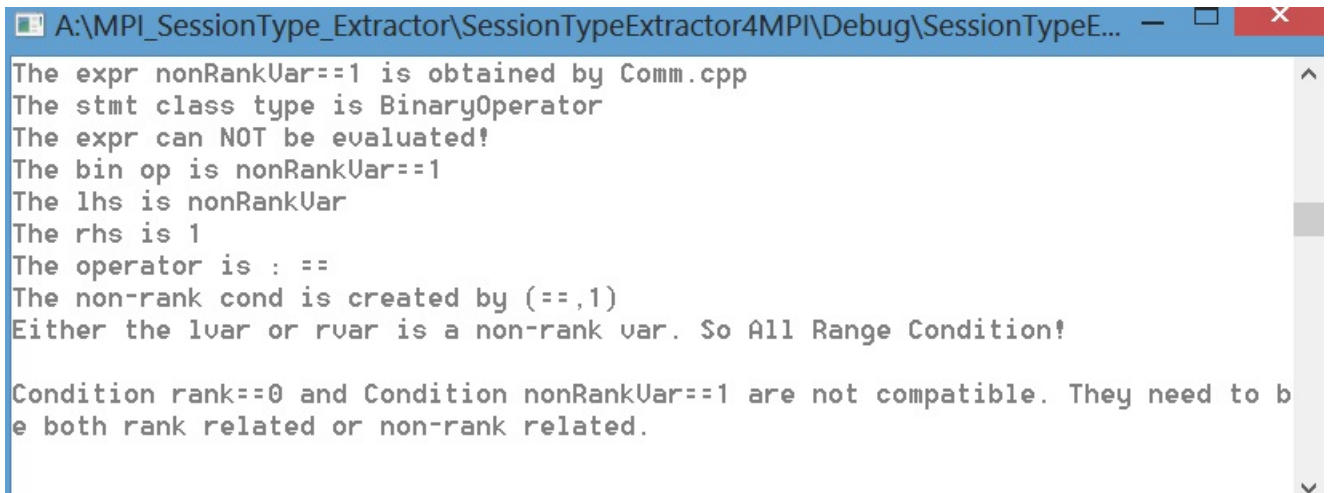
1  MPI_Init (&argc , &argv);
2  MPI_Comm_size ( MPLCOMM_WORLD , &nprocs);
3  MPI_Comm_rank ( MPLCOMM_WORLD , &rank);
4
5  int nonRankVar=2;
6
7  if (rank==0 && nonRankVar==1){

```

```

8  MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
9  }
10
11 if(rank==1){
12  MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD,&status);
13 }
14
15 MPI_Barrier (MPLCOMM_WORLD);
16
17 MPI_Finalize ();

```



```

A:\MPI_SessionType_Extractor\SessionTypeExtractor4MPI\Debug\SessionTypeE...
The expr nonRankUar==1 is obtained by Comm.cpp
The stmt class type is BinaryOperator
The expr can NOT be evaluated!
The bin op is nonRankUar==1
The lhs is nonRankUar
The rhs is 1
The operator is : ==
The non-rank cond is created by (==,1)
Either the lvar or rvar is a non-rank var. So All Range Condition!

Condition rank==0 and Condition nonRankUar==1 are not compatible. They need to be both rank related or non-rank related.

```

Figure 4.4: Output of analysing code 4.6 from cmd

As shown in the output 4.4, an exception will be thrown and application terminates with a warning message.

Listing 4.7 : Unsupported Nesting

```

1  MPI_Init (&argc, &argv);
2  MPI_Comm_size (MPLCOMM_WORLD, &nprocs);
3  MPI_Comm_rank (MPLCOMM_WORLD, &rank);
4
5  int nonRankVar=2;
6
7  if(rank==0){
8      if(nonRankVar==0){
9          MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
10     }
11     else {
12         MPI_Send (buf0, buf_size, MPI_INT, 2, 0, MPLCOMM_WORLD);
13     }
14 }
15
16 if(nonRankVar==1){
17     if(rank==1)
18         MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD,&status);
19 } else {

```

```

20 |   if (rank==2)
21 |       MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD,&status);
22 | }

```

This kind of unsupported nesting has been discussed in section 4 of design & implementation chapter. An exception with error message “The current node is not allowed to insert MPI operation!” will be thrown.

Listing 4.8 : Unsupported Non-deterministic sender

```

1 | MPI_Init (&argc, &argv);
2 | MPI_Comm_size (MPLCOMM_WORLD, &nprocs);
3 | MPI_Comm_rank (MPLCOMM_WORLD, &rank);
4 |
5 | if (rank==0)
6 |     MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
7 |
8 | if (rank==1)
9 |     MPI_Recv (buf0, buf_size, MPI_INT, MPLANY_SOURCE, 0, MPLCOMM_WORLD,&status);

```

The *MPLANY_SOURCE* is not supported in the current application. Therefore, an exception with message “The *MPLANY_SOURCE* is not supported at present, sorry about that.” will be thrown when encounter such situation.

2.4 Deadlock tests

When a deadlock is detected, the application terminates and prints the stack of pending operations when the deadlock occurs. This can help the debugging a little.

2.4.1 Classic deadlock

The first deadlock example is a classic one and it comes from the ISP website [5].

Listing 4.9 : Classic Deadlock

