Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

SYNTHESIS OF CORRECT-BY-CONSTRUCTION MPI PROGRAMS

Filipe Emanuel Ventura Pires de Matos Lemos

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software
2014

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

SYNTHESIS OF CORRECT-BY-CONSTRUCTION MPI PROGRAMS

Filipe Emanuel Ventura Pires de Matos Lemos

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos e co-orientada pelo Prof. Doutor Eduardo Resende Brandão Marques

2014

Agradecimentos

Quero agradecer a todas as pessoas que de alguma forma contribuiram para o sucesso desta tese.

Em primeiro lugar um enorme obrigado ao Professor Eduardo Marques, meu orientador, pelo constante encorajamento e apoio. Foi um prazer e um privilégio trabalhar e aprender consigo. Agradeço igualmente ao meu orientador Professor Vasco Vasconcelos por me ter apresentado este projecto desafiante e por me ter aceite na equipa do mesmo. Tenho também de agradecer aos restantes membros do projecto ATSMP, César Santos e Professor Francisco Martins, por toda a ajuda ao longo da tese.

Um agradecimento muito especial à minha família. Aos meus pais, Emanuel Lemos e Antónia Lemos, por todo o afecto e por sempre me ajudarem e incentivarem a ser melhor. Ao meu irmão Daniel por me ter aturado durante toda a vida e por participar em todas as minhas brincadeiras. Sem vocês não teria chegado aqui.

Não podia faltar um agradecimento à minha amada Bianca Lai, por iluminar todos os dias da minha vida com o seu sorriso, por fazer de mim um homem melhor, por toda a ternura e pelo apoio incondicional. Dás rumo à minha vida.

Por fim, um grande abraço a todos os que me acompanharam durante o meu percurso académico na FCUL e que me ajudaram a trilhar este caminho. Em especial ao Kleomar Almeida que tantas noitadas fez comigo, à Lara Caiola pela constante boa disposição, ao Tiago Silva por todas as maluqueiras e ao Míguel Simões por todo o companheirismo e amizade.

Ao meu irmão João, espero fazer-te orgulhoso

Resumo

MPI (Message Passing Interface) é o padrão de programação concorrente mais usado em super-computadores. Devido à sua portabilidade, diversidade de funções e desenvolvimento constante, MPI é usado mundialmente em diversos campos de investigação, como na pesquisa de curas para o cancro, previsões metereológicas, investigação de fontes de energia alternativas e mais. Este tipo de programas contém trocas de mensagens complexas e operações colectivas que dificultam a sua implementação. Independentemente do talento ou cuidado do programador, é fácil introduzir erros que bloqueiam toda a execução do programa e a sua análise torna-se igualmente complicada. Dada a complexidade e importância das áreas em que o MPI é usado, ferramentas que facilitem o desenvolvimento e análise deste tipo de programas são necessárias. Ferramentas actuais para análise de código MPI usam técnicas refinadas como a verificação de modelos ou execução simbólica. Na maioria dos casos estas soluções não acompanham a escalabilidade dos programas, sendo fiáveis até 1024 processos, pouco quando comparado com as aplicações MPI que correm em computadores com milhares de processadores.

Os programadores de aplicações paralelas têm lidar com duas partes principais: a parte de computação e a parte de comunicação. Computação representa a parte lógica e aritmética do programa, enquanto que a coordenação refere-se à interação entre processos / participantes.

A programação paralela inclui frequentemente programação de baixo nível. Mesmo primitivas MPI usadas para enviar mensagens contém um mínimo de seis parâmetros. Ao desenvolver aplicações paralelas complexas, lidar com funções de baixo nível pode ser um aborrecimento. Se de alguma forma fosse possível contornar esta questão, o desenvolvimento de aplicações paralelas seria muito mais fácil. Misturar computação com comunicação complica ainda mais esta tarefa, pelo que há uma necessidade de dividir as duas partes, tanto para fins de modificabilidade como de organização.

Esqueletos de aplicações paralelas podem ser definidos de muitas maneiras, mas o conceito principal é que o esqueleto permite que os programadores tenham uma visão de topo da parte de comunicação do programa, dando-lhes uma estrutura correcta

à partida, aliviando o seu trabalho. Assim sendo, o foco do desenvolvimento recai na parte de computação do programa.

A ferramenta MPI Sessions, criada antes do ínicio desta tese mas na qual esta se insere, difere das restantes ferramentas de verificação MPI pela possibilidade de usar análise estática para verificar propriedades ligadas à concorrencia, possivelmente em tempo polinomial. A framework, criada antes do ínicio desta tese, baseia-se na teoria dos tipos de sessão multi-participante.

A teoria dos tipos de sessão passa por caracterizar o padrão de comunicação entre vários processos de um programa e, verificando-se que os participantes seguem o protocolo de comunicação especificado, é garantido que certas propriedades como a ausencia de interbloqueio e coerencia de tipos são garantidas. O desafio encontra-se na forma de analisar se certo programa C+MPI obedece a um protocolo de comunicação especificado. Para este fim é usado o VCC, um verificador de programas C desenvolvido pela Microsoft Research.

O processo de uso do MPI Sessions passa por primeiro especificar um protocolo de comunicação, numa linguagem desenvolvida para este fim, num plugin Eclipse que avalia automaticamente se o protocolo está bem formado. Se sim, o plugin traduz automaticamente o protocolo para um formato reconhecido pelo VCC. Para guiar o processo de validação são necessárias anotações no código fonte, nomeadamente nas funções MPI e operadores de controlo de fluxo. Estas anotações ajudam a identificar os campos das funções MPI que vão ser comparadas com o que está especificado no protocolo, a fim de identificar se a ordem de operações no código corresponde à que está no protocolo.

No caso das funções MPI criou-se uma biblioteca especial que contém contractos para cada função. Esta biblioteca é incluida no ficheiro fonte pelo que apenas as anotações relacionadas com o controlo de fluxo têm de ser inseridas manualmente.

Este tese estende as funcionalidades da ferramenta MPI Sessions, possibilitando a síntese de um esqueleto C+MPI fiel ao protocolo de comunicação especificado. Este esqueleto contém a parte de comunicação programa e dita o seu fluxo de execução. O esqueleto é composto por funções MPI e por chamadas a funções a ser definidas pelo utilizador, a parte de computação do programa. A ferramenta complementa automáticamente o código gerado com anotações VCC, permitindo a verificação da fidelidade do código ao protocolo - portanto, o programa contém uma prova que certifica a sua correcção.

As funções de computação complementares são fornecidas pelo utilizador sob a forma de funções “callback”, usadas para definir os buffers de dados a serem transmitidos ou armazenados durante/após a comunicação, as condições envolvidas na implementação de ciclos e escolhas, e as funções de processamento de dados. Devido a uma especificação de protocolos aumentada, com marcas simples que identificam

os pontos de funções "callback", é possível ligar esqueleto ao código a ser definido pelo utilizador. A ferramenta consome estas marcas e gera um cabeçalho C com os protótipos necessários (assinaturas) para as funções "callback" que serão implementadas pelo utilizador. O código gerado e o definido pelo utilizador podem então ser compilados juntamente com a biblioteca MPI instalada no sistema para obter um executável.

Em suma, a nossa abordagem tem as seguintes características principais:

1. Protocolos de comunicação apartir dos quais é feita a síntese automática de programas C+MPI. As etapas manuais do desenvolvimento do programa são apenas a definição do próprio protocolo, e a implementação de funções "callback" necessárias ao controle do fluxo de dados e as funções de processamento.

2. A correção do código sintetizado é certificada, também de forma automatizada, através das anotações de verificação que são copuladas com o esqueleto C+MPI.

3. A abordagem utiliza a infra-estrutura MPI Sessions já em existente para a especificação e verificação de protocolos MPI. Os aperfeiçoamentos compreendem a síntese "back-end" que se alimenta da estrutura de protocolos e o uso de marcas adicionais dentro do protocolo para definir funções "callback".

As nossa solução foi testada com recurso a sete programas MPI, obtidos em livros de texto e da "suite" de "benchmark" FEVS [2]. Através da especificação de protocolos de comunicação, baseados nesses programas, comparámos os tempos de execução dos programas síntetizados com os programas originais (a fim de verificarmos a qualidade dos programas gerados), o número de linhas de código geradas automáticamente, a diferença de linhas de código entre as duas versões (a fim de termos uma primeira noção se o esforço do utilizador é diminuído seguindo o nosso método). É também apresentada uma análise dos tempos de verificação para cada um dos exemplos e as anotações geradas automáticamente necessárias para essa verificação.

Concluíndo, é definido o processo de síntese de código C+MPI funcional e correcto, com base em protocolos de comunicação aumentados para esse propósito e que seguem a teoria de tipos de sessão multi-participante. São explicadas as alterações feitas ao plugin para acomodar as mudanças necessárias para suportar a síntese de código, bem como outras funcionalidades adicionadas para facilitar o uso do plugin. É validado o nosso trabalho através de vários testes que avaliam tanto a parte funcional dos programas gerados (tempo de execução), bem como a vantagem da nossa solução face ao processo desenvolvimento normal - os programas gerados são correctos por construção. Na conclusão são propostas várias ideias para

a continuidade do melhoramento não só da vertente de síntese da ferramenta, bem como do projecto como um todo. O resultado é uma ferramenta funcional e útil, disponibilizada publicamente na internet, incluindo exemplos prontos a usar e um manual práctico.

Palavras-chave: MPI, tipos de sessão, síntese de código, verificação formal, programação paralela

Abstract

Message Passing Interface (MPI) is the *de facto* standard for programming high performance parallel applications, with implementations that support hundreds of thousands of processing cores. MPI programs, written in C or Fortran, adhere to the Single Program Multiple Data (SPMD) paradigm, in which a single program specifies the behavior of the various processes, each working on different data, and that communicate through message-passing.

Errors in MPI programs are easy to introduce, given the complexity in designing parallel programs and of MPI itself, plus the low-level and unchecked semantics of C and Fortran. One can easily write programs that cause processes to deadlock waiting for messages, or that exchange data of mismatched type or length. In the general case, it is impossible to certify that a program avoids these errors, or that it overall complies with an intended communication pattern.

To deal with this general problem, we propose the synthesis of C+MPI programs from well-founded protocol specifications, based on the theory of multiparty session types. The idea is to depart from a high-level protocol specification that defines an intended communication pattern, and automatically synthesize a C+MPI code skeleton that is faithful to the source protocol specification. To define the functionality of a complete program, the skeleton needs only to be complemented by user-provided functions that do not employ any MPI calls, and merely implement the bindings for the skeleton's control and data flow logic. The underlying multiparty session types' theory ensures the preservation of key properties from protocol specification to synthesized code, such as deadlock freedom and type safety. For certification of the whole synthesis process, the C+MPI skeleton also includes annotations that are processed by a software verifier to mechanically verify compliance with the protocol.

This thesis presents the design, implementation, and evaluation of the proposed approach, within a software framework called MPI Sessions.

Keywords: MPI, session types, parallel programming, code synthesis, software verification

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Message Passing Interface (MPI) [4] is the *de facto* standard for programming high performance parallel applications, with implementations that support hundreds of thousands of processing cores, like the Blue Gene/P machine in Figure 1.1 that has with 163,840 cores and attains a peak performance 557 TeraFlops/sec [6].

Figure 1.1: The Blue Gene/P Open Science – Argonne National Laboratory.

MPI programs are written according to the Single Program Multiple Data (SPMD) paradigm, in which a single program, written in C or Fortran, specifies the behaviour of the various processes, each working on different data. Programs make calls to MPI primitives whenever they need to exchange data. MPI offers different forms of communication, notably point-to-point, collective, and one-sided communication. Working with MPI primitives raises several problems though: one can easily write

programs that cause processes to block indefinitely waiting for messages, or that exchange data of unexpected sorts or lengths. Certifying the correctness of programs in regard to these properties is far from trivial, and stumbles upon a scalability problem. Even advanced techniques such as model checking or symbolic execution allow only for the verification of programs, with a small bound on the number of processes for real-world programs [6, 26, 25]. The verification is further complicated by the different communication semantics and wide variety of MPI primitives. In general, it is also intractable to tell if a program follows an intended communication pattern.

1.2 Proposal

In the Advanced Type Systems for Multicore Programming (ATSMP) project [20], concurrency abstractions based on type systems and static analysis methods have been considered for multicore systems. In particular, we considered the verification of conformance of existing C+MPI programs against protocol specifications based on the theory of multiparty session types [14, 15]. Although we had some interesting verification results, such as scalable verification times that are independent on the number of processes [15], the process is partially hindered by a non-trivial process of program annotation that is required for verification.

In this thesis, we consider the orthogonal approach of deriving programs from protocol specifications, to explore the possibility of automated generation of correct programs. The idea is that the protocol works as skeleton for the program's behavior, and C+MPI code is automatically synthesized in adherence to the protocol. Key properties are then preserved from protocol specification to synthesized code, such as deadlock freedom and type safety. For certification of the whole synthesis process, the generated code may also include annotations that are processed by a software verifier, employing the pre-existing framework for MPI program verification. In this sense, synthesized C+MPI code will be an instance of proof-carrying code [21], given that it will "carry" a proof of its own correctness.

The contribution of this thesis comprises the design, implementation, and experimental evaluation of the above methodology. The work has been integrated into our general software toolchain for MPI protocol design and verification, called MPI Sessions. During the process, the tool has been made available online at `http://gloss.di.fc.ul.pt/MPISessions`.

1.3 Thesis structure

The remainder of this thesis is organised as follows:

Chapter 2 provides background on MPI programs, and discusses related work.

Chapter 3 introduces the language used for specifying MPI protocols, and the methodology for verification of protocol compliance by C+MPI programs. We also describe the underlying support provided by the MPI Sessions plugin.

Chapter 4 describes our approach for synthesising correct-by-construction MPI programs in detail. We describe how protocol specifications are slightly extended for synthesis, the synthesis of C+MPI code together with verification annotations from these protocols, the definition of user-code

Chapter 5 describes the main implementation aspects of this thesis' work within the MPI Sessions toolchain.

Chapter 6 provides an evaluation of our approach for example MPI programs.

Chapter 7 discusses our conclusions and proposes plans for future work.

Chapter 2

Background and Related Work

This chapter concentrates on providing a background on the concepts required to understand this thesis, such as the Message Passing Interface (MPI) in section 2.1, the Session Type theory (section 2.2) and the VCC verifier (section 2.3). We also provide an overview of the existing verification tools for MPI (section 2.4).

2.1 MPI

MPI [4] is a library standard for message-based parallel programs. The library defines official bindings for the C and Fortran programming languages, and can be used in a variety of computing infrastructures, from networks of workstations to supercomputers. The library interface comprises a great variety of primitives, accessible as functions to C/Fortran programs, for aspects such as point-to-point communication, collective communication, definition of communication topologies, or process groups. MPI programs typically have an SPMD (Single Program Multiple Data) nature, i.e., a program consists of a set of processes that run the same code but each have a distinct memory address space. During execution, data is exchanged among processes using MPI primitives, and the behaviour of each process may typically differ according to the unique identifier of each process, known as the process rank.

The C+MPI program of Figure 3.4 illustrates the basic traits of MPI. The program at stake is a simple one that calculates the value of pi up to some tolerance using numerical integration. The program starts with `MPI_Init` (line 7) which initializes the MPI execution environment, whereas `MPI_Finalize` on line 36 terminates it. The calls to `MPI_Comm_rank` (line 8) and `MPI_Comm_size` (line 9) yields the process rank and the total number of processes.

If the process rank id is 0 then it reads the number of intervals for the pi integration and enters a loop where it sends a message with that value to every other process (lines 13–16). If the process rank is not 0 then it waits to receive the message

```
1  int main(int argc, char **argv) {
2    int n, rank, procs, i;
3    double PI25DT = 3.14159265358979323846264 3;
4    double mypi, pi, h, sum, x;
5    MPI_Status status;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &procs);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10   if (rank == 0) {
11     printf("Enter the number of intervals: ");
12     scanf("%d",&n);
13     for (i = 1; i < procs; i++) {
14       // Send message to every other process
15       MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
16     }
17   } else{
18     // Receive a message from process 0
19     MPI_Recv(&n, 1, MPI_INT, 0 , 0, MPI_COMM_WORLD, &status);
20   }
21   // Computation
22   h = 1.0 / (double) n;
23   sum = 0.0;
24   for (i = rank + 1; i <= n; i += procs) {
25     x = h * ((double)i - 0.5);
26     sum += (4.0 / (1.0 + x*x));
27   }
28   mypi = h * sum;
29   // Reduction using sum of mypi from all processes
30   // Value becomes available at process 0.
31   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
32   if (rank == 0) {
33     printf("pi is approximately %.16f, Error is %.16f\n",
34     pi, fabs(pi - PI25DT));
35   }
36   MPI_Finalize();
37   return 0;
38 }
```

Figure 2.1: Pi MPI+C program.

from the process with rank 0.

