

Thèse de doctorat

pour obtenir le grade de
Docteur de l'université de Paris-Est
discipline : Informatique
présentée et soutenue publiquement par

Jean FORTIN

le 14 octobre 2013

BSP-Why: a Tool for Deductive Verification of BSP Programs

Machine-checked semantics and application to distributed state-space algorithms

Composition du jury

Président :

Un examinateur/One examiner

Rapporteurs/Referees :

Dr. Hab. Jean-Christophe FILLIÂTRE Univ. of Paris-South/CNRS

Dr. Alan STEWART Queen's Univ. of Belfast

Pr. Sandrine BLAZY Univ. of Rennes

Examineurs/Examiners :

Pr. Bart JACOBS Katholieke Univ. of Leuven

Pr. Jan-Georg SMAUS Univ. of Toulouse (Paul Sabatier)

Pr. Roberto DI COSMO Univ. of Paris VII

Directeur/Advisor :

Pr. Élisabeth PELZ Univ. of Paris-East

Direction scientifique/Supervisor :

Dr. Hab. Frédéric GAVA Univ. of Paris-East

Notations

In this document, we note: `[[X]]` (where X is a number) for a link to the url of a software. These urls appear at the end of the document.

Remerciement/Acknowledgements

This work was carried out under the supervision of Pr. Elisabeth PELZ and Frédéric GAVA in the **Algorithmic, Complexity and Logic Laboratory (LACL)**, of the University of Paris-Est Créteil.

I would like to thank all the jury members for their presence.

(...)

Contents

1	Introduction	1
1.1	Context of the Work: Generalities and Background	1
1.1.1	Why Verify Parallel Programs?	1
1.1.2	Why is Verification of Parallel Programs Hard?	2
1.2	Machine-checked Proof of Programs	3
1.2.1	Common Methods for Having Machine-checked Correct Programs	3
1.2.2	Hoare Logic and Deductive Verification of Programs	4
1.3	Parallel Computing	6
1.3.1	Different Forms of Parallelism	6
1.3.2	The Bulk-Synchronous Parallelism	8
1.4	Verification of Parallel Programs	10
1.4.1	Generalities	10
1.4.2	Advantages of Bridging Models for Correctness of Parallel Programs	10
1.4.3	Our Approach: Deductive Verification of BSP Programs	11
1.5	Outline	14
2	Imperative BSP Programming	15
2.1	Different Kinds of BSP Operations	15
2.1.1	Generalities	15
2.1.2	Description of these Routines	17
2.2	Some Details About Some BSP Libraries	20
2.2.1	The Message Passing Interface (MPI)	20
2.2.2	BSPLIB Like Libraries for C	21
2.2.3	BSP Programming over Multi-cores and GPUs	25
2.2.4	BSP Programming in Java	27
2.2.5	Other Libraries	30
2.2.6	Which Routines for BSP-Why	34
2.3	Common Errors Introduced in BSP Programming	35
2.3.1	Common Errors and Bugs	35
2.3.2	These Errors in BSP Programs	36
2.4	Related Works	38
2.4.1	Parallel Libraries and Imperative Programming	38
2.4.2	Other Parallel Paradigms	38
3	The BSP-Why Tool	41
3.1	The BSP-Why-ML Intermediate Language	41
3.1.1	Syntax of BSP-Why	42
3.1.2	The Parallel Memory Model in Why	46
3.2	Transformation of BSP-Why-ML Programs: the Inner Working	48
3.2.1	Generalities	48
3.2.2	Identification of Sequential Blocks	49
3.2.3	Block Tree Transformation	53
3.2.4	Translation of Sequential Blocks	54
3.2.5	Translation of Logic Assertions	58
3.2.6	Dealing with Exceptions	59
3.3	Dealing with Subgroup Synchronisation	61
3.3.1	Subgroup Definition	62
3.3.2	Transformation of Programs with the Subgroup Synchronisation	64

3.4	Related Work	67
3.4.1	Other Verification Condition Generators	67
3.4.2	Concurrent (Shared Memory) Programs	67
3.4.3	Distributed and MPI Programs	68
3.4.4	Proof of BSP Programs	69
4	Case Studies	71
4.1	Simple BSP Algorithms	71
4.1.1	Parallel Prefix Reductions	71
4.1.2	Parallel Sorting Algorithm	73
4.1.3	Mechanical Proof	75
4.2	Parallel State-space Construction	75
4.2.1	Motivations and Background	75
4.2.2	Definitions and Verification of a Sequential State-space Algorithm	77
4.2.3	Verification of a Generic Distributed State-space Algorithm	79
4.2.4	Dedicated Algorithms for Security protocols	82
4.3	Related Work	87
4.3.1	Other Methods for Proving the Correctness of Model-checkers	87
4.3.2	Verification of Security Protocols	88
4.3.3	Distributed State-space Construction	89
5	Formal Semantics	91
5.1	Big-steps Semantics	92
5.1.1	Semantics Rules	92
5.1.2	Co-inductive Semantics	93
5.1.3	Adding the Subgroup Synchronisation	95
5.2	Small-steps Semantics	97
5.2.1	Semantics Rules	97
5.2.2	Co-inductive Semantics	99
5.2.3	Equivalence Between the Semantics	99
5.2.4	Adding the Subgroup Synchronisation	100
5.3	Semantics in Coq	101
5.3.1	Mechanized Semantics in Coq	101
5.3.2	Memory Model	103
5.3.3	Environment of Execution	103
5.3.4	Semantics Rules	104
5.3.5	Adding the Subgroup Synchronisation	106
5.4	Proof of the Translation	108
5.4.1	Program Correctness	108
5.4.2	Equivalence Between Elements of the Semantics	110
5.4.3	Elements of Proof in Coq	114
5.5	Related Work	116
6	Conclusion	117
6.1	Summary of the Contributions	117
6.2	Future Work and Perspectives	121
6.2.1	Close Future Work	121
6.2.2	Long Term Perspectives	122
	Bibliography	125

1 Introduction

Contents

1.1 Context of the Work: Generalities and Background	1
1.1.1 Why Verify Parallel Programs?	1
1.1.2 Why is Verification of Parallel Programs Hard?	2
1.2 Machine-checked Proof of Programs	3
1.2.1 Common Methods for Having Machine-checked Correct Programs	3
1.2.2 Hoare Logic and Deductive Verification of Programs	4
1.3 Parallel Computing	6
1.3.1 Different Forms of Parallelism	6
1.3.2 The Bulk-Synchronous Parallelism	8
1.4 Verification of Parallel Programs	10
1.4.1 Generalities	10
1.4.2 Advantages of Bridging Models for Correctness of Parallel Programs	10
1.4.3 Our Approach: Deductive Verification of BSP Programs	11
1.5 Outline	14

The main topics of this thesis are formal proofs, using methods inspired by Hoare logic, and the mechanical proof checking of parallel programs.