```

1 | /* -- Mode: C; -- */
2 | /* Creator: Bronis R. de Supinski (bronis@llnl.gov) Fri Mar 17 2000 */
3 | /* no-error.c -- do some MPI calls without any errors */
4 |
5 | #include <stdio.h>
6 | #include "mpi.h"
7 |
8 | #define buf_size 128
9 |
10 | int main (int argc, char **argv)
11 | {
12 |     int nprocs = -1;
13 |     int rank = -1;
14 |     char processor_name[128];
15 |     int namelen = 128;

```

```

16  int buf0[buf_size];
17  int buf1[buf_size];
18  MPI_Status status;
19
20  /* init */
21  MPI_Init (&argc, &argv);
22  MPI_Comm_size (MPLCOMM_WORLD, &nprocs);
23  MPI_Comm_rank (MPLCOMM_WORLD, &rank);
24  MPI_Get_processor_name (processor_name, &namelen);
25  printf ("%d is alive on %s\n", rank, processor_name);
26  fflush (stdout);
27
28  MPI_Barrier (MPLCOMM_WORLD);
29
30  if (nprocs < 2)
31  {
32      printf ("not enough tasks\n");
33  }
34  else if (rank == 0)
35  {
36      memset (buf0, 0, buf_size);
37
38      MPI_Recv (buf1, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD, &status);
39
40      MPI_Send (buf0, buf_size, MPI_INT, 1, 0, MPLCOMM_WORLD);
41  }
42  else if (rank == 1)
43  {
44      memset (buf1, 1, buf_size);
45
46      MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD, &status);
47
48      MPI_Send (buf1, buf_size, MPI_INT, 0, 0, MPLCOMM_WORLD);
49  }
50
51  MPI_Barrier (MPLCOMM_WORLD);
52
53  MPI_Finalize ();
54  printf ("%d Finished normally\n", rank);
55 }
56
57 /* EOF */

```

As shown in output figure 4.5, the deadlock is detected during the simulation. Because both processes 1 and 2 are performing the blocking operation: **MPI_Recv**, no one can move forward to the sending operation. As a result, the deadlock occurs and leave the unfinished operations on the pending list.

2.4.2 Deadlock caused by MPI_Wait

```

Deadlock occurs!!!
The current pending operations are:
MPI_Recv (buf1, buf_size, MPI_INT, 1, 0, MPI_COMM_WORLD,
&status)
MPI_Recv (buf0, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status)

No pending collective operation!

```

Figure 4.5: Output of analysing code 4.9

Listing 4.10 : Deadlock caused by misusing synchronisation operation MPI.Wait

```

1  MPI.Init (&argc, &argv);
2  MPI.Comm_size ( MPLCOMM_WORLD , &nprocs); /*nprocs=100*/
3  MPI.Comm_rank ( MPLCOMM_WORLD , &rank);
4
5  if(rank==0){
6    MPI_Isend (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD, &req);
7
8    MPI.Wait (&req, &status);
9
10   MPI_Irecv (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD, &req2);
11 }
12
13 if(rank==1){
14   MPI_Isend (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&req2);
15
16   MPI.Wait (&req2, &status);
17
18   MPI_Irecv (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&req);
19 }
20
21 MPI_Barrier (MPLCOMM_WORLD);
22 MPI_Finalize ();

```

```

Deadlock occurs!!!
The current pending operations are:
MPI_Isend (buf0, buf_size, MPI_INT, rank+1, 0, MPI_COMM_WORLD,
&req)
MPI_Isend (buf0, buf_size, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
&req2)

No pending collective operation!

```

Figure 4.6: Output of analysing code 4.10

As shown in the diagram 4.6, a deadlock is detected when analysing this program. Both process 0 and 1 are trying to send data to the other but no one is prepared to receive data before entering the blocking node *MPI.Wait*. Therefore, both processes are blocked and no one can proceed to the

receive statement. As a result, an impasse happens.

2.4.3 Deadlock caused by collective operations

Listing 4.11 : Deadlock by Collective Operations

```
1  MPI_Init (&argc , &argv);
2  MPI_Comm_size ( MPLCOMM_WORLD , &nprocs); /*nprocs=100*/
3  MPI_Comm_rank ( MPLCOMM_WORLD , &rank);
4
5  MPI_Comm comm=MPLCOMM_WORLD;
6  int gatherRoot=0;
7  int reduceRoot=5;
8
9  if(rank==0 || rank==2){
10  MPI_Gather( sendarray , 100, MPI_INT, buf0 , 1, MPI_INT, gatherRoot , comm);
11
12  MPI_Reduce( buf0 , buf1 , 100, MPI_INT, MPI_SUM, reduceRoot , comm );
13 }
14
15 if(rank>=1){
16  MPI_Gather( sendarray , 100, MPI_INT, buf0 , 1, MPI_INT, gatherRoot , comm);
17
18  MPI_Reduce( buf0 , buf1 , 100, MPI_INT, MPI_SUM, reduceRoot , comm );
19 }
```

```
Deadlock occurs!!!
The current pending operations are:

