Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

PROTOCOL BASED PROGRAMMING OF CONCURRENT SYSTEMS

César Augusto Ribeiro dos Santos

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2014

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

PROTOCOL BASED PROGRAMMING OF CONCURRENT SYSTEMS

César Augusto Ribeiro dos Santos

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Francisco Cipriano da Cunha Martins
e co-orientada pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

2014

Acknowledgments

This thesis would not have been possible without the amazing support of a lot of people.

My family, first and foremost. I thank my father for not even letting me think about dropping all of this and spending the rest of my life working at McDonalds. I am still not sure if that would not have been the right choice. I would also like to thank my mother, who supported me the whole time, even when I wanted to drop all of this and go working for McDonalds.

A special thank-you goes to my late grandfather, who was a major inspiration in my life, and without whom I might not have been able to finish this course.

I would also like to thank my supervisors for the opportunity to do this thesis and their support. A special thank-you goes out to professor Eduardo Marques, for constantly changing the format of the VCC export, which in turn forced me to massively refactor the plugin, greatly improving the quality of the code. I also thank him for being there in the lab with his students whenever he could, helping even when I needed.

I thank my friends for always brightening up my day with some good laughs, with a special thanks to Fernando Alves for actually taking the time to review my thesis and helping me improve it.

Finally, I thank my girlfriend, for giving me the strength to pull through to the end, and somehow putting up with my incessant whining about every tool I use being terrible and nothing working right. I dedicate this thesis to you Solange, it wouldn't have been finished without you.

Dedicatória.

Resumo

Desenvolver sistemas de software concorrentes (e paralelos) seguros é difícil. É muito mais difícil verificar software paralelo do que é fazer o mesmo para aplicações sequenciais. A *Message Passing Interface (MPI)* é uma especificação independente de linguagem para protocolos de comunicação, utilizada para programar computadores paralelos e baseada no paradigma de troca de mensagens, seguindo o paradigma *Single Program Multiple Data (SMPD)* em que um único pedaço de código é partilhado por todos os processos. Programas MPI apresentam uma séria de desafios de correção: o tipo de dados trocados na comunicação pode não corresponder, o que resulta em computações erradas, ou o programa pode entrar numa situação de impasse resultando em recursos desperdiçados. Este trabalho tem como objective melhorar o estado-da-arte da verificação de programas paralelos. O estado-da-arte na verificação de programas *MPI* utiliza técnicas de verificação de modelos que não escalam com o número de processos e são tipicamente feitas em tempo de execução, o que desperdiça recursos e está dependente da qualidade do conjunto de testes.

A nossa abordagem é inspirada em tipos de sessão multi-participante (*multi-party session types*). A teoria dos tipos de sessão parte da caracterização da comunicação entre vários processos de um programa de um ponto de vista global (*mensagem do processo 0 para o processo 1* corresponderia a *envio para o processo 1* no processo 0 e *recebo do processo 0* no processo 1 localmente). A esta caracterização dá-se o nome de *protocolo*. Protocolos bem formados são por construção livres de impasse e a comunicação é correcta (o tipo de dados enviado é o mesmo que o tipo de dados recebido). Se for provado que um programa segue um protocolo, então essas mesmas propriedades (ausência de impasses e correcção na comunicação) são preservadas para o programa.

Primeiro, um protocolo é especificado numa linguagem desenhada para o efeito. As primitivas suportadas pela linguagem de protocolos são baseadas nas primitivas *MPI*, com a adição de escolhas e ciclos colectivos. Estas primitivas representam tomadas de decisão colectivas que não recorrerem a comunicação. Tomadas de decisão colectivas acontecem em programas *SPMD* porque todos os processos partilham o mesmo código, e como tal é possível garantir que todos os programas calculam o mesmo valor num certo ponto do programa se os dados forem iguais. Além disso, foi

adicionada uma primitiva *foreach*, que expande um pedaço de protocolo para uma gama de valores. Esta primitiva permite que protocolos sejam paramétricos quanto ao número de processos. Os dados enviados nas mensagens podem ter restrições associadas, especificadas recorrendo a tipos dependentes.

Os protocolos são escritos num *plugin Eclipse*. O *plugin* valida a boa formação dos protocolos com base numa teoria, utilizando o *SMT solver Z3* da *Microsoft Research* para provar certas propriedades, e que as restrições nos tipos dependentes são congruentes. O *plugin* foi implementado recorrendo a *Xtext*, uma *framework* para desenvolvimentos de *plugins Eclipse*.

O *plugin*, além de validar a boa formação, compila os protocolos para vários formatos, entre os quais o formato *Why*. *Why* é uma das linguagens da plataforma *Why3*, uma plataforma para verificação dedutiva de software. A linguagem *Why* é utilizada para escrever teorias, e é aliada à linguagem *WhyML* (inspirada em *OCaml*) para escrita de programas.

Foi desenvolvida em *Why* uma teoria para protocolos, em que protocolos são especificados como um tipo de dados da linguagem. O ficheiro gerado pelo *plugin* especifica um protocolo utilizando os construtores deste tipo de dados. Para especificar as restrições de tipos dependentes, uma teoria de funções anónimas incluída com a plataforma *Why3* é utilizada.

Além disso, foi desenvolvida uma biblioteca *WhyML* para programação paralela. Esta biblioteca inclui primitivas inspiradas em MPI, e primitivas para escolhas colectivas e ciclos colectivos. Estas primitivas têm como pré-condição que a cabeça do protocolo é a primitiva esperada, e que os dados a serem enviados respeitam a restrição do tipo de dados a ser enviado. Todas as primitivas têm como parâmetro o estado actual do protocolo, e na sua pós-condição consomem a primitiva à cabeça. Graças a estas anotações é possível saber se o programa segue o protocolo, confirmando no final do programa se o protocolo foi consumido por completo.

Dado um programa paralelo escrito em *WhyML*, o protocolo em formato *Why* e a teoria de protocolos, o programador pode utilizar o *Why3 IDE* para verificar a conformidade do programa face ao protocolo. O *Why3 IDE* permite dividir a prova em partes, e provar cada parte com um *SMT solver* diferente. Caso nenhum *SMT solver* seja capaz de provar uma das sub-provas, o programador pode recorrer a um *proof assistant* e tratar da sub-prova manualmente.

Além das anotações de primitivas colectivas, o programador também precisa de por um anotação no final de cada bloco que verifica se o protocolo está vazio naquele ponto (sem a qual não é possível garantir que o protocolo foi seguido), de marcar que partes do código correspondem a que expansão da primitiva *foreach*, e precisa de adicionar variantes e invariantes aos ciclos.

Em resumo, a nossa abordagem é a seguinte:

1. O programador escreve um protocolo que explicita globalmente a comunicação que deve ocorrer.

2. Este protocolo, se bem formado, é correcto por construção. A comunicação é correcta e é livre de impasses.

3. A boa formação do protocolo é feita num *plugin Eclipse*, que também o compila para vários formatos, entre os quais o formato *Why*.

4. O programador escreve o seu programa paralelo em *WhyML*, recorrendo a uma biblioteca de programação paralela inspirada em *MPI* desenvolvida neste projecto.

5. As primitivas da biblioteca são anotadas com pré e pós-condições que verificam a conformidade do programa face ao protocolo.

6. Oprograma, aliado ao protocolo em formato *Why* e a uma teoria de protocolos, é verificado no *Why3 IDE*

7. O *Why3 IDE* permite dividir a prova em partes, e provar cada parte com um *SMT solver* diferente. No caso em que nenhum SMT solver consiga tratar da sub-prova, o programador pode recorrer a um *proof assistant* e tratar da sub-prova manualmente.

8. Se o programa passar na verificação, as propriedades do protocolo (correcção na comunicação e ausência de empasses) são garantidas para o programa.

Estas anotações impõe trabalho extra ao programador, mas são dentro do esperado para este tipo de ferramenta. A nossa solução foi testada com recurso a três programas *MPI*, obtidos em livros de texto.

Foram verificados vários exemplos clássicos de programação paralela, adaptados de livros de texto. Comparativamente à ferramenta mais próxima (que utiliza o verificador de programas *C*, *VCC*), o número de anotações da nossa solução é menor, as anotações enquadram-se melhor com o código e o tempo de verificação é semelhante.

No entanto, a nossa solução recorre a uma linguagem que não é apropriada para a industria, a linguagem *WhyML*. As linguagens *C* ou *Fortran* aliadas à biblioteca *MPI* são o *gold standard* para programação paralela de alta performance, e são as linguagens com que os programadores no ramo estão familiarizados.

Como tal, para trabalho futuro, propomos o desenvolvimento de uma linguagem de alto nível o mais semelhante possível a *C* ou *Fortran*, com primitivas de programação paralela *built-in*, incluindo escolhas colectivas e ciclos colectivos. Esta linguagem deverá compilar para código *C* ou *Fortran*, convertendo as primitivas da

linguagem em primitivas *MPI*. Este passo de conversão pode também optimizar o programa, recorrendo a primitivas de comunicação *MPI* mais eficientes que as usadas, sendo esta uma funcionalidade importante para programação paralela de alta performance cuja principal preocupação é a eficiência do programa.

Compilando também para *WhyML*, estes programas podem ser verificados. E como as primitivas de programação paralela são *built-in* na linguagem, a grande maioria das anotações necessárias pode ser gerada automaticamente.

É também necessário suportar mais funcionalidades *MPI*, incluindo comunicação assíncrona, topologias e comunicadores.

Palavras-chave: Programação Paralela, MPI, Verificação de Software, Tipos de Sessão, Sistemas de Tipos

Abstract

Developing safe, concurrent (and parallel) software systems is hard. It is much more difficult to debug or verify parallel software than it is to do the same for sequential applications. The Message Passing Interface (MPI) [12] is a language-independent specification for communication protocols, used to program parallel computers and based on the message-passing paradigm. MPI programs present a number of correctness challenges: communication may not match, resulting in errant computations, or the program may deadlock resulting in wasted resources.

This work hopes to improve the state-of-the-art of MPI program verification. The state-of-the-art in MPI program verification relies on model-checking techniques which do not scale with the number of processes and are typically done at runtime, which wastes resources and is dependent on the quality of the test set. Our approach is inspired by multi-party session types. First, a protocol is specified in a language designed for the purpose. A well-formed protocol is communication-safe and deadlock free. An Eclipse plugin verifies the well-formation of the protocol, and compiles it to Why, a language of a deductive software verification platform called Why3. This compiled protocol, allied with a theory for protocols also written in Why, is used to verify parallel WhyML programs, WhyML being another language of the Why3 platform. If a program passes verification, the properties of communication safety and deadlock freedom are preserved for the program. This verification occurs at compile time and is not dependent on any kind of test set, avoiding the issues of model-checking techniques.

We verified several parallel programs from textbooks using our approach.

Keywords: Parallel Programming, MPI, Software Verification, Session Types, Type Systems

Contents

Chapter 1

Introduction

1.1 Motivation

Parallel programming is difficult to get right. The Message Passing Interface (MPI) [12], a language-independent specification for communication protocols based on the message-passing paradigm, is the *de facto* standard for High Performance Computing (HPC). Some of the challenges in developing correct MPI programs include: communication that does not match, resulting in incorrect computations, and deadlocks, which result in wasted time and resources.

High performance computing (HPC) bugs are very costly, high-end HPC centres cost hundreds of millions to commission, and the machines become obsolete within six years. On many of these centres, over 3 million dollars are spent in electricity costs alone each year and research teams apply for computer time through competitive proposals, spending years planning experiments [15]. A deadlocked program represents an exorbitant amount of wasted money as such situations are hard to detect at runtime without resource wasting monitors. One must also consider the societal costs, since there is a reliance on the results of these experiments (weather simulations for example).

Verifying the correctness of concurrent (and parallel) software systems is also hard. Several methodologies are employed in the formal verification of MPI programs, such as model checking and symbolic execution. These approaches typically face a scalability problem, since the verification state space grows exponentially with the number of processes. Verifications of real-world applications may restrict the number of processes in the state space to only a few [33].

1.2 Objectives

This work aims at improving the state-of-the-art of parallel program verification. Our approach is inspired by *multi-party session types* [21]. The theory of session

types allows the specification of a protocol as a type. Such a type describes not only the data types exchanged in messages, but also the state transitions of the protocol and hence the allowable sequence of messages. A well-formed protocol is communication-safe and deadlock free. With session types it is possible to verify, at compile-time, that participants communicate in accordance with the protocol and, if so, the properties of communication-safety and deadlock freedom are preserved for the program.

The idea is as follows: first, a protocol is specified in a language designed for the purpose. The plugin verifies if the protocol is well-formed and, if it does, compiles it to a format that can be processed by a deductive program verifier. A parallel program is then checked against the generated protocol. For that, it is necessary that all parallel primitives be annotated with pre and post-conditions based on the current state of the protocol. If all pre-conditions match, and the protocol fully processed at the end, then program matches the protocol and is therefore type-safe and free from deadlocks.

This work is included in a larger ongoing project of parallel verification, Advanced Type Systems for Multicore Programming (PTDC/EIA-CCO/122547/2010), where verification was carried out on C+MPI code through the usage of a deductive software verifier for C, VCC [7]. The C language has very complex operational semantics [27, 32], with many different specifications attempted [27], none of which fully specifies the language. Because of that it is not possible to guarantee the progression of a C program according to a protocol.

Programming MPI in C is very error prone and requires many annotations regarding concurrency and pointer arithmetic. Using an existing higher level language with MPI support does not solve the problem, communication primitives must be a fundamental part of it, otherwise errors like mismatched types in messages and deadlocks can still occur. We present an implementation of a higher level language with first class support for parallel MPI-like primitives using WhyML, a language that is part of Why3 [11] a deductive software verification platform that also features a rich well-defined specification language called Why. All the primitives are protocol aware, and using Why3 the programmer can guarantee his program is correct.