In this introduction, we provide a general background on program verification, and different forms of parallelism. We will then introduce two technologies that form the basis of our work: **Bulk Synchronous Parallelism (BSP)** and the **WHY** tool for deductive verification of programs.

1.1 Context of the Work: Generalities and Background

1.1.1 Why Verify Parallel Programs?

Currently, computing is evolving from sequential programming towards the parallelisation of calculations. New smartphones now include quad-core processors. Personal computers use increasing number of cores. Software now uses GPUs to perform massively parallel computations. On another scale, supercomputers reach new records of performance, allowing massively parallel programs to be executed in various domains, such as meteorology, scientific calculations, *etc.*

However, the common practices of sequential programming (code reuse and structuring) that evolved from Djisktra’s “Goto statement considered harmful” paper [87] are not the norm for parallel code. Most parallel programming tasks are still performed using low level tools and languages, leading to poor quality code and high debugging and upkeep costs [136].

Because parallel code is the norm in many areas (an attempt to list all of them would certainly fail¹ [153]), formal verification [161,171] of parallel programs is necessary. Indeed formal verification seems essential when considering the growing number of parallel architectures (GPUs, multi-cores, *etc.* [12]), the complexity of parallel architectures and the *cost* of conducting large-scale simulations (the losses due to faulty programs, unreliable results, unexpected crashing simulations, *etc.*). This is especially true when parallel programs are executed on architectures which are expensive and consume many resources.

Checking programs for correctness has always been a major issue. In critical domains, such as in aeronautics, medicine, or for military applications, a bug can be disastrous, causing the loss of billions

¹These include numerical computations, analysis of texts in biology, social sciences and humanities, symbolic calculations such as model-checking, analysis of large graphs representing, for instance, social networks.

of dollars, or even human lives². Early programmers³ were fully aware that ensuring the correctness of programs was an unavoidable issue. Given the strong heterogeneity of massively parallel architectures and their complexity, a frontal attack of the problem is a daunting task which is unlikely to materialise. Therefore, it seems more appropriate to find errors before the said programs are executed. This is called “*a priori* verification”.

1.1.2 Why is Verification of Parallel Programs Hard?

First we consider the reasons why many researchers think that verification of parallel programs is hard:

- **Added complexity**; there is no magic method for designing a task decomposition and data distribution; if the wrong decomposition or distribution is chosen, poor performance results (due to too much communication between machines) and unexpected results can appear; for example, think of distributed graph algorithms [51] that mainly make the hypothesis of a sorted distribution of edges. Compared to the verification of sequential programs, there are three main inherent difficulties in verifying parallel programs:
 1. There is not just one *flow* of execution but many (as there are multiple units of computation);
 2. The *interleaving* of instructions can introduce *data-races*⁴ and unexpected results by the presence of non-determinism. A choice should be made between forbidding these cases (by raising a warning or an exception) or computing all possible *traces* of executions and results.
 3. In distributed computations the use of a network for reading/writing values in different memories and primitives of communications are introduced. This makes the reading and the semantics of programs harder: synchronous/buffered sending can generate a *deadlock*⁵ and an unexpected result.

Tools for the verification of parallel codes exist but are not much used, mainly because they are usually inferior to their sequential counterparts. Most researchers on parallel computing are not accustomed to formal methods [135].

- **Parallel programming is error-prone** even for “Gurus” of parallel computing, some subtle errors can be introduced and only appear after a large number of program executions; for example, a lack of verification of the call of MPI’s collective operators mixed with asynchronous sending and multi-threading can crash the whole machine quickly [136]. Parallel libraries are generally well documented but even experienced programmers can misunderstand these APIs, especially when they are informally described.
- **Too few powerful abstractions are available** in the languages used for parallel programming [188]. Parallel programming languages are often very low level [136], as each communication or synchronisation operation has to be managed in detail by the programmer. Verification becomes much easier when the programmer can rely on powerful libraries to encapsulate complex behaviours — these generate less possible cases/traces of execution to prove. Work on *algorithmic skeletons* and *high-level patterns* is established but there is little knowledge of innovative parallel programming languages and tools for the verification of codes. Furthermore, it is difficult to know when you can compose parallel codes and if the results are as intended. For example, how to call a MPI code within an OPENMP [[1]] one?
- **Parallel code testing is hard**. Many libraries (*e.g.* MPI, OPENMP, *etc.*) allow you to test your parallel programs on your personal computer by emulating parallelism using OS processes; but not all interleavings can be tested in this way and some combinations that generate a deadlock may be missed; only formal verification gives this assurance. Testing is a first step and without test try, it can be hard to know “where to go” in a proof of correctness.

We conclude that **High Performance Computing** (HPC) programming is hard because every detail of parallelism is left to the programmer. Large fractions of code are technical and program complexity becomes excessive, resulting in a truly incessant debugging and proof. Because HPC is widely used in research in science and engineering (these themes are present in HPC conferences), for the long term viability of this area it is absolutely essential that we have verification tools to help application developers gain confidence in their software.

²Two famous and dramatic examples are the first Ariane-V rocket, which was destructed following a software bug and the code controlling the Therac-25 radiation therapy machine, which was directly responsible for some patient deaths.

³In 1949 already, Turing writes an article in which he formally proves an algorithm.

⁴When at least two computation units access together to a common resource.

⁵When at least two computation units are each waiting for the other to finish.

In the last decades, methods have been developed to study the correctness of sequential programs. For the verification of parallel programs it should be possible to avoid starting again. A natural question immediately follows: how easy is it to adapt the methods developed to ensure the safety of sequential programs for a parallel environment? In the following, we will summarize some of the methods that are frequently used to ensure that programs are error-free, focusing on machine-checked approaches. Then we will describe more precisely what we mean by parallel computing and which model we will use. Finally pre-existing tools for verification of parallel codes (algorithms and programs) will be presented.