MPI_Reduce( buf0 , buf1 , 100 , MPI_INT , MPI_SUM , reduceRoot ,
comm )
```

Figure 4.7: Output of analysing code 4.11

The deadlock shown in figure 4.7 is caused by process 2. According to the program, process 2 will perform gather operation twice while all the other processes only execute that operation once. When the first gather operation happens successfully, process 0 and 2 are still inside the first IF statement. The other processes with *condition* $\{[1..1],[3..N-1]\}$ will continue visiting the reduce node and report the *MPI_Reduce* operation to *CollectiveOperationManager*. Some time later, process 2 comes to the second IF statement and report the gather operation again. The *CollectiveOperationManager* will detect the deadlock immediately as soon as it knows process 2 is trying to perform a gather operation.

2.5 Testing of complex program with multiple source files

The session type extractor developed in this project supports the analysis of an MPI project which involves multiple source files. To demonstrate this feature, a project which contains multiple source files in different levels is created. Its hierarchy is shown in figure 4.8.

The complete list of these source files can be found in the appendix 1.

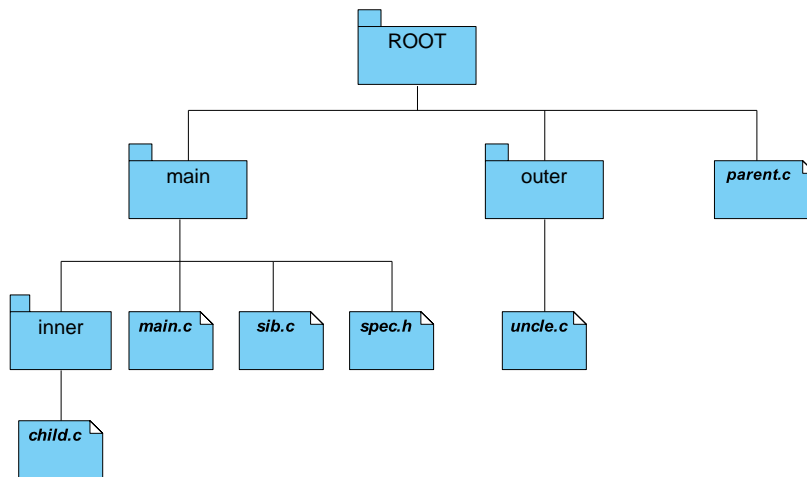


Figure 4.8: Project structure in test case

There are five source files in the project and the file “main.c” contains the main function. Each of the other four files defines one function which contains MPI operations. The first function being called from main function is “helloUncle()” from “uncle.c” and within that function, “void testMulti2()” is defined. The function *testMulti2()* is quite interesting because in the header of FOR loop, the variable *i* is bound to *range* [0..2]. Then in the conditional expression, the rank variable is bound to variable *i*, which makes the executor of the MPI operations within the IF block have *range* [0..2]. Therefore, the actually happened operation is processes [0..2] send data to their right neighbours [1..3]. In the protocol, the indices of the processes are either represented by *range* or bounded variables.

The function *testDiffRanksCallSameMethod()* in *sib.c* is quite complex and it tests two ways of calling the function “testFor2()”: call it from rank-related node and call it from non-rank-related node. The conditional expression ‘*nprocs* > 9 && *nprocs* < 20’ is a non-rank-related global choice; therefore both *then* block and *else* block will be analysed and in the final output; the sub-protocol for *then* block and the sub-protocol for *else* block will be generated separately and combined by the keyword *or*. There are several nested structures within the *testFor2()* function. The most interesting one is the last inner FOR loop “*for(int j = 17; j >= 0; j --)*”. In that loop, processes with *range* [2..8] send and then receive data from process 1. Process 1 uses a FOR loop “*for(int c = 2; c <= 8; c ++)*” to gather from processes [2..8] and multicast to processes [2..8].

The function “void testWait()” is defined in “parent.c” and a similar test case has been given in the deadlock testing. However, the function *testWait* does not have deadlock. Even though there are multiple MPI.Wait, the repeated one will be ignored.

The function “void testMultiSenderAndMultiRecver()” in “child.c” is quite challenging because it involves multicast performed by multiple processes. There are two ways to interpret the interactions in the function. They can be regarded as all processes within *range* [3..7] perform multicast to processes [0..2] independently, or processes with *range* [0..2] gather data from process [3..7] separately.

The full protocol generated by the application is available in the appendix.

2.6 LFP examples

Listing 4.12 : An MPI program having stable protocol

```
1  MPI_Init (&argc , &argv);
2  MPI_Comm_size ( MPLCOMM_WORLD , &nprocs);
3  MPI_Comm_rank ( MPLCOMM_WORLD , &rank );
4
5  if (rank==5)
6  MPI_Send (buf0 , buf_size , MPI_INT, rank+1, 0, MPLCOMM_WORLD);
7
8  if (rank==6)
9  MPI_Recv (buf0 , buf_size , MPI_INT, rank-1, 0, MPLCOMM_WORLD,&status);

/*The protocol generated is stable!*/

const N= 8..Inf

global protocol lfp_ProToCoL (role MPI_COMM_WORLD[0..N-1])
{
  Data(MPI_INT) from MPI_COMM_WORLD[rank:5..5] to
  MPI_COMM_WORLD[rank+1];
}
```

Figure 4.9: Output of analysing code 4.12 using 100 processes