The `MPI_Send` and `MPI_Recv` point-to-point primitives are used to send and receive data from other processes, respectively. These primitives should be assumed as synchronous, meaning a call to `MPI_Send` only returns when the corresponding `MPI_Recv` has started. After the message exchange, every process performs some computation where its local pi is calculated (lines 21–28). `MPI_Reduce`(line 31) is a collective operation in which every process partakes, used to reduce a set of numbers to a smaller set of numbers via functions such as sum, max, min and so on. In this cased it is used to sum every process local value of pi, and thus the value of pi is obtained.

The pi calculation executes without any problems, though bugs can easily be introduced leading up to deadlock. For instance, take `MPI_Recv` at line 19. Each `MPI_Send` needs a matching `MPI_Recv` as these are blocking primitives. So, if line 19 was removed no process would be waiting to receive a message from process 0, so the execution would block on the first message exchange.

Also, if the `MPI_Reduce` (line 31) was inside the conditional statement in line 32, the program would block because only process 0 is attempting to perform the reduce operation, which requires all processes to participate.

2.2 Session Types

Session types are used to express the interactions between multiple participants in a concurrent program. When the concept was first introduced [10], only linear binary interactions where considered. The theory was later generalised to multiple participants [9], so called multiparty session types.

The idea behind multi-party session types starts from a global protocol that expresses the interaction among multiple participant processes. From the global protocol, a local protocol (projection) can be derived for each participant, representing its role (the participant's local view) in the global protocol, as depicted in Figure 2.2.

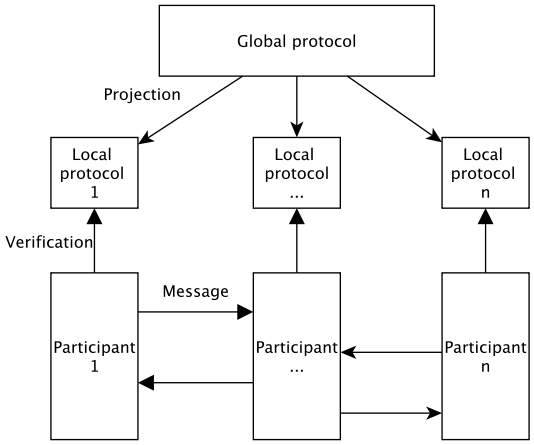

Figure 2.2: Multi party session types structure.

Well-formed protocols guarantee by construction properties such as deadlock freedom or type safety. These properties are preserved by participants in a program, as long as each participant conforms to the corresponding local protocol.

2.2.1 Session Java and Session C

Session Java [11] is the first example of a full implementation of a session language based on session types integrated in an object oriented language - Java. The interaction between participants is expressed by session types specifications. Session Java relies on its compiler to verify that a given implementation conforms to the corresponding protocol and also uses run-time checks to ensure correctness. Session C [22] follows the same idea for C programs as Session Java. Both toolchains depend on individual protocol specification for each participant and are tailored for asynchronous communication, as it was built around the multi-agent session type theory. To note that for each participant a specific session type must be given.

In the MPI Sessions toolchain, described next, a single protocol is used to describe the global interaction between processes. It supports synchronized communication, where message order matters, causing some difficulties employing the Session Type theory. Also, the toolchain supports collective choices which are not considered in Session C or Session Java.

2.3 The VCC verifier

VCC[1] is a sound verification tool used to check functional properties of concurrent C code. It uses annotations for pre and post-conditions, invariants, state assertions and ghost code, in the same vein of other tools like ESC/Java [3]. VCC tests each method in isolation solely based on the contracts of the said method. When running VCC, after the given C source code is analysed and deemed valid, it is translated to Boogie code [13], an intermediate language used by multiple verifications tools. That code is verified by the Z3 SMT [18] solver to assert the correctness of the Boogie/VCC code.

2.4 Formal verification of MPI programs

2.4.1 MUST

MUST [8] is a verification tool based on runtime analysis, in succession to other tools for MPI verification, Marmot[12] and Umpire[29]. It relies on PnMPI[24] structure to dynamically link tools to MPI programs, removing the need to recompile MPI applications to comply with a given tool and allowing code reuse (between tools) through modules.

MUST is used to detect deadlocks and lost messages. It needs state information to perform checks, so it relies on extra processes called “state trackers” to keep this informations. MUST guarantees Correctness and is proved to support full MPI tracing, up to 1024 processes.

2.4.2 DAMPI

DAMPI [30] is a dynamic analyzer for MPI programs, centered around Lamport clocks, used to maintain the causality between communication exchanges. Lamport clocks are imprecise, but were chosen as an alternative to Vector clocks which aren't scalable. DAMPI captures deadlocks and resource-leaks. Like MUST[8] it uses PnMPI[24] to analyze MPI calls so there is no need to change the MPI program code. Completeness is not assured. DAMPI was tested on medium to large benchmarks running on thousands of processes.

2.4.3 ISP

ISP [28] is a MPI verification tool that uses run-time model checking methods to examine possible execution paths. It detects deadlocks and local assertion violations. The authors use a solution that reduces inter-leavings exploitation, increasing scalability compliance. ISP has been tested on current laptop computers and provides support for applications running with 32 processes [6].

2.4.4 TASS

TASS [6] is a suite of tools used to verify safety properties of concurrent programs in MPI+C, such as deadlocks or out-of-bound references. It uses symbolic execution and was designed to analyse computer science programs. TASS also asserts if a parallel program is equivalent to a sequential program, that is, if given the same input both programs produce the same output.

2.4.5 Automatic skeleton extraction

Sottile et al. [27] present a framework used to generate MPI program skeletons, used for performance analysis and benchmarks. The process used to obtain a skeleton is based on using annotations in the MPI source code to perform static slicing. This approach was followed because creating a MPI skeleton manually is subject to human errors, deriving it from source code directly guarantees that it represents the intended program. These annotations are introduced manually by the user. Compile directives help the user control the output.

Chapter 3

Specification and Verification of MPI Protocols

This chapter presents the main support for the specification and verification of communication protocols for C+MPI programs, embedded in the MPI Sessions software toolchain. We start (Section 3.1) with an overview of the main traits of protocol specification and verification. We then describe protocol specification and verification in turn. Regarding protocol specification (Section 3.2), we cover the custom language used for specifying protocols, present example protocols, and describe an Eclipse plugin for protocol specification. As for protocol verification (Section 3.3), we describe the support for verifying C+MPI programs using the VCC software verifier, covering the representation of protocols in VCC logic, the definition of verification contracts for MPI primitives, the annotation of source code in aid of verification, and an illustration of the use of VCC in practice.

3.1 Overview

The use of MPI Sessions is illustrated in Figure 3.1. The main idea is to assert the program's congruence with the protocol, based on a C+MPI program and a corresponding communication protocol.

The intended work flow starts by developing a protocol specification. This protocol depicts the communication pattern of the program, encoded in a language based on the multi-party session types theory. Our Eclipse plugin validates whether the protocol is well-formed and translates it to VCC verifier syntax.

It is also necessary to annotate the C+MPI source code with verification logic to aid VCC match code and protocol. A library with logic verification and contracts for MPI operations was created for this purpose, and is included in the source code. Annotations related to control flow must be manually added.

Next, both the translated protocol and the annotated code are submitted to VCC to check if the code complies with the protocol. The verifier also checks if the

protocol is well-formed. Lastly, VCC does an extensive verification to assert if any of the wanted properties is not satisfied.

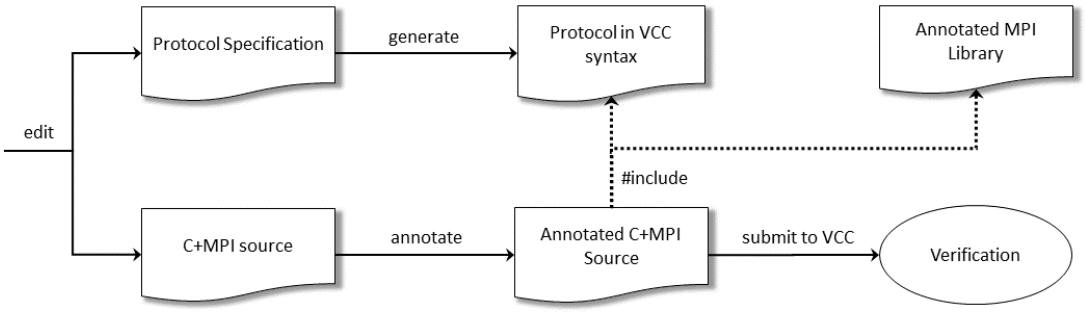

Figure 3.1: The MPI Sessions overview.

3.2 Protocol language

P	::= skip	empty protocol
	| message i , i D	point-to-point message
	| broadcast i x: D	`MPI_Bcast`
	| scatter i D	`MPI_Scatter`
	| gather i D	`MPI_Gather`
	| reduce op D	`MPI_Reduce`
	| allgather D	`MPI_Allgather`
	| allreduce op D	`MPI_Allreduce`
	| val x: D	variable
	| P; P	sequence
	| foreach x: $i..i$ do P	repetition
	| loop P	collective loop
	| choice P or P	collective choice
D	::= integer | float | $D[i]$ | $\{x\colon D \mid p\}$ | ...	refined types
i	::= x | n | $i + i$ | max(i, i) | length(i) | $i[i]$ | ...	index terms
p	::= true | $i \leq i$ | p and p | $a(i, \ldots i)$ | ...	index propositions
op	::= max | min | sum | ...	reduction

Figure 3.2: Grammar for the protocol language.

Protocols are specified using a custom language. An outline of the language grammar is shown in Figure 3.2. We first go through a description of each language

construct, then present two example protocols. The constructs of the language are as follows:

- The `skip` term represents the empty protocol. It is matched by any program fragment without any communication logic.
- A `message a, b D` term represents a blocking point-to-point message from `a` to `b` with data of type `D`. It is matched by a `MPI_Send` call to `b` in the process with rank `a`, and by a `MPI_Recv` call from `a` in the process with rank `b`. Grammar-wise, the `a` and `b` arguments are index terms, defined as constants, variables in context, or inductively using arithmetic operators. As for `D`, it is a refined type instance, representing a primitive integers or floating point number, an arrays of those types, possibly with additional domain constraints. For example, `val n : { x : integer | x % 5 = 0 }` defines that variable n is an integer divisible by 5, and `val n : { x : float[] | len(x)> 3 and x[0] > 5 = 0 }` defines an array of floating points values with 3 elements, where the value of the first element is greater than `5`.
- The `broadcast`, `scatter`, `gather`, `reduce`, `allgather`, `allreduce` constructs are in correspondence to MPI collective communication primitives. Like `message`, index terms and refined types are employed for process ranks and message payloads. For instance, `gather 0 float` term represents a `MPI_Gather` operation, with data of type `float` gathered at rank `0`. The `op` argument in `reduce` and `allreduce` is in correspondence to MPI reduction operators (e.g., `max` for `MPI_MAX`). A `broadcast r x:D` term introduces variable `x`, representing a global value that is shared by all processes.
- A `P; Q` term, where `;` is the sequence operator, represents protocol `P` followed by protocol `Q`. The control flow of a matching program must first match `P`, then `Q` in sequence. For instance `message 0 1 float; broadcast 0 integer` represents a point-to-point message, followed by a broadcast operation.
- A `foreach x:a..b P` term stands for the repetition in sequence of `P` for variable `x` ranging from `a` to `b`, i.e., it is equivalent to `P [x / a] ; P [x / a + 1]; ... ; P [x / b]` where `P [x / v]` stands for `P` with `x` replaced by `v`. For instance, the term `foreach i:1..10 message 0, i integer` stands for the a sequence of messages sent by process `0` to all processes from `1` to `10`.
- A `choice P or Q` term specifies a collective choice between protocols `P` and `Q`, i.e., participant processes should either all branch to `P` or all to `Q`. Typically, a term of this kind will be matched by a program fragment of the form `if (cond)BlockP else BlockQ` , such that `cond` is guaranteed to evaluate

the same (true or false) on all processes, and, as such, all of them proceed to `BlockP` and then match `P`, or to `BlockQ` and match `Q`.

- A **`loop`** `P` protocol defines a collective loop executed in synchrony by all program participants, i.e., `P` will be executed the same number of times (zero or more) for all participant processes. Protocols of this kind are matched by program loops, e.g., **`while`** `(cond){ BlockP }`, such that `BlockP` matches `P` and where, as in the case of collective choices, the loop condition is guaranteed to evaluate the same for all processes in synchrony.

- A **`val`** `x:D` term introduces a variable in the protocol that represents a global value that is known by all processes, e.g. a value needed from the command line.

3.2.1 Examples

We now present two examples of protocols. The protocols are encoded in a concrete syntax that instantiates the protocol grammar, and that is recognized by the Eclipse plugin tool, described later on. The first protocol is for the simple Pi program of Chapter 3, illustrating the use of communication primitives, sequential composition, and **`foreach`** terms. The second protocol is for a Finite Differences program, which is presented in context, and illustrates more complex aspects, such as the matching of collective control flow using **`choice`** and **`loop`** constructs.

Pi

```
1  protocol pi p: {x: integer | x > 1}{
2    foreach i: 1 .. p-1 {
3      message 0, i {x: integer | x > 1}
4    }
5    reduce 0 sum float
6  }
```

Figure 3.3: Protocol specification for pi calculation program.

In Figure 3.3 we present a communication protocol for the example Pi calculation program of Chapter 2. The first line names the protocol, `pi`, and introduces variable `p` to represent the total number of processes. The refined type domain for `p: {x: integer | x > 1}`, imposes that there should more than one process. In line with the MPI program, a **`foreach`** term (lines 2–4) defines a sequence of messages sent from process `0` to all other processes, `1` to `p-1`, **`message`** `0, i {x: integer | x > 1}`, and the protocol ends (line 5) with a reduction operation, **`reduce`** `0` **`sum float`**, to stand for the aggregation of a reduced value at rank `0`.

Finite Differences

```
1  int main(int argc, char** argv) {
2    int procs;              // Number of processes
3    int rank;               // Process rank
4    MPI_Init(&argc, &argv);
5    MPI_Comm_size(MPI_COMM_WORLD, &procs);
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    ...
8    int n = atoi(argv[1]);            // Global problem size
9    if (rank == 0)
10      read_vector(work, lsize * procs);
11   MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0,
         MPI_COMM_WORLD);
12   int left  = (procs + rank - 1) % procs;  // Left neighbour
13   int right = (rank + 1) % procs;          // Right neighbour
14   int iter = 0;
15   // Loop until minimum differences converged or max iterations attained
16   while (!converged(globalerr) && iter < MAX_ITER)) {
17     ...
18     if (rank == 0) {
19       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
20       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
21       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
22       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
23     } else if (rank == procs - 1) {
24       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
25       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
26       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
27       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
28     } else {
29       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
30       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
31       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
32       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status)
33     }
34     ...
35     MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
36     ...
37   }
38   ...
39   if (converged(globalerr)) { // Gather solution at rank 0
40     MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0,
           MPI_COMM_WORLD);
41     ...
42   }
43   ...
44   MPI_Finalize();
45   return 0;
46 }
```

Figure 3.4: Finite Differences C+MPI program.

The code of a Finite Differences calculation program, adapted from [5], is shown in Figure 3.4. The program departs from an initial solution X^0 and iteratively calculates X^1, X^2,..., until a minimum error threshold is attained or a maximum number of iterations is executed. The program has some features that are similar to the Pi program of Chapter 2, but additionally illustrates collective flow control logic.