1.2 Machine-checked Proof of Programs

Developing tools and methods to find logical *bugs* or to help programmers have “from the start” (or at a given time of the development) correct programs is an old but still pertinent area of research. For sequential (imperative, functional, object or real-time) algorithms and programs, many methods (and their associated tools) exist: programming through theorem proving, B method, model-checking, test, deductive verification, abstract interpretation, algebraic rules, *etc.*

A common and well accepted approach is conducting machine-checked proofs. These are formal derivations in some given formal system, assuming that the underlying logic is coherent. Humans, even great mathematicians, are fallible. The use of a formal system for doing proofs on a machine forbids false proofs. For example, special cases that seem *a priori* trivially true (or very similar to other cases and therefore unattractive) are often forgotten about but later revealed to be false. The use of a formal system gives thus greater confidence in the results. Parallel programming tends to be much more complex than sequential computing. This means that rigorous program development techniques are all the more necessary.

1.2.1 Common Methods for Having Machine-checked Correct Programs

There are different methods and tools for proving the correctness of programs and systems or for generating machine-checked programs. Without being exhaustive, we present here some of the best known methods.

(a) Testing, Debugging and Model-checking

This first class of methods we consider is generally not used for proving the correctness of programs but rather for finding bugs. Software testing [174] involves running a program for different scenarios (inputs of the program) and comparing the traces of execution against a formal specification of expected results. Scenarios can be automatically extracted by a testing tool from the program (for example, when there is a division to test the program against a division by zero) or given by hand by the programmer. Some languages such as Python have a specific syntax for describing the expected results of a function (which is also used to document the programs). Testing can not prove the correctness of a given program but it is very useful for finding most of the small careless mistakes of programmers.

Verification through *model checking* [61] consists in: (1) defining a formal model of the system to be analysed; (2) expressing expected properties formally (generally in a temporal logic); (3) using automated tools to check whether the model fulfils the properties. This is done by calculating the state-space of the model representing all the different configurations of the execution of the program. The state-space construction problem is that of computing the explicit representation of a given model from the implicit one. In most cases, this space is constructed by exploring all the states reachable through a successor function from an initial one. Building a state-space is the first step in verification by model-checking of logical properties. Temporal logics are mainly used to compare all traces of execution against a given formal specification. Systems that are to be analysed are mostly concurrent communicated “agents”, *i.e.* small programs. Model-checking can only find unexpected results for a given finite number of agents but not for any number of agents. However, model-checkers use different techniques to reduce the state-space (or the time to compute it) and to check the logical formula. That allows the checking of more agents or more complicated systems (since less memory and time are needed in the verification).

(b) Static Analysis, Abstract Interpretation and Symbolic Execution

This class of automatic methods is based on the analysis of a (full) source code, sometimes on the executable code (after compilation) and therefore performed *without* truly executing the code. These

methods do not prove the “full” correctness of programs but rather prove that some properties hold during the entire execution of the programs. Those properties can be safe execution, absence of deadlock or (integer/buffer/...) overflow, termination of the computation, security issues, *etc.*

Static program analysers are mainly based on some kind of type systems. If the source code is valid for the type system then some simple properties hold during the execution of the program. Type systems give some (polymorphic) types to elementary objects and sub-expressions of the code. Then, some specific rules are used to compose these types. ML, HASKELL, JAVA, ADA (and many others) are common examples of typed languages. There are other kinds of specific analysers which extend the traditional type systems of programming languages.

Abstract interpretation is a generalisation of type systems, which uses Galois connections and monotonic functions over ordered sets [70]. It can be viewed as a partial execution of a computer program which gains information about its semantics by running an approximation of the execution. Approximation is used to allow for vague answers to questions; this simplifies problems making them amenable to automatic solutions. One crucial requirement is to add sufficient vagueness so as to make problems manageable while still retaining enough precision for answering the important questions (such as “will the program crash?”). Abstract interpretation thus works on abstract domains of the data of the analysed program. When one chooses an abstract domain, one typically has to strike a balance between keeping fine-grained relationships, and high computational costs. Different kind of abstractions exist such as convex polyhedra, congruence relations, *etc.*

Symbolic execution is a special case of abstract interpretation and works by tracking symbolic rather than actual values of the code and is used to reason about all the inputs that take the same path through a program. Mainly, true values are substituted by symbolic variables and a constraint solver is used to test assertions in the program or to test if a given property holds during a symbolic execution. For branching constructs, all feasible paths run symbolically. But the number of feasible paths in a program grows exponentially with an increase in program size. Thus the method is inappropriate for large programs.

(c) Programming Through Theorem Proving and B Method

Machine-checked proofs (such as mechanized semantics of programming languages) are realised using theorem provers. A theorem is given to the system together with some tactics for solving the theorem using the logic rules of the system. Some theorem provers are fully automatic while others have special tactics to check automatically some simple goals. Several theorem provers (such as COQ) are based on the Curry-Howard isomorphism which claims that “a proof is a program, the formula it proves is a type for the program”. In this way, proving a property is as giving a (functional) program that computes the property. For example, the property “for any list of objects, there exist a list with the same objects but arranged in an ordered fashion” corresponds to giving a function for sorting lists. Programming through theorem proving [22] is thus proving that the formal specification (a property) is valid and then extracting the program from the proof. The resulting program would be automatically correct (assuming a correct extraction) since it is taken from the mechanized proof.

In contrast, top-down design of a program starts from an abstract formal specification. The specification is then refined into a more concrete form that incorporates design decisions reflecting a commitment to algorithmic and data representation. An executable code can be then extracted. The B method yields such a refinement technique of abstract machines. Correctness of abstract machines, mainly with logical invariants of the computations, are performed using a kind of Hoare logic (described below).

1.2.2 Hoare Logic and Deductive Verification of Programs

Hoare triples. In [160], Hoare introduces the use of triples to describe the execution of a piece of code. A Hoare triple is written in the form $\{P\} e \{Q\}$, where P and Q are predicates, and e is a program expression (a command). P is called the *precondition*, and Q the *postcondition*. The meaning of a triple in Hoare logic is that when executed from an environment that satisfies the predicate P , the program e will result in an environment satisfying the predicate Q .