```
/*The protocol generated is NOT stable!*/
/*The current protocol is only applicable when number of
processes is 5*/

const N= 5;

global protocol lfp_ProToCoL (role MPI_COMM_WORLD[0..N-1])
{}
```

Figure 4.10: Output of analysing code 4.12 using 5 processes

The figure 4.9 and the figure 4.10 analyses the same code 4.12 but using different number of processes. As we can see, their results are quite different. When there are only 5 processes, the only interaction from the program cannot actually happen and therefore, nothing will be produced. However, if the number of processes is greater than or equal to 8, then the protocol will remain the same forever. This is an easy example for showing the value of *LFP* in analysing the communication patterns of MPI programs.

2.7 Matching examples

Listing 4.13 : An MPI program with perfect matching

```
1 MPI_Comm_size ( MPLCOMM_WORLD , &nprocs );
2 MPI_Comm_rank ( MPLCOMM_WORLD , &rank );
3
4 if (rank==(nprocs/5))
5 MPI_Send ( buf0 , buf_size , MPI_INT , rank+1 , 0 , MPLCOMM_WORLD );
6
7 if (rank==(nprocs/5)+1){
8 MPI_Recv ( buf0 , buf_size , MPI_INT , rank-1 , 0 , MPLCOMM_WORLD,&status );
9 MPI_Send ( buf0 , buf_size , MPI_INT , rank+1 , 0 , MPLCOMM_WORLD );
10 }
11
12 if (rank==(nprocs/5)+2)
13 MPI_Recv ( buf0 , buf_size , MPI_INT , rank-1 , 0 , MPLCOMM_WORLD,&status );

/*The protocol generated is stable!*/

const N= 4..Inf

global protocol combi_ProToCoL (role MPI_COMM_WORLD[0..N-1])
{
  Data(MPI_INT) from MPI_COMM_WORLD[rank:(N/5)..(N/5)+1] to
  MPI_COMM_WORLD[rank+1];
}
```

Figure 4.11: Output of analysing code 4.13

As we can see in diagram 4.11, the protocol is stable because the MPI operations in the program will always happen in the same pattern: processes with range $[N/5..N/5+1]$ will send data to their right neighbours. The protocol always has the same content as long as the number of processes is greater than or equal to the fixed point '4'. Here, the least fixed point is estimated conservatively, considering the possibility that the largest known rank '2' may communicate with the process with rank 'N-1'; therefore, the largest rank number may reach 3 and as a result, 4 is chosen as the minimum number of processes.

Listing 4.14 : An MPI program with imperfect matching

```
1 MPI_Comm_size ( MPLCOMM_WORLD , &nprocs );
2 MPI_Comm_rank ( MPLCOMM_WORLD , &rank );
3
4 if (rank==(nprocs/5))
5 MPI_Send ( buf0 , buf_size , MPI_INT , rank+1 , 0 , MPLCOMM_WORLD );
6
7 if (rank==21){
8 MPI_Recv ( buf0 , buf_size , MPI_INT , rank-1 , 0 , MPLCOMM_WORLD,&status );
9 MPI_Send ( buf0 , buf_size , MPI_INT , rank+1 , 0 , MPLCOMM_WORLD );
10 }
11
```

```

/*The protocol generated is NOT stable!*/

/*The current protocol is only applicable when number of
processes is 100*/

/*There is imperfect matching of MPI operations; Plz use
perfect matching.*/

const N=100;

global protocol combi2_ProToCoL (role MPI_COMM_WORLD[0..N-1])
{
Data(MPI_INT) from MPI_COMM_WORLD[rank:(N/5)] to
MPI_COMM_WORLD[rank+1];

Data(MPI_INT) from MPI_COMM_WORLD[rank:21..21] to
MPI_COMM_WORLD[rank+1];

}

```

Figure 4.12: Output of analysing code 4.14

```

/*The protocol generated is stable!*/

const N= 2..Inf

global protocol online_ProToCoL (role MPI_COMM_WORLD[0..N-1])
{
Data(MPI_INT) from MPI_COMM_WORLD[my_rank:0..N-1] to
MPI_COMM_WORLD[(my_rank-1+N)%N];