As usual, the Finite Differences program begins by initializing the MPI environment, and reading the number of processes and the process rank (lines 4–6). Rank `0` starts by reading the input vector X^0 (line 10) and then distributes it by all participants (line 11, call to `MPI_Scatter`), such that each process is responsible for the calculation of a local part of the vector. The program then enters a loop (lines 15–

35), specifying point-to-point message exchanges between each process and its `left` and `right` neighbors, on a ring topology. The various message exchanges distribute boundary values necessary to local calculations, and the different send/receive orders for different ranks (lines 9–22, lines 24–27, and lines 29–32) aim at avoiding deadlock, given the use of blocking (synchronous and unbuffered) primitives, `MPI_Send` and `MPI_Recv`. After value exchange, and still inside the loop, the global error is calculated via a reduction (`MPI_Allreduce`) operation and communicated to all participants (line 35). The loop ends when the convergence condition is attained, or after a pre-defined number of iterations. After the loop, if the algorithm converged, rank `0` gathers the solution, receiving from each participant (including itself) a part of the vector (using `MPI_Gather`, lines 37–38). The program ends with the usual call to `MPI_Finalize`.

The protocol for the Finite Differences program is listed in Figure 3.5.

```
1  protocol fdiff p: {x: integer | x > 1}{
2    val n: {x: positive | x % p = 0} //problem size
3    scatter 0 float[n]
4    loop {
5      foreach i: 0 .. p - 1 {
6        message i, (i - 1 >= 0 ? i-1 : p-1) float
7        message i, (i + 1 <= p-1 ? i+1 : 0) float
8      }
9      allreduce max float
10   }
11   choice
12     gather 0 float[n]
13   or
14     {}
15 }
```

Figure 3.5: Finite differences protocol.

The protocol for the Finite Differences program is listed in Figure 3.5. As in the Pi protocol the number of processes, `p` is bigger than 1 (line 1). The protocol starts with the introduction of `n` (line 2), the problem size, constrained to be a multiple of the number of processes. The protocol then proceeds with the initial data scatter (line 3), followed by the computation loop (lines 4–9). The protocol loop corresponds to the program loop with point-to-point neighbor message exchange, using a **`foreach`** construct (lines 5–8), and a reduction step at the end of each iteration (line 9). Following the loop, the protocol ends with the optional gathering step, encoded by the **`choice`** construct (lines 11–14).

3.2.2 MPI Sessions plugin for Eclipse

The MPI Sessions plugin for Eclipse can be downloaded from the MPI Sessions web page [19]. A screenshot of the plugin is shown in Figure 3.6. The tool provides an environment where one can design protocols and check their good formation, alerting the developer of possible errors. If the IDE doesn't report any errors, then

the protocol is well-formed and can be automatically translated to VCC syntax.

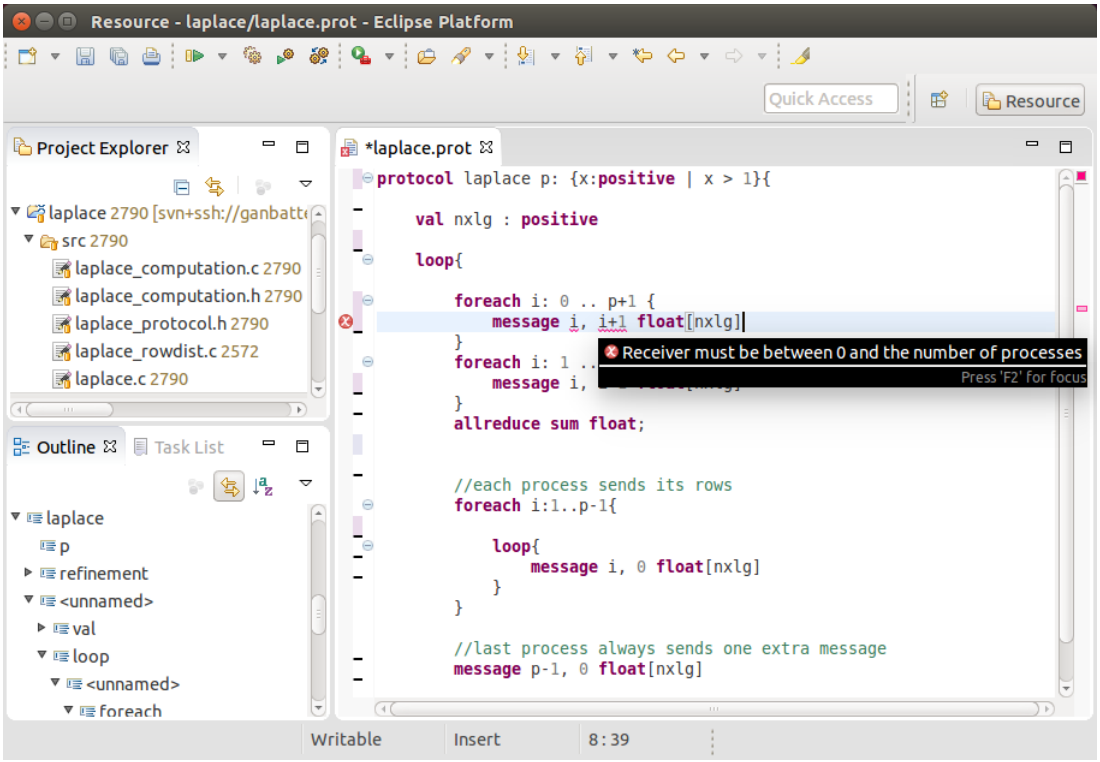

Figure 3.6: Screenshot of the MPI Sessions plugin in use.

3.3 Protocol verification

To match a given C+MPI program against a communication protocol, the MPI Sessions toolchain employs the VCC verifier for C [1]. VCC takes as input the source code of the program, augmented with deductive verification annotations. The necessary annotations for C+MPI code comprise a representation of the communication protocol in VCC form, contracts for MPI function primitives, and ghost annotations in the body of the program.

The goal of these annotations is guiding the progressive match of the protocol against the C+MPI program flow, from MPI initialization (the call to `MPI_Init`) to shutdown (the call to `MPI_Finalize`).

3.3.1 VCC protocol format

For a given protocol, the MPI Sessions Eclipse plugin automatically generates the protocol representation in an abstract syntax tree format understood by the VCC software verifier. As an example, the VCC format for the Finite Differences protocol is listed in Figure 3.7. The protocol is expressed as the result of a ghost

```
1  _(pure Protocol program_protocol ()
2      _(reads {})
3      _(ensures \result ==
4          seq(size(\lambda \integer x; x > 1),
5           seq(abs(\lambda \integer p;
6            seq(val(\lambda \integer x; x > 0 && x % p == 0),
7             seq(abs(\lambda \integer n;
8              seq(scatter(0, floatRefinement(\lambda float* x; \integer x_length; x_length == n)),
9              seq(loop(
10                  seq(foreach(0, p - 1, \lambda \integer i;
11                      seq(message(i, i - 1 >= 0 ? i - 1 : p - 1, floatRefinement(\lambda float* _x5;
                            \integer _x5_length; \true && _x5_length == 1)),
12                      seq(message(i, i + 1 <= p - 1 ? i + 1 : 0, floatRefinement(\lambda float* _x6;
                            \integer _x6_length; \true && _x6_length == 1)),
13                      skip()))
14                  ),
15                  seq(allreduce(MPI_MAX, floatRefinement(\lambda float* _x7; \integer _x7_length; \
                        true && _x7_length == 1)),
16                  skip()))
17              ),
18              seq(choice(
19                  seq(gather(0, floatRefinement(\lambda float* _x10; \integer _x10_length; _x10_length
                        == n / p)),
20                  skip())
21              ,// or
22                  skip()
23              ),
24              skip()))
25             ),
26            skip()))
27           ),
28          skip()))
29      );
30  )
```

Figure 3.7: Protocol for the Finite Differences program in VCC syntax.

`program_protocol` function of type `Protocol`. In spite of a different syntactic style, a `Protocol` term closely resembles the source protocol specification (Figure 3.5).

Almost all constructors have the same name, e.g., **`choice`**, **`loop`**, **`foreach`**, as well as all the constructors for MPI primitives like **`allreduce`** and **`gather`**. The `seq` constructor represents the sequence operator (`;`) and the `skip` constructor corresponds to an empty protocol (`{ }`). The `abs` terms, provide a convenient method to deal with term substitution, and employ VCC anonymous functions (**`\lambda`** terms) with an integer parameter, e.g., as in lines 5 and 7 of Figure 3.7. Refined types are represented by `intRefinement` and `floatRefinement` terms, for integer and floating point data, respectively, which have an associated anonymous VCC function for representing the restriction, e.g., `\floatRefinement(\lambda float * x; \integer x_length; x_length == n)` at line 8 for **`scatter`** in Figure 3.7 is in correspondence to **`float`**`[n]` in the original protocol specification in Figure 3.5, line 3.

3.3.2 MPI primitives contracts

The goal of the deductive verification annotations is guiding the progressive match of the protocol against the C+MPI program flow, from MPI initialization

```
int MPI_Send
(void *buf, int count, MPI_Datatype datatype, int to, int tag, MPI_Comm comm
_(ghost Param* param)
_(ghost Protocol in)
_(out   Protocol out)
)
// Annotations on memory use
_(reads param)
_(maintains \wrapped(param))
// Type-safe memory validation: this must be stated here
_(requires datatype == MPI_INT && count == 1 ==> \thread_local((int*) buf))
_(requires datatype == MPI_INT && count > 1 ==> \thread_local_array((int*) buf, (unsigned) count))
_(requires datatype == MPI_FLOAT && count == 1 ==> \thread_local((float*) buf))
_(requires datatype == MPI_FLOAT && count > 1 ==> \thread_local_array((float*) buf, (unsigned) count)
    )
// Protocol verification
_(requires isMessageTo(head(in, param->rank),to))
_(requires datatype == MPI_INT ==> verifyDataI(depType(head(in, param->rank)), (int*) buf, count))
_(requires datatype == MPI_FLOAT ==> verifyDataF(depType(head(in, param->rank)), (float*) buf, count)
    )
_(ensures out == continuation(in, param->rank))
_(ensures \result == MPI_SUCCESS)
...
```

Figure 3.8: VCC contract for the `MPI_Send` primitive.

(the call to `MPI_Init`) to shutdown (the call to `MPI_Finalize`). The contract of `MPI_Init` initializes a ghost `Protocol` term using the `program_protocol` ghost function described in the previous section, and the `MPI_Finalize` checks that the same ghost variable is equivalent to the empty protocol, `skip()`, at the end. Along the program flow, any MPI communication calls by the program are also matched against the protocol.

To accomplish this, a contract is defined for each MPI primitive that is supported by the toolchain. All such contracts are included in a program's definition through a mock MPI header, i.e., a replacement of the usual `mpi.h` C header. As an example, a fragment of the contract for `MPI_Send` is shown in Figure 3.8. In the traditional style of design-by-contract, the VCC contract of a function defines extra ghost parameters for the function, and lists sets of pre-conditions and post-conditions for the function.

Regarding the three ghost parameters in Figure 3.8, the `Param* param` defines the numerical constraints regarding the number of processes and the rank of the process. Its definition is as follows:

```
typedef struct {
  int procs;
  _(invariant procs >= 2)
  int rank;
  _(invariant rank >= 0)
  _(invariant rank < procs)
}
```

The restrictions are expressed by the invariant conditions: the number of processes, `procs`, is greater than 2, and the process rank is between 0 and `procs-1`. As for the `in` and **`out`** ghost parameters, they are used to represent the protocol before and after the `MPI_Send` primitive is matched, respectively.

The progressive reduction of a protocol is exemplified by the pre and post-conditions of the `MPI_Send` contract. These employ some pre-defined logic in the

form of VCC predicates, functions, and relations for which we omit details (see [15]). The `isMessageTo(head(in, param->rank),to)` pre-condition expresses that a `message(a,b, ...)` term is expected at the head of the protocol, with `a = param->rank` and `b = to`, where `to` is the destination parameter of the `MPI_Send` call. The progress in verification is expressed by the `out == continuation(in, param->rank)` post-condition, which states that the protocol after `MPI_Send` is obtained by removing the `message` term at its head. The other two pre-conditions of the contract involve predicates `verifyDataI` and `verifyDataF`. They are used to verify that the transmitted data—jointly expressed by `datatype`, `buf` and `count`–matches the refined type's restriction for the `message` term that is matched.

3.3.3 Program Annotations

Beyond the base verification logic defined for MPI primitives, the source code of a MPI program must be extended with complementary annotations. A large part of the process can be automated, as described in [15, 16], but some manual annotations may have to be introduced by the programmer. In the following discussion we make no such distinction, since it is only relevant to our purpose to discuss their nature. All such annotations are derived automatically for the synthesized programs discussed in Chapter 4.

The more trivial program annotations are illustrated in Figure 3.9. The C code must include the header file containing the protocol's definition. Secondly, the prototype of the `main` function (the entry point of the program) needs to be changed to introduce two ghost parameters similar to those discussed for MPI primitives, one of type `Param` called `_param`, another one of type `Protocol` called `_protocol`. These ghost variables are added as arguments to each MPI call, as illustrated in the figure for `MPI_Init`, `MPI_Send`, and `MPI_Finalize`. Finally, an `assert \false` statement is typically added after the `MPI_Finalize` call to ensure that the verification is sound. The assertion is supposed to fail. If it is deemed as proved by VCC, this will imply a contradiction in the verification logic.

The second major group of annotations relates to program control flow, and the corresponding match with `choice`, `loop`, and `foreach` protocols. The process is similar for the three types of constructors, and is illustrated in Figure 3.10, by the annotation of a C `while` loop that is matched against a `loop` protocol. We can see that the verification flow is driven by the syntax of both the C loop and the `loop` constructor. Just before the `while` loop, two ghost `Protocol` variables are introduced: the `_loop_body` and `_loop_cont` variables are initialized to hold the body of the `loop` protocol, and its continuation, respectively. Within the `while` loop. the `_protocol` body is initialized to `_loop_body`, and after the `while` loop the `_protocol` is initialized to `_loop_cont`.

```
#include <mpi.h>
#include "program_header.h"
...
int main( int argc, char *argv[]
// VCC {
_(ghost Param* _param)
_(ghost Protocol _protocol)
// }
) {
// Initialization
MPI_Init(&argc, &argv _(ghost param) _(out _protocol));
...
// MPI communication call
MPI_Send(... _(ghost param) _(ghost _protocol) _(out protocol))
...
// Shutdown
MPI_Finalize(_(ghost param) _(ghost _protocol));
// Check sanity of verification
_(assert \false)
}
...
```

Figure 3.9: Basic program annotations.

```
...
// Introduce ghost variables
_(ghost Protocol loop_body = loopBody(_protocol);)
_(ghost Protocol loop_cont = loopCont(_protocol);)
while(fdiff_diverged(__ud))
..
{
// Math loop body
_(ghost _protocol = loop_body;)
...
// verify loop body was entirely matched
_(assert congruent(cleanup(_protocol,__rank), skip()))
}
// verify rest of the protocol
_(ghost _protocol = loop_cont;)
...
```

Figure 3.10: Example of VCC loop annotations.

Finally, some complementary annotations may be required for miscellaneous reasons, e.g., regarding the use of memory, the introduction of values in the programmer, or programmer unstated assumptions which must be made explicit [15, 16]. Some instances of these annotations are presented in Chapter 4.

3.3.4 Verification

In Figure 3.11 we present the results of verifying the finites difference program with VCC. A variety of information is displayed, however, in our case, the relevant data is in lines 10–17 and 32.

In lines 10–13 is reported that errors occurred during verification. This errors are related to the annotations that are purposely intended to always fail in order to guarantee that the verification is sound - if VCC does not report these assertions as false then some contradiction exists in the verification logic. As these are the only errors reported, the verification was successful, code and protocol are congruent.