In [160], Hoare gives a set of rules for reasoning about the instructions of a minimalist programming language. This allows valid Hoare triples to be built for a whole program incrementally. The rules are given in figure 1.1. A few points need to be underlined:

- In the assignment rule, $P[E/x]$ denotes the expression P where all the instances of x have been replaced with E . It is important to understand that this rule is only valid when the language does not contain aliases to the same value. For instance, the Hoare triple $\{y = 3\} x := 2 \{y = 3\}$ is

$$\begin{array}{c}
\frac{}{\{P\} \text{ skip } \{P\}} \quad \frac{}{\{P[E/x]\} x := E \{P\}} \\
\frac{\{P\} e1 \{Q\} \quad \{Q\} e2 \{R\}}{\{P\} e1; e2 \{Q\}} \\
\frac{\{P \wedge B\} e1 \{Q\} \quad \{P \wedge (\neg B)\} e2 \{Q\}}{\{P\} \text{ if } B \text{ then } e1 \text{ else } e2 \{Q\}} \\
\frac{}{\{P \wedge B\} e \{P\}} \\
\frac{}{\{P\} \text{ while } B \text{ do } e \text{ done } \{P \wedge (\neg B)\}} \\
\frac{P' \rightarrow P \quad \{P\} e \{Q\} \quad Q \rightarrow Q'}{\{P'\} e \{Q'\}}
\end{array}$$

Figure 1.1. Inference rules for the Hoare logic

valid according to the Hoare logic, but would be invalid in a program where x and y designate the same value. This limitation also holds for methods that are based on the Hoare logic, such as the WHY verification condition generator. We will discuss more about such consequences when we introduce WHY.

- The **while** rule introduces the notion of a loop *invariant*, P , that must remain true at the start and end of each iteration of the loop. We can state that P is true at the end of the loop, assuming that it was true initially. Hence the given Hoare triple for while.
- The last rule is used to strengthen a precondition, or weaken a postcondition.

Total correctness. The rules given by Hoare in [160] have a limitation, discussed by the author in the original article. Even if Hoare triples are proved correct, there is still no guarantee that the program terminates. It is possible that a **while** statement will loop indefinitely. For this reason, a Hoare triple $\{P\} e \{Q\}$ should be read as: *provided that e successfully terminates, the result of its execution is as described by Q* . Classical Hoare logic only proves *partial* correctness.

However, it is possible to enrich the declaration of the loop rule, so that it includes the definition of a *variant*, a quantity that is decreasing according to a well-founded order. The new rule for a loop statement is written as follows:

$$\frac{wf(<) \quad \{P \wedge B \wedge v = v_0\} e \{P \wedge v < v_0\}}{\{P\} \text{ while } B \text{ do } e \text{ done } \{P \wedge (\neg B)\}}$$

Assuming that the relation $<$ is well-founded, this means that in every iteration of the loop the quantity v , called the variant, decreases. A finite chain is formed, ensuring the termination of the loop.

Weakest Precondition. Hoare logic allows us to write triples that do not carry meaningful information. For instance, the triple $\{false\} e \{P\}$ is always true, but very rarely relevant. Dijkstra's *weakest precondition* (wp) calculus [88] can be understood to give the most general precondition, for a program and its postcondition. As such, $\{wp(e, P)\} e \{P\}$ is always true, and for all Q such as $\{Q\} e \{P\}$ holds, we have $Q \rightarrow wp(e, P)$. Most modern verification condition generators (VCG) are based on the Weakest Precondition calculus.

One of the main advantages of the weakest precondition calculus, compared to the basic Hoare logic, is that it is more easily adapted to the generation of mechanical proofs of programs. The wp-calculus can be computed by the formula given in figure 1.2. However, in Hoare logic, to prove a simple triple such as $\{P\} e1; e2 \{Q\}$, one has to devise a middle predicate that will allow the use of the induction rule. This can usually be done by a human; for a mechanical proof, the wp-calculus is more appropriate.

$$\begin{aligned}
wp(skip, R) &= R \\
wp(x := E, R) &= R[E/x] \\
wp(e_1; e_2, R) &= wp(e_1, wp(e_2, R)) \\
wp(\text{if } B \text{ then } e_1 \text{ else } e_2, R) &= (B \rightarrow wp(e_1, R)) \wedge (\neg B \rightarrow wp(e_2, R)) \\
wp(\text{while } B \text{ do } e \text{ done}, R) &= I \\
&\quad \wedge \forall \omega, (B \wedge I \rightarrow wp(e, I)) \\
&\quad \wedge \forall \omega, (\neg B \wedge I \rightarrow R)
\end{aligned}$$

I is the loop invariant, ω is the set of modified variables

Figure 1.2. Typical rules of the weakest precondition calculus.

1.3 Parallel Computing

1.3.1 Different Forms of Parallelism

(a) Flynn's Classification

A parallel computer or multi-processor system is a computer utilizing more than one processor (or unit of computation). It is common to classify parallel computers by distinguishing them by how processors access the system's main memory. Memory access heavily influences the usage and programming of a system. Flynn defines a classification of computer architectures, based upon the number of concurrent instructions (or controls) and data streams available in the architecture [92, 107]. Two major classes of distributed memory computers can be distinguished: distributed memory and shared memory systems. Flynn's classification is as follow:

	Single Instruction	Multiple Instructions
Single data	SISD	MISD
Multiple data	SIMD	MIMD

where:

- **SISD** is “**S**ingle **I**nstruction, **S**ingle **D**ata stream” that is a sequential machine;
- **SIMD** is “**S**ingle **I**nstruction, **M**ultiple **D**ata streams” that is mostly array processors and GPU;
- **MISD** is “**M**ultiple **I**nstruction, **S**ingle **D**ata stream” that is pipeline of data (pipe skeleton);
- **MIMD** is “**M**ultiple **I**nstruction, **M**ultiple **D**ata streams” that is clusters of CPUs.

Distributed memory **N**o **R**emote **M**emory **A**ccess (**NORMA**) computers do not have any special hardware to support access to another node's local memory directly. The nodes are only connected through a computer network. Processors obtain data from remote memory by exchanging messages over this network between processes on the requesting and the supplying node. Computers in this class are sometimes also called **N**etwork **O**f **W**orkstations (**NOW**). In case of shared memory systems, **R**emote **M**emory **A**ccess (**RMA**) computers allow access to remote memory via specialized operations implemented by hardware. However the hardware does not provide a global address space.