Why3 allows the user to split proofs in parts and prove each part using a different Satisfiability Modulo Theories (SMT) solver. In cases where the solvers cannot handle part of the proof, Why3 can generate files for use with proof assistants like Coq [3], allowing the user to handle those parts manually.

1.3 Contributions

The contributions of this work are:

1. A plugin for the development of protocols,
2. A compiler of protocols to Why format,
3. A theory for protocols in Why,
4. Verification of sample WhyML programs against a given protocol.

1.4 Document structure

This document is organised as follows:

- **Chapter 1 - Introduction:** Describes the motivation, objectives and contributions of the work.
- **Chapter 2 - Related Work:** Describes the tools this work builds upon and compares it to similar tools.
- **Chapter 3 - Protocol Language:** Describes the protocol language and the protocol validator.
- **Chapter 4 - Programming language:** Describes the programming language and verification using Why3.
- **Chapter 5 - Implementation:** Explains in detail how the various contributions were implemented.
- **Chapter 6 - Evaluation:** Describes the results obtained and evaluates them in comparison to alternatives.
- **Chapter 7 - Conclusion:** Presents conclusions and possible future work.

Chapter 2

Related work

2.1 MPI

MPI [12] is a language-independent specification for communication protocols used to program parallel computers. Based on the message-passing paradigm, it is both a programmer interface and a specification for how its features must behave in any implementation. MPI is not sanctioned by a standards body but it has become a *de facto* standard for programming parallel applications that run on distributed memory systems, such as super computers and clusters. It is the dominant model in high-performance computing.

MPI programs are typically written in C or Fortran (although there are non-official bindings for many other programming languages). They consist of one or more processes, each with its own private memory, that communicate with each other through various kinds of message exchanges, and follow the *Single Program Multiple Data* paradigm (SPMD). In SPMD, multiple processes execute the same program at independent points. Serial sections of the program are implemented by identical computation on all processes, rather than computing the result on one process and sending it to the others. This also means that program control-flow follows the same path if it is only dependent on data that is known to all processes. We call these collective control-flow operations *collective choices* and *collective loops*.

Participant specific behaviour can be separated with conditionals based on the participant's *rank*, a unique identifier that is attributed to each process. The two most commonly used types of message exchanges are *point-to-point* messages (sends and receives) and *collective operations* (broadcasts and reductions for example).

Messages can be synchronous or asynchronous, blocking or non-blocking. Blocking operations wait until data is transmitted, while synchronous operations wait until the other party signals it is ready to perform an operation, it is a subtle difference but it is important to note. This work focuses on blocking synchronous operations like those in MPI.

```c
int main(int argc, char** argv) {
  int procs;              // Number of processes
  int rank;               // Process rank
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...
  int n = atoi(argv[1]);            // Global problem size
  int lsize = n / procs;
  if (rank == 0)
     read_vector(work, n);
  MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
  int left  = (procs + rank - 1) % procs;  // Left neighbour
  int right = (rank + 1) % procs;          // Right neighbour
  int iter = 0;
  // Loop until minimum differences converged or max iterations attained
  while (!converged(globalerr) && iter < MAX_ITER)) {
    ...
    MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
    MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
    MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
    ...
    MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
    ...
  }
  ...
  if (converged(globalerr)) { // Gather solution at rank 0
    MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
    ...
  }
  ...
  MPI_Finalize();
  return 0;
}
```

Figure 2.1: Finite Differences C+MPI program.

MPI also allows processes and communication spaces to be structured using *topologies* and *communicators*. It also supports persistence, user-defined datatypes, one-sided communications, file I/O, among others. Figure 2.1 shows an example MPI program written in C.

2.2 Multi-party session types

A session is a series of reciprocal interactions between two parties, possibly with branching and recursion, and serves as a unit of abstraction for describing interaction. It is based on the message-passing style of parallel programming. Session types [20] are a type discipline for sessions.

Multi-party session types [21] extend the theory to an arbitrary number of participants. Parameterised multi-party session types [9] further extend the theory to support a parametric number of participants. Multi-party session types begin with the design of a global protocol which specifies the intended interaction between participating processes. The global protocol implicitly defines a local protocol for each participant, through the notion of projection. Figure 2.2 shows the approach followed by multi-party session types.

A local protocol defines the role a certain participant has in the global interac-

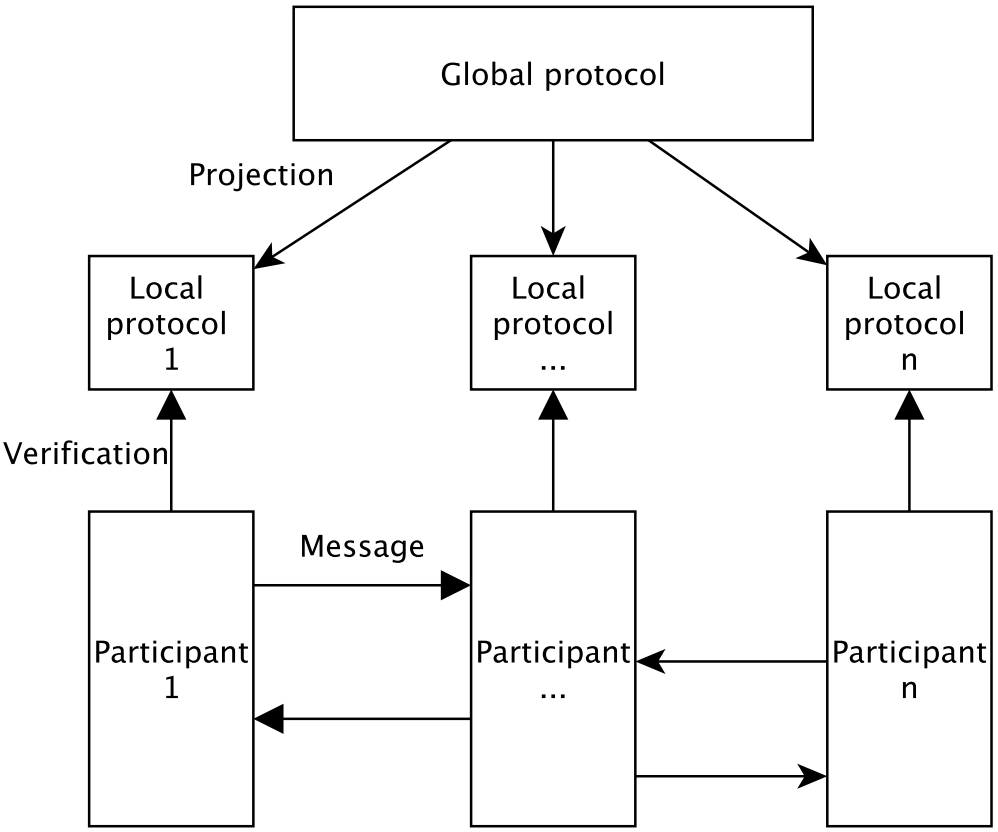

Figure 2.2: Multi-party session types approach

tion. Well-formed protocols verify, by construction, the properties of type safety, communication safety and deadlock freedom. These properties extend to the participants if they conform to the corresponding local protocol. The approach followed by this work is inspired by multi-party session types.

2.3 Why3

Why3 [4, 11] is the current generation of Why, a platform for deductive program verification. It provides a rich language of specification and programming, called WhyML, and relies on external theorem provers, both automated and interactive, to discharge verification conditions. It is mainly motivated by the necessity to model the behaviour of programs and formally prove their properties [4]. WhyML is used as an intermediate language for the verification of programs in a variety of languages such as C, Java and Ada [11]. It can also work as a programming language inheriting features and syntax from ML. WhyML has two components: a specification component and a programming component.

The specification component is used to write program annotations and back-

ground logical theories. It is based on first-order logic with rank-1 polymorphic types [25] and several extensions. The specification language does not depend on features of the programming language and can serve as a common format for theorem proving problems as Why3 can dispatch proof obligations to various provers.

The programming component has two important restrictions: it has no higher-order functions, and does not have a memory model, imposing instead a static control of aliases. This enables the generation of first-order proof obligations that are simultaneously tractable for provers and readable for users. Functions are annotated with pre and post-conditions for normal and exceptional termination, and loops are annotated with invariants. While-loops and recursive functions can be given variants to ensure termination. Programs written in C can be verified by modelling the algorithm in WhyML, and using Why3 to formally verify it [10]. This work uses Why3 to verify MPI programs written in WhyML.

2.4 Xtext

Xtext [1] is an Eclipse [14] based framework for the development of programming languages and domain specific languages. It covers all aspects of a complete language infrastructure: parser, linker, compiler, interpreter and even Eclipse IDE integration. Xtext provides a set of domain-specific languages and APIs to describe different aspects of a programming language. The compiler components of the language are independent of Eclipse and can be used in any Java environment. This includes the parser, the abstract syntax tree, the serialiser, the code formatter, the scoping framework, the linker, compiler checks and static analysis and the code generator or interpreter. The Eclipse plugin for the protocol specification language was created with Xtext.

2.5 VCC

VCC [7] 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. 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 [2], an intermediate language used by multiple verification tools. That code is verified by the Z3 [8] SMT solver to assert the correctness of the Boogie/VCC code. The plugin developed as part of this work can also output protocol in VCC format, an example for the finite differences problem can be seen in Figure 2.3.

```
_(pure Protocol program_protocol ()
    _(reads {})
    _(ensures \result ==
        seq(size(\lambda \integer x; x > 1),
         seq(abs(\lambda \integer p;
          seq(val(\lambda \integer x; x >= 0 && x % p == 0),
           seq(abs(\lambda \integer n;
            seq(scatter(0, floatRefinement(\lambda float* _x4; \integer _x4_length; _x4_length == n))
                ,
            seq(foreach(0, p - 1, \lambda \integer i;
                 seq(message(i, (i + 1) % p, intRefinement(\lambda int* _x5; \integer _x5_length; \
                     true && _x5_length == 1)),
                 skip())
            ),
            seq(loop(
                 seq(bcast(0, intRefinement(\lambda int* _x6; \integer _x6_length; \true &&
                     _x6_length == 1)),
                  seq(abs(\lambda \integer z;
                   seq(allreduce(MPI_MAX, floatRefinement(\lambda float* _x7; \integer _x7_length; \
                       true && _x7_length == 1)),
                   skip())
                  ),
                 skip()))
            ),
            seq(choice(
                 seq(gather(1, intRefinement(\lambda int* x; \integer x_length; (\forall \integer _x9
                     ; (0 <= _x9 && _x9 < x_length) ==> x[_x9] <= 50) && x_length > 0)),
                 skip())
            ,// or
                 skip()
            ),
            skip()))))
           ),
          skip()))
         ),
        skip()))
    );
)
```

Figure 2.3: Protocol in VCC format.

2.6 Scribble

The Scribble language [19] is a platform-independent description language for the specification of asynchronous, multiparty message passing protocols, built on a rigorous mathematical basis. Scribble is based on the theory of multi-party session types. Protocols are specified using the type language of Scribble, which is the most abstract level of the description layers in Scribble. Conversation models provide a foundation for design-by-contract through assertions written in a logical language. Finally there are languages for describing detailed behaviour, reaching executable descriptions.

Scribble can be used to statically validate parallel applications based on message-passing, but also dynamic validation (monitoring) of message exchanges which is useful for protecting against untrusted participants. Figure 2.4 shows a protocol written in Scribble. Although similar to our approach, Scribble does not follow the synchronous semantics of MPI, nor does it feature process ranks as identifiers or collective primitives.

```
type <xsd> "ProductId" from "ProductId.xsd" as ProductID;
type <xsd> "Calendar" from "Calendar.xsd" as Calendar;
type <java> "java.lang.Integer" from "rt.jar" as int;

global protocol BuyerBrokerSupplier(role Buyer, role Broker, role Supplier) {
    rec START { // Recursion point for the "Redo" scenario
        // the common initial four steps
        query(ProductID) from Buyer to Broker;
        query(ProductID) from Broker to Supplier;
        price(int) from Supplier to Broker;
        price(int) from Broker to Buyer;
        choice at Buyer { // Buyer decides the protocol scenario to follow
            // "Redo" scenario
            redo() from Buyer to Broker;
            redo() from Broker to Supplier;
            continue START; // Protocol flow returns to START recursion point
        } or {
            // "Accept" scenario
            accept() from Buyer to Broker;
            confirm() from Broker to Supplier;
            date(Calendar) from Supplier to Broker;
            date(Calendar) from Broker to Buyer;
        } or {
            // "Reject" scenario
            reject() from Buyer to Broker;
            cancel() from Broker to Supplier;
        }
    }
}
```

Figure 2.4: Example protocol in the Scribble language

2.7 Session C

Session C [24] is a multiparty session-based programming framework for message-passing parallel algorithms in C. It follows an approach similar to ours, starting with the specification of a global protocol (written in Scribble) for a certain parallel algorithm, from which a projection algorithm generates endpoint protocols, based on which each endpoint C program is designed and implemented with a small number of session primitives, which can be statically validated. Our approach does not use a projection algorithm, relying instead on deductive techniques.

Session C guarantees deadlock freedom, type-safety, communication-safety and global progress for any well-typed programs. The underlying theory ensures that the complexity of the toolchain stays in polynomial time against the size of programs. The biggest limitation is that this approach requires the mastering of a new API, while our approach targets MPI, the dominant model for high-performance computing.

2.8 ISP

ISP [30] is a dynamic analyser of MPI programs that employs run-time model checking methods based on dynamic partial order reduction (DPOR) using a fixed test suite. ISP can check for deadlocks, violations of assertions placed by the user and exceptions thrown at runtime. It exhaustively explores all relevant interleavings

of the given MPI process as determined by DPOR. The first interleaving is chosen at random by following a standard depth-first search. Afterwards it traverses up the stack, having DPOR identify points where adding interleavings might be useful, and continues the search. ISP uses P^{N}MPI [31] instrumentation to trap MPI calls.