}

```

Figure 4.13: Output of analysing code 9

```

12 | if (rank==22)
13 | MPI_Recv (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD,&status);

```

As we can see from figure 4.12, the protocol is only applicable when the number of processes is 100. This is caused by the imperfect matching of SEND and RECV operation. In the program, the process with rank $N/5$ is trying to send data to its right neighbour; however, number N is not bound until the MPI program starts executing. If number N is set to 100 by the user when the MPI program is executed, then the interaction between process 20 (computed from $N/5$) and 21 can happen; otherwise, a deadlock will happen.

2.8 Real MPI Program Test: Ring topology

The figure 4.13 is the protocol for an on-line tutorial MPI program 9. That tutorial program is used for showing non-blocking operations within a ring topology and the communication part is quite simple: only two MPI operations are involved. The trick here is, the target of the operation is repre-

sented by a rank-related variable; as a result, the target process of the operation is dependent on the executor. Although there is only one line of “*MPI_Isend(&my_rank,1,MPI_INT,left_neighbor,tag,MPI_COMM_WORLD,&reqSend);*” in the source code, it will be interpreted differently by different processes during the execution. Therefore, we need to capture the communication pattern in a generic way; otherwise, the protocol has to be changed each time the number of processes changes. After analysing and generalising, it is found that the program describes a communication pattern in a wraparound ring, where everyone sends data to its left neighbour. However, both the executor and the target of the operation have *range* [0..N-1]. Therefore, we need to figure out the relations between the senders and receivers and compute the relative indices of target processes; otherwise, the senders and receivers will be the same. To resolve this problem, an extra field “*execString*” is created for class *Condition* and aims to hold the string presentation of the right hand side expression each time an assignment operation is encountered. To make it consistent in the generated protocol, inside this string, the variable the number of processes will be replaced by ‘N’. After an assignment operation, the *condition* for the right hand side expression will be generated and associated with the l.h.s variable. Therefore, when the l.h.s variable is encountered in the MPI operation, the associated *condition* can be retrieved from the mapping. Because the “*execString*” field of the *condition* has been set in the assignment, therefore, we can finally find the string representation of the initial expression.

Another difficulty is to extract the target condition from the complex mathematical expression. In the code 9, variable ‘*left_neighbor*’ and ‘*right_neighbor*’ are represented by mathematical expressions which involve plus, minus and modulo operations. Thanks to the well-defined recursive extraction function, the complex expression can be decomposed repeatedly and gradually solved. Although the mathematical expression is simulated, the complexity is not as high as that of executing the real program: the calculation in the loops are not simulated multiple times.

5 Conclusion

In conclusion, the major goals that were mentioned in section “This project” (1) have been achieved. To summarise, the main aim is to extract the parameterised protocols from MPI programs. In order to achieve the final goal, three sub-goals are established to facilitate the realisation of the final mission: 1) extract the communication skeleton of the MPI program (see 9.2); 2) prune the raw communication skeleton to get the tree of interactions that actually happened (see 10); 3) gather the information from the pruned communication tree and generate parameterised protocol (see 15). These three sub-goals are achieved in sequence and the output of the previous one is the input of the next one. To achieve the first sub-goal, the traversal of the AST of the MPI source code is performed with the help of Clang’s recursive AST visitor. After getting the common structure and MPI primitives nodes, a simulation is conducted to estimate the interactions that actually happened in the real execution and the relevant information is recorded in a tree structure. Finally, the tree which stores the information about the actually happened interactions is traversed and the protocol is generated according to Scribble’s syntax.

The accuracy of the static analysis will be weakened when the checker only uses the direct knowledge from the static structure without inference. The poor accuracy of previous application [23] is an example of this. The application developed in this project enhances the semantics support and deadlock detection ability by simulation. Much useful information can be inferred during the simulation, such as which process can interact with which process and whether the matching of two unilateral MPI operations is perfect or not.

In addition to generating the protocols, the application tries to make the protocols as generic as possible and investigate the applicability of the generated protocols. It has quite a few limitations due to the time limits; however, the application developed in this project, together with this report, try to capture and specify the limitations of the application. Through the diagnostic information provided by the application, the users of the application are more likely to know whether a problem is caused by the MPI program or the limitation of the application. Through the unsupported cases explained in the report, the later developers can be more clear about the difficulties.

To summarise, this project made contributions to the development of the MPI type checking via session type theories and produced a working software application to achieve the goal. The software can be used by MPI programmers to judge whether an MPI program behaves as expected in terms of communication. The theoretical innovations of this report may enlighten the future researchers and developers in some aspect.

1 Future Work

There are several possible directions for future developments in the project of static analysis of MPI programs using Scribble Protocol.

Further analysis on currently unsupported cases As explained in the design & implementation chapter, in order to make the application perform well in its supported domain, it ignores some cases and simply throws an exception when encounter these cases. If more time is available, some further deduction can be performed to make the analysis more accurate; and the diagnostic information for the deadlocks can be made more expressive, E.G, including the reason of deadlock and the location where error occurs.

Interprocedural analysis The interprocedural analysis is interesting because it may involve both arguments passing and sophisticated control flow analysis. A function can have local variables and receive arguments from invoker. The values of local variables are only valid inside the block where they reside. Therefore, if multiple variables with the same name are defined in different places, then they need to be checked carefully to ensure that they have correct values at the places where we encounter them. Although in general it is hard to analyse the values stored in variables statically, we can start from the variables which are important in MPI analysis(the variables inside the MPI operation's target expression) and then perform a backward slicing.

Not only the complexity of data flow analysis increases in function invocation, the control flow analysis is also increased by function recursion. It is expected by the author that the function recursion can be transformed to some kind of *do-while* loop. For example, the code snippet 5.1 shows the basic idea;

Listing 5.1 : Idea about transform function recursion to do while loop

```
1 //the recursion function
2 void rec(int test){
3
4     if(test < 0){/*throw exception*/}
5
6     else if(test == 0){/*MPI operations: block 0*/}
7
8     else if(test == 1){/*MPI operations: block 1*/}
9
10    else{
11        /*MPI operations: block X*/
12        rec(test - 1);
13    }
14 }
15
16
17
18
19
20 //after the transformation:
21 do{
22     if(test < 0){/*throw exception*/}
23
24     else if(test == 0){/*MPI operations: block 0*/}
25
26     else if(test == 1){/*MPI operations: block 1*/}
27
```