The major advantage of distributed memory systems is their ability to scale to a very large number of nodes. In contrast, a shared memory architecture provides (in hardware) a global address space, *i.e.* all memory locations can be accessed via load and store operations. Such a system is much easier to program. Shared memory systems can only be scaled to moderate numbers of processors. We concentrate on the **M**IND streams model, and especially on the so-called **S**ingle **P**rogram **M**ultiple **D**ata (**SPMD**) model, which is widely used for programming parallel computers. In the **SPMD** model, the same program runs on each processor but it computes on different parts of the data (distributed over the processors).

There are two main programming models: message passing and shared memory. These offer different features for implementing applications parallelized by domain decomposition. Shared memory allows multiple processes to read and write data from the same location. In the message passing model each process can send messages to the other processes.

(b) Shared Memory Model

In the shared memory model, a program starts as a single process (known as a master thread) which executes on a single processor. The programmer designates parallel regions in the program. When the

master thread reaches a parallel region, a fork operation is executed. This creates a team of threads, which execute the parallel region on multiple processors. At the end of the parallel region, a join operation terminates the fork, leaving only the master thread to continue on a single processor. Now, we give some well known examples of libraries for the shared memory programming model.

OPENMP [21,53] is a directive-based programming interface for the shared memory programming model. It consists of a set of directives and runtime routines for Fortran, and a corresponding set of pragmas for C and C++ (1998). Directives are special comments that are interpreted by the compiler. Directives have the advantage that the code is still sequential and can be executed on sequential machines (by ignoring the directives/pragmas). There is no need to maintain separate sequential and parallel versions.

Intel Threading Building Blocks (Intel TBB [[2]]) library [184] which is a library in C++ that supports scalable parallel programming. The evaluation is specifically for pipeline applications which are implemented using the filter and pipeline class provided by the library. Various features of the library which help during pipeline application development are evaluated. Different applications are developed using the library and are evaluated in terms of their usability and expressibility [172].

Recently Graphic Processing Units (GPU) have been used in HPC, due to their tremendous computing power, favorable cost effectiveness, and energy efficiency. The Compute Unified Device Architecture (CUDA) [[3]] [219] has enabled graphics processors to be explicitly programmed as general-purpose shared-memory multi-core processors with a high level of parallelism. In recent years, graphics processing units (GPUs) have advanced from being specialized fixed-functions to being programmable and parallel computing devices. With the introduction of CUDA, GPUs are no longer exclusively programmed using graphics APIs. In CUDA, a GPU can be exposed to the programmer as a set of general-purpose shared-memory SIMD multi-core processors. The number of threads that can be executed in parallel on such devices is currently in the order of hundreds and is expected to multiply soon. Many applications that are not yet able to achieve satisfactory performance on CPUs can benefit from the massive parallelism provided by such devices.

Some take the view that models (generally concurrent languages) based on shared memory are easier to program because they provide an abstraction for a single, shared address space. While shared memory reduces the need for placement, it created a need to control simultaneous access to the same location. This requires either careful crafting of programs, or expensive lock management. Implementing shared-memory abstractions requires a large fraction of a computer's resources to be devoted to communication and the maintenance of coherence. Worse, the technology required to provide the abstraction is not at all a commodity nature, and hence even more expensive [188].

(c) Distributed Memory Model and the Message Passing Interface (MPI)

The message passing model is based on a set of processes with private data structures. Processes communicate by exchanging messages with special send and receive operations. It is widely used for programming distributed memory machines but it can be also used on shared memory computers.

The most popular message passing technology is the Message Passing Interface (MPI) [[4]] [259], a message passing library for C and FORTRAN. MPI is an industry standard and is implemented on a wide range of parallel computers. Details of the underlying network protocols and infrastructure are hidden from the programmer. This helps achieve portability while enabling programmers to focus on writing parallel code rather than networking code.

It includes routines for point-to-point communication, collective communication, one-sided communication, parallel IO, and dynamic task creation. Later in this thesis more details about the sub-part of MPI (that interests us) will be given.

(d) Hybrid Architectures

In addition to shared-memory and distributed models, modern parallel architectures now provide hybrid models: we have now clusters of clusters of ... of multi-processors of multi-cores with GPUs. These architectures need a new way of programming.

Clusters of Multi-processors/cores. Clusters have become the de-facto standard in parallel processing due to their high performance to price ratio. Symmetric MultiProcessing (SMP) clusters are also gaining popularity, mainly under the assumption of fast interconnection networks and memory buses. SMP clusters can be thought of as an hierarchical two-level parallel architecture, since they combine features of shared and distributed memory machines. As a consequence, there is interest in hybrid parallel programming models, *e.g.* models that perform communication both through message passing and memory access.

Intuitively, a paradigm that uses memory access for intra-node communication and message passing for internode communication seems to exploit better the characteristics of SMP clusters [91]. Hybrid models have already been applied to scientific applications [154], including probabilistic model-checking [147].

Skeleton Paradigm. Many parallel algorithms can be characterised and classified by their adherence to a small number of generic patterns of computation — farm, pipe, map, reduce, *etc.* Skeletal programming proposes that such patterns be abstracted and provided as a programmer’s toolkit with specifications which transcend architectural variations but implementations which recognise them to enhance performance [64, 97, 133, 137]. The core principle of skeletal programming is conceptually straightforward. Its simplicity is its strength. There are two kinds of skeletons [182]:

1. **Flow skeletons;** These manipulate a “flow” of data by applying functions; All data flows (asynchronously) from machines to machines until a global condition occurs; For example, the OCAML P3L [67] [[5]] skeletons language (P3L [10]’s set of skeletons for OCAML);
2. **Data-parallel skeletons;** They mainly manipulate distributed data-structures such as arrays, lists, trees, *etc.* by applying functions to the data.

Skeletons are patterns of parallel computations [133, 235]. They can be seen as high-order functions that provide parallelism. They fall into the category of functional extensions [183] — following their semantics [1]. Most skeleton libraries extend a language (mostly JAVA, HASKELL, ML, C/C++) to provide high level primitives.

Currently, the best known library is Google’s MAPREDUCE [80]. It is a framework to process embarrassingly parallel problems across huge datasets — originally for the page-ranking algorithm. Different implementations for JAVA or C exist. But only two skeletons are provided which limits expressiveness.