Some of the biggest problems with ISP are that it requires restarting from MPI_Init to explore each new interleaving which causes a huge overhead [36] and it is only compatible with MPICH2. It is not scalable, not even to a small number of MPI primitives, and being a runtime verifier, it is dependent on the quality of the tests. Our approach does not have any of these problems.

2.9 DAMPI

DAMPI [37] is a distributed dynamic analyser of MPI programs, the first that guarantees scalable coverage of the space of non-determinism. It does so by employing heuristics to focus coverage to regions of interest. It detects deadlocks and resource-leaks in real applications using over a thousand processes. It is a followup to ISP, having a decentralised scheduling algorithm based on Lamport clocks [23]. Much like ISP, it uses P^{N}MPI instrumentation to trap MPI calls. DAMPI cannot detect the deadlock that occurs when two processes are trying to send a message to each other, and can give false positives due to the timeout based deadlock detection [18]. Although DAMPI does not require the writing of a protocol or model and, unlike ISP, is scalable, it is a runtime verifier. As such, it is dependent on the quality of the tests, unlike our approach. Our approach does not present false positives.

2.10 MUST

MUST [17, 18], preceded by Marmot [22] and Umpire [5], utilises an approach similar to DAMPI. Like DAMPI, MUST utilises P^{N}MPI, and focuses on the same runtime verifications. A big advantage over DAMPI is that it does not have false positives for deadlock detection and it enables a comprehensive understanding of the source of the deadlock. It also features local verifications, such as pointer validations, memory access, and resource leaks. It can handle up to 1024 processes with the full verification set. MUST has the same problems shared by runtime verifications that our approach lacks.

2.11 TASS

TASS [34] is a suite of tools for the formal verification of MPI-based parallel programs. TASS takes an integer n larger than one and a C+MPI program, and constructs an abstract model of the program with n processes. The model is explored using symbolic execution and explicit state space enumeration.

A number of techniques are used to reduce the time and memory consumed. TASS performs a number of checks: absence of deadlocks, buffer overflows, reading of uninitialised variables, division by zero, memory leaks and assertion violations; that the type and size of a message received is compatible with the receive buffer type; proper use of malloc and free, among others. It can also verify the functional equivalence between an MPI program and its sequential version. Unlike tools such as ISP or DAMPI, it can reason about an infinite input space. The biggest problem with TASS is that it does not scale with the number of processes, which is not a problem for our approach.

2.12 Parallel data-flow analysis

Parallel data-flow analysis [6] is a compiler analysis framework that extends traditional data-flow analyses to message passing applications, such as MPI based applications, on an arbitrary number of processes. Parallel data-flow analysis makes it possible to analyse the interactions between different processes and to detect the shape of the communication topology by statically matching pairs of send and receive operations that may communicate at runtime. The communication model considers send and receive operations exclusively.

While parallel control flow graphs (pCFGs) may be infinite in size (the number of possible processes is unbounded), this framework focuses on finite pCFGs that are bounded to represent only a finite number of process sets. This is sufficient for some real world applications, which typically divide processes into several groups, each of which performs a specific role or pattern of communication operations.

The analysis performed by this approach allows for optimisations of the communication pattern to maximise performance for a given network, constant propagation, and detection of potential bugs including message leaks and inconsistent types on the sender and receiver. Unlike our approach it makes no attempts at deadlock detection, and lacks support for collective MPI operations.

Chapter 3

Protocol language

Our approach for the development of parallel software involves the creation of a protocol that specifies the communication that is to occur. This protocol, if well-formed, is correct by construction. The messages exchanged are type safe and it is not possible to write a protocol that will deadlock. If we can guarantee a program follows the protocol, we can guarantee those properties for the program as well.

We developed a language for the specification of such protocols. The language is backed by a theory, and we developed an Eclipse plugin for the creation and validation of these protocols. That plugin can also compile the protocol into VCC and Why3 formats. VCC verification is outside the scope of this work.

Compiling to Why3 format allows the programmer to later verify a parallel program written in WhyML. The full workflow can be seen in Figure 3.1.

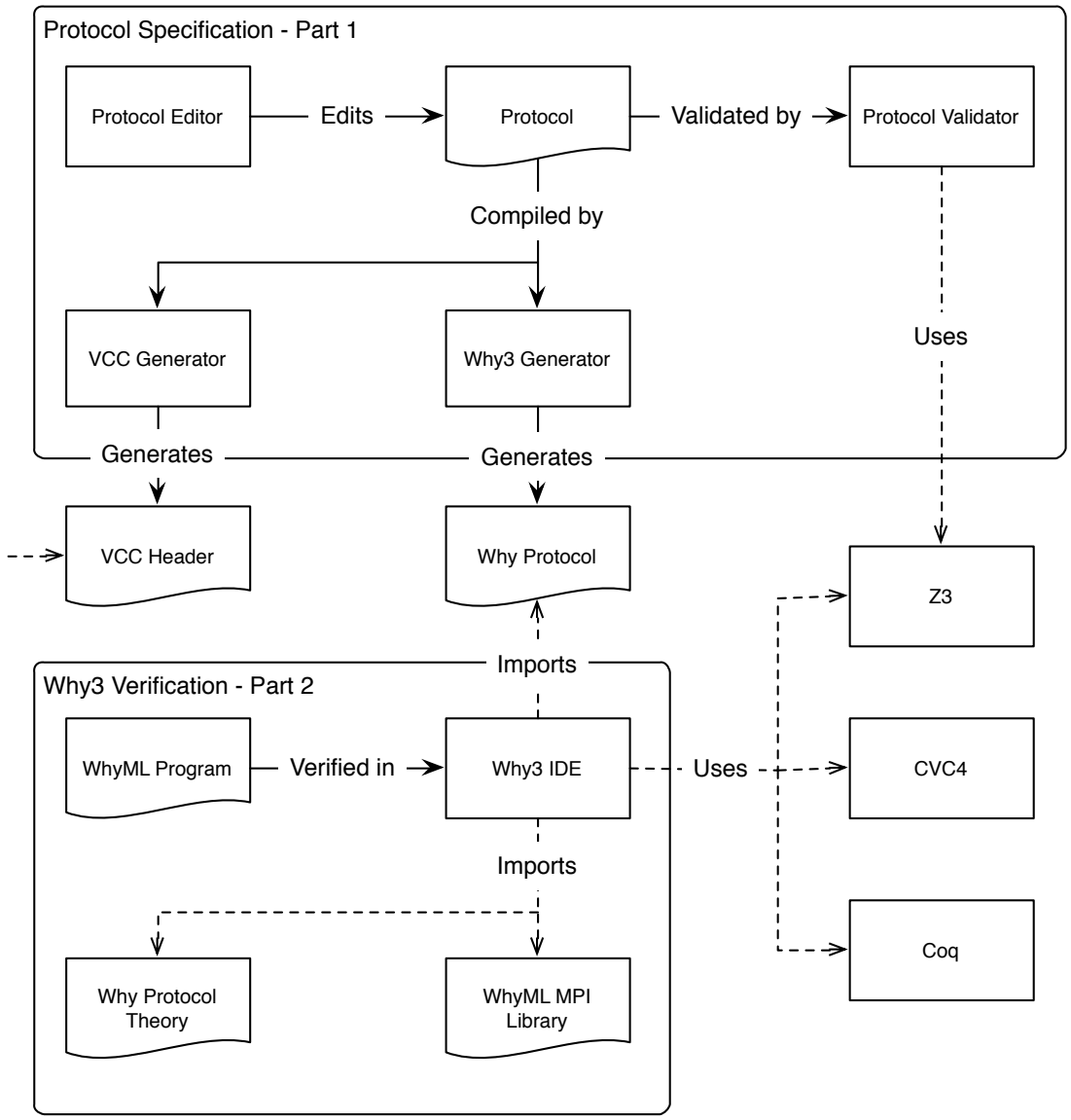

Figure 3.1: Workflow

3.1 The finite differences problem

To illustrate the protocol language and how software is developed using this methodology we present a parallel program that solves the finite differences problem. This program is a classic MPI example, and it uses every major feature of the protocol language.

Finite differences is a method of solving differential equations. The program starts with an initial solution X_0, and calculates $X_1, X_2, X_3, \ldots$ iteratively until it reaches a certain error threshold or a maximum number of iterations are executed. The problem vector is split amongst all processes, each calculating their part of the problem, and then joined at the end. The processes are setup in a ring topology

like in Figure 3.2 to exchange boundary values necessary for the calculation of the differences.

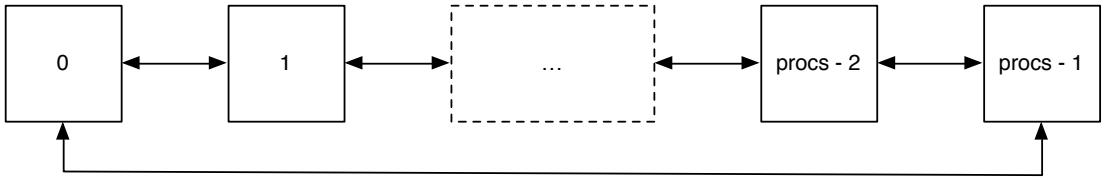

Figure 3.2: Processes in a ring topology

Programs in our approach function like MPI, following the *Single Program Multiple Data* (SPMD) paradigm. A single piece of code is executed for every process, with different processes executing different code based on their rank (a process identifier). Protocols have a global view of the communication, explicitly stating which processes are communicating when. A protocol for the finite differences problem can be seen in Figure 3.3.

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

Figure 3.3: Finite differences protocol

Every protocol specification starts with the keyword **`protocol`** (line 1), followed by a protocol name, and a constant that represents the number of processes. The name of the constant does not have to be the same used in the program, in this example the constant is given the name `p` and type `{x:int | x >=2}`. Types in the protocol language can be refined, that is, they can have a restriction attached to them. The type for `p` means *any integer greater than or equal to 2*. The language supports some abbreviations for refined types: **`natural`**, which is *any number greater than or equal to 0*, and **`positive`**, which is *any number greater than 0*.

The protocol starts by specifying a global value, the size of the work vector. A global value is known by every process but was not exchanged by communication. Such values are introduced with the keyword **`val`** (line 2). Much like with the number of processes, the value is given a name, `n`, and a type restriction, in this case *any natural divisible by the number of processes*.

Process 0 performs a **scatter** operation splitting an array of size `n` to all processes (line 3). Type **float**`[n]` is an abbreviation of `{x: float[] | length(x) = n}`, it is possible to specify restrictions on the size of the array and every value it contains. Scatter does not introduce a value as its result is not global and could not be referred to in the rest of the protocol.

The protocol enters a collective **loop** (lines 4–10). A *collective loop* is a loop where every process reaches the same decision at the same time without exchanging any message for the purpose. This sort of collective operation is the major difference between our approach and something like Scribble[19]. Inside the loop is a **foreach** operation (lines 5–8). The **foreach** operation is not a communication primitive, but a macro that expands its contents for every `i` between the two bounds. This macro allows protocols to be parametric regarding the number of processes for example, or another value known by all processes. If we were to expand it, it would result in message exchanges like Figure 3.4.

```
1  message 0, p-1 float
2  message 0, 1 float
3  message 1, 0 float
4  message 1, 2 float
5  message 2, 1 float
6  message 2, 3 float
7  ...
8  message p-1, p-2 float
9  message p-1, 0 float
```

Figure 3.4: Expanded foreach

Process 0 sends a **message** to the process on its left, then to the process on its right (lines 6–7). Process 1 does the same and so on until process $p - 1$. Note that there can be messages being exchanged simultaneously, a restriction on the order of messages being sent or received happens on a per-process level.

The rest of the protocol is simple, the loop has an **allreduce** operation at the end (line 9), this operation has every process send every other process its local error, and then calculates the global error, which is the maximum of all local errors. Finally, there's the collective choice, where either process 0 performs a **gather** operation in case the solution was found, or no communication occurs (lines 11–14, `{}` or **skip** is the empty protocol). A *collective choice*, like the loop, has every process reach the same decision at the same time without exchanging any message for the purpose.

A full grammar for the protocol language can be seen in Figure 3.5.

$$
\begin{array}{rll}
T & ::= \mathsf{skip} & \text{empty} \\
& \mid \mathsf{message}\ i\ , i & \text{point-to-point comm.} \\
& \mid \mathsf{broadcast}\ i\ x\colon D & \text{broadcast operation} \\
& \mid \mathsf{scatter}\ i & \text{scatter operation} \\
& \mid \mathsf{gather}\ i & \text{gather operation} \\
& \mid \mathsf{reduce}\ op & \text{reduce operation} \\
& \mid \mathsf{allgather} & \text{allgather operation} \\
& \mid \mathsf{allreduce}\ op\ x\colon D & \text{allreduce operation} \\
& \mid P; Q & \text{sequence} \\
& \mid \mathsf{foreach}\ x\colon i..i\ T & \text{repetition} \\
& \mid \mathsf{loop}\ T & \text{collective loop} \\
& \mid \mathsf{choice}\ T\ \mathsf{or}\ T & \text{collective choice} \\
& \mid \mathsf{val}\ x\colon D & \text{variable} \\
D & ::= \mathsf{integer} \mid \mathsf{float} \mid D[i] \mid \{x\colon D \mid p\} \mid ... & \text{refined types} \\
i & ::= x \mid n \mid i + i \mid \mathsf{max}(i, i) \mid \mathsf{length}(i) \mid i[i] \mid \ \ldots & \text{index terms} \\
p & ::= \mathsf{true} \mid i \leq i \mid p\ \mathsf{and}\ p \mid a(i, \ldots i) \mid \ \ldots & \text{index propositions} \\
op & ::= \mathsf{max} \mid \mathsf{min} \mid \mathsf{sum} \mid \ \ldots & \text{functions for reduce}
\end{array}
$$

Figure 3.5: Protocol language grammar

3.2 Protocol validator

The protocols are written in an eclipse plugin. This plugin (Figure 3.6) does syntax highlighting, real-time validation and compilation.