```
28  else{
29      /*MPI operations: block X*/
30      test--;
31  }
32 } while(!test<0 && !test==0 && !test==1)
```

As shown in the code snippet 5.1, the loop condition of the *do-while* loop is obtained by analysing the condition of the branch in which the recursive function is invoked. The function *recur* is invoked in the *else branch*, therefore, the condition of the *do-while* loop is the negation of that condition. The idea is very clear, but again, due to the limitation of the time available, this part has to be implemented in the future.

Conformance to given protocol Due to the limited time, the current application does not support the function of projecting the generated global protocol to local protocols and reading external protocols. If more time is available, these feature should be added and the actual protocol of MPI program can be checked against the expected protocol, making the type checking more comprehensive.

More languages Both the previous and the current project only analyse the MPI programs written in C language but in fact in can be written in several other languages, such as C++. The analysis of object-oriented (O-O) languages is very challenging because both the syntax and semantics of O-O languages are more complex.

More communication groups Currently, only the global communication group '*MPI_COMM_WORLD*' is supported. However, there are a considerable amount of MPI operations on manipulating communication groups and many MPI programs use these operations to create new communication groups. In order to make the analysis tool support more language features and applicable to more MPI programs, it is quite desirable for the application to support more communication groups.

Acknowledgements

Thanks my supervisor Professor Yoshida, for the guidelines and suggestions about the topic, the materials provided by her is really helpful. Thanks Socrates Katsoulacos, the author of the previous report [23], his report gave me an initial impression on static analysis of MPI programs. Thanks Mr. Nicholas Ng for the discussion and test cases, and the notations used in the project is greatly influenced by his unpublished report [26]. Finally, thanks my lovely girlfriend for giving me support in my everyday life and I also got inspiration from some discussion with her.

Bibliography

- [1] A Solution to A simple Jacobi iteration. <http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobi/C/solution.html>. [Online; accessed 3-September-2013].
- [2] Clang - Features and Goals. <http://clang.llvm.org/features.html#diverseclients>. [Online; accessed 20-August-2013].
- [3] Clang: a C language family frontend for LLVM. <http://clang.llvm.org>. [Online; accessed 20-August-2013].
- [4] Compiling multiple files. <http://crasseux.com/books/ctutorial/Compiling-multiple-files.html>. [Online; accessed 28-August-2013].
- [5] ISP Test Results. http://www.cs.utah.edu/formal_verification/ISP_Tests/. [Online; accessed 3-September-2013].
- [6] Lecture 5. <http://www.cfm.brown.edu/people/gk/APMA281A/LECTURES/>. [Online; accessed 3-September-2013].
- [7] The LLVM Compiler Infrastructure. <http://llvm.org>. [Online; accessed 20-August-2013].
- [8] MPI: A Message-Passing Interface Standard Version 2.2. <http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>, September 2009. [Online; accessed 26-August-2013].
- [9] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, pages 418–433, 2008.
- [10] Greg Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 1–12. IEEE Computer Society, 2009.
- [11] Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, pages 272–286, 2012.
- [12] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446, 2011.
- [13] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen F. Siegel, Rajeev Thakur, William Gropp, Ewing L. Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Formal analysis of MPI-based parallel programs. *Commun. ACM*, 54(12):82–91, 2011.
- [14] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing Second Edition*. Benjamin/Cummings, 2003.

- [15] Ian Hodkinson. lecture notes of modal and temporal logic. note.
- [16] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Denielou, and Nobuko Yoshida. Structuring communication with session types. note.
- [17] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, pages 55–75. Springer, 2011.
- [18] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, pages 122–138. Springer, 1998.
- [19] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.
- [20] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-Safe Eventful Sessions in Java. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 329–353. Springer, 2010.
- [21] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pages 516–541. Springer, 2008.
- [22] Michio Kaku. Tweaking Moore’s Law and the Computers of the Post-Silicon Era. <http://www.youtube.com/watch?v=bm6ScvNygUU/>, April 2012. [Online; accessed 17-April-2013].
- [23] Socrates Katsoulacos. Safety checking for mpi programs via multiparty session types. Master’s thesis, Imperial College London, 2012.
- [24] Rumyana Neykova. Communication assurance with session types. note.
- [25] Rumyana Neykova. Session types go dynamic or how to verify your python conversations. In *PLACES*, 2013.
- [26] Nicholas Ng and Nobuko Yoshida. Practical message-passing programming with parameterised session types. note.
- [27] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, pages 202–218. Springer, 2012.
- [28] Oracle. Socket Programming. <http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html>. [Online; accessed 1-May-2013].
- [29] Dale R. Shires, Lori L. Pollock, and Sara Sprenkle. Program Flow Graph Construction For Static Analysis of MPI Programs. In *Proceedings of the International Conference on Parallel*

and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA, pages 1847–1853. CSREA Press, 1999.

- [30] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. Data-Flow Analysis for MPI Programs. In *2006 International Conference on Parallel Processing (ICPP 2006), 14-18 August 2006, Columbus, Ohio, USA*, pages 175–184. IEEE Computer Society, 2006.
- [31] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.
- [32] The Scribble team. Scribble Language Reference. <http://www.doc.ic.ac.uk/~rhu/scribble/main.html>, July 2013. [Online; accessed 27-August-2013].
- [33] wikipedia. Message Passing Interface. http://en.wikipedia.org/wiki/Message_Passing_Interface. [Online; accessed 18-August-2013].
- [34] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Appendix: Source Files of Test Cases 2.5 and 4.13

Listing 2 : parent.c