If at any place the verification detected that code and protocol don't match

```
1  D:\ProgramingPrograms\eclipse-dsl-luna-R-win32\runtime-EclipseXtext\fdiff\src>mpiv fdiff.c
2  -- Executing: vcc /z3:/memory:2048 /time -p:-ID:\ProgramingPrograms\ampi\\include fdiff.c
3  Verification of Param#adm succeeded. [0,81]
4  Verification of swprintf succeeded. [0,02]
5  Verification of vswprintf succeeded. [0,00]
6  Verification of _swprintf_l succeeded. [0,01]
7  Verification of _vswprintf_l succeeded. [0,00]
8  Verification of strnlen_s succeeded. [0,00]
9  Verification of wcsnlen_s succeeded. [0,02]
10 Verification of main failed. [12,09]
11 D:\ProgramingPrograms\eclipse-dsl-luna-R-win32\runtime-EclipseXtext\fdiff\src\fdiff.c(146,11) : error
       VC9500: Assertion '\false' did not verify.
12 Verification of __falseAssertionSanityCheck__ failed. [0,05]
13 D:\ProgramingPrograms\ampi\include\mpi.h(58,53) : error VC9500: Assertion '\false' did not verify.
14 Verification of program_protocol#reads succeeded. [0,01]
15 Verification of paramGetProcs#reads succeeded. [0,02]
16 Verification of paramGetRank#reads succeeded. [0,01]
17 Verification errors in 2 function(s)
18                         Total 15,456
19                   FELT Visitor 0,585
20                   Total Plugin 14,781
21                        Prelude 0,090
22                         Boogie 0,042
23                     Boogie AST 0,000
24                      Boogie AI 0,153
25                  Boogie Resolve 0,034
26                Boogie Typecheck 0,045
27                    VC Optimizer 0,000
28              Boogie Verify Impl. 13,959
29                  Boogie Save BPL 0,000
30                         Pruning 0,000
31                AST transformers 0,235
32 Exiting with 3 (2 error(s).)
33 -- Execution time: 15.63 seconds
```

Figure 3.11: Verification results.

an error would be reported. Altering the Finite Differences protocol to expect a `MPI_Scatter` with root in 1 instead of 0, but maintaining the `MPI_Scatter` with root 0 in the code, results on the error displayed in Figure 3.12. From lines 2–6 we can see the error report, indicating that the scatter operation did not verify and its localization in the code, line 49 of file fdiff.c. It also mentions what pre-condition in the associated function contract failed, giving further hints about the parameter that was not satisfied.

```
1   Verification of main failed. [1,68]
2   D:\ProgramingPrograms\eclipse-dsl-luna-R-win32\runtime-EclipseXtext\fdiff\src\fdiff.c(49,2) :
3     error VC9502: Call '_MPI_Scatter(__outData, (n)/__procs, MPI_FLOAT, __inData, (n)/__procs,
4     MPI_FLOAT, 0, 0xFFFFFFFF _(ghost _param) _(ghost _protocol) _(out _protocol))' did not verify.
5   d:\programingprograms\ampi\include\MPI_Scatter.h(43,14) :
6     error VC9599: (related information) Precondition: 'isScatter(head(in, param->rank),root)'.
7   D:\ProgramingPrograms\eclipse-dsl-luna-R-win32\runtime-EclipseXtext\fdiff\src\fdiff.c(146,11) :
8     error VC9500: Assertion '\false' did not verify.
9   Verification of __falseAssertionSanityCheck__ failed. [0,05]
10  D:\ProgramingPrograms\ampi\include\mpi.h(58,53) : error VC9500: Assertion '\false' did not verify.
```

Figure 3.12: Verification error, incongruence in `MPI_Scatter`.

Chapter 4

Synthesis of correct-by-construction MPI programs

This chapter describes the synthesis of C+MPI programs from protocol specifications. We begin with an overview of the workflow for program synthesis (Section 4.1). We then provide an exposition of the synthesis process in detail (Section 4.2). We end with a description of the changes that were made to the MPISessions Eclipse plugin (Section 4.3).

4.1 Overview

The synthesis of C+MPI programs is illustrated in the diagram of Figure 4.1. As in a standard use of the toolchain (discussed in Chapter 3), we depart from a communications protocol, which is translated automatically to VCC syntax (step 1 in the figure). The novelty (2) is that we also synthesise a C+MPI program skeleton, along with a C header for user-defined callback functions.

Along with the code, the program skeleton also contains verification annotations, that allow protocol fidelity to be verified by VCC—hence, the program carries a proof that can be used to certify its own correctness. The aim is that the resulting code may run as a normal C+MPI program (3), and also be verified for compliance with the protocol using VCC (4). The generation of both code and verification annotations proceeds in syntax-driven (compositional) manner, according to each possible type of protocol construct.

By itself, the C+MPI program skeleton only defines the invocation of MPI primitives and program control flow. The complementary data flow must be supplied by user in the form of callback functions, defining the data buffers to be transmitted or stored to/after communication, the predicates involved in the implementation of loops and choices, and data processing procedures. The skeleton and user-code

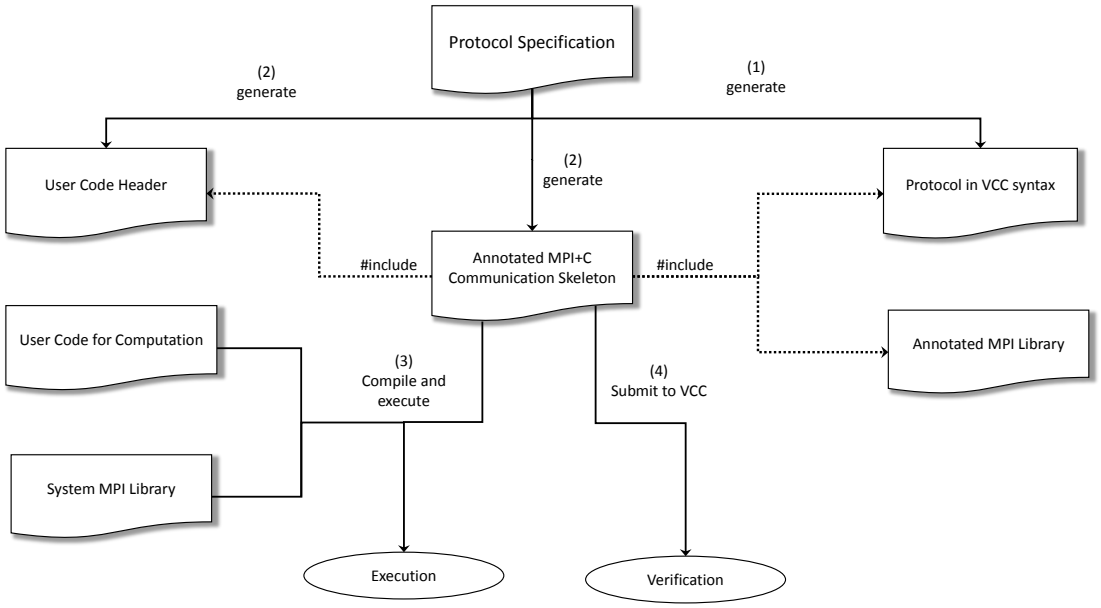

Figure 4.1: Workflow for synthesized C+MPI programs.

are glued together by an augmented protocol specification, thanks to simple marks that identify the callback function points. Feeding on these marks, the toolchain generates a C header file with the necessary prototypes (signatures) for the callback functions that are to be implemented by the user. The skeleton and user code can then be compiled and linked with a host MPI library to obtain an executable binary.

In summary, our approach has the following three core traits:

1. Communication protocols lead to the automated synthesis of C+MPI programs. The manual steps in program development are only the definition of the protocol itself, and the implementation of callback user functions for data flow and processing.

2. The correctness of the synthesised code process is certified, also in automated manner, thanks verification annotations that are bundled together with the skeleton code.

3. The approach uses the MPI Sessions infrastructure already in place for the specification and verification of MPI protocols. The refinements comprise a synthesis back-end that feeds on the protocol structure and the use of extra callback function marks within the protocol.

4.2 The synthesis process

4.2.1 Protocol synthesis marks

We begin by explaining the extension to protocol specification. It amounts only to the introduction of special marks in aid of the synthesis process, with otherwise no effect on the semantic significance of the protocol, i.e., a protocol without marks will be an equivalent one in terms of the prescribed behavior for a program that complies with it.

```
1  protocol @synthesis fdiff p: {x: integer | x > 1} {
2
3    @in getProblemSize
4    val n: {x: positive | x % p = 0}
5
6    @out getInitialSol @in setLocalData
7    scatter 0 {x: float[] | length(x) = n}
8
9    @condition diverged
10   loop {
11     foreach i: 0 .. p - 1 {
12
13       @out getLeftMost @in setRightBorder
14       message i, (i - 1 >= 0 ? i-1 : p-1) float
15
16       @out getRightMost @in setLeftBorder
17       message i, (i + 1 <= p-1 ? i+1 : 0) float
18     }
19     @exec compute
20     @out getLocalError @in setGlobalError
21     allreduce max float
22   }
23
24   @condition converged
25   choice
26     @out getLocalSol @in setFinalSol
27     gather 0 float[n/p]
28   or
29     {}
30 }
```

Figure 4.2: Finite differences protocol with synthesis marks.

We illustrate the extension in Figure 4.2, containing the Finite Differences protocol in a suitable format for program synthesis. The specification makes use of the following marks:

- The **`@synthesis`** mark (line 1) just indicates that the protocol will trigger the synthesis of a program.
- The **`@out`** and **`@in`** marks are used in association to communication operation to identify callback functions that will yield the outgoing (**`@out`**) or incoming (**`@in`**) data buffers. For instance, these annotations at line 6 associate to the following **`scatter`** operation at line 7, and dictate that process 0 (the root process for the operation in this case) should call `getInitialSol()` to obtain the data to scatter, and every process should store the data obtained through the operation in the data buffer defined by `setLocalData()`.
- The **`@condition`** mark identifies a predicates that should be used as condition guards for the loop or choice behavior of the program, as defined by **`loop`** or

choice blocks. In the example, predicates `diverged` (line 9) and `converged` (line 24) respectively associate to the **loop** and **choice** blocks that immediately follow.

- Finally , an **@exec** mark identifies a data computation procedure that should be called at some point in the protocol. In the example, it is used at line 19, implying that procedure `compute` should be called after the implementation of the **foreach** block and just before the **allreduce** operation.

4.2.2 Overall program structure

The general pattern for the program structure of a synthesised C+MPI program is illustrated in Figure 4.3 for the Finite Differences example.

In the structure shown, first note that some C header files of relevance are included (lines 2–4) by the code: the traditional MPI header (`mpi.h`), the C header containing callback function prototypes (`fdiff_computation.h`), and the protocol specification in VCC format (`fdiff_protocol.h`). The skeleton code is then defined entirely in the `main` function, the entry point of the program (in line with the usual C convention). The `main` function signature declares the usual C parameters, plus the ghost parameters (lines 10–11) required for verification (discussed back in Chapter 3). Along with these parameters, a number of variables are then declared

```
1  (...)
2  #include <mpi.h>
3  #include "fdiff_computation.h"
4  #include "fdiff_protocol.h"
5
6  int main
7  (
8    int argc,
9    char *argv[]
10   _(ghost Param* _param)
11   _(ghost Protocol _protocol)
12 )
13 {
14   FDiffUserData* __ud;
15   int __rank;
16   int __procs;
17   void* __inData;
18   void* __outData;
19   int __receiver = 0;
20   int __sender = 0;
21   ...
22   // == INITIALIZATION ==
23   MPI_Init(&argc, &argv _(ghost _param) _(out _protocol));
24   MPI_Comm_size(MPI_COMM_WORLD, &__procs _(ghost _param));
25   MPI_Comm_rank(MPI_COMM_WORLD, &__rank _(ghost _param));
26   __ud = fdiff_init(argc, argv, __rank, __procs);
27
28   // == IMPLEMENTATION OF PROTOCOL ==
29   int p = __procs;
30   (...)
31   // <--
32
33   // == SHUTDOWN ==
34   fdiff_shutdown(__ud);
35   MPI_Finalize(_(ghost protocol));
36   ...
37 }
```

Figure 4.3: Overall structure of synthesized C+MPI programs.

in the body of `main()`. The variables (lines 14–20) and their meaning are as follows:

- `__ud` is a pointer to a user-defined data structure of type `FdiffUserData` that will be manipulated by user callback functions throughout the program;
- `__rank` and `__procs` store the process rank and the number of processes;
- `__inData` and `__outData` store the addresses of incoming and outgoing data buffers, respectively, as returned by callback functions;
- `__sender` and `__receiver` are used to store sender or receiver process ranks, when necessary.

In terms of actual functionality, the program begins with a typical MPI initialization sequence (lines 21–23), followed by a special call to the `fdiff_init()` user function (line 25). The latter informs user code of global runtime parameters, and returns a pointer to a user data that will used throughout the program, referenced by `__ud`. In symmetry, the program shutdown sequence begins with a call to the `fdiff_shutdown()` function for user-code cleanup (line 31), just before the usual `MPI_Finalize()` call at the end of the program. In between the initialisation and shutdown sequences (that are similar for all protocols), the actual code for the protocol implementation is located. Note that the various MPI calls shown in the figure include the ghost parameters for the verification, cf. Chapter 3.

4.2.3 Communication code

In correspondence to communication operations, calls to MPI communication primitives are synthesised. The code generation is driven by the operation at stake, and the associated **`@in`** and **`@out`** marks for user-code data flow. For instance, consider the following protocol fragment from the Finite Differences example:

```
@out getInitialSol @in setLocalData
scatter 0 {x: float[] | length(x) = n}
```

The protocol fragment leads to the generation of the code

```
__sender = 0;
if (__rank == __sender)
  __outData = fdiff_getInitialSol(__ud, -1, n);
else
  __outData = 0;
__inData = fdiff_setLocalData(__ud, __sender, (n)/__procs);
MPI_Scatter(__outData, (n)/__procs, MPI_FLOAT,
            __inData,  (n)/__procs, MPI_FLOAT, __sender,
            MPI_COMM_WORLD
            _(ghost _param) _(ghost _protocol) _(out _protocol));
```

Protocol language	Synthesized C+MPI code

`@out SrcBuf`
`@in DstBuf`
`message from,to D`
	`__receiver = <from>;`
	`__sender = <to>;`
	`if (__rank == __sender) {`
	`  __outData = <PName>_SrcBuf(__ud, __receiver, <len(D)>);`
	`  MPI_Send(__outData, <len(D)>, <MPI_Type(D)>,`
	`  __receiver, 0, MPI_COMM_WORLD`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`
	`} else if (__rank == __receiver) {`
	`  __inData = <PName>_DstBuf(__ud, __sender, <len(D)>);`
	`  MPI_Recv(__inData, <len(D)>, <MPI_type(D)>,`
	`  __sender, 0, MPI_COMM_WORLD, &__status`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`
	`}`

`@out SrcBuf`
`@in DstBuf`
`allgather D`
	`__outData = <PName>_SrcBuf(__ud, -1, <len(D)>;`
	`__inData = <PName>_DstBuf(__ud, -1, <len(D)> * __procs);`
	`MPI_Allgather(__outData, <len(D)>, <MPI_type(D)>,`
	`  __inData, <len(D)>, <MPI_type(D)>, MPI_COMM_WORLD`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`

`@out SrcBuf`
`@in DstBuf`
`allreduce op D`
	`__outData = <PName>_SrcBuf(__ud, -1, <len(D)>);`
	`__inData = <PName>_DstBuf(__ud, -1, <len(D)>);`
	`MPI_Allreduce(__outData, __inData,`
	`  <len(D)>, <MPI_op(op)>, MPI_COMM_WORLD`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`

`@out SrcBuf`
`@in DstBuf`
`broadcast root D`
	`__sender = <root>;`
	`if (__rank == __sender)`
	`  __outData = <PName>_SrcBuf(__ud, -1, <len(D)>);`
	`else`
	`  __inData = <PName>_DstBuf(__ud, __sender, <len(D)>);`
	`MPI_Bcast(__rank == __sender ? __outData : __inData,`
	`  <len(D)>, <MPI_type(D)>, __sender, MPI_COMM_WORLD`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`

`@out SrcBuf`
`@in DstBuf`
`gather root D`
	`__receiver = <root>;`
	`__outData = <PName>_SrcBuf(__ud, __receiver, <len(D)>);`
	`if(__rank == __receiver)`
	`  __inData = <PName>_DstBuf(__ud, -1, <len(D)> * __procs);`
	`else`
	`  __inData = 0;`
	`MPI_Gather(__outData, <len(D)>, <MPI_type(D)>,`
	`  __inData, <len(D)>, <MPI_type(D)>,`
	`  __receiver, MPI_COMM_WORLD`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`

`@out SrcBuf`
`@in DstBuf`
`reduce root op D`
	`__receiver = <root>;`
	`__outData = <PName>_SrcBuf(__ud, __receiver, <len(D)>);`
	`if (__rank == __receiver)`
	`  __inData = <PName>_DstBuf(__ud, -1, <len(D)>);`
	`else`
	`  __inData = 0;`
	`MPI_Reduce(__outData, __inData, <len(D)>, <MPI_type(D)>,`
	`  <MPI_op(op)>, __receiver, MPI_COMM_WORLD`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`

`@out SrcBuf`
`@in DstBuf`
`scatter root D`
	`__sender = <root>;`
	`if(__rank == __sender)`
	`  __outData = <PName>_SrcBuf(__ud, -1, <len(D)>);`
	`else`
	`  __outData = 0;`
	`__inData = <PName>_DstBuf(__ud, __sender, <len(D)> / __procs);`
	`MPI_Scatter(__outData, <len(D)> / __procs, <MPI_type(D)>,`
	`  __inData, <len(D)> / __procs, <MPI_type(D)>,`
	`  __sender, MPI_COMM_WORLD`
	`  _(ghost _param) _(ghost _protocol) _(out _protocol));`

Table 4.1: Synthesis of communication code.