Protocols are validated according to a theory that specifies in what circumstances a protocol is well-formed. For example, Figure 3.7 shows the rules for the well-formation of each MPI primitive and datatypes. The syntax is not the same as the one used in the plugin, as that is made to look more familiar to C developers. Whenever a value is introduced by a primitive, which only occurs for primitives where the value ends up being known by all processes, it is placed on the context (a symbol table from variables to datatypes). Primitives that introduce a value have the rest of the protocol as a continuation. Primitives that do not are joined using a sequencing operator ($T;T$). An explanation for some of the rules follows:

- The rule for val states that the value sent must be a valid datatype, and that value will be propagated to the rest of the protocol by placing it in the context and moving to the continuation.
- The rule for reduce states that the rank of the root process must be between

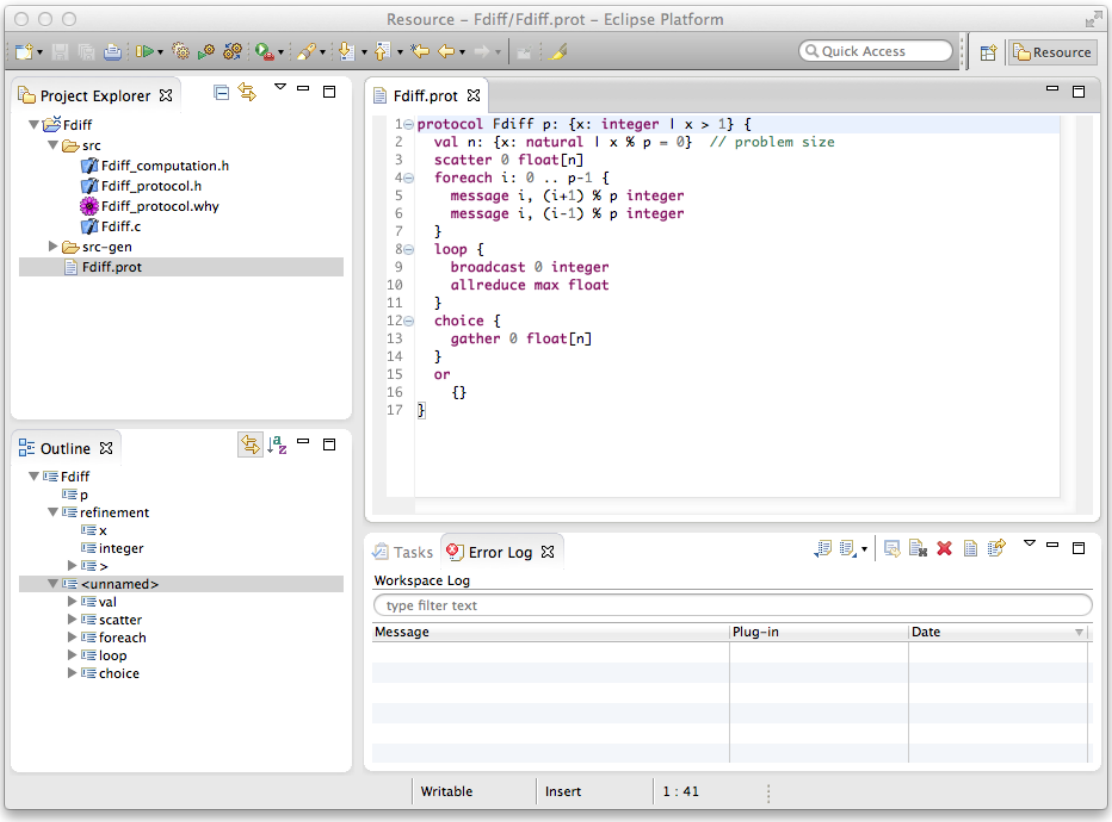

Figure 3.6: MPI Sessions plugin

0 and the number of processes, **sat** meaning this restriction must be proved satisfiable by an SMT solver.

- The rule for message also states that the ranks of the sender and receiver must be between 0 and the number of processes, but also that they cannot be the same number. The data exchanged in the message must be a valid datatype, but that value is not propagated to the rest of the protocol.
- The rule for broadcast also has a rank restriction and a value introduced.
- The rule for gather has a restriction on subtyping (<:), the data exchanged must be a subtype of an array.
- The rule for foreach introduces a value into the context with a range of values and then checks the body.
- The rule for skip, which signifies the empty protocol, checks that there are no inconsistent refinements and no repeated names in the context.
- The rule for sequence $(T;T)$ checks all the sub protocols in order, the rule for looping (T^*) checks the loop body, and the rule for choice $(T+T)$ checks both

Type formation, $\Gamma \vdash T : \mathbf{type}$

$$\frac{\Gamma, x\colon D \vdash T : \mathbf{type}}{\Gamma \vdash \mathsf{val}\ x\colon D.T : \mathbf{type}} \qquad \frac{\Gamma \vdash 1 \le i \le \mathsf{p}\ \mathbf{sat}}{\Gamma \vdash \mathsf{reduce}\ i : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \le i_1, i_2 \le \mathsf{p} \wedge i_1 \neq i_2\ \mathbf{sat} \quad \Gamma \vdash D : \mathbf{dtype}}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \le i \le \mathsf{p}\ \mathbf{sat} \quad \Gamma, x\colon D \vdash T : \mathbf{type}}{\Gamma \vdash \mathsf{broadcast}\ i\ x\colon D.T : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \le i \le \mathsf{p}\ \mathbf{sat} \quad \Gamma \vdash D_1 <: D_2\ \mathsf{array}}{\Gamma \vdash \mathsf{gather}\ i\ D_1 : \mathbf{type}}$$

$$\frac{\text{see gather}}{\Gamma \vdash \mathsf{scatter}\ i\ D : \mathbf{type}} \qquad \frac{\Gamma, x\colon \{y\colon \mathsf{integer} \mid i_1 \le y \le i_2\} \vdash T : \mathbf{type}}{\Gamma \vdash \mathsf{foreach}\ x \in i_1..i_2\ \mathsf{do}\ T : \mathbf{type}}$$

$$\frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{skip} : \mathbf{type}} \qquad \frac{\Gamma \vdash T_1 : \mathbf{type} \quad \Gamma \vdash T_2 : \mathbf{type}}{\Gamma \vdash T_1; T_2 : \mathbf{type}}$$

$$\frac{\Gamma \vdash T : \mathbf{type}}{\Gamma \vdash T^* : \mathbf{type}} \qquad \frac{\Gamma \vdash T_1 : \mathbf{type} \quad \Gamma \vdash T_2 : \mathbf{type}}{\Gamma \vdash T_1 + T_2 : \mathbf{type}}$$

Datatype formation, $\Gamma \vdash D : \mathbf{dtype}$

$$\frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{integer} : \mathbf{dtype}} \qquad \frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{float} : \mathbf{dtype}}$$

$$\frac{\Gamma \vdash D : \mathbf{dtype}}{\Gamma \vdash D\ \mathsf{array} : \mathbf{dtype}} \qquad \frac{\Gamma, x\colon D \vdash p\ \mathbf{sat}}{\Gamma \vdash \{x\colon D \mid p\} : \mathbf{dtype}}$$

Proposition formation, $\Gamma \vdash p : \mathbf{prop}$

$$\frac{\Gamma \vdash p_1 : \mathbf{prop} \quad \Gamma \vdash p_2 : \mathbf{prop}}{\Gamma \vdash p_1\ \mathsf{and}\ p_2 : \mathbf{prop}} \qquad \frac{\Gamma \vdash i_1\colon \mathsf{integer} \quad \Gamma \vdash i_2\colon \mathsf{integer}}{\Gamma \vdash i_1 \le i_2 : \mathbf{prop}} \qquad \frac{\Gamma \vdash i_1\colon \mathsf{float} \quad \Gamma \vdash i_2\colon \mathsf{float}}{\Gamma \vdash i_1 \le i_2 : \mathbf{prop}}$$

Context formation, $\Gamma : \mathbf{context}$

$$\frac{}{\cdot : \mathbf{context}} \qquad \frac{\Gamma : \mathbf{context} \quad \Gamma \vdash D : \mathbf{dtype} \quad x \notin \Gamma, D}{\Gamma, x\colon D : \mathbf{context}}$$

Figure 3.7: Formation rules

alternatives.

3.3 Why3 Protocol

Compiling the protocol to Why3 format generates a Why3 theory containing the protocol as a constant of a Why3 datatype, Figure 3.8. That datatype is specified in the theory sessiontypes . Protocols (Figure 5.7, chapter 5, section 3). Theory sessiontypes . ConstantArrays is for immutable arrays. We had to make that theory

```
theory Fdiff
    use import int.Int
    use import int.ComputerDivision
    use import real.RealInfix
    use import HighOrd
    use import sessiontypes.Protocols
    use import sessiontypes.ConstantArrays

    axiom size_ref: size >= 3
    constant p : int = size

    constant fdiff_protocol : protocol =
        Val (CInt (\n. n > 0) (\n.
        Scatter 0 any_array_float (
        Loop (
            ForEach 0 (p - 1) (\i.
                Message i (if i > 0 then i-1 else p-1) any_float (
                Message i (if i < p-1 then i+1 else 0) any_float (
                Skip))
            )(
            AllReduce Max (CFloat (\x.true) (\_x4.
            Skip)))
        )(
        Choice (
            Gather 0 any_array_float (
            Skip)
        )(
            Skip
        )
        Skip))))
end
```

Figure 3.8: Protocol in Why3 format.

because Why3 does not support arrays on the theory side (only on WhyML programs), and protocols may have arrays as values. The other imported theories are part of the Why3 Standard library.

The constructors of the datatype match those of the protocol language except that they appear capitalised. The big difference is in how values are introduced and how the protocol is sequenced. For example, the constructor **Val** has as its sole parameter a datatype. Since **Val** introduces a value to the protocol, the datatype must have a continuation so that the value can be referenced on other primitives. Other constructors like **Gather** do not introduce a value, and have the remainder of the protocol as a parameter (the first parameter to **Gather** being the root process just like in the protocol). The datatype passed should only have a predicate and not a continuation.

There are eight possible constructors for datatypes, all of which have a predicate attached to specify a restriction. This predicate is implemented as a higher order function. The first four constructors (**Int**, **Float**, **ArrayInt**, **ArrayFloat**) only have a restriction, while the other four, with a capital C in the name (**CInt**, **CFloat**, **CArrayInt**, **CArrayFloat**), have a restriction and a continuation. This continuation is a higher order function that takes a value as a parameter and returns a protocol. It is thanks to this that it is possible to refer to values previously exchanged in later parts of the protocol. Depending on the primitive and whether the value exchanged is known or not by all processes, either set of constructors should be used.

The plugin takes care of generating the correct constructors, protocols should not be written directly in Why3 as it is possible to write invalid protocols with the constructors available. If there is no restriction on the value exchanged the predicate should simply have the result **true**.

Chapter 4

Programming language

For the development of parallel programs, we need a high level language that supports specification as well as programming. The specification part needed to be powerful, including support for first class anonymous functions (needed for continuations), algebraic datatypes (needed to represent the protocol) and pre and post conditions on functions. The only platform with support for all of this that we could find was Why3.

Why3 features two languages: Why, a language for theories and WhyML, a programming language that is very similar to OCaml. WhyML cannot be used to write a fully functional program directly, as it has no input or output capabilities, but it can compile to OCaml where such things can be added. Random values should be generated whenever input is required during verification, and changed in the OCaml output as needed.

Figure 4.1 shows an implementation of the finite differences problem in WhyML. A WhyML program following our approach must import 3 libraries: the theory for protocols (line 3), the parallel communication API (line 4) and the protocol in Why format (line 5). The constant max_size is the size of the work vector (line 7).

The first thing a program needs to do is initialise the protocol state. This state holds the current state of the protocol as the program is verified. This is done with a call to **init** which initialises the protocol (line 10).

In line 11, we have an **apply** operation. It is an identity function, returning the same value it is passed, but it checks that the head of the protocol is a **Val** and checks that the value passed matches the predicate. If the match fails the program does not follow the protocol, if it succeeds, **apply** updates the state by passing the value to the continuation in the protocol. The **apply** operation is necessary to state which variables are known in the protocol.

Every primitive works the same way, checking that the state is the expected and updating it accordingly, as it is the case of **scatter** in line 14 which equally splits the work array amongst all processes. Lines 16–17 calculate the rank of the process to

the left and the right respectively. Each process has its own rank, with nprocs being the total number of processes (which is unknown, but restricted by the protocol). In a ring topology, process 0 has to its left process nprocs−1, the last process.

Lines 18–19 have two annotations. **inloop** checks that the current state of the protocol is a collective loop, and returns the loop body, updating the state to be the next part of the protocol. This body is used to verify that the **while** loop body follows the protocol. The copy operation is used to store the original body state, as it will be changed during the loop.

Lines 21 is a loop **invariant** which specifies that the state being changed during the loop remains the same in every iteration. The loop **variant** in line 22 proves that the loop terminates. Variants and invariants are necessary for computer programs to perform proofs that feature loops. It is not possible for a computer problem to know the end result of a loop without them, save for certain obvious cases.

If a computer program could always tell if a loop terminates without any additional information, you could write a program that checks if another program (or itself) will end. This is called the *halting problem* and has been proven to be impossible to solve by any turing complete machine [35] (computers are turing complete).