Many authors have provided skeleton libraries for different language. For example, [98] provides a set of flow skeletons for C++. Templates are used to provide an efficient compilation of programs: for each program, a graph of communicating processes is generated which is then transformed into a classical MPI program. Templates have also been used in [150]. For JAVA, many libraries exist such as [3, 194] and [2]. The latter has been extended for multi-core architectures [60]. A study of how to type JAVA’s skeletons is given in [48]. The authors note that some libraries of data-flow skeletons use a unique generic type for data (even if it is an integer or a string), which can cause a clash of the JVM. They explain how to avoid this problem, using a simple type system. There is also the work of [64] which describes how to add skeletons in MPI (the *eskel* library), as well as giving some experiments. It also gives convincing and pragmatic arguments to mix message passing libraries and skeleton programming. Some benchmarks of an OCAML implementation of data-flow skeletons for a numerical problem are described in [62].

Skeletons programs can be optimised to run on hybrid architectures (even if currently, this has not been really done).

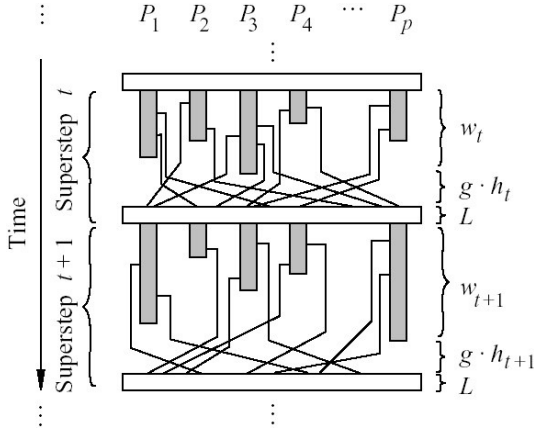
1.3.2 The Bulk-Synchronous Parallelism

(a) The Model

The **Bulk-Synchronous Parallel (BSP)** model is a *bridging model* [204] between abstract execution and concrete parallel systems and was introduced by Valiant in [278] and developed by McColl *et al.* [258]. Its initial goal is to have a portable and scalable performance prediction for parallel programs. Without dealing with low-level details of parallel architectures, the programmer can focus on algorithm design — complexity, correctness, *etc.* An introduction to its “philosophy” can be found in [258] and a complete book of BSP numerical algorithms is [25]. A recent presentation on BSP can be found in [177].

A BSP computer has three components: (1) a homogeneous set of uniform processor-memory pairs; (2) a communication network allowing inter processor delivery of messages; (3) a global synchronisation unit which executes collective requests for a *synchronisation barrier*.

A wide range of actual architectures can be seen as BSP computers. For example shared memory machines could be used so that each processor only accesses a sub-part of the shared memory (which is then “private”) and communications could be performed using a dedicated part of the shared memory. Moreover the synchronisation unit is very rarely a hardware entity but rather a software component [158]. Supercomputers, clusters of PCs [25], multi-cores [131, 287] and GPUs [165], *etc.* can be thus considered as BSP computers. There are different libraries and languages for BSP programming. We will describe them in the next chapter.



A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive disjoint phases (see left): (1) each processor only uses its local data to perform sequential computations and to request data transfers to/from other nodes; (2) the network delivers the requested data; (3) a global (collective) synchronisation barrier occurs, making the transferred data available for the next super-step.

The performance of the BSP machine is characterised by 4 parameters: (1) the local processing speed r ; (2) the number of processor p ; (3) the time L required for a barrier; (4) and the time g for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word.

The network can deliver an h -relation (every processor receives/sends at most h words) in time $g \times h$. To accurately estimate the execution time of a BSP program these 4 parameters could be benchmarked [25]. The execution time (cost) of a super-step s is the sum of the maximal of the local processing, the data delivery and the global synchronisation times. It is expressed by the following formula:

$$\text{Cost}(s) = \max_{0 \leq i < p} w_i^s + \max_{0 \leq i < p} h_i^s \times g + L$$

where w_i^s = local processing time on processor i during superstep s and h_i^s is the maximal number of words transmitted or received by processor i during superstep s . The total cost (execution time) of a BSP program is the sum of its super-steps's costs.

(b) Advantages and Disadvantages

It is stated in [81] that “A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures between the late eighties and the time from the mid-nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain”.

This *structured* model of parallelism enforces a strict separation of communication and computation: during a super-step, no communication between the processors is allowed; only at a synchronisation barrier can information be exchanged⁶. This execution policy has two main advantages. First, it removes non-determinism and guarantees the absence of deadlocks. This is the most visible aspect of a parallel model that shifts the responsibility for timing and synchronisation issues from the applications to the communications library⁷. Second, it allows for an accurate model of performance prediction based on the throughput and latency of the interconnection network, and on the speed of processors. This performance prediction model can even be used at runtime to dynamically make decisions, for instance choose whether to communicate in order to re-balance data, or to continue an unbalanced computation.

However, on many basic distributed architectures, barriers are often expensive when the number of processors dramatically increases — *e.g.* more than 10 000. But proprietary architectures and future shared memory architecture developments (such as multi-cores and GPUs) may make barriers much faster. Barriers have a number of advantages: it is harder to introduce livelock, since barriers do not create circular data dependencies. Barriers also permit novel forms of fault tolerance [258].

The BSP model considers communication actions *en masse*. This is less flexible than asynchronous messages, but easier to debug, since there are many simultaneous communication actions in a parallel program, and their interactions are typically complex. Bulk sending offers better performance since, from an implementation point of view, grouping communication together in a separate program phase permits a global optimisation of the data exchange by the communications library. Moreover it is easy to measure during the execution of a BSP program, the time spent to communicate and to synchronise by just comparing the time before and after the primitive of synchronisation. This is mainly use to compare different algorithms.

Since BSP programs are portable and have a cost estimate which may model power consumption, they have the potential to be implemented in the cloud [9]: we can imagine a scheduler server that distributes a BSP program depending on its BSP cost so as to optimise power consumption and network traffic.

⁶For performances issue, a BSP library can send messages during the computation phase of a super-step, but it is hidden to programmers.

⁷BSP libraries are generally implemented using MPI [259] or low level routines of the given specifics architectures.

As with other low/high level design decisions, the applications programmer gains simplicity but gives up some flexibility and performance. In fact, the performance issue is not as simple as it seems: while a skilled programmer can in principle always produce more efficient code with a low-level tool (be it message passing or assembly language), it is not at all clear that a program, produced in a finite amount of time, can actually realise that theoretical advantage, especially when the program is to be used on a wide range of machines [136, 188].