As we can see, the user callback functions are first called to obtain the data addresses of the outgoing and incoming data buffers. Note that for a **scatter** / MPI_Scatter operation, a root process alone provides the data to be distributed for all participants, the one with rank 0 in this case, so __outData is undefined for all other processes. In the MPI_Scatter call, the arguments obey the MPI conventions and are defined by the protocol specification in terms of the process root, the MPI "datatype" (MPI_FLOAT) and the the data payload length (n) [1]. In complement, ghost arguments are also specified for program verification by VCC.

The synthesis process for other communication operators proceeds in similar manner, as illustrated in detail in Table 4.1. In the table, we use notation <E> or <f(E)> to denote a translation point that depends on a protocol expression E.

Protocol language	Synthesized C+MPI code
@condition Predicate P ; Q	<synthesize(P)> <synthesize(Q)> _(assert congruent(cleanup(_protocol,__rank), skip())) _(ghost _protocol = _cCont<I>;)

@condition Predicate
choice {
  P
} or {
  Q
}
	_(ghost Protocol _cTrue<I> = choiceTrue(_protocol);)
	_(ghost Protocol _cFalse<I> = choiceFalse(_protocol);)
	_(ghost Protocol _cCont<I> = choiceCont(_protocol);)
	if (<PName>_Predicate(__ud)) {
	  _(ghost _protocol = _cTrue;)
	  <synthesize(P)>
	} else {
	  _(ghost _protocol = _cFalse;)
	  <synthesize(Q)>
	}
	_(assert congruent(cleanup(_protocol,__rank), skip()))
	_(ghost _protocol = _cCont<I>;)

foreach var : a .. b {
  P
}
	_(ghost ForeachBody _fBody<I> = foreachBody(_protocol);)
	_(ghost Protocol _fCont<I> = foreachCont(_protocol);)
	for (int var = a; var <= b; var++) {
	  _(ghost _protocol = _fBody<I>[var];)
	  <synthesize(P)>
	  _(assert congruent(cleanup(_protocol,__rank), skip()))
	}
	_(ghost _protocol = _fCont<I>;)

@condition Predicate
loop {
  P
}
	_(ghost Protocol _lBody<I> = loopBody(_protocol);)
	_(ghost Protocol _lCont<I> = loopCont(_protocol);)
	while(<PName>_Predicate(__ud)) {
	  _(ghost _protocol = _lBody<I>;)
	  <synthesize(P)>
	  _(assert congruent(cleanup(_protocol,__rank), skip()))
	}
	_(ghost _protocol = _lCont<I>;)

@in Func val x : D	__inData = <PName>_Func(__ud,-1); <C_type(D)> name = * (<C_type(D)> *) __inData; _(assume <HasType(D,name)>)

Table 4.2: Synthesis of control flow code.

[1] The argument n/__procs is used in place of the protocol-specified n, since the payload length arguments for MPI_Scatter must define the data length to be received by each process.

4.2.4 Control flow

The synthesis of control flow code and associated verification logic follows the source protocol specification in AST-driven compositional manner, in the form summarised in Table 4.2. As shown, code and verification annotations are generated for instances of the `;` (sequence), **`choice`**, **`foreach`**, **`loop`**, and **`val`** constructs. The verification annotations and the C control code are in line with the discussion of Chapter 3. The table uses notation `<synthesize(P)>` to denote the compositional synthesis for a subprotocol `P`, and `<I>` to denote a fresh variable index (with the purpose of avoiding variable name clashes).

F.D. protocol	F.D. C+MPI code

```
<P1>
@condition diverged
loop {
    foreach i: 0 .. p - 1 {
      <P2>
    }
    <P3>
  }
  @condition converged
  choice
    <P4>
  or
    <P5>
}
```

```
<synthesize(P1)>
_(ghost Protocol _lBody1 = loopBody(_protocol);)
_(ghost Protocol _lCont1 = loopCont(_protocol);)
while (fdiff_diverged(__ud)) {
  _(ghost _fBody2 = ForeachBody(_protocol);)
  _(ghost _fCont2 = ForeachCont(_protocol);)
  for (int i = 0; i <= p-1; i++) {
    _(ghost _protocol = _fBody2[i];)
    <synthesize(P2)>
    _(assert congruent(cleanup(_protocol,__rank), skip()))
  }
  _(ghost _protocol = _fCont2;)
  <synthesize(P3)>
  _(assert congruent(cleanup(_protocol,__rank), skip()))
}
_(ghost _protocol = _lCont1;)
_(ghost Protocol _cTrue3 =  choiceTrue(_protocol);)
_(ghost Protocol _cFalse3 = choiceFalse(_protocol);)
_(ghost Protocol _cCont3  =  choiceCont(_protocol);)
if (fdiff_converged(__ud)) {
  _(ghost _protocol = _cTrue3;)
  <synthesize(P4)>
} else {
  _(ghost _protocol = _cFalse3;)
  <synthesize(P5)>
}
_(assert congruent(cleanup(_protocol,__rank), skip()))
_(ghost _protocol = _cCont3;)
```

Figure 4.4: Control flow code for the Finite Differences example.

As an example of this synthesis process at work, we present a skeleton of the Finite Differences protocol (fully given in Figure 4.2) and the generated control flow logic in Figure 4.4.

4.2.5 Data processing

Apart from communication and control flow handling, **`@exec`** marks indicate points in the protocol where there is a need to call a computation procedure. This type of mark can be applied to any protocol construct, and implies that the identified user-callback function will execute before the implementation code for the construct at stake. For instance, in the Finite Differences protocol of Figure 4.2, the **`@exec`**

`compute` (line 19) associates to the subsequent **`allreduce`** operation, and leads to a call of the form `fdiff_compute(__ud)` just before the communication code for **`allreduce`** (cf. Table 4.1).

4.2.6 User code definition

```
1  // User data
2  typedef struct _fdiffUserData {
3    int rank;
4    int procs;
5    int n;
6    int problemSize;
7    int iter;
8    float lError;
9    float gError;
10   float* data;
11   float* solution;
12 } fdiffUserData;
13
14 // Initialization callback function
15 fdiffUserData* fdiff_init(int argc, char** argv, int rank, int procs) {
16   fdiffUserData* ud = (fdiffUserData*) malloc(sizeof(fdiffUserData));
17   ud->rank = rank;
18   ud->procs = procs;
19   ud->problemSize = atoi(argv[1]);
20   ...
21   return ud;
22 }
23
24 // Shutdown callback function
25 void fdiff_shutdown( fdiffUserData *ud) {
26   if (ud->rank == 0) {
27     printf("iter %d error %f\n", ud->iter, ud->gError);
28     free(ud->solution);
29   }
30   free(ud->data);
31   free(ud);
32 }
33
34 // @in / @out callback function
35 float* fdiff_setLocalData( fdiffUserData *ud, int peer, int len) {
36   return ud->data + 1;
37 }
38
39 // @condition callback function
40 int fdiff_converged(fdiffUserData *ud) {
41   return (ud->gError <= SMALL_ERROR) || (ud->iter == MAX_ITER);
42 }
43
44 // @exec callback function
45 void fdiff_compute(fdiffUserData *ud) {
46   float* v = ud->data;
47   float e = 0;
48   int i;
49   for (i = 1; i <= ud->n; i++) {
50     float v0 = v[i];
51     v[i] = 0.25 * (v [i-1] + 2 * v0 + v[i+1]) ;
52     e += fabs(v[i] - v0);
53   }
54   ud->lError = e;
55 }
```

Figure 4.5: Example of user definition of the callback functions.

In Figure 4.5, we depict a C fragment containing the definition of user callback functions and the data structure that is used by them. The example at stake is again the Finite Differences program.

Lines 1 to 12 define the user data structure, `fdiffUserData`. The structure contains relevant information required by user code, like the number of processes, the process rank, data buffers, state variables, etc. The `fdiff_init` function (lines 14–22) is called by the C+MPI skeleton to obtain an instance of `FdiffUserData`, that should be properly allocated and initialised by the user code. In symmetry, the `fdiff_shutdown` (lines 24–32) function is called by the skeleton at the end of execution, for cleanup actions, e.g., freeing up memory.

In lines 34–37, we have a callback function, `fdiff_setLocalData` that associates to an **`@in`** or **`@out`** protocol mark. These type of functions return pointers to memory areas that will act as buffers for the MPI calls. Regarding the function arguments, `ud` argument points to the user-data (kept by the skeleton), `peer` provides the rank of the peer for communication in some cases (cf. Table 4.1), and `length` stipulates the length of the data array that should be returned. In the example at stake, the `peer` and `length` arguments are ignored, but they convey relevant information in the general case.

The function in lines 39–42 implements a predicate in correspondence to a **`@condition`** mark for **`loop`** and **`choice`** blocks. The function is supposed to return `0` if the predicate at stake does not hold, and a value other than `0` otherwise. In the case at stake, the implemented predicate corresponds to the convergence condition of the Finite Differences algorithm.

Finally, the `compute()` function in lines 44–55 corresponds to an **`@exec`** mark for data computation. The code basically implements the sequential Finite Differences algorithm.

4.3 Using the MPI Sessions for program synthesis

The functionality of the MPI Sessions Eclipse plugin was extended in support of C+MPI program synthesis. The user can now create an MPI Sessions project in order to create a protocol. After writing the protocol, the user may trigger the synthesis process, and then compile and execute the resulting C+MPI program. The plugin is available at `http://download.gloss.di.fc.ul.pt/mpi-sessions` .

A screenshot of MPI Sessions plugin at work is provided in Figure 4.6. Let us briefly explain how it works in terms of user interaction.

After creating a protocol, the “Generate” button on the toolbar (marked as 1 in Figure 4.6) can be used to synthesize code. The default button generates both the protocol in VCC syntax and the C+MPI code. To individually generate either just the protocol or the MPI code without annotations, some extra buttons are available from the drop-down menu. The generated files should be visible (mark 5) in the top-left pane “Project Explorer”, including the C+MPI program skeleton and the

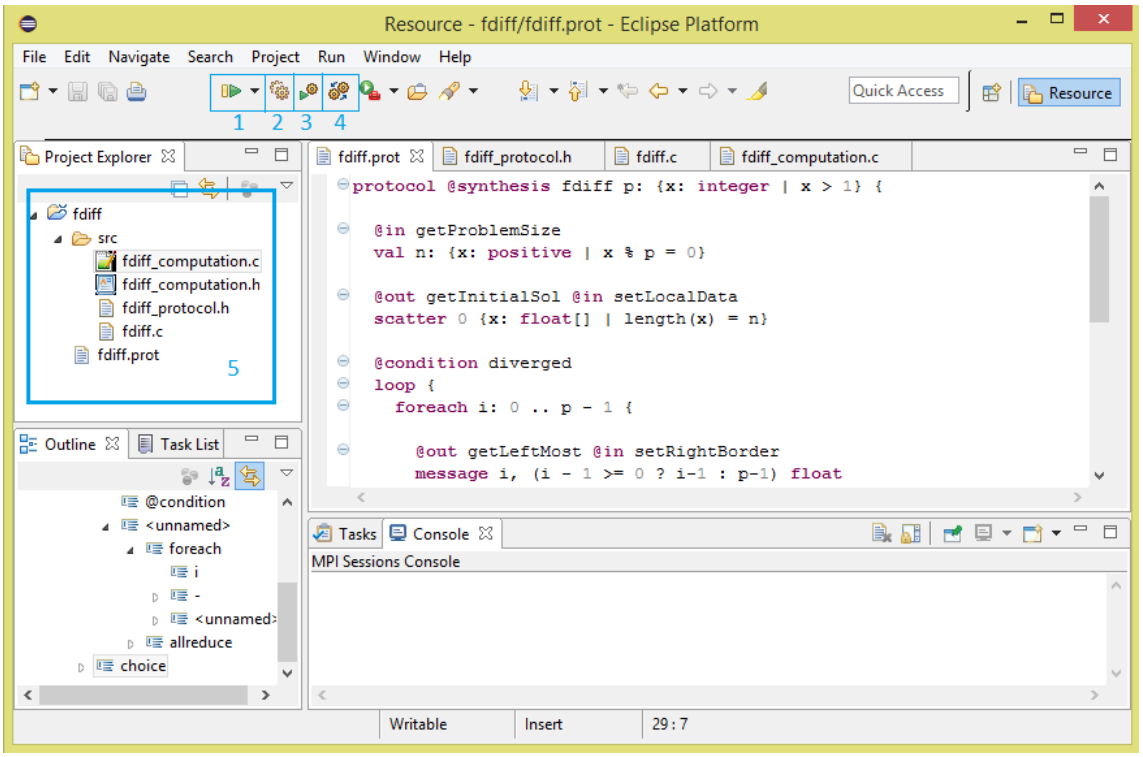

Figure 4.6: Using the MPI Sessions plugin for program synthesis.

user-code header file.

The user-code should then be implemented. When ready, the compilation of the entire program can be fired by the “Compile” button (mark 2). If the compilation is successful, we can execute the program directly by pushing the “Execute” button (mark 3). The compilation and execution steps require an MPI runtime system installed, as the plugin fires the usual `mpicc` and `mpirun` MPI scripts for these tasks. The MPI configuration can be setup in the plugin preferences page, shown in Figure 4.7.

Finally, it is also possible to directly verify if the program conforms with the protocol, if VCC and the annotated MPI library are installed. For this purpose, we can use the “Verify” button (mark 4).

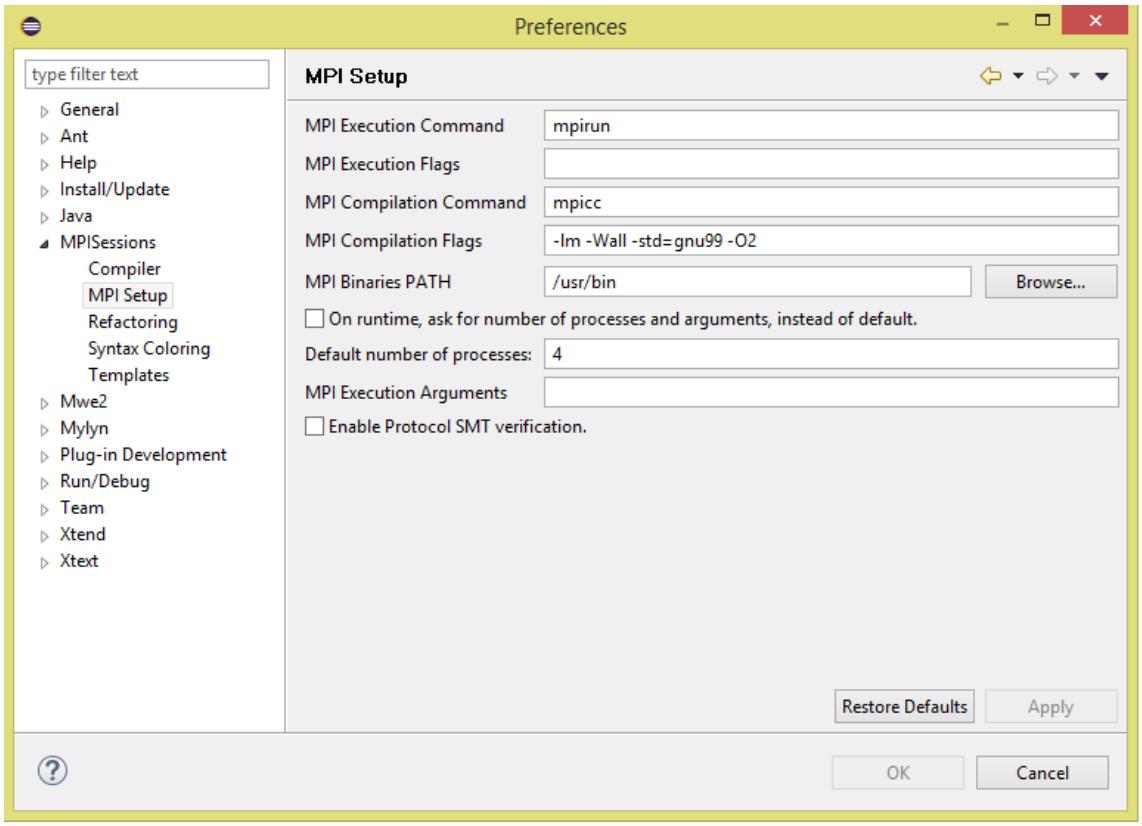

Figure 4.7: MPI Sessions preferences page.