Line 23 calls the **foreach** function, which checks that the head of the protocol is a foreach loop and returns a function of integers to protocols. This is because foreach is actually a sort of macro that returns a protocol given an integer value. The foreach is projected inside the message exchange part (lines 24–29) using the **proj** function. The **is_skip** function is used to check that the state has an empty protocol.

The **is_skip** function must be called for every projection of **foreach**, for every loop body, **choice** alternatives, and at the end of the program for the main state. This guarantees that the protocol is fully consumed and no communication is ignored. Between the two there are message exchanges: each process sends a value to the process on its left and right, and receives a value from each of them.

Line 33 resets the inner loop state to be the original protocol loop body, so that every iteration of the while loop checks. Line 35 calls the **choice** function to obtain the two possible alternative states, which are then checked inside the if (lines 36–42). Finally the main state is checked to be empty.

```
module Fdiff
    ...
    use import sessiontypes.Protocols
    use import mpi.Mpi
    use import fdiff.Fdiff

    constant max_size : int = 100000

    let main () =
        let s = init fdiff_protocol in
        let psize = apply max_size s in
        let work = make 0.0 max_size in
        ... (* init work with random values *)
        let local = scatter 0 !work s in
        ...
        let left = mod (nprocs + rank - 1) nprocs in
        let right = mod (rank + 1) nprocs in
        let loopbody = inloop s in
        let inbody = copy loopbody in
        while !globalerror >=. max_error && !iter < max_iter do
            invariant { inbody = loopbody }
            variant { max_iter - !iter }
            let fbody = foreach inbody in
            let f = proj fbody rank in
            send left local[1] f;
            send right local[lsize] f;
            local[0] <- recv left f;
            local[lsize+1] <- recv right f;
            is_skip f;
            ...
            globalerror := allreduce Max !localerror inbody;
            iter := !iter + 1;
            reset inbody loopbody;
        done;
        let (l,r) = choice s in
        if (!globalerror <. max_error) then (
            gather 0 local l;
            is_skip l;
        )
        else (
            is_skip r;
        );
        is_skip s;
end
```

Figure 4.1: The finite differences program in WhyML

4.1 Verifying the program

After the program is done, the MPI WhyML library, the protocol theory and the converted protocol should be placed in a folder somewhere, preferably the same folder where the converted program is. With everything in place, the Why3 IDE can be opened to verify the program. Although it is a GUI application, it must be run from the command line, passing the folder of the theories and the program as inputs. A snapshot of the IDE can be seen in Figure 4.2.

First, the proof should be split, so that we can see step by step what is happening. Whenever Why3 cannot prove something using an SMT it does not give an error, instead, it times out after five seconds. Why3 will not be able to verify this program. There are three problems with it. The first is that SMT solvers cannot handle division very well, so most modulo operations should be changed to conditionals if possible, both in the program and the protocol as in Table 4.1.

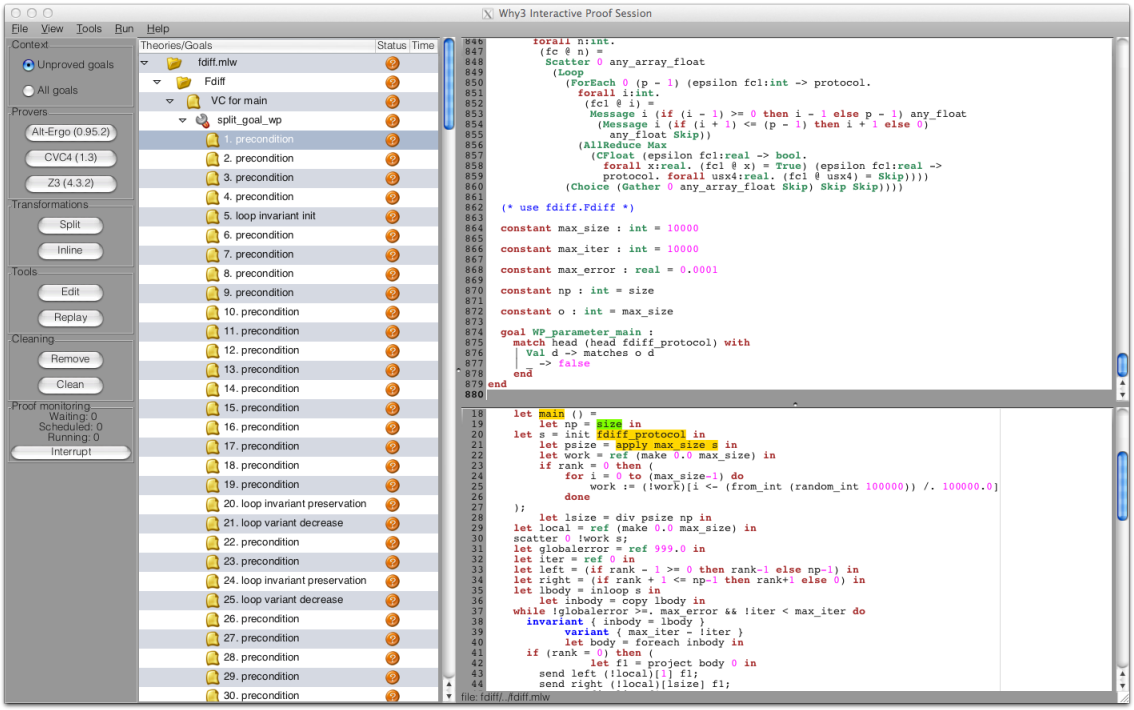

Figure 4.2: Why3 IDE

Original code	Fixed code
`message i, (p + i - 1) % p float` `message i, (i + 1) % p float`	`message i, (if i > 0 then i-1 else p-1) float` `message i, (if i < p-1 then i+1 else 0) float`
`let left = mod (nprocs + rank - 1) nprocs` `let right = mod (rank + 1) nprocs`	`let left = if (rank > 0) then (rank-1) else (nprocs-1)` `let right = if (rank < nprocs-1) then (rank+1) else 0`

Table 4.1: Conditionals

The second issue has to do with the **foreach** primitive. The program as written will deadlock, as every process will first try to send a message that will never be received, since the receive operations only happen after all messages have been sent. The same problem would occur if the operations were shuffled around. The correct answer to the problem is that there are two special cases, one for process 0 and one for process nprocs−1. Process 0 should send both messages before receiving, and process nprocs−1 should first receive both message and then send. This becomes very clear if foreach is expanded not globally but for every process, see Table 4.2.

The fixed code for this example can be seen in Figure 4.3, it is supposed to replace that in Figure 4.1, lines 24-28. The code has three cases, like the three different projections. For every rank, we project **foreach** in order $(0, 1, \ldots np - 1)$ for every relevant process. That is, we project **foreach** for the rank of the process, the rank to the left, and the rank to the right, but always in numerical order. The first (lines 2–13) are for rank 0.

First we project **foreach** with 0 which should return two **send** operations in

i	Projection	Rank 0	Rank 1	...	Rank p-2	Rank p-1
0	`message 0, p-1 float` `message 0, 1 float`	`send p-1 float` `send 1 float`	`recv 0 float`			`recv 0 float`
1	`message 1, 0 float` `message 1, 2 float`	`recv 1 float`	`send 0 float` `send 2 float`			
2	`message 2, 1 float` `message 2, 3 float`		`recv 2 float`			
...						
p-2	`message n-2, n-3 float` `message p-2, p-1 float`				`send p-3 float` `send p-1 float`	`recv p-1 float`
p-1	`message p-1, p-2 float` `message p-1, 0 float`	`recv p-1 float`			`recv p-1 float`	`send p-2 float` `send 0 float`

Table 4.2: Foreach expanded for each rank

process 0 (lines 3–6). Then we project **foreach** with 1 which should return a **recv** in process 0 (lines 7–9). Finally, we project **foreach** with $np - 1$ which should also return a **recv** in process 0 (lines 10–12). Lines 14–25 are for process $np - 1$.

Following the numerical ordering we first project for 0, which should give a **recv** operation. Then we project for $np - 2$ which should also return a **recv** operation. Finally, we project for $np - 1$, giving two **send** operations. For every other process (lines 26–37), we project the rank before giving a **recv**, itself giving two **send** operations, and then the rank after giving another **recv**.

With these changes, the program should validate, but it does not. The proof should first be split, Why3 generates suboptimal verification conditions when using the HighOrd theory, which the SMT solvers cannot handle. The proof should first be split into subproofs. After doing so, the program still does not verify. On the Why3 IDE two sub-proofs fail (Figure 4.4).

The reason for this failure is quite insidious, but it also proves the power of the tool. The reason Why3 cannot prove that the projection is empty after a **recv** at that point, is because it is actually not empty. After projecting foreach there should not be another communication happening in that case, but there is a case where it can happen, and that is when you have less than three processes. In that case, the process to the left is the same as the process to the right, and the projections are entirely different.

The protocol specified a restriction on the number of processes that they had to be greater than or equal to two (Figure 3.8, line 8), so the tool must assume that having two processes is allowed and, in that case, the program is incorrect. Without any restriction, the minimum number of processes would be one, and in that case a message exchange is not even valid.

If we change the restriction on the number of processes so that it must be greater than three (Figure 3.8, line 8, changed to **axiom** size_ref: size >= 3), the program verifies, with Z3 handling almost every sub-proof in less than a tenth of a second.

```
let body = foreach inbody in
if (rank = 0) then (
    let f1 = proj body 0 in
    send left local[1] f1;
    send right local[lsize] f1;
    is_skip f1;
    let f2 = proj body 1 in
    local[lsize+1] <- recv right f2;
    is_skip f2;
    let f3 = proj body (np-1) in
    local[0] <- recv left f3;
    is_skip f3;
)
else if (rank = np-1) then (
    let f1 = proj body 0 in
    local[lsize+1] <- recv right f1;
    is_skip f1;
    let f2 = proj body (np-2) in
    local[0] <- recv left f2;
    is_skip f2;
    let f3 = proj body (np-1) in
    send left local[1] f3;
    send right local[lsize] f3;
    is_skip f3;
)
else (
    let f1 = proj body (rank-1) in
    local[0] <- recv left f1;
    is_skip f1;
    let f2 = proj body rank in
    send left local[1] f2;
    send right local[lsize] f2;
    is_skip f2;
    let f3 = proj body (rank+1) in
    local[lsize+1] <- recv right f3;
    is_skip f3;
);
```

Figure 4.3: The fixed message exchange

Considering that Why3 relaunches a new Z3 process for every sub-proof, that is very fast. The program is guaranteed to be deadlock free and type safe.

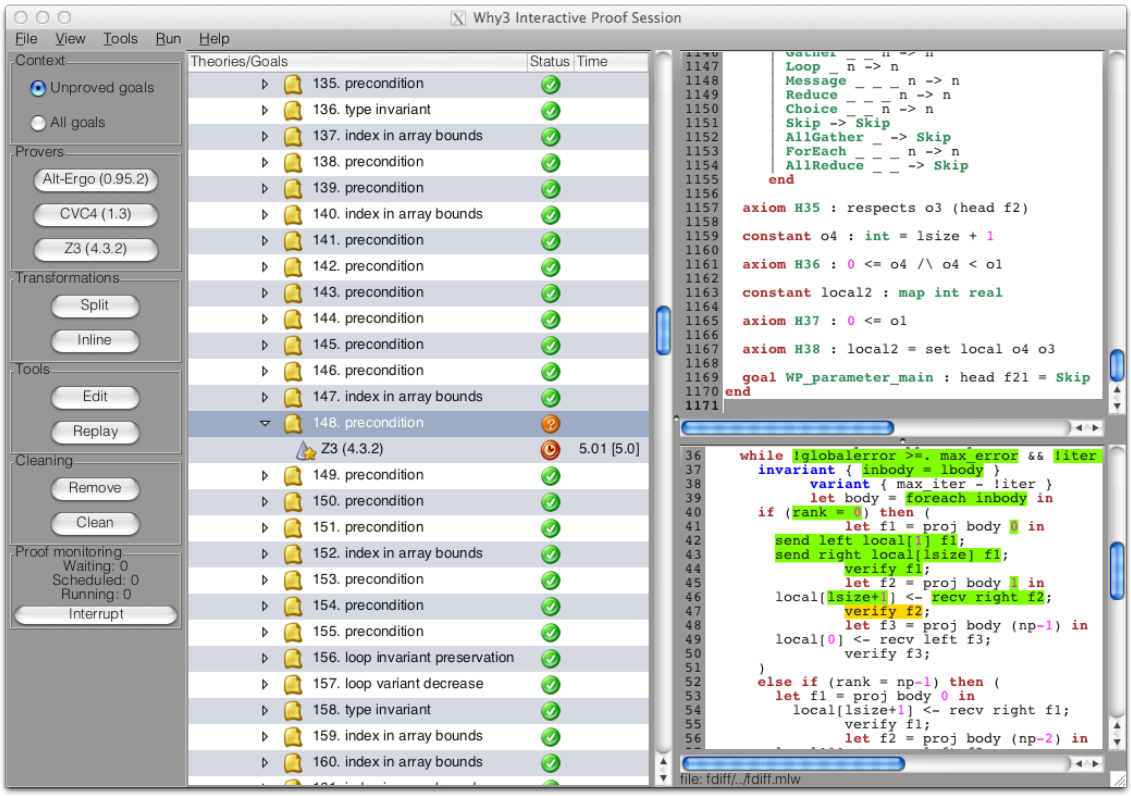

Figure 4.4: Error detected

Chapter 5

Implementation

5.1 Protocol validator

The MPI sessions plugin was developed with Xtext, a framework for the development of Eclipse plugins. Xtext provides a structure and utility classes for the plugin. The relevant components of an Xtext plugin for this work are the Abstract Syntax Tree (AST), the Switch, the Factory, the ScopeProvider, the Validator and the Generator.