```
1 #include "main/spec.h"
2
3 void testWait(){
4
5 if(rank>=0 && rank<=7){
6 MPI_Irecv (buf0, buf_size, MPLINT, rank+1, 0, MPLCOMMLWORLD, &req);
7 MPI_Isend (buf0, buf_size, MPLINT, rank+1, 0, MPLCOMMLWORLD, &req2);
8 MPI_Recv (buf0, buf_size, MPLINT, rank+1, 0, MPLCOMMLWORLD,&status);
9
10 MPI_Wait (&req, &status);
11 MPI_Wait (&req2, &status);
12 MPI_Wait (&req, &status);
13 MPI_Wait (&req2, &status);
14 }
15
16 if(rank>=1 && rank<=8){
17 MPI_Irecv (buf0, buf_size, MPLINT, rank-1, 0, MPLCOMMLWORLD,&req);
18 MPI_Isend (buf0, buf_size, MPLINT, rank-1, 0, MPLCOMMLWORLD,&req2);
19 MPI_Send (buf0, buf_size, MPLINT, rank-1, 0, MPLCOMMLWORLD);
20
21 MPI_Wait (&req2, &status);
22 MPI_Wait (&req, &status);
23 MPI_Wait (&req, &status);
24 MPI_Wait (&req2, &status);
25 }
26 }
27
```

```

28 void helloParent () {
29     printf ("Hello from %s\n", "Parent");
30
31     testWait ();
32 }

```

Listing 3 : uncle.c

```

1 #include "../main/spec.h"
2
3 void testMulti2 () {
4
5     for (int i=0; i<3; i++){
6         if (rank==i)
7             MPI_Isend (buf0, buf_size, MPI_INT, rank+1, 0, MPLCOMM_WORLD, &req2);
8
9         if (rank==i+1)
10            MPI_Recv (buf0, buf_size, MPI_INT, rank-1, 0, MPLCOMM_WORLD, &status);
11     }
12
13 }
14
15 void helloUncle () {
16     printf ("Hello from %s\n", "Uncle");
17
18     testMulti2 ();
19 }

```

Listing 4 : main.c

```

1 #include "spec.h"
2
3
4     int nprocs = -1;
5     int rank = -1;
6     int sendarray [100];
7     char processor_name [128];
8     int namelen = 128;
9
10 int main (int argc, char **argv)
11 {
12     MPI_Init (&argc, &argv);
13     MPI_Comm_size (MPLCOMM_WORLD, &nprocs);
14     MPI_Comm_rank (MPLCOMM_WORLD, &rank);
15     MPI_Get_processor_name (processor_name, &namelen);
16
17     MPI_Comm comm=MPLCOMM_WORLD;
18
19     helloUncle (); //testMulti2 ();
20     helloSib (); //testDiffRanksCallSameMethod ();
21     helloParent (); //testWait ();
22     helloChild (); //testMultiSenderAndMultiReceiver ();

```

```

23
24
25 MPI_Barrier (comm); //release mode: 7.156s
26
27
28 int check;
29 printf ("Checking if processor is available...");
30 if (system(NULL)) puts ("Ok");
31     else exit (EXIT_FAILURE);
32 printf ("Executing command DIR...\n");
33 check=system ("dir");
34 printf ("The value returned was: %d.\n",check);
35
36
37 MPI_Finalize ();
38 printf ("(%d) Finished normally\n", rank); return 0;
39
40 }

```

Listing 5 : sib.c

```

1 #include "spec.h"
2
3
4 void testFor2 () {
5 for (int i=2; i<7; i++){
6
7 if (rank>=0 && rank<7)
8 MPI_Bcast (buf0 , buf_size , MPI_INT , i , MPLCOMM_WORLD) ;
9 else
10 MPI_Bcast (buf0 , buf_size , MPI_INT , i , MPLCOMM_WORLD) ;
11
12 if (rank>2 && rank <8)
13 MPI_Send (buf0 , buf_size , MPI_INT , rank+2, 0 , MPLCOMM_WORLD) ;
14
15 if (rank>=5 && rank<=9)
16 MPI_Recv (buf0 , buf_size , MPI_INT , rank-2, 0 , MPLCOMM_WORLD,&status);
17
18
19 for (int j=17; j>=0; j--){
20 if (rank>1 && rank <9){
21 MPI_Send (buf0 , buf_size , MPI_INT , 1 , 0 , MPLCOMM_WORLD) ;
22 MPI_Recv (buf0 , buf_size , MPI_INT , 1 , 0 , MPLCOMM_WORLD,&status) ;
23 }
24
25 if (rank==1)
26 for (int c=2; c<=8; c++){
27 MPI_Recv (buf0 , buf_size , MPI_INT , c , 0 , MPLCOMM_WORLD,&status) ;
28 MPI_Send (buf0 , buf_size , MPI_INT , c , 0 , MPLCOMM_WORLD) ;
29 }
30 }
31
32 }}

```

```

33
34 void testDiffRanksCallSameMethod () {
35     if (nprocs > 9 && nprocs < 20) {
36         if (rank == 0) {
37             testFor2 ();
38         } else {
39             testFor2 ();
40         } else {
41             MPI_Barrier (MPLCOMM_WORLD);
42             testFor2 ();
43         }
44
45
46
47 void helloSib () {
48     printf ("Hello from %s\n", "Sibling1");
49
50     testDiffRanksCallSameMethod ();
51
52 }

```

Listing 6 : spec.h