Chapter 5

Design and Implementation

Xtext is a framework designed for the development of new domain specific languages (DSLs), completely integrated in the Eclipse IDE. Based on a syntax grammar defined by the user, parsers and formatters are automatically generated, resulting in a functional text editor. Artifacts developed within the text editor are automatically parsed as an Abstract Syntax Tree (AST). Additional classes are also automatically generated from the grammar specification and provide support for writing rules of validation, scoping and other means of manipulating the AST's, such as visitors for generation, on which we focused our work. In Section 5.1 we explain the syntax grammar, an important feature in which the framework revolves. In Section 5.2 we explain the process behind the generation, where the transformation from protocol language to C+MPI code is explained.

5.1 Grammar specification

The grammar syntax used in Xtext plugin presented in Figure 5.1 is based on the formal protocol language presented in Chapter 3. We present the correlation between the grammar syntax and the protocol language in table 5.1. This specification was developed previously to this thesis, only the annotation related rules where added to support code synthesis.

The grammar syntax specifies the rules to process the protocols. The `Protocol` defined in the syntax acts as a first rule from which the AST will be built:

```
Protocol
: ''protocol'' (synthesis=''@synthesis'')? name=ID vardecl=
   VariableDeclaration type=Datatype body=Type;
```

The structure to process the protocol is defined: **`protocol`** indicates a keyword, `(synthesis=''@synthesis'')?` is an optional variable (indicated by "`?`") that expects a **`@synthesis`** keyword, `name=ID` expects the protocol name, `vardecl= VariableDeclaration type=Datatype` requires the declaration of the program

```
Protocol
  : "protocol" (synthesis="@synthesis")? name=ID vardecl=VariableDeclaration type=Datatype body=Type;

Annotation
  : type=('@in' | '@out' | '@exec' | '@condition') udfunc=ID
  ;


Type
  : {Message} ann+=Annotation* 'message' sender=Expression ',' receiver=Expression type=Datatype
      (';')?
  | {Broadcast} ann+=Annotation* 'broadcast' root=Expression vardecl=VariableDeclaration? type=
      Datatype (';')?
  | {Gather} ann+=Annotation* 'gather' root=Expression type=Datatype (';')?
  | {Scatter} ann+=Annotation* 'scatter' root=Expression type=Datatype (';')?
  | {Reduce} ann+=Annotation* 'reduce' root=Expression op=ReduceOp type=Datatype (';')?
  | {AllReduce} ann+=Annotation* 'allreduce' op=ReduceOp vardecl=VariableDeclaration? type=Datatype
      (';')?
  | {AllGather} ann+=Annotation* 'allgather' vardecl=VariableDeclaration? type=Datatype (';')?
  | {Val} ann+=Annotation* 'val' vardecl=VariableDeclaration type=Datatype (';')?
  | {ForEach} ann+=Annotation* 'foreach' vardecl=VariableDeclaration from=Expression '..' to=
      Expression body=Type
  | {Choice} ann+=Annotation* 'choice' left=Type 'or' right=Type
  | {Loop} ann+=Annotation* 'loop' body=Type
  | {Skip} ann+=Annotation* 'skip' (';')?
  | {Sequence} ann+=Annotation* '{' elements+=Type* '}' (';')?
  ;
```

Figure 5.1: Grammar Syntax.

size (a name and a datatype) and `body=Type` is the protocol's body that delegates to the `Type` rule. This rule contains definitions of all common constructs that constitute the protocol language, including the `Sequence` sub rule that contains an arbitrary number(*) of `Types` added to the identifier `elements`, used to represent the primitives sequence.

Other sub rules follow the same intuition. For example, the following `Message` definition:

```
{Message} ann+=Annotation* 'message' sender=Expression ','
   receiver=Expression type=Datatype (';')?
```

After the `'message'` keyword the rule expects an `Expression` that will identify the message sender, after a comma (`','`) another `Expression` indicates the message receiver, `type=Datatype` specifies the data type to be transmitted. The `(';')?` indicates that a semicolon may or may not be present to delimit the **`message`**, it is optional.

The `ann+=Annotation*` specifies an arbitrary number of annotations before the **`message`** keyword. The `Annotation` rule was added in the ambit of this thesis to augment protocol specification and support code synthesis. It is merely a list of the valid annotation keywords.

Protocol Language	Grammar Syntax
`protocol @synthesis protocol_name p: {x: integer | x > 1} {...}`	`Protocol : "protocol" (synthesis="@synthesis")? name=ID vardecl=VariableDeclaration type=Datatype body=Type;`
`@out srcbuf @in rcvbuf allgather datatype`	`{AllGather} ann+=Annotation* 'allgather' vardecl=VariableDeclaration? type=Datatype (';')?`
`@out srcbuf allreduce operation datatype`	`{AllReduce} ann+=Annotation* 'allreduce' op=ReduceOp vardecl=VariableDeclaration? type=Datatype (';')?`
`@out srcbuf broadcast process_root datatype`	`{Broadcast} ann+=Annotation* 'broadcast' root=Expression vardecl=VariableDeclaration? type=Datatype (';')?`
`@condition eval choice (...) or (...)`	`{Choice} ann+=Annotation* 'choice' left=Type 'or' right=Type`
`foreach i: a .. b { (...) }`	`{ForEach} ann+=Annotation* 'foreach' vardecl=VariableDeclaration from=Expression '..' to=Expression body=Type`
`@out srcbuf @in rcvbuf gather root datatype`	`{Gather} ann+=Annotation* 'gather' root=Expression type=Datatype (';')?`
`@condition eval loop (...)`	`{Loop} ann+=Annotation* 'loop' body=Type`
`@out srcbuff @in rcvbuff message origin, destination datatype`	`{Message} ann+=Annotation* 'message' sender=Expression ',' receiver=Expression type=Datatype (';')?`
`@out srcbuff @in rcvbuff reduce root operation datatype`	`{Reduce} ann+=Annotation* 'reduce' root=Expression op=ReduceOp type=Datatype (';')?`
`;`	`{Sequence} ann+=Annotation* '{' elements+=Type* '}' (';')?`
`@out srcbuff @in rcvbuff scatter root datatype`	`{Scatter} ann+=Annotation* 'scatter' root=Expression type=Datatype (';')?`
`@in src val name: datatype`	`{Val} ann+=Annotation* 'val' vardecl=VariableDeclaration type=Datatype (';')?`

Table 5.1: Correlation between the protocol language and the grammar syntax.

5.2 Code synthesis

5.2.1 Skeleton generation

From the syntax grammar rules Java classes for the various components are automatically generated. Each `Type` in a protocol can now be manipulated for our purposes, as every AST node is an object.

Code synthesis works by visiting the AST nodes, interpreting the `Types` and writing the correct MPI calls and control flow constructs to the output file. Xtend has double dispatching which allows us to define a different method to handle each primitive type with ease. On those methods, a corresponding MPI call template is in place waiting to be filled with the given the primitive parameters: sender, receiver, buffer size, buffer type, etc.

The generation starts in the `compile()` method in Figure 5.2. It takes a `Protocol` object, creates a stub for the C+MPI code based on the template in place and continues visiting the `Protocol`'s body (line 42). Note that there are

several VCC annotations that can be generated if a variable is enabled(lines 6–10, 16–21, 38–40, 47–49), this is the rationale applied in the remaining methods that visit the AST.

```
1   def String compile(Protocol p) {
2     (...)
3     '''
4     (...)
5
6     <<IF vcc_annot>>
7     // VCC {
8     #include "<<protocolName>>_protocol.h"
9     // }
10    <<ENDIF>>
11
12    int main
13    (
14    int argc,
15    char *argv[]
16    <<IF vcc_annot>>
17    // VCC {
18    _(ghost Param* _param)
19    _(ghost Protocol _protocol)
20    // }
21    <<ENDIF>>
22    ) {
23      int __rank;
24      int __procs;
25      <<protocolName>>UserData* __ud;
26      void* __inData;
27      void* __outData;
28      MPI_Status __status;
29      int __receiver = 0;
30      int __sender = 0;
31
32      MPI_Init(&argc,&argv);
33      MPI_Comm_size(MPI_COMM_WORLD,&__procs);
34      MPI_Comm_rank(MPI_COMM_WORLD,&__rank);
35
36      __ud = <<protocolName>>_init(argc, argv, __rank, __procs);
37      int <<name>> = __procs;
38      <<IF vcc_annot>>
39      <<IF restriction != null>>><<restriction>><<ENDIF>>
40      <<ENDIF>>
41
42      <<p.body.process>>
43
44      // == SHUTDOWN ==
45      <<protocolName>>_shutdown(__ud);
46      MPI_Finalize();
47      <<IF vcc_annot>>
48      _(assert \false)
49      <<ENDIF>>
50    }
51    '''
52  }
```

Figure 5.2: The compile method that initiates generation by visiting the AST.

5.2.2 Communication code

In `p.body.process` (Figure 5.2, line 42) the `process()` method takes the `protocol p`'s body and dispatches it to the appropriate function, as it is a double dispatch method. Typically the body is of Sequence type, containing a sequence of Type objects. The specific `process()` for the Sequence type is presented in Figure 5.3. It takes a list of types inside the variable `elements` and dispatches each one to the corresponding method.

```
1   def dispatch String process(Sequence o) {
2     '''
3     <<FOR t : o.elements>>
4     << t.process >>
5     <<ENDFOR>>
6     '''
7   }
8
9   def dispatch String process(Broadcast o) {
10
11    var obj = o.type
12    var bufsize = obj.printSize
13    var buftype = obj.convert.base.print.mpi
14
15    '''
16    <<printAnnotFunc(null, protocolName,"@exec", o.ann,"-1")>>
17    <<printAnnotFunc(bufsize, protocolName,"@out", o.ann,o.root.expPrint)>>
18    <<IF o.vardecl != null>>
19    <<variableDeclaration(o.vardecl.name, o.type.convert, o.ann)>>
20    <<ENDIF>>
21    MPI_Bcast(__outData, <<bufsize>>, <<buftype>>, <<o.root.expPrint>>, MPI_COMM_WORLD);
22    '''
23  }
```

Figure 5.3: The `process()` methods that handle Sequence Types and MPI broadcast types .

Consider the following broadcast type: **`@out`** `getN` **`broadcast`** `0 n:`**`integer`** . The excerpt describes a `MPI_Bcast` operation where process 0 sends an integer `n` to all processes and `getN()` is the function that returns the pointer for the buffer used to perform this operation.

In figure 5.3 is presented the method used to handle Broadcast primitives. Note that all the necessary information for code generation is present in the protocol primitive so we only fill the "template" in place for each case. The method returns a string that will be written in the output file. Lines 15 to 22 represents the template in place for the MPI Broadcast operation. Inside the double angle quotation marks are functions that are either void or return strings to be included in the result. This is useful to handle special cases, such as annotations or possible variable declarations that are not always generated. Notice that in our example the integer disseminated by the broadcast operation comes with a variable declaration, `n`, that can be used in the protocol from this point on. In this case, MPI generation has to reflect this variable declaration so lines 18 to 20 are used to verify if there is a variable declaration. If there is, the method `variableDeclaration()` returns the correct declaration of this variable.

The function `printAnnotFunc()` in lines 16–17 checks for annotations and handles them accordingly. Line 16 checks for **`@exec`** annotations that imply local computation code. Line 17 checks for the function that returns the pointer for the output buffer.

All MPI call methods are somewhat similar in structure, with some different details.

5.2.3 Control flow code

```
1  def dispatch String process(Loop l) {
2      var loop_count = count;
3      count = count+1;
4      '''
5      <<printAnnotFunc(null, protocolName,"@exec", l.ann,"-1")>>
6
7      <<IF vcc_annot>>
8      // VCC {
9      _(ghost Protocol _t<<loop_count>>_loop_body = loopBody(_protocol);)
10     _(ghost Protocol _t<<loop_count>>_loop_cont = loopCont(_protocol);)
11     // }
12     <<ENDIF>>
13     while(<<printAnnotFunc(null, protocolName,"@condition", l.ann,"-1")>>)
14       <<IF vcc_annot>>
15       // VCC (write clauses for loop)
16       _(writes {})
17       // }
18       <<ENDIF>>
19       {
20       <<IF vcc_annot>>
21       // VCC (loop body match) {
22       _(ghost _protocol = _t<<loop_count>>_loop_body;)
23       // }
24       <<ENDIF>>
25       <<l.body.process>>
26       <<IF vcc_annot>>
27       // VCC (end of loop body matching) {
28       _(assert congruent(cleanup(_protocol,__rank), skip()))
29       // }
30       <<ENDIF>>
31     }
32     <<IF vcc_annot>>
33     // VCC (loop continuation) {
34     _(ghost _protocol = _t<<loop_count>>_loop_cont;)
35     // }
36     <<ENDIF>>
37     '''
38   }
```

Figure 5.4: Method used to process **loop** primitives in the generator.

Figure 5.4 presents the `process()` method used to generate **loop** primitives. The template begins by declaring any call back functions defined by a **@exec** annotation (line 5), then declaring the while operator (line 22), processing the loop body (line 25) similar to what is done in the compile method (back in section 5.2.2). The `process()` function for the remaining control flow primitives (**foreach** and **choice**) follows the same pattern.

5.2.4 User header

Computation header generation is straightforward. We created another generator purposely for this purpose, however, the strategy is the same as with the C+MPI skeleton: we visit the AST nodes(Figure 5.5) and generate code accordingly. Instead of MPI calls and control flow operators we synthesize a function header for each annotation(Figure 5.6) and a list keeps track of the declared functions to avoid repetition.

```
def dispatch String processH(Broadcast o) {


    var obj = o.type
    var bufsize = obj.printSize
    var buftype = obj.convert.base.print.to_C_type

    var inName = getUDFunc(o.ann,"@in");
    var outName = getUDFunc(o.ann,"@out");
    '''
    <<IF !inName.equals("")>>
    <<declareFunction(inName, buftype, bufsize)>>
    <<ENDIF>>
    <<IF !outName.equals("")>>
    <<declareFunction(outName, buftype, bufsize)>>
    <<ENDIF>>
    '''
  }
```

Figure 5.5: One of the method used to visit the AST and generate function headers

```
  /**
   * This method produces a method declaration for user functions,
   * given a name, a type and a size. If size is "1", then the len
       parameter is
   * omitted.
   * Declared function always have a "*" after its type.
   */
  def String declareFunction(String name, String type, String size){

    //checks if the function was already declared
    if(funcList.contains(name))
      return ""
    funcList.add(name);

    var boolean noSize = size.equals("1");
    '''
    <<IF noSize == false>>
    <<type>>* <<protocolName>>_<<name>>
    (
      <<protocolName>>UserData *ud, /* in out */
      int peer, /* in */
      int len /* in */
    )
    <<ELSE>>
    <<type>>* <<protocolName>>_<<name>>
    (
      <<protocolName>>UserData *ud, /* in out */
      int peer /* in */
    )
    <<ENDIF>>
    <<IF vcc_annot>>
    <<IF noSize>>
    _(ensures \thread_local(\result))
    <<ELSE>>
    _(requires len > 0)
    _(ensures \thread_local_array(\result, (unsigned) len))
    <<ENDIF>>
    <<ENDIF>>
    ;
    '''
  }
```

Figure 5.6: One of the methods used to generate function headers

Chapter 6

Evaluation

This chapter provides an evaluation of our approach, considering a set of MPI programs. We begin by describing the examples (Section 6.1), and then provide an evaluation of results regarding code synthesis (Section 6.2), program execution (Section 6.3), and program verification (Section 6.4).

6.1 Sample MPI programs and protocols

We tested our toolchain on seven different programs, taken from textbooks and benchmark suites. The protocol specifications for each example are provided in Appendix A.4.