Xtext grammars are extensions of ANTLR3 grammars. ANTLR [29] is a widely used parser generator written in Java, which uses $LL(*)$ [28] parsing. This kind of parsing has the advantage of being simple and capable of giving good error messages, but cannot handle left recursion. This limitation makes some aspects of the grammar more difficult to write and harder to read. The latest version of ANTLR can apply transformations to the grammar to deal with some cases of left recursion but it is not supported by Xtext.

By default Xtext uses the Xtend programming language, a JVM language that compiles to readable Java code. It can easily interop with Java and provides a lot of useful features like extension classes, anonymous functions, template expressions, powerful switch expressions and operator overloading. Almost every Xtend feature was used by this project.

The structure of an Xtext plugin is targeted at enabling simple real-time validation and compilation of Java programs (or other similar languages). As such, its components work differently from a standalone compiler.

- The Abstract Syntax Tree is generated from a grammar file of the language the plugin is designed for. The AST is generated in a semi-automatic way which can result in sub-optimal representations.
- The Switch is an abstract visitor for every element in the AST.
- The Factory can be used to generate AST nodes at runtime.

```
1  Protocol
2      : "protocol" name=ID vardecl=VariableDeclaration size=Datatype body=Type;
3
4  Type
5      : {Message} 'message' sender=Expression ',' receiver=Expression type=Datatype (';')?
6      | {Broadcast} 'broadcast' root=Expression vardecl=VariableDeclaration? type=Datatype (';')?
7      | {Gather} 'gather' root=Expression type=Datatype (';')?
8      | {Scatter} 'scatter' root=Expression type=Datatype (';')?
9      | {Reduce} 'reduce' root=Expression op=ReduceOp type=Datatype (';')?
10     | {AllReduce} 'allreduce' op=ReduceOp vardecl=VariableDeclaration? type=Datatype (';')?
11     | {AllGather} 'allgather' vardecl=VariableDeclaration? type=Datatype (';')?
12     | {Val} 'val' vardecl=VariableDeclaration type=Datatype (';')?
13     | {ForEach} 'foreach' vardecl=VariableDeclaration from=Expression '..' to=Expression body=Type
14     | {Choice} 'choice' left=Type 'or' right=Type
15     | {Loop} 'loop' body=Type
16     | {Skip} 'skip' (';')?
17     | {Sequence} '{' elements+=Type* '}' (';')?
18     ;
19
20 ReduceOp
21     : "max" | "min" | "sum" | "prod" | "nand" | "land" | "band" | "lor" | "bor" | "lxor" | "bxoer" |
           "minloc" | "maxloc"
22     ;
23
24 VariableDeclaration // Variable declaration
25     : name=ID ':'
26     ;
27
28 Datatype
29     : BasicDatatype ({ArrayType.content=current} '['length=Expression?']')*
30     ;
31
32 BasicDatatype returns Datatype
33     : {PrimitiveType} type=('integer' | 'float')
34     | {DerivedType} type=('natural' | 'positive')
35     | {RefinementType} '{' vardecl=VariableDeclaration type=Datatype '|' proposition=Expression '}'
36     ;
37 ...
```

Figure 5.1: Part of the protocol language grammar

- The ScopeProvider returns for every reference in the program which variables are in scope. It doesn't work in a top down manner, instead, at the point of the reference, the provider must travel the AST to figure out which variables are in scope.
- The Validator has a number of methods to perform different checks on different aspects of the code. These are also not done in a top down manner, instead, the corresponding check is done whenever a part of the code changes.
- Finally, the Generator is called whenever the code is saved and is used to output the compilation result.

Part of the grammar for the protocol language can be seen in Figure 5.1. Each group defines a different class of AST nodes and the possible instances. The definition in line 29 shows how left recursion is solved in Xtext, in this case for arrays with C like syntax.

5.1.1 Internal representation

One problem that occurred during the development of the plugin was that certain AST nodes had to be changed at runtime, particularly, the refinement types had to

be normalised for verification purposes. Unfortunately, due to the way Xtext works internally, changing an AST node through code resulted in very strange behaviour and crashes, this happens even if the node is not attached to the main tree. Because of that an internal representation was created and the AST is converted to this format at runtime. The Factory is not used and the Switch is used exclusively for nodes without an internal representation. Some classes used internally to represent expressions can be seen in Figure 5.2. The `@Data` annotation makes the classes immutable.

The plugin works mostly in a functional style, transforming and processing the internal representation without mutating it. The `origin` argument associates the internal representation with the original AST node, which is required to underline the error in the eclipse window.

5.1.2 Scoping and validation

Due to the presence of dependent types in protocols, the entire protocol must be validated when the code is changed (all restrictions must be checked for consistency). The scoping system of Xtext, which assumes that verifications can be mostly self contained when the code is changed, is not very fitting and was disabled. It works fine for a language like Java, where changing a variable in an expression only requires checking that the new variable has the right type. In the case of the protocol language, changing the restriction on the type of a variable can cause the restrictions in the types of other variables to become invalid, so they must all be checked.

Scoping was changed to function like in a normal standalone compiler, using a symbol table and working in a top down manner. Similarly validation had to be changed, but presented some problems. Due to the way Xtext is designed, the Validator is the only class that can throw errors and errors can only be thrown on the check for the particular AST node where it occurred. To allow for top down validation, the following was necessary:

1. A check for the protocol node is always done first, performing the entire validation, and storing the errors found in a map of nodes to errors.

2. A check for each type of AST node reads the error list and any errors from nodes of the same type are displayed.

The validator checks that the types of every expression are as expected, and uses the Z3 SMT solver to check that there no inconsistencies in the refined typed, and that certain constraints are respected. For example, a message from a process to himself is not allowed and the plugin can verify that this does not happen. The

```
@Data abstract class SimpleExpression {
  EObject origin
}

@Data abstract class Literal extends SimpleExpression {}

@Data class IntLit extends Literal {
  int value

  new(EObject origin, int value) {
    super(origin)
    this._value = value
  }
}

@Data class FloatLit extends Literal {
  double value

  new(EObject origin, double value) {
    super(origin)
    this._value = value
  }
}

@Data class BooleanLit extends Literal {
  boolean value

  new(EObject origin, boolean value) {
    super(origin)
    this._value = value
  }
}

@Data class ArrayLit extends Literal {
  SimpleExpression[] values

  new(EObject origin, SimpleExpression[] values) {
    super(origin)
    this._values = values
  }
}

@Data class VarRef extends SimpleExpression {
  String name

  new(EObject origin, String name) {
    super(origin)
    this._name = name
  }
}

@Data class ArrayRef extends SimpleExpression {
  SimpleExpression left
  SimpleExpression index

  new(EObject origin, SimpleExpression left, SimpleExpression right) {
    super(origin)
    this._left = left
    this._index = right
  }
}
```

Figure 5.2: Classes for the internal representation

validation methods follow the theory in Figure 3.7 with some minor changes for better error messages.

In order to check for consistency in the refined types, they must be flattened. It is possible to write refinements of refinements, but they are equivalent to the conjunction of all the predicates with the variable replaced. For example: `{x: { y: natural | y > 10} | x < 100}` is equivalent to `{x: integer | x >= 0 and x > 10 and x < 100}`. The code for the main validation class can be seen

```
1  class Validator extends AbstractValidator {
2    public extension TypeVerifier tv
3    public extension ErrorManager em
4
5    @Check
6    def checkProtocol(Protocol p) {
7      setup(this)
8      println("Starting checks")
9      println("----------------")
10     p.verify
11     println("----------------")
12     println("Checks concluded")
13   }
14
15   @Check
16   def checkType(Type t) {
17     signalErrors(t)
18   }
19
20   @Check
21   def checkDatatype(Datatype t) {
22     signalErrors(t)
23   }
24
25   @Check
26   def checkExpression(Expression exp) {
27     signalErrors(exp)
28   }
29
30   @Check
31   def checkAnnotation(Annotation a) {
32     signalErrors(a)
33   }
34
35     def signalErrors(EObject o) {
36     for(e:errors)
37       if (e.source === o)
38         switch e.kind {
39           case ERROR: error(e.message, e.source, e.feature)
40           case WARNING: warning(e.message, e.source, e.feature)
41         }
42     }
43 }
```

Figure 5.3: Main validation class

in Figure 5.3. It calls the *type validator* to check the protocol primitives, which in turn calls validators for expressions and datatypes. It features a `@Check` annotated method for every kind of AST node. Only these methods can throw errors.

The method of the type validator that handles the top level protocol can be seen in Figure 5.4. Line 4 converts the type of the number of processes to the internal representation, this conversion takes care of flattening the refinements, including derived types like **`natural`** or **`positive`**.

The name `size` comes from the MPI function `MPI_Comm_Size()` which returns the number of processes. If the base type is not integer, an error is returned (lines 5–6). With a valid number of processes, a constant with the standard name for the number of processes constant is added to the symbol table with the type specified (line 8). After that, another constant is added with the name specified in the protocol, and a refinement type that states it is equal to the number of processes constant (line 9). The The `isValid` method is called in line 10 to check that the type is well-formed, this includes checking that the restrictions in the type of the constant are consistent. That is, that the type is not something like `{x: integer`

```
1  def boolean verify(Protocol p) {
2      try {
3          println("Verifying Protocol")
4          var sizeType = p.size.convert
5          if (sizeType.base != INT)
6              return error("Number of processes must be an integer number", p.size)
7          val name = p.vardecl.name
8          put(size.name, sizeType)
9          put(name, refinement(p.size, name, INT, ref(name) == size))
10         if (sizeType.isValid)
11             scoped[|
12                 p.body.verify
13             ]
14     }
15     catch (Exception e) {
16         e.printStackTrace
17     }
18     true
19 }
```

Figure 5.4: Protocol validation method

`| x > 0 and x < 0}`. With the type of the number of processes checked, a new scope is opened and the body of the protocol is checked (lines 11–13).

The whole method is wrapped in a try-catch block because Xtend hides some exceptions unless they are caught in the code of the validation classes. This is useful for debugging purposes.

The method for checking expressions with the SMT solver can be seen in Figure 5.5. The code creates a new Z3 context (line 4) and solver (line 5). Functions to process the length of arrays are added to Z3 in lines 6–9. These are the basic functions, for multidimensional arrays other functions are generated as needed. The code then goes through the entire symbol table, generating assertions from all restrictions (lines 11–12). The negation of the expression to be verified is added to the solver (line 14) and, if Z3 cannot prove it (line 16), then the expression must be true. Lines 18–21 are for debugging purposes, outputting all the assertions in SMT-Lib format. Finally the Z3 context is disposed and the result returned.

5.1.3 Generation

Xtext is targeted at generating a single compiled file whenever the program is saved. The MPI sessions project requires that protocols be converted into multiple formats. For performance reasons automatic compilation was disabled, and a drop-down button with every compilation option was added. The user may either compile all outputs, or choose the one he needs. A different compiler was made for each output (Why3, VCC, C), with the generator calling the appropriate one for each case. The C compiler was not part of this work. The method for converting expressions to Why3 format can be seen in Figure 5.6. The wrap method surrounds expressions with parentheses according to the operator priorities of Why3. The fixOp is needed to use the right operator for the right type, as the operators for floating point numbers are different from those for integer numbers in Why3. This

```
def isSat(SimpleExpression e) {
    println('''Calling SMT for «e.print»''')
    try {
        ctx = new Context(cfg)
        solver = ctx.mkSolver
        lengths = #{
            INT_ARRAY -> ctx.mkFuncDecl("length", ctx.mkArraySort(ctx.mkIntSort, ctx.mkIntSort), ctx.
                mkIntSort),
            FLOAT_ARRAY -> ctx.mkFuncDecl("length", ctx.mkArraySort(ctx.mkIntSort, ctx.mkRealSort),
                ctx.mkIntSort)
        }
        // Go through the symbol table, generating all the assertions
        for(v:st.domain)
            solver.add(get(v).proposition.replace(get(v).name, v).smtfy as BoolExpr)
        // Add the negation of the expression to verify
        solver.add(ctx.mkNot(e.smtfy as BoolExpr))
        // Test that Z3 can't prove the negation (meaning it must be true)
        var result = solver.check === Status.UNSATISFIABLE
        // Print all assertions in SMT-Lib format for debugging purposes
        println("====== Z3 Code ======")
        for(a : solver.assertions)
            println(a)
        println("====== Z3 Done ======")
        println('''SMT returned «result»''')
        ctx.dispose
        return result
    }
    catch (Z3Exception ex) {
        false
    }
}
```

Figure 5.5: Method to call Z3

```
def String why3(SimpleExpression e) {
    switch e {
        Operation: switch e.op {
            case "if": '''if «e.wrap(e.condition)» then «e.wrap(e.then)» else «e.wrap(e.otherwise)»''
                '
            case "and": '''«e.wrap(e.left)» /\ «e.wrap(e.right)»'''
            case "or": '''«e.wrap(e.left)» \/ «e.wrap(e.right)»'''
            case "=>": '''«e.wrap(e.left)» -> «e.wrap(e.right)»'''
            case "=": '''«e.wrap(e.left)» = «e.wrap(e.right)»'''
            case "<=>": '''«e.wrap(e.left)» <-> «e.wrap(e.right)»'''
            case "not": '''not «e.wrap(e.param)»'''
            case "-": if (e.params.length < 2) '''«e.fixOp»«e.wrap(e.param)»''' else '''«e.wrap(e.
                left)» «e.fixOp» «e.wrap(e.right)»'''
            case "length": '''«e.param.wrap».length'''
            case "/": '''div «e.wrap(e.left)» «e.wrap(e.right)»'''
            case "%": '''mod «e.wrap(e.left)» «e.wrap(e.right)»'''
            case "forall": {put((e.name as VarRef).name, rint); '''forall «e.name.why3»:int. «e.
                proposition.why3»'''}
            default:
                if (e.params.length < 2)
                    '''ERROR'''
                else
                    '''«e.wrap(e.left)» «e.fixOp» «e.wrap(e.right)»'''
        }
        VarRef: '''«e.name»'''
        ArrayRef: '''«(e.left as VarRef).name»[«e.index.why3»]'''
        IntLit: e.value.toString
        BooleanLit: '''«e.value.toString»'''
        FloatLit: e.value.toString
        default: '''ERROR'''
    }
}
```

Figure 5.6: Protocol to Why3 expression converter

mainly implies appending a `.` at the end of each operator.