One last advantage of BSP is that it greatly facilitates debugging. The computations going on during a super-step are completely independent and can thus be debugged independently. This facility will be used here to formally prove the correctness of our algorithms. This simplicity (for programming, debugging and proof) combined with its efficiency makes it a good framework for teaching.

The runtime system of BSP knows precisely which computations are independent. In an asynchronous message-passing system as MPI, the independent sections tend to be smaller, and identifying them is much harder. But, using BSP, programmers should be aware that some parallel patterns are not really BSP friendly, *e.g.* in pipeline and master/slave paradigm (also known as farm of processes). It is still possible to have reasonably efficient BSP programs for these schemes as we will see later. Some parallel computations and optimisations are unsuited for BSP [94]. This is the disadvantage of all restricted models of computations.

Note that there are other (bridging) models of parallel computations. The best known is “logp” [24, 73] but there are also D-BSP, E-BSP, CLUMPS, QSM, *etc.* A comprehensive list can be found in [46]. Except for logp, which has also been used to optimise some programs [59, 210] and MPI’s collective operations implementations [234], only BSP is widely used. The BSP model has also been used with success in a wide variety of problems such scientific computing [25, 26, 84, 125, 127, 164, 272], parallel data-structure [126, 145], genetic algorithms [42] and genetic programming [90], neural network [240], parallel data-bases [13–15], constraints solver [141], graphs [51, 100, 186, 272], geometry [82], string search [84, 99, 181], implementation of tree skeletons [212], search engine (queries to textual databases) [68], 3-SAT solver [35], algebra [38, 242, 243], discrete event simulation [45], bio-computing [166, 181], scheduling threads [76], multi-agent services [56], image processing [75, 165], *etc.* BSP was adopted because it represents a common model for writing successful parallel programs that exhibit phase-based computational behaviour [153].

1.4 Verification of Parallel Programs

1.4.1 Generalities

The correctness of parallel programs is of paramount importance, especially considering the growing number of parallel architecture (GPUs, multi-cores, *etc.*) and the cost of conducting large-scale simulations — the losses due to faulty programs, unreliable results or crashing simulations is enormous. Also, avoiding deadlocks is insufficient to ensure that the programs will not crash. We need to check buffer and integer overflows (safety) and liveness. For critical systems and libraries, one needs to ensure that results are as intended. Formal verification tools that display parallel concepts are useful for program understanding and debugging. With multi-cores, GPUs and peta-scale revolutions looming, such tools are long overdue.

Given the strong heterogeneity of these massively parallel architectures and their complexity, a frontal attack of the problem of verification of parallel programs is a daunting task that is unlikely to materialize. An approach would be to consider well-defined subsets that include interesting structural properties [47]. In fact, many programs are not as unstructured as they appear: it is the skeletons and BSP main idea.

1.4.2 Advantages of Bridging Models for Correctness of Parallel Programs

One inherent difficulty in the development of scientific software is the reconciliation of the requirements that code be both correct and efficient. Today, the number of parallel computation models and languages may exceed the number of different architectures available. Most are inadequate because they make it hard to achieve portability, correctness and performance. Such systems are prone to deadlocks. Furthermore, the performance of such programs is typically difficult to predict because of the interaction of large number of individual data transfers. Placing too much emphasis on correctness may result in an abstract, but inefficient, programming model. Alternatively, striving for optimal efficiency can run the risk of comprising software correctness and can result in the employment of architecture-specific programming models. A common (but still insufficient) solution is the use of a bridging model.

In computer science, a bridging model is an abstract model of a computer which provides a conceptual bridge between the physical implementation of the machine and the abstraction available to a programmer

of that machine; in other words, it is intended to provide a common level of understanding between hardware and software engineers. A bridging model provides software developers with an attractive escape route from the world of architecture-dependent parallel software. A successful bridging model is one which can be efficiently implemented in reality and efficiently targeted by programmers; in particular, it should be possible for a compiler to produce good code from a typical high-level language. The term was introduced in Leslie Valiant’s 1990 paper “A Bridging Model for Parallel Computation” [278], which argued that the strength of the von Neumann model was largely responsible for the success of computing. The paper goes on to develop the BSP model as an analogous model for parallel computing.

There exist other tentatives of bridging models such as Logp [24, 73] (and many variants such as CGM [51], LOGGP, LOGPC [112]), CLUMPS, E-BSP, D-BSP, SQM, *etc.* None are as effective as BSP. For hybrid and hierarchical architectures, new bridging models such as the **H**ierarchical **H**yper **C**lusters of **H**eterogeneous **P**rocessors (HIHCOHP) model [46] or the Multi-BSP model [278] have been designed. But these models are too complex in comparison with BSP: compare algorithms is thus much harder (even if in a low-level parallel model, some trick optimisations are easier to find) and especially, for our purpose, proof of correctness would be harder because parallel routines could be much harder (more side-effects, unspecified behaviour *etc.*) to formalize.

The only sensible way to evaluate an architecture-independent model of parallel computation such as BSP is to consider it in terms of all of its properties, that is (a) its usefulness as a basis for the design and analysis of algorithms; (b) its applicability across the whole range of general-purpose architectures and its ability to provide efficient, scalable performance on them; (c) its support for the design of fully-portable programs; and (d) software engineering tools such as those for correctness or debug can be easily adapt to programs of this bridging model.

Take for example, a proof of correctness of a GPU-like program. Although interesting in itself it cannot be used directly for clusters of PCs. A bridging model has the advantage that if a program is correct, then this is the case for “all” physical architectures. Note that it is also the case for portable libraries such as MPI but algorithm design would be clearly architecture independent, which will be not the case using a bridging model. Moreover, it is known and accepted that correctness of programs is more costly in terms of work than just programming and designing algorithms. Hence the choice in this thesis of the BSP bridging model to provide both portability for proofs of correctness and a model for algorithmic design and efficient programs.

1.4.3 Our Approach: Deductive Verification of BSP Programs

(a) The Traditional Solutions for the MPI Standard

Some works on the formal verification of programs based on standard MPI exist [135]. We can find dynamic testing and runtime verification [134, 203, 281]⁸ or debugging solutions [85]. The traditional solution for checking MPI codes is model-checking [143, 224] which works most of time only for some specific properties [252].