Finite Differences: used previously as a running example, the one-dimensional finite differences problem takes a vector X^0 and calculates X^n iteratively using a recursive formula until either a convergence condition is verified or a certain number of iterations is reached. The code is adapted from [5]; in particular the original program had to be adjusted to prevent deadlock in case the runtime MPI system does not provide buffering.

Jacobi Iteration: solves a $Ax = b$ linear equation using Jacobi's method, from [23].

Laplace: this program calculates a solution to the 2-D Laplace equation recursively. This example is part of the FEVS [2] benchmark suite.

Matrix Multiplication: a simple matrix multiplication algorithm, adapted from [17].

N-body simulation: a classic N-body simulation program, from [7].

Pi: a toy program that calculates an approximation of pi through numerical integration, from [7]. This has also been used as a running example in Chapter 3.

Vector dot product: calculates the dot product of two vectors, from [23].

6.2 Code Synthesis

Program	Procotol	Synthesized code	User Code	Total	Original code
Finite differences	23	258 (81,7%)	58 (18,3%)	316	108
Jacobi iteration	22	238 (61,2%)	151 (38,8%)	389	176
Laplace	32	311 (85,7%)	52 (14,3%)	363	128
Matrix multiplication	15	170 (68,5%)	78 (31,5%)	248	116
N-body simulation	28	261 (56,2%)	203 (43,8%)	464	163
Pi	9	126 (86,3%)	20 (13,7%)	146	40
Vector dot product	23	241 (76,3%)	75 (23,7%)	316	90

Table 6.1: Code synthesis results (LOC)

The results for code synthesis are presented in Table 6.1. For each program (column 1), the lines of code are shown for: the protocol specification (2); synthesised C+MPI code (3); complementary user-code; the total size of the program (4); and, finally, the original program (5). For a fair comparison, the LOC count for the synthesised programs does not account for verification annotations, discussed further on in this chapter.

The first observation is that protocol specifications are generally much smaller than the corresponding C+MPI code that is synthesised. On average, the synthesised C+MPI code is 11,0 times larger than the protocol. This is also true in comparison to the size of original programs, which are on average 4,4 times larger than the protocol. Thus, relatively verbose and complex C+MPI code can be derived from succinct and high-level protocol specifications.

Regarding the size of user-code, a measure for programmer effort beyond protocol specification, we can also observe that it is generally much smaller than synthesised code. User-code amounts to 26 % of program code, taking the average of all programs. Compared to the original code, user-code is in some cases more verbose, due to the user-function callback scheme; even if the core computation code is the same, (many) user callback functions and the user-data structure definition lead to an increase in the size of the code.

Finally, the total size of programs defined using our approach tends to be higher than the original code, 2,9 times larger on average. However, if we consider non-synthesised C code (the user-code) only, the ratio is 0,7.

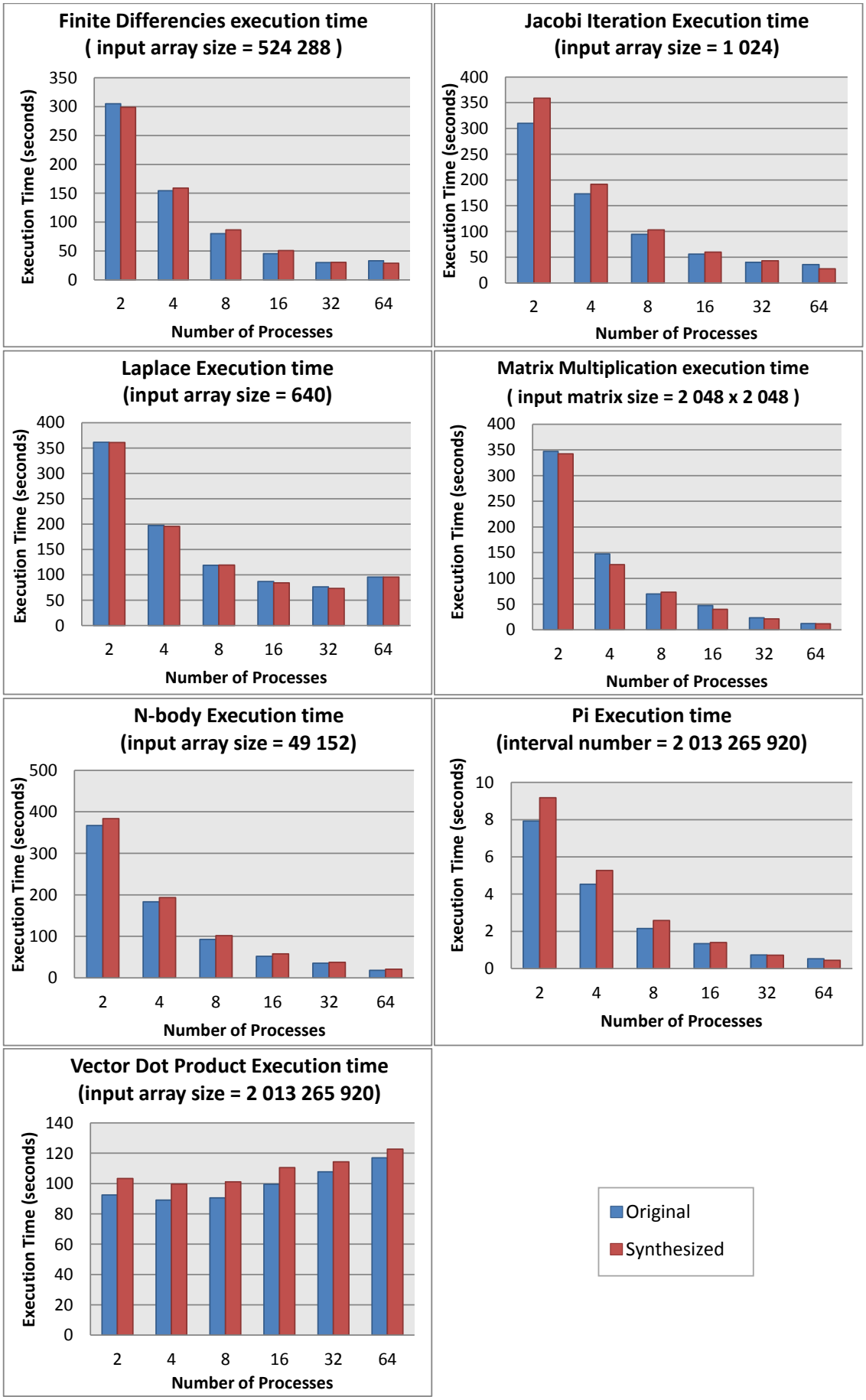

Figure 6.1: Program execution times.

6.3 Execution time

We measured the execution time of the example programs, for both the synthesised and original versions in each case.

Our benchmark setup was as follows. The tests were conducted in one of our department's computers, Hydra, a 64-core machine with 46 GB ram running Linux with the MPICH2 MPI distribution. Every program relies on an array size parameter, except for the Pi program that takes a maximum number of intervals for integration. We tuned the values for these parameters to obtain execution times that expose the scalability of each example at stake appropriately. Also, in all examples but Pi, we made sure the program's input arrays were initialised with random input values, using a fixed seed for the random number generator (for repeatable tests). Finally, we fixed the maximum number of iterations to 100000 for the (convergence-based) Finite Differences and Jacobi Iteration examples.

In Figure 6.1 we depict the average results of 6 runs for each program. The execution times are plotted for each example for a varying number of employed processors, from 2 to 64. The synthesised programs have similar scalability and performance to the original code's counterpart. Note that the Vector Product program does not scale; this textbook example [23] is designed in this manner on purpose.

6.4 Verification results

Program	Time (s)	LOC	Annot.	MPI calls	Control constructs	Domain restrictions
Finite differences	16,0	316	45	5	3	4
Jacobi iteration	23,3	389	41	4	1	3
Laplace	7,33	363	57	5	5	0
Matrix multiplication	5,3	248	33	3	2	3
N-body simulation	5,3	464	46	3	4	2
Pi	2,5	146	17	2	1	1
Vector dot product	7,4	316	40	5	3	3

Table 6.2: VCC verification related analysis.

The results for protocol verification are shown in Table 6.2. For each example (column 1), we indicate the verification time (2), the number of synthesised verification annotations (4), plus characteristics for the protocols/programs at stake: the number of MPI communication primitives (5), the number of control flow (**`choice`**, **`loop`** and **`foreach`**) constructs (6) and the number of refined type restrictions (7). Regarding the verification times shown, they are the average of 6 executions of VCC

over each example program, using a Windows 7 machine with two 2.4 GHz/64-bit CPU cores and 4 GB of RAM.

Overall, all examples take less than 25 seconds to verify, and all but two take less than 10 seconds. The correlation between these times and protocol/program characteristics is not immediate. We must take into consideration the combination of program characteristics, and sometimes little details represent a significant difference in verification time. For example, the Finites Differences program takes an average 16 seconds to verify. To characterize message exchange in a ring topology we use ternary operations. However, if we remove those ternary operators, as illustrated in Figure 6.2 and alter the protocol so that message exchange does not occur between process `0` and process `p-1`, the verification time is reduced to under 8 seconds, half of the original time. The resulting program no longer follows the ring topology, we just refer it as an example of some aspects that may affect verification performance.

```
//original foreach with ternary operators
foreach i: 0 .. p - 1 {
  @out getLeftMost @in setRightBorder
  message i, (i - 1 >= 0 ? i-1 : p-1) float
  @out getRightMost @in setLeftBorder
  message i, (i + 1 <= p-1 ? i+1 : 0) float
}

//altered cycle (note: this does not follow a ring topology anymore)
foreach i: 1 .. p - 2 {
  @out getLeftMost @in setRightBorder
  message i,i-1 float
  @out getRightMost @in setLeftBorder
  message i, i+1 float
}
```

Figure 6.2: Small alterations in the **`foreach`** cycle.

Chapter 7

Conclusion

7.1 Summary of contributions

This thesis presented a framework for synthesizing correct C+MPI programs, derived from protocol specifications based on the multi-party session type theory. The programs also contain annotations that allow its verification by the VCC verifier, acting as proof of correctness.

In detail, the core contributions of this thesis were as follows:

1. We expanded the functionalities of the MPI Sessions framework. We enhanced the protocol specification with annotations specific to the synthesis process. The result of the synthesis process is a correct C+MPI program skeleton that flawlessly follows the communication protocol.

2. From a set of communication protocols, based on existing programs, we synthesized the corresponding C+MPI program, complemented with appropriate callback functions, and thoroughly compared its execution results with the original ones. Analysing the execution times we concluded that there is similar performance and scalability between the two versions. We also presented the number of automatically generated lines of code (to assert how much of the program is synthesized) and compared the number of lines that required manual input in both cases.

3. Using the verification architecture created prior to this thesis, we submitted our synthesized programs and asserted its correctness. We also provided a small analysis about the differences in verification time between all processes.

4. We improved the overall MPI Sessions plugin experience. We added a project wizard to the Eclipse plugin as a mean to create a MPI Sessions project that automatically opens a protocol ready to be filled. Using the plugin, the diverse tasks of synthesis, compilation, execution, and verification, can be accomplished in an user-friendly manner.

5. We released the tool online for public usage. The release includes an Eclipse update site that facilitates the plugin's installation, the program examples developed in this thesis (ready to use) and an User Manual with a practical insight and simple instructions on how to use the plugin, create protocols, etc, without the technicalities behind it.

7.2 Future work

In terms of future work, we propose the following challenges:

1. An empirical study of the framework in a controlled group, to evaluate the advantages of the approach from a software engineering perspective, e.g., to compare the effort of a programmer to develop an MPI program in a standard way vs. the effort of doing so using our approach;

2. The use of protocols for larger, more complex programs, for a better assessment of the applicability of the approach in the real world;

3. To expand the subset of MPI primitives supported by the protocol language, non-blocking primitives in particular [4], and also extend the program synthesis process in that regard. This affects the whole MPI Sessions framework as a whole.

Appendix A

Listings

A.1 Finite differences communication code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <mpi.h>
5  #include "fdiff_computation.h"
6
7  // VCC {
8  #include "fdiff_protocol.h"
9  // }
10
11 int main
12 (
13 int argc,
14 char *argv[]
15 // VCC {
16 _(ghost Param* _param)
17 _(ghost Protocol _protocol)
18 // }
19 ) {
20   int __rank;
21   int __procs;
22   fdiffUserData* __ud;
23   void* __inData;
24   void* __outData;
25   MPI_Status __status;
26   int __receiver = 0;
27   int __sender = 0;
28
29   MPI_Init(&argc,&argv);
30   MPI_Comm_size(MPI_COMM_WORLD,&__procs);
31   MPI_Comm_rank(MPI_COMM_WORLD,&__rank);
32
33   __ud = fdiff_init(argc, argv, __rank, __procs);
34   int p = __procs;
35   _(assume (p > 1));
36
37   __inData = fdiff_getProblemSize(__ud,-1);
```

```
38  int n = * (int *) __inData;
39  // VCC {
40  _(assume (n > 0 && n % p == 0));
41  apply(n);
42  // }
43
44  __inData = fdiff_setLocalData(__ud,0,(n)/__procs);
45  if(__rank == 0)
46    __outData = fdiff_getInitialSol(__ud,0,n);
47  else
48    __outData = 0;
49  MPI_Scatter(__outData, (n)/__procs, MPI_FLOAT, __inData, (n)/__procs,
        MPI_FLOAT, 0, MPI_COMM_WORLD);
50
51  // VCC {
52  _(ghost Protocol _t0_loop_body = loopBody(_protocol);)
53  _(ghost Protocol _t0_loop_cont = loopCont(_protocol);)
54  // }
55  while(fdiff_diverged(__ud))
56    // VCC (write clauses for loop)
57    _(writes {})
58    // }
59    {
60    // VCC (loop body match) {
61    _(ghost _protocol = _t0_loop_body;)
62    // }
63
64    // VCC (foreach matching) {
65    _(ghost ForeachBody _t1_foreach_body = foreachBody(_protocol);)
66    _(ghost Protocol _t1_foreach_cont = foreachCont(_protocol);)
67    // }
68    for(int i =0; i <= p - 1; i++)
69    // VCC (write clauses for foreach)
70    _(writes {})
71    // }
72    {
73      // VCC (foreach body matching) {
74      _(ghost _protocol = _t1_foreach_body[i];)
75      // }
76      __receiver = i - 1 >= 0 ? i - 1 : p - 1;
77      __sender = i;
78      if (__rank == __sender) {
79        __outData = fdiff_getLeftMost(__ud,__receiver);
80        MPI_Send(__outData,1,MPI_FLOAT,__receiver,0,MPI_COMM_WORLD);
81      } else if (__rank == __receiver) {
82        __inData = fdiff_setRightBorder(__ud,__sender);
83        MPI_Recv(__inData,1,MPI_FLOAT,__sender,0,MPI_COMM_WORLD,&__status);
84      }
85      __receiver = i + 1 <= p - 1 ? i + 1 : 0;
86      __sender = i;
87      if (__rank == __sender) {
88        __outData = fdiff_getRightMost(__ud,__receiver);
89        MPI_Send(__outData,1,MPI_FLOAT,__receiver,0,MPI_COMM_WORLD);
90      } else if (__rank == __receiver) {
91        __inData = fdiff_setLeftBorder(__ud,__sender);
92        MPI_Recv(__inData,1,MPI_FLOAT,__sender,0,MPI_COMM_WORLD,&__status);
```