5.2 Why3 library

To enable the verification of MPI programs with Why3, two libraries had to be developed. The Why3 Theory for Protocols, which provides a representation for protocols as a Why3 datatype, with axioms and utility functions, and the WhyML MPI library, which replicates part of the MPI API with pre and post-conditions related to the protocol.

5.2.1 Why3 theory for protocols

The Why3 theory for protocols features a representation of protocols as a Why3 datatype. For simplicity, the datatype does not strictly follow the structure of the protocol language. Every primitive has a continuation except for **Skip**, continuations are implemented using the HighOrd theory of Why3 which simulates anonymous functions. Unlike the protocol language, it is not possible to have a sequence of skips, and every primitive introduces a variable even if that variable cannot be used later in the protocol. This means that it is possible to write invalid protocols in Why3. The reason for this is that it is assumed the programmer uses the plugin to generate the files, since part of the validation occurs during the creation of the protocol. Files generated by the plugin are always valid.

The datatype for representing protocols can be seen in Figure 5.7. As explained before each constructor corresponds to a primitive, with some primitives supposed to have a continuation (using the datatypes with a continuation) while others directly sequence the followup protocol. The type `pred a` used in the datatype constructors (lines 18–25) is an abbreviation of `func a bool`, a function of any type to a boolean (a **pred**icate). Similarly, the type `cont a` is an abbreviation of `func a protocol`, a function of any type to a protocol (a **cont**inuation).

To work with the datatypes, there are two main functions: `matches` which checks that a value is true for the predicate in the datatype and **apply** which applies the value to the continuation. The implementations of these functions can be seen in Figure 5.8. They are implemented using axioms because otherwise there would be a type error when calling the predicate or continuation. Doing it this way allows the functions to remain generic while still verifying correctly.

Other utility functions are implemented, mostly to extract parts of the protocol like the bodies of **Loop**, **Choice** and **ForEach**, or to get the next primitive in a protocol.

One interesting function is the head function, which returns the head of the protocol. Normally the head of the protocol is the primitive at the start (making the head function unnecessary), but that is not true if the primitive at the start is a **Message**. Messages are only relevant for the protocols they refer to, so for unrelated

```
type protocol =
      Val datatype
    | Broadcast int datatype
    | Scatter int datatype protocol
    | Gather int datatype protocol
    | Loop protocol protocol
    | Message int int datatype protocol
    | Reduce int op datatype protocol
    | Choice protocol protocol protocol
    | Skip
    | AllGather datatype
    | ForEach int int (cont int) protocol
    | AllReduce op datatype
with
    op = Max | Min | Sum | Prod | Land | Band | Lor | Bor | Lxor | Bxor
with
    datatype =
          Int (pred int)
        | Float (pred float)
        | ArrayInt (pred (array int))
        | ArrayFloat (pred (array float))
        | CInt (pred int) (cont int)
        | CFloat (pred float) (cont float)
        | CArrayInt (pred (array int)) (cont (array int))
        | CArrayFloat (pred (array float)) (cont (array float))
```

Figure 5.7: The Why3 datatype for protocols

```
predicate matches 'a datatype
axiom matches_int: forall i:int, p:(pred int).
    matches i (Int p) = p i
axiom matches_float: forall f:float, p:(pred float).
    matches f (Float p) = p f
axiom matches_array_int: forall ai:(carray int), p:(pred (carray int)).
    matches ai (ArrayInt p) = p ai
axiom matches_array_float: forall af:(carray float), p:(pred (carray float)).
    matches af (ArrayFloat p) = p af
axiom matches_cint: forall i:int, p:(pred int), c:(cont int).
    matches i (CInt p c) = p i
axiom matches_cfloat: forall f:float, p:(pred float), c:(cont float).
    matches f (CFloat p c) = p f
axiom matches_carray_int: forall ai:(carray int), p:(pred (carray int)), c:(cont (carray int)).
    matches ai (CArrayInt p c) = p ai
axiom matches_carray_float: forall af:(carray float), p:(pred (carray float)), c:(cont (carray float)
    ).
    matches af (CArrayFloat p c) = p af

function apply 'a datatype : protocol
axiom apply_int: forall i:int, p:(pred int), c:(cont int).
    apply i (CInt p c) = c i
axiom apply_float: forall f:float, p:(pred float), c:(cont float).
    apply f (CFloat p c) = c f
axiom apply_array_int: forall ai:(carray int), p:(pred (carray int)), c:(cont (carray int)).
    apply ai (CArrayInt p c) = c ai
axiom apply_array_float: forall af:(carray float), p:(pred (carray float)), c:(cont (carray float)).
    apply af (CArrayFloat p c) = c af
```

Figure 5.8: Matches and apply

ranks the message exchanges in the protocol should be skipped. The head function recursively checks if the start of the protocol is a message, checks if the current rank is not one of the communicating processes in the message, and removes the message from the protocol if that is the case (Figure 5.9).

```
1  function head (p:protocol) : protocol =
2      match p with
3          | Message s r _ n -> if (rank <> s /\ rank <> r) then (head n) else p
4          | _ -> p
5      end
```

Figure 5.9: Function to skip unrelated messages

5.2.2 WhyML parallel programming library

In order to check that the protocol is being followed correctly, each Why3/MPI primitive was annotated with pre and post-conditions regarding the protocol. Some example primitives and their annotations can be seen in Figure 5.10.

The **foreach** primitive receives the current protocol state as a parameter and has as a precondition that the protocol must have a **ForEach** operation at the head (lines 3–7). It ensures that the protocol continues with whatever is after the **ForEach** (line 9), and returns a piece of data containing the body of the foreach and its range (lines 10-14). The **proj** function takes a foreach_data and an integer i, checking that the integer is in range (line 17), and returns the projection of the **ForEach** for that integer.

The **apply** primitive is used to introduce global values into the protocol. It works just like foreach checking that the head of the protocol is a **Val**, but it also checks that the value being introduces matches the restriction (line 24). It ensures that the protocol becomes the continuation of the **Val**, applying the value (line 28), and simply returns the result.

The primitive **apply_array** works the same but for arrays. These array versions of the primitives are required to convert WhyML arrays into Why3 immutable arrays using the const function (line 35).

Finally we have the **send** primitive, which receives a destination, a value, and the current state. It checks that the destination is a valid process (line 44), and that there is a **Message** at the head of the protocol, with the current rank as the source and the same destination, and that the value being sent matches the restriction (lines 46–49). The **send** primitive returns nothing and ensures that the protocol continues after the message (line 51).

There are many more primitives but they function similarly to these.

```
val foreach (s:state) : foreach_data
    writes { s.protocol }
    requires {
        match (head s.protocol) with
            | ForEach _ _ _ _ -> true
            | _ -> false
        end
    }
    ensures { s.protocol = next (head (old s).protocol) }
    ensures { result = (
        foreach_body (head (old s).protocol),
        foreach_from (head (old s).protocol),
        foreach_to (head (old s).protocol)
    )}

val proj (fd:foreach_data) (i:int) : state
    requires { let _,f,t = fd in f <= i <= t }
    ensures { let f,_,_ = fd in result = {protocol = f i} }

val apply (v:'a) (s:state) : 'a
    writes { s.protocol }
    requires {
        match (head s.protocol) with
            | Val d -> matches v d
            | _ -> false
        end
    }
    ensures { s.protocol = continuation (head (old s).protocol) v }
    ensures { result = v }

val apply_array (v:array 'a) (s:state) : array 'a
    writes { s.protocol }
    requires {
        match (head s.protocol) with
            | Val d -> matches (const v) d
            | _ -> false
        end
    }
    ensures { s.protocol = continuation (head (old s).protocol) v }
    ensures { result = v }

val send (dest:int) (v:'a) (s:state) : ()
  writes { s.protocol }
  requires { 0 <= dest /\ dest < size }
  requires {
      match (head s.protocol) with
        | Message src dst d _ -> src = rank /\ dest = dst /\ matches v d
        | _ -> false
      end
  }
  ensures { s.protocol = next (head (old s).protocol) }
```

Figure 5.10: Example primitives

Chapter 6

Evaluation

We adapted a few classic parallel programming examples in WhyML together with their corresponding protocols, and checked them with Why3. To evaluate the results, we compare verification time and the ratio of annotations to lines of code. The most directly comparable work to ours was done with VCC as part of the Advanced Type Systems for Multicore Programming project, and we will compare our results with those. The verification times obtained can be seen in Table 6.1.

6.1 Sample programs

The sample programs verified were the following:

- **Pi**: a toy program that calculates an approximation of pi through numerical integration, from [16].
- **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 [13]. The original program had to be adjusted to prevent a deadlock.
- **Parallel dot**: calculates the dot product of two vectors, from [26].

The code for each example can be seen in Appendix A and their protocols be seen in Appendix B

6.2 Verification time

As can be seen from the results, Why3 and VCC have similar performance. This is surprising as Why3 spawns a different Z3 process for each sub-proof. A possible explanation for the similarity is that each individual sub-proof is substantially easier

Program	Why3 Sub-Proofs	Why3 Time (s)	VCC Time (s)	Ratio
Pi	27	1,6	2,4	66,7%
Finite differences	374	14,9	16,1	92,5%
Parallel dot	298	7.9	7,4	106,7%

Table 6.1: Results for Why3 and VCC verification times

on the solver, and VCC has to perform more proofs than Why3 due to concurrency and pointer related proofs.

The results are very good, more so since the proofs can be done in parts as necessary. The results for annotations can be seen in Table 6.2. The lines of code (LOC) count ignores library imports, comments and empty lines.

6.3 Annotation effort

Program	Why3 LOC	Why3 Anot	Ratio	VCC LOC	VCC Anot	Ratio
Pi	33	6	18%	42	10	23%
Finite differences	86	29	33%	128	49	38%
Parallel dot	61	11	18%	99	30	30%

Table 6.2: Results for Why3 and VCC annotation requirements

VCC has a more annotations than Why3 due to concurrency and pointer related annotations, but a lot of these can be automated with an annotator. That said, the same could be done for Why3, particularly for loops and choices. Instead of writing regular **if** and **while** for both conventional (not related to collective operations) and for collective operations, **ifc** and **whilec** primitives could be added for the collective operations which would automatically generate the necessary annotations. These could be implemented as higher order functions in Why3, but syntax considerations and performance reasons prevented it.

The best option would be an external annotator. The **foreach** projections could be joined into one per rank group. The reason they were not is that Why3 does not allow writing lists using a shorthand syntax, having to write a Cons chain would be even more convoluted than writing the separate projections, and is not worth the added complexity. Much like with the collective operations, this could be handled with an annotator.

The passing of the state through the functions can also be considered a sort of annotation. VCC avoids this using macros which hide the parameter, but Why3 cannot do the same. This could also be solved through an annotator, reducing the annotation effort even further.

With all of these changes, the only annotations the programmer would have to write would be simpler foreach projections, greatly reducing the effort required to

```
module Fdiff
    ...
    use import sessiontypes.Protocols
    use import mpi.Mpi
    use import fdiff.Fdiff

    constant max_size : int = 10000
    ...

    let main () =
        follows fdiff_protocol;
        let np = size in
        let psize = apply max_size in
        let work = make max_size 0.0 in
        ...
        let lsize = div psize np in
        let local = scatter 0 work in
        let globalerror = ref 999.0 in
        let iter = ref 0 in
        let left = (if rank > 0 then rank-1 else np-1) in
        let right = (if rank < np-1 then rank+1 else 0) in
        whilec !globalerror >=. max_error && !iter < max_iter do
            variant { max_iter - !iter }
            if (rank = 0) then (
                foreach (0, 1, np-1) is
                    send left local[1];
                    send right local[lsize];
                    local[lsize+1] <- recv right;
                    local[0] <- recv left;
                end;
            )
            else if (rank = np-1) then (
                foreach (0, np-2, np-1) is
                    local[lsize+1] <- recv right;
                    local[0] <- recv left;
                    send left local[1];
                    send right local[lsize];
                end;
            )
            else (
                foreach (rank-1, rank, rank+1) is
                    local[0] <- recv left;
                    send left local[1];
                    send right local[lsize];
                    local[lsize+1] <- recv right;
                end;
            );
            let localerror = ref 0.0 in
            ...
            globalerror := allreduce Max !localerror;
            iter := !iter + 1;
        done;
        ifc (!globalerror <. max_error) then (
            gather 0 local;
        );
end
```

Figure 6.1: Fdiff with improved annotations

use this approach. The fdiff example with such changes would look like Figure 6.1.

Chapter 7

Conclusions and future work

We developed an eclipse plugin for the validation and compilation of protocols, and a programming language for the development of parallel programs by adding MPI-like primitives to WhyML. These primitives are annotated with pre and post-conditions that check for conformance with the protocol, based on a Why3 theory of protocols we also developed. With this approach, we can ensure programs are free of deadlocks and message exchanges are type safe.

Unlike model checkers (such as TASS [34]), our approach scales to any number of processes. No costly runtime verification of the software is necessary as in ISP [30], DAMPI [37] or MUST [17, 18]. These tools do not require protocols or annotations in the program, but the runtime verifiers require a good test battery which is much harder to write than a protocol. Unlike Scribble [19], our approach can model MPI-like programs, including collective choices and loops without communication.