```

1 #ifndef spec_H
2 #define spec_H
3
4 #include <stdio.h>
5 #include <string.h>
6 #include <mpi.h>
7 #include <stdlib.h>
8
9 #define buf_size 128
10 int buf0 [buf_size];
11 int buf1 [buf_size];
12 MPI_Status status;
13 MPI_Request req;
14 MPI_Request req2;
15
16 extern int nprocs;
17 extern int rank;
18 extern int sendarray [100];
19 extern char processor_name [128];
20 extern int namelen;
21
22 void helloUncle ();
23 void helloSib ();
24 void helloParent ();
25 void helloChild ();
26
27
28 #endif

```

Listing 7 : child.c

```

1 #include "../spec.h"
2
3 void testMultiSenderAndMultiRecver() {
4     if(rank >= 3 && rank < 8)
5     for(int i=0; i<3; i++)
6     MPI_Isend (buf0, buf_size, MPLINT, i, 0, MPLCOMM_WORLD, &req2);
7
8     if(rank >= 0 && rank < 3)
9     for(int i=3; i<8; i++)
10    MPI_Recv (buf0, buf_size, MPLINT, i, 0, MPLCOMM_WORLD, &status);
11
12 }
13
14
15
16 void helloChild() {
17
18     printf ("Hello from %s\n", "child");
19
20     testMultiSenderAndMultiRecver();
21
22 }

```

Listing 8 : protocol for the whole program

```

1
2 /*The protocol generated is NOT stable!*/
3
4 /*The current protocol is only applicable when number of processes is 100*/
5
6 /*There is imperfect matching of MPI operations; Plz use perfect matching.*/
7
8 const N=100;
9
10 global protocol main_ProToCoL (role MPLCOMM_WORLD[0..N-1])
11 {
12     foreach (i:0..2) {
13     Data(MPLINT) from MPLCOMM_WORLD[rank:i] to MPLCOMM_WORLD[rank+1];
14
15     }
16
17     choice at MPLPROGRAM
18     {
19     foreach (i:2..6) {
20     Data(MPLINT) from MPLCOMM_WORLD[i] to MPLCOMM_WORLD[0..N-1]
21     }
22
23     foreach (i:2..6) {
24     Data(MPLINT) from MPLCOMM_WORLD[rank:3..7] to MPLCOMM_WORLD[rank+2];
25
26     }

```

```

27
28 foreach (i:2..6){
29 foreach (j:17..0){
30 Data(MPLINT) from MPLCOMMLWORLD[2..8] to MPLCOMMLWORLD[1];
31
32 }
33 }
34
35 foreach (i:2..6){
36 foreach (j:17..0){
37 foreach (c:2..8){
38 Data(MPLINT) from MPLCOMMLWORLD[1..1] to MPLCOMMLWORLD[c];
39
40 }
41 }
42 }
43 }
44
45 or
46
47 {
48 foreach (i:2..6){
49 Data(MPLINT) from MPLCOMMLWORLD[i] to MPLCOMMLWORLD[0..N-1]
50
51 Data(MPLINT) from MPLCOMMLWORLD[rank:3..7] to MPLCOMMLWORLD[rank+2];
52
53
54 foreach (j:17..0){
55 Data(MPLINT) from MPLCOMMLWORLD[2..8] to MPLCOMMLWORLD[1];
56
57
58 foreach (c:2..8){
59 Data(MPLINT) from MPLCOMMLWORLD[1..1] to MPLCOMMLWORLD[c];
60
61 }
62 }
63 }
64 }
65
66 Data(MPLINT) from MPLCOMMLWORLD[rank:0..7] to MPLCOMMLWORLD[rank+1];
67
68
69 Data(MPLINT) from MPLCOMMLWORLD[rank:1..8] to MPLCOMMLWORLD[rank-1];
70
71
72 Data(MPLINT) from MPLCOMMLWORLD[rank:1..8] to MPLCOMMLWORLD[rank-1];
73
74
75 foreach (i:0..2){
76 Data(MPLINT) from MPLCOMMLWORLD[3..7] to MPLCOMMLWORLD[i];
77
78 }
79 }

```


Listing 9 : A ring topology [6]

```

1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char **argv)
5 {
6     int my_rank, ncpus;
7     int left_neighbor, right_neighbor;
8     int data_received=-1;
9     int tag = 101;
10    MPI_Status statSend, statRecv;
11    MPI_Request reqSend, reqRecv;
12
13    MPI_Init(&argc, &argv);
14    MPI_Comm_rank(MPLCOMM_WORLD, &my_rank);
15    MPI_Comm_size(MPLCOMM_WORLD, &ncpus);
16
17    left_neighbor = (my_rank-1 + ncpus)%ncpus;
18    right_neighbor = (my_rank+1)%ncpus;
19
20    MPI_Isend(&my_rank, 1, MPI_INT, left_neighbor, tag, MPLCOMM_WORLD, &reqSend); // comm
    start
21    MPI_Irecv(&data_received, 1, MPI_INT, right_neighbor, tag, MPLCOMM_WORLD, &reqRecv);
22
23    // maybe do something useful here
24
25    MPI_Wait(&reqSend, &statSend); // complete comm
26    MPI_Wait(&reqRecv, &statRecv);
27
28    printf("Among %d processes, process %d received from right neighbor: %d\n",
29        ncpus, my_rank, data_received);
30
31    // clean up
32    MPI_Finalize();
33    return 0;
34 }

```