```
93        }
94        // VCC (end of foreach body matching) {
95        _(assert congruent(cleanup(_protocol,__rank), skip()))
96        // }
97      }
98      // VCC (foreach continuation) {
99      _(ghost _protocol = _t1_foreach_cont;)
100     // }
101     // == Computation ==
102     fdiff_compute(__ud);
103
104     __outData = fdiff_getLocalError(__ud,-1);
105     __inData = fdiff_setGlobalError(__ud,-1);
106     MPI_Allreduce (__outData, __inData, 1, MPI_FLOAT,MPI_MAX, MPI_COMM_WORLD);
107     _(assert congruent(cleanup(_protocol,__rank), skip()))
108   }
109   // VCC (loop continuation) {
110   _(ghost _protocol = _t0_loop_cont;)
111   // }
112
113   // VCC (choice matching) {
114   _(ghost Protocol _t2_choice_T    = choiceTrue(_protocol);)
115   _(ghost Protocol _t2_choice_F    = choiceFalse(_protocol);)
116   _(ghost Protocol _t2_choice_cont = choiceCont(_protocol);)
117   // }
118   if(fdiff_converged(__ud)){
119     // VCC (choice match -- first case) {
120     _(ghost _protocol = _t2_choice_T;)
121     // }
122     if(__rank == 0)
123       __inData = fdiff_setFinalSol(__ud,0,(n / p) * __procs);
124     else
125       __inData = 0;
126     __outData = fdiff_getLocalSol(__ud,0,n / p);
127     MPI_Gather(__outData, n / p, MPI_FLOAT, __inData, n / p, MPI_FLOAT, 0,
          MPI_COMM_WORLD);
128     // VCC (end of choice match -- first case) {
129     _(assert congruent(cleanup(_protocol,__rank), skip()))
130     // }
131   } else {
132     // VCC (choice match -- second case) {
133     _(ghost _protocol = _t2_choice_F;)
134     // }
135     // VCC (end of choice match -- second case) {
136     _(assert congruent(cleanup(_protocol,__rank), skip()))
137     // }
138   }
139   // VCC (end of choice matching) {
140   _(ghost _protocol = _t2_choice_cont;)
141   // }
142
143   // == SHUTDOWN ==
144   fdiff_shutdown(__ud);
145   MPI_Finalize();
146   _(assert \false)
147 }
```

A.2 Finite differences computation header

```
#ifndef __fdiff_gen_h__
#define __fdiff_gen_h__

struct _FdiffUserData;

typedef struct _FdiffUserData FDiffUserData;

FDiffUserData* fdiff_init
(
  int argc, /* in */
  char** argv, /* in */
  int rank, /* in */
  int procs /* in */
);

void fdiff_shutdown
(
  FDiffUserData *ud /* in out */
);

void fdiff_getProblemSize
(
  FDiffUserData *ud, /* in out */
  int* n /* out */
);

void fdiff_getInitialSol
(
  FDiffUserData *ud, /* in out */
  int len, /* in */
  float* v /* out */
);

void fdiff_setLocalData
(
  FDiffUserData *ud, /* in out */
  int len, /* in */
  float* v /* out */
);

int fdiff_converged
(
  FDiffUserData *ud /* in out */
);

void fdiff_getLeftCell
(
  FDiffUserData *ud, /* in out */
  float *v /* out */
);

void fdiff_getRightCell
(
```

```
54    FDiffUserData *ud, /* in out */
55    float *v /* out */
56  );
57
58  void fdiff_setLeftBorder
59  (
60    FDiffUserData *ud, /* in out */
61    float v /* in */
62  );
63
64  void fdiff_setRightBorder
65  (
66    FDiffUserData *ud, /* in out */
67    float v /* in */
68  );
69
70  void fdiff_getLocalSol
71  (
72    FDiffUserData *ud, /* in out */
73    float *v, /* out */
74    int n
75  );
76  void fdiff_setFinalSol
77  (
78    FDiffUserData *ud, /* in out */
79    float *v, /* in */
80    int n
81  );
82  void fdiff_compute
83  (
84    FDiffUserData *ud /* in out */
85  );
86  void fdiff_getLocalError
87  (
88    FDiffUserData *ud, /* in out */
89    float *v /* out */
90  );
91  void fdiff_setGlobalError
92  (
93    FDiffUserData *ud, /* in out */
94    float v /* in */
95  );
96  #endif
```

A.3 Finite differences computation implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5  #include "fdiff-computation.h"
6
7  static const int MAX_ITER = 100000;
```

```
8  static const float SMALL_ERROR = 1e-05f;
9  static const float HUGE_ERROR = 1e+05f;
10
11 struct _FdiffUserData {
12   int rank;
13   int procs;
14   int n;
15   int iter;
16   float lError;
17   float gError;
18   float* data;
19 };
20
21 FDiffUserData* fdiff_init
22 (
23   int argc, /* in */
24   char** argv, /* in */
25   int rank, /* in */
26   int procs /* in */
27 )
28 {
29   FDiffUserData* ud = (FDiffUserData*) malloc(sizeof(FDiffUserData));
30   ud->rank = rank;
31   ud->procs = procs;
32   ud->n = atoi(argv[1]) / procs;
33   ud->iter = 0;
34   ud->lError = HUGE_ERROR;
35   ud->gError = HUGE_ERROR;
36   ud->data = (float*) malloc( (ud->n + 2) * sizeof(float));
37   return ud;
38 }
39
40 void fdiff_shutdown
41 (
42   FDiffUserData *ud /* in out */
43 )
44 {
45   if (ud->rank == 0)
46     printf("iter %d error %f\n", ud->iter, ud->gError);
47   free(ud->data);
48   free(ud);
49 }
50
51 void fdiff_getProblemSize
52 (
53   FDiffUserData *ud, /* in out */
54   int* n /* out */
55 )
56 {
57   *n = ud->n * ud->procs;
58   srand48(0L);
59 }
60
61 void fdiff_getInitialSol
62 (
63   FDiffUserData *ud, /* in out */
```

```
64    int len, /* in */
65    float* v /* out */
66  )
67  {
68    int i;
69    for (i=0; i < len; i++) {
70      v[i] = drand48();
71    }
72  }
73
74  void fdiff_setLocalData
75  (
76    FDiffUserData *ud, /* in out */
77    int len, /* in */
78    float* v /* out */
79  )
80  {
81    memcpy(ud->data + 1, v, len * sizeof(float));
82  }
83
84  int fdiff_converged
85  (
86    FDiffUserData *ud /* in out */
87  )
88  {
89    ud->iter ++;
90    return (ud->gError <= SMALL_ERROR) || (ud->iter == MAX_ITER);
91  }
92
93  void fdiff_getLeftCell
94  (
95    FDiffUserData *ud, /* in out */
96    float *v /* out */
97  )
98  {
99    *v = ud->data [1];
100 }
101
102 void fdiff_getRightCell
103 (
104   FDiffUserData *ud, /* in out */
105   float *v /* out */
106 )
107 {
108   *v = ud->data [ud->n];
109 }
110
111 void fdiff_setLeftBorder
112 (
113   FDiffUserData *ud, /* in out */
114   float v /* in */
115 )
116 {
117   ud->data [0] = v;
118 }
119
```

```
120 void fdiff_setRightBorder
121 (
122   FDiffUserData *ud, /* in out */
123   float v /* in */
124 )
125 {
126   ud->data [ud->n + 1] = v;
127 }
128
129 void fdiff_getLocalSol
130 (
131   FDiffUserData *ud, /* in out */
132   float *v, /* out */
133   int n
134 )
135 {
136   memcpy(v, ud->data + 1, n * sizeof(float));
137 }
138
139 void fdiff_setFinalSol
140 (
141   FDiffUserData *ud, /* in out */
142   float *v, /* in */
143   int n
144 )
145 {
146   // TODO
147 }
148 void fdiff_compute
149 (
150   FDiffUserData *ud /* in out */
151 )
152 {
153   float* v = ud->data;
154   float e = 0;
155   int i;
156   for (i = 1; i <= ud->n; i++) {
157     float v0 = v[i];
158     v[i] = 0.25 * (v [i-1] + 2 * v0 + v[i+1]) ;
159     e += fabs(v[i] - v0);
160   }
161   ud->lError = e;
162 }
163
164 void fdiff_getLocalError
165 (
166   FDiffUserData *ud, /* in out */
167   float *v /* out */
168 )
169 {
170   *v = ud->lError;
171 }
172
173 void fdiff_setGlobalError
174 (
175   FDiffUserData *ud, /* in out */
```

```
176   float v /* in */
177 )
178 {
179   ud->gError = v;
180 }
```

A.4 Program Protocols

```
1  protocol @synthesis fdiff p: {x: integer | x > 1} {
2    @in getProblemSize
3    val n: {x: positive | x % p = 0}
4
5    @out getInitialSol @in setLocalData
6    scatter 0 {x: float[] | length(x) = n}
7
8    @condition diverged
9    loop {
10     foreach i: 0 .. p - 1 {
11
12       @out getLeftMost @in setRightBorder
13       message i, (i - 1 >= 0 ? i-1 : p-1) float
14
15       @out getRightMost @in setLeftBorder
16       message i, (i + 1 <= p-1 ? i+1 : 0) float
17     }
18   @exec compute
19     @out getLocalError @in setGlobalError
20     allreduce max float
21   }
22
23   @condition converged
24   choice
25     @out getLocalSol @in setFinalSol
26     gather 0 float[n/p]
27   or
28     {}
29 }
```

Figure A.1: Finite Differences protocol.

```
protocol @synthesis parallel_jacobi p: {x: integer | x > 1} {

  @in getN
  val n : {y: integer | y > 0 and y * p <= 100000 and y * y * p <= 100000}

  @out readMatrix @in getA_local
  scatter 0 float[n*n*p]

  @out readVector @in getB_local
  scatter 0 float[n*p]

  @out getB_local @in getX_temp1
  allgather float[n]

  @exec before_iteration
  @condition converged
  loop{
    @exec jacobi_iteration
    @out getX_local @in getX_new
    allgather float[n]
  }

  @condition distance_tol
  choice {
    @out getX_local @in getTemp
    gather 0 float[n]
  } or { }

}
```

Figure A.2: Jacobi iteration protocol.

```
protocol @synthesis laplace p: {x:positive | x > 1}{

  @in get_nxlg
  val nxlg : positive

  //jacobi method
  @condition converged
  loop{

      foreach i: 0 .. p-2 {
        @out old_nylg2 @in old0
        message i, i+1 float[nxlg];
      }
      foreach i: 1 .. p-1 {
        @out old1 @in old_nylg1
        message i, i-1 float[nxlg];
      }
    @exec computation
    @out error @in global_error
    allreduce sum float;

    //print frame
    @exec printBegin

    //each process sends its rows
    foreach i:1..p-1{
      @exec initRow
      @condition get_row
      loop{
        @out grid_row @in get_rbufx
        message i, 0 float[nxlg]
        @exec printIter skip //print what is received
      }
    }

    //last process always sends one extra message
    @out grid_nylg1 @in get_rbufx
    message p-1, 0 float[nxlg]
    @exec printIter skip //print this

    //swap old for new
    @exec switchroo skip
  }
}
```

Figure A.3: Laplace protocol.

```
protocol @synthesis matrixmul p :{x: integer | x > 1}{

  @in getN
  val nDouble : { x:integer | x > 0 and x % p = 0}

  foreach i: 1.. p-1{

    @out getApart @in getRemoteApart
    message 0, i float[nDouble/p]
  }

  @in getB @out getB
  broadcast 0 float[nDouble]

  @exec calculate

  foreach i:1..p-1{

    @out getCremote @in getCpart
    message i, 0 float[nDouble/p];
  }

}
```

Figure A.4: Matrix Multiplication protocol.

```
protocol @synthesis nbodypipe p: {x: integer | x > 1}{

  @in init_npart
  val n: {x: natural | x >= p and x <= 1000000}

  @out get_npart @in get_counts
  allgather integer[1]; // was p instead of 1 before

  @exec sizes_displacements_initparticles
  @condition count
  loop {
    @exec load_initialBuf
    @condition pipe
    loop {
      @condition pipe_procs
      choice {
        foreach i: 0 .. p-1 {
          @out get_sendbuf @in get_recvbuf
          message i, (i + 1 <= p-1 ? i+1 : 0) float[n * 4]
        }
      }
      or
        { }
     @exec computeForces skip
     }

    @exec computeNewPos
    @out get_dt_est @in get_dt_new
    allreduce min float;
    @exec modifyTimeStamp skip
  }
}
```

Figure A.5: N-body simulation protocol.

```
protocol pi p: {x: integer | x > 1}{

  foreach i: 1 .. p-1 {

    @in getN @out getN
    message 0, i {x: integer | x > 1}
  }

  @exec computation
  @out getMyPi @in setPi
  reduce 0 sum float
}
```

Figure A.6: Pi protocol.

```
1  protocol @synthesis parallel_dot p: {x: integer | x > 1 }{
2
3    @out getN
4    broadcast 0 n:{x: integer | x > 0 and x % p = 0}
5
6    @exec readFirstVectorPart_X
7    foreach i: 1 .. p-1{
8      @out scanPartVector_X @in getLocalX
9      message 0, i float[n/p]
10    }
11
12   @exec readFirstVectorPart_Y
13   foreach i: 1 .. p-1{
14     @out scanPartVector_Y @in getLocalY
15     message 0,i float[n/p]
16   }
17
18   @exec serialDot
19   @out getLocalDot @in getDot
20   allreduce sum float
21
22   //print results
23   foreach i: 1 .. p-1{
24     @out getDot @in getRemoteDot
25     message i, 0 float;
26    @exec print_received skip
27   }
28 }
```

Figure A.7: Vector dot protocol.

Bibliography

[1] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[2] FEVS: A Functional Equivalence Verification Suite. `http://vsl.cis.udel.edu/fevs/index.html`.

[3] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. PLDI, pages 234–245. ACM, 2002.

[4] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard — Version 3.0.* 2012.

[5] I. Foster. *Designing and building Parallel programs.* Addison-Wesley, 1995.

[6] G. Gopalakrishnan, Robert M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky. Formal analysis of MPI-based Parallel programs. *CACM*, 54(12):82–91, 2011.

[7] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2^{nd} Ed.): Portable Parallel Programming with the Message-passing Interface.* MIT Press, 1999.

[8] Tobias Hilbrich, Martin Schulz, Bronis R. Supinski, and Matthias S. Muller. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In *Tools for High Performance Computing*, pages 53–66. Springer Berlin Heidelberg, 2010.

[9] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *SPPL*, POPL, pages 273–284. ACM, 2008.

[10] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381, pages 22–138. Springer, 1998.

[11] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. ECOOP, pages 516–541. Springer-Verlag, 2008.

[12] B. Krammer, K. Bidmon, M.S. Muller, and M.M. Resch. Marmot: An mpi analysis and checking tool. In *Parallel Computing Software Technology, Algorithms, Architectures and Applications*, volume 13 of *Advances in Parallel Computing*, pages 493 – 500. North-Holland, 2004.

[13] K. Rustan M. Leino. This is Boogie 2. Technical report, 2008. Microsoft Research, Redmond, WA, USA.

[14] Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, Nicholas Ng, and Nuno Martins. Towards deductive verification of MPI programs against session types. In *PLACES*, 2013.

[15] Francisco Martins, Nicholas Ng, César Santos, Vasco T. Vasconcelos, and Nobuko Yoshida. Specification and Verification of MPI Protocols. Technical report, 2013.

[16] Nuno Martins. Formal Verification of Parallel C+MPI Programs. Master's thesis, Department of Informatics, University of Lisbon, 2013.

[17] Matrix Multiplication using MPI, from Arizona University. `http://www.cs.arizona.edu/classes/cs522/fall12/examples/mpi-mm.c`.

[18] L.De Moura and N.Bjørner. Z3: an efficient SMT solver. In *TPS*, TACAS/ETAPS, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] MPI Sessions. `http://gloss.di.fc.ul.pt/MPISessions`.

[20] ATSMP - Advanced Type Systems for Multicore Programming. `http://gloss.di.fc.ul.pt/types-multicore`.

[21] George C. Necula. Proof-carrying Code. In *POPL*, pages 106–119. ACM Press, 1997.

[22] N. Ng, N. Yoshida, and K. Honda. Multiparty session C: safe parallel programming with message optimisation. In *ICOMCP*, TOOLS, pages 202–218, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[24] Martin Schulz and Bronis R. de Supinski. PNMPI tools: a whole lot greater than the sum of their parts. In *SC*, pages 30:1–30:10. ACM, 2007.

[25] S.F. Siegel and L.F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In *EuroPVM/MPI*, volume 5205 of *LNCS*, pages 274–282, 2008.

[26] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *TOSEM.*, 17(2):1–34, 2008.

[27] Matthew Sottile, Amruth Dakshinamurthy, Gilbert Hendry, and Damian Dechev. Semi-automatic Extraction of Software Skeletons for Benchmarking Large-scale Parallel Applications. SIGSIM-PADS, pages 1–10. ACM, 2013.

[28] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A Tool for Model Checking MPI Programs. PPoPP, pages 285–286. ACM, 2008.

[29] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *SC*, page 51. IEEE Computer Society, 2000.

[30] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *SC*. IEEE Computer Society, 2010.