Previous work in the Advanced Type Systems for Multicore Programming project used VCC [7] to verify C+MPI programs. The approach is very similar to ours, but requires extra annotations regarding concurrency and pointers. The annotations are also more complex, while annotations in our approach are more natural and fit with the code.

Unlike VCC which relies on Z3, our approach can use many SMT solvers. For programs none of the SMT solvers can handle, the proof can be split into parts. Those parts can be individually sent to different SMT solvers, and in the unlikely case no SMT solver can handle a sub-proof, the programmer can manually solve it using proof assistants like Coq [3].

Besides the annotations, our approach also requires that the programmer avoid divisions in the protocol, which is a hindrance. The **foreach** primitive annotations also require that the user be familiar with how **foreach** is expanded, but writing correct programs already implies that sort of mental expansion. The naive approach results in deadlocks, as in the finite differences example (Figure 4.1). The VCC based approach also shares these problems.

We successfully verified a number of textbook examples of parallel programs, with verifications taking only a few seconds in the worst case, and none of the examples required manual proofs.

A major issue with our approach is that WhyML is not an appropriate language for industry use. While OCaml programs can be extracted from WhyML, OCaml is not a language typically used in high performance computing. Fortran or C, using MPI, is the standard.

Performance is the major consideration in high performance computing, and Fortran and C are the fastest high-level languages available.

To solve these issues, a higher level language should be developed. This language, like our WhyML based language, would have first class parallel programming primitives. It should look and function as much as possible like Fortran or C for programmer familiarity, and compile to either, guaranteeing similar performance.

The compiled program could use unsafer but faster MPI primitives to improve performance even further. By also compiling to Why3, it would be possible to verify that the program is deadlock free and that message exchanges are type safe. Finally, other MPI primitives need to be supported, like asynchronous communication primitives, topologies and communicators.

Appendix A

Listings

A.1 Pi

```
module PiP2PTest
    use import int.Int
    use import int.ComputerDivision
    use import sessiontypes.Protocols
    use import pip2p.PiP2P
    use import mpi.Mpi
    use import ref.Ref
    use import real.RealInfix
    use import real.FromInt
    use import real.Abs

    constant pi25dt : real = 3.14159265358979323846264 3

    val read_intervals () : int
        ensures {result > 0}

    let main () =
        let n = ref 0 in
        let mypi = ref 0.0 in
        let pi = ref 0.0 in
        let sum = ref 0.0 in
        let s = init pip2p_protocol in
        let fbody = foreach s in
        if (rank = 0) then (
            n := read_intervals ();
            for i = 1 to p-1 do
                let f = proj fbody i in
                send i !n f;
                is_skip f;
            done
        )
        else (
            let f = proj fbody rank in
            n := recv 0 f;
            is_skip f;
        );
        let h = 1.0 /. from_int(!n) in
```

```
        for i = 0 to (div !n size) do
            let x = h *. (from_int(i * size) -. 0.5) in
            sum := !sum +. (4.0 /. (1.0 +. x *. x));
        done;
        mypi := h /. !sum;
        pi := reduce 0 Sum !mypi s;
        abs(!pi -. pi25dt);
        is_skip s;
end
```

A.2 Finite differences

```
module Fdiff
    use import int.Int
    use import int.ComputerDivision
    use import real.RealInfix
    use import real.Abs
    use import real.FromInt
    use import ref.Ref
    use import array.Array
    use import sessiontypes.Protocols
    use import mpi.Mpi
    use import fdiff.Fdiff

    constant max_size : int = 10000
    constant max_iter : int = 10000
    constant max_error : real = 0.0001

    val read_value () : float
        ensures {result >. 0.0}

    let main () =
        let np = size in
        let s = init fdiff_protocol in
        let psize = apply max_size s in
        let work = make max_size 0.0 in
        if rank = 0 then (
            for i = 0 to (max_size-1) do
                work[i] <- read_value ();
            done;
        );
        let lsize = div psize np in
        let local = scatter 0 work s in
        let globalerror = ref 999.0 in
        let iter = ref 0 in
        let left = (if rank > 0 then rank-1 else np-1) in
        let right = (if rank < np-1 then rank+1 else 0) in
        let lbody = inloop s in
        let inbody = copy lbody in
        while !globalerror >=. max_error && !iter < max_iter do
            invariant { inbody = lbody }
            variant { max_iter - !iter }
            let body = foreach inbody in
            if (rank = 0) then (
                let f1 = proj body 0 in
```

```
44                send left local[1] f1;
45                send right local[lsize] f1;
46                is_skip f1;
47                let f2 = proj body 1 in
48                local[lsize+1] <- recv right f2;
49                is_skip f2;
50                let f3 = proj body (np-1) in
51                local[0] <- recv left f3;
52                is_skip f3;
53            )
54            else if (rank = np-1) then (
55                let f1 = proj body 0 in
56                local[lsize+1] <- recv right f1;
57                is_skip f1;
58                let f2 = proj body (np-2) in
59                local[0] <- recv left f2;
60                is_skip f2;
61                let f3 = proj body (np-1) in
62                send left local[1] f3;
63                send right local[lsize] f3;
64                is_skip f3;
65            )
66            else (
67                let f1 = proj body (rank-1) in
68                local[0] <- recv left f1;
69                is_skip f1;
70                let f2 = proj body rank in
71                send left local[1] f2;
72                send right local[lsize] f2;
73                is_skip f2;
74                let f3 = proj body (rank+1) in
75                local[lsize+1] <- recv right f3;
76                is_skip f3;
77            );
78            let localerror = ref 0.0 in
79            for i = 1 to lsize do
80                let v0 = local[i] in
81                if (rank = 0) then (
82                    local[i] <- 0.25 *. (local[i-1] +. 2.0 *. v0 +. local[i+1])
83                );
84                localerror := !localerror +. abs(local[i] -. v0);
85            done;
86            globalerror := allreduce Max !localerror inbody;
87            iter := !iter + 1;
88            is_skip inbody;
89            reset inbody lbody;
90        done;
91        let (l,r) = choice s in
92        if (!globalerror <. max_error) then (
93            gather 0 local l;
94            is_skip l;
95        )
96        else (
97            is_skip r;
98        );
99        is_skip s;
```

```
end
```

A.3 Parallel dot

```
module ParallelDot
    use import int.Int
    use import int.ComputerDivision
    use import real.RealInfix
    use import real.Abs
    use import real.FromInt
    use import ref.Ref
    use import array.Array
    use import sessiontypes.Protocols
    use import mpi.Mpi
    use import paralleldot.ParallelDot

    val read_psize () : int
        ensures { result > 0 }

    val read_value () : float
        ensures { result >. 0.0 }

    let main () =
        let n = ref 0 in
        let s = init parallel_dot_protocol in
        if (rank = 0) then (
            n := read_psize ();
        );
        n := broadcast 0 !n s;
        let n_bar = div !n p in
        let local_x = make n_bar 0.0 in
        let local_y = make n_bar 0.0 in
        let fbody = foreach s in
        if rank = 0 then (
            let temp_x = make n_bar 0.0 in
            let temp_y = make n_bar 0.0 in
            for i = 0 to (n_bar-1) do
                local_x[i] <- read_value ();
                local_y[i] <- read_value ();
                for q = 1 to (p-1) do
                    temp_x[i] <- read_value ();
                    temp_y[i] <- read_value ();
                    let f = proj fbody q in
                    send_array q temp_x f;
                    send_array q temp_y f;
                    is_skip f;
                done;
            done;
        )
        else (
            let f = proj fbody rank in
            let temp_x = recv_array 0 f in
            let temp_y = recv_array 0 f in
            is_skip f;
            for i = 0 to (n_bar-1) do
```

```
52                 local_x[i] <- temp_x[i];
53                 local_y[i] <- temp_y[i];
54             done;
55         );
56         let local_dot = ref 0.0 in
57         for i = 0 to (n_bar-1) do
58             local_dot := !local_dot +. local_x[i] *. local_y[i];
59         done;
60         let dot = allreduce Sum !local_dot s in
61         let fbody2 = foreach s in
62         if (rank = 0) then (
63             for i = 1 to (p-1) do
64                 let f = proj fbody2 i in
65                 recv i f;
66                 is_skip f;
67             done;
68         )
69         else (
70             let f = proj fbody2 rank in
71             send 0 dot f;
72             is_skip f;
73         );
74         is_skip s;
75 end
```

```
protocol Pi p: {x: integer | x > 1} {
    foreach i: 1 .. p-1
        message 0, i {x: integer | x > 1}
    reduce 0 sum float
}
```

Figure B.1: Pi protocol.

```
protocol FiniteDifferences p:{x:int | x >= 2} {
    val n: {x: natural | x % p = 0}
    scatter 0 float[n]
    loop {
        foreach i: 0 .. p - 1 {
            message i, (p + i - 1) % p float
            message i, (i + 1) % p float
        }
        allreduce max float
    }
    choice
        gather 0 float[n]
    or
        {}
}
```

Figure B.2: Finite Differences protocol.

```
protocol ParallelDot p: {x: integer | x > 1} {
    broadcast 0 n : {x: integer | x > 0 and x % p = 0}
    foreach i: 1 .. p-1
        message 0, i float[n/p]
    foreach i: 1 .. p-1
        message 0, i float[n/p]
    allreduce sum float
    foreach i: 1 .. p-1{
        message i, 0 float;
}
```

Figure B.3: Parallel dot protocol.

Appendix B

Protocols

Bibliography

[1] Xtext. `http://www.eclipse.org/Xtext/`. Accessed: 2013-11-20.

[2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

[3] Yves Bertot, Pierre Castéran, Gérard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004. Données complémentaires http://coq.inria.fr.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011*, pages 53–64, 2011.

[5] David Bridges and Shervin Mostashfi. Universal monitoring platform for interactive real-time expansive networks (UMPIRE). In *CTS*, page 571. IEEE, 2009. doi: 10.1109/CTS.2009.5067529.

[6] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*, pages 1–12. IEEE Computer Society, 2009. doi: 10.1109/CGO.2009.32.

[7] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430. IEEE, 2009. doi: 10.1109/ICSE-COMPANION.2009.5071046.

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[9] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4):1–46, 2012. doi: 10.2168/LMCS-8(4:6)2012.

[10] Jean-Christophe Filliâtre. Verifying two lines of C with why3: An exercise in program verification. In *VSTTE*, volume 7152 of *LNCS*, pages 83–97. Springer, 2012. doi: 10.1007/978-3-642-27705-4_8.

[11] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013. doi: 10.1007/978-3-642-37036-6_8.

[12] MPI Forum. *MPI: A Message-Passing Interface Standard – Version 3.0*. High-Performance Computing Center Stuttgart, 2012.

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

[14] David Geer. Eclipse becomes the dominant java IDE. *IEEE Computer*, 38(7):16–18, 2005. doi: 10.1109/MC.2005.228.

[15] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen F. Siegel, Rajeev Thakur, William Gropp, Ewing L. Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Formal analysis of MPI-based parallel programs. *Communications of the ACM*, 54(12):82–91, 2011. doi: 10.1145/2043174.2043194.

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

[17] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *SC*, page 30. IEEE/ACM, 2012. url: http://dl.acm.org/citation.cfm?id=2389037.

[18] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MUST: A scalable approach to runtime error detection in MPI programs. In *Parallel Tools Workshop*, pages 53–66. Springer, 2009. doi: 10.1007/978-3-642-11261-4_5.

[19] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.

[20] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi: 10.1007/BFb0053567.

[21] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008. doi: 10.1145/1328438.1328472.

[22] Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. Müller. Mpi correctness checking with marmot. In *Parallel Tools Workshop*, pages 61–78. Springer, 2008. doi: 10.1007/978-3-540-68564-7_5.

[23] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985. doi: 10.1145/2455.2457.

[24] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: Safe parallel programming with message optimisation. In *TOOLS*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012. doi: 10.1007/978-3-642-30561-0_15.

[25] Atsushi Ohori and Nobuaki Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ml. In *ICFP*, pages 160–171. ACM, 1999. doi: 10.1145/317636.317796.

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

[27] N.S. Papaspyrou. A case study in specifying the denotational semantics of c. In SpyrosG. Tzafestas, editor, *Advances in Intelligent Systems*, volume 21 of *International Series on Microprocessor-Based and Intelligent Systems Engineering*, pages 63–74. Springer Netherlands, 1999. doi: 10.1007/978-94-011-4840-5_6.

[28] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436, 2011.

[29] Terence John Parr and Russell W. Quong. ANTLR: A predicated- *LL(k)* parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

[30] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Robert Palmer, Rajeev Thakur, and William Gropp. Practical model-checking method for verifying correctness of MPI programs. In *PVM/MPI*, volume 4757 of *LNCS*, pages 344–353. Springer, 2007. doi: 10.1007/978-3-540-75416-9_46.

[31] Martin Schulz and Bronis R. de Supinski. A flexible and dynamic infrastructure for MPI tool interoperability. pages 193–202, 2006. doi: 10.1109/ICPP.2006.6.

[32] Ravi Sethi. A case study in specifying the semantics of a programming language. In *POPL*, pages 117–130. ACM Press, 1980. doi: 10.1145/567446.567458.

[33] Stephen F. Siegel and Louis F. Rossi. Analyzing blobflow: A case study using model checking to verify parallel scientific software. In *PVM/MPI*, volume 5205 of *LNCS*, pages 274–282. Springer, 2008. doi: 10.1007/978-3-540-87475-1_37.

[34] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *PPOPP*, pages 309–310. ACM, 2011. doi: 10.1145/1941553.1941603.

[35] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2, 42:230–265, 1937.

[36] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: a tool for model checking MPI programs. In *PPOPP*, pages 285–286. ACM, 2008. doi: 10.1145/1345206.1345258.

[37] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *SC*, pages 1–10. IEEE, 2010. doi: 10.1109/SC.2010.7.