Model-checkers of MPI programs [251, 284] work on the C/MPI source code by using an abstraction of the MPI calls (the schemes of communications). An engineer, by push-button, can mainly verify that the program does not contain any possible deadlocks. A mechanised semantics of C/MPI primitives has been done in [197] (using TLA^+ [[6]], the **T**emporal **L**ogic of **A**ctions) which can be used for the model-checking of MPI programs — and dynamically testing logical assertions. Model-checking of C/MPI programs has also been done in [180] [[7]] [251] [[8]] [277, 284] [[9]] [[10]].

The drawback of these methods is that checking is limited to a predefined number of processors: it is impossible to verify that the program is deadlock free for any number of processors (a scaling problem). Increasing the number of processes increases the verification time (at most exponentially) even if partial order reductions are used to reduce the state space. Most of these works are aimed at standard concurrency properties, rather than the functional correctness of the computations carried out by an MPI program. Those tools do not check/prove statically the correctness of the results — only assertion violations. They focus on important properties as deadlocks, resource leaks or data-races. This greatly increases the confidence that we have in the programs but it is not enough if we want to guarantee correctness.

Proving a program for any number of processors brings an additional difficulty: many properties are proved by induction (over \mathbf{p} the number of processors). This kind of proof is problematic for automatic provers. We have sought another solution for verifying BSP programs and for the “BSP like” like subpart

⁸The verification is performed during the execution of the program: this is not a *a priori* verification.

of MPI which is collective routines. Our proposed solution is deductive verification of BSP programs.

(b) Proposed Solution: Deductive Verification of BSP Programs

Avoiding deadlocks (or data-races) is not sufficient to ensure that programs will not crash. We need to check buffer and integer overflows or liveness. And one can also want a better trust in the code: are results as intended? It is recognised that “correctness” is not only the safe execution of a program but also a formal characterisation of the intended results and formal properties that hold during the execution. Using a Verification Condition Generator (VCG) tool is the proposed solution. A VCG takes an annotated program as input and produces verification conditions (proof obligations) as output to ensure correctness of the properties given in the annotations. An advantage of this approach is that manual proof of properties using proof assistants can be mixed with automatised checks of simple properties using automatic decision procedures.

As stated by J.-C. Filliâtre: *Deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it.*

Deductive verification emphasises the use of a logic, to specify and to prove. This discipline is interested in the verification of small and challenging programs rather than huge systems. Deductive verification can be used to verify algorithms as well as programs, if the tool has been designed to verify codes that are executable — in our current work we only provide a tool for algorithm verification. The main concept (known nowadays as Hoare triple [160]) binds together a *precondition* P , a program statement s , and a *postcondition* Q . This triple is usually noted $\{P\}s\{Q\}$. The basic method is the insertion of logical annotations in the programs (called annotated programs) *e.g.* an *invariant* of a loop⁹.

From this annotated program (or algorithm), a VCG would produce verification conditions for provers — proof assistants or automatic provers. Dijkstra’s weakest preconditions calculus (wp calculus) is used. The key advantages of using a VCG are:

- It allows the verification of simple properties (such as “no overflow”) of a program without formally proving its entire correctness;
- Some tools are able to automatically insert annotations for some simple properties¹⁰;
- Using automatic provers enables the quick detection of simple errors;
- The manual proof of properties (using proof assistants) can be mixed with automatised checks of simple properties using automatic decision procedures

We provide a tool for the verification of properties of a special class of parallel programs by providing annotations and generation of proof obligations using a VCG. We choose BSP programs for three main reasons: (1) MPI programs with collective operators can be seen as BSP programs; (2) it is intrinsically a deterministic model; (3) as in COQ proofs of BSMML programs [40, 117, 118, 130, 271], the structured nature of BSP programs (sequences of super-steps with clear separation between communications and computations) allows BSP programs to be executed in a sequential manner. This latter property will be used by our tool to generate sequential programs from BSP ones and by using a traditional VCG as a back-end to generate goals — logical conditions. Also, the structured nature of the BSP programs allows programs to be decomposed into sequences of blocks of code, each block mainly corresponding to a super-step.

Writing a proof assistant or a VCG is a huge task which should be left to the experts. The main idea of our work is to simulate BSP parallelism by transforming the parallel code into a sequential form. Therefore, my goal is to use a “well defined” verification tool for sequential programs as a back-end for our own verification tool of parallel programs. Furthermore, implementing a VCG for a realistic programming language needs a lot of work: too many constructs require specific treatment. Reducing the VCG to a core language is a good approach. Another advantage of generating a sequential program with assertions is that we would be able to use any kind of dedicated tools for code analysis that work on annotated programs; thus avoiding the need to recreate these tools for the BSP model.

⁹A property that is logically valid before the loop and during any step of the loop.

¹⁰Many engineers do not want to write anything else than the programs and thus want push-button tools (such as model-checking) to provide at least safety and liveness, there are some tools [179] that automatically provide annotations for some properties. But defining the exact meaning of a computation is clearly a human activity. We have not yet studied this problem and we add assertions manually.

(c) The VCG Why and its programming language

We choose the VCG WHY [33, 101, 103, 105] [[11]]. First, it takes as input a small language (WHY-ML, close to ML) avoiding the need to handle all the constructs of a real language¹¹. Instead, realistic programming languages can be compiled into the WHY input language: WHY currently interprets C programs, JAVA and ADA programs with the help of companion tools [104, 105, 146] — KRAKATOA for JAVA and HIT-LIT for ADA. Second, WHY is currently interfaced with the main proof assistants (COQ [[12]], PVS [[13]], ISABELLE [[14]], HOL [[15]]) and automatic decision procedures (SIMPLIFY [[16]], ALT-ERGO [[17]], Z3 [[18]], CVC3 [[19]], YICES [[20]], VAMPIRE [[21]]) as back-ends for proofs obligations. These provers can be used for the proof obligations obtained from the parallel programs.

One might ask, why not work directly on parallel C programs then? After all, the main BSP library are written for the C language, and the programs that an end-user will want to prove are likely to be written in C, too. There are two aspects to answering this question. First, from a practical point of view, it is much easier to transform a parallel program in a syntax close to WHY-ML to a sequential WHY-ML program. The transformation (simulation) itself becomes almost a direct rewriting in many parts, and there is the added advantage that we were able to re-use the WHY parser, slightly modified to handle the parallel specificities, for our tool (BSP-WHY). The other reason is that, much as WHY-ML is intended to be an intermediate language, BSP-WHY-ML is ultimately aiming at being an intermediate in a larger chain. The framework FRAMA-C [[22]] allows the transformation of C programs into WHY-ML programs, with the use of the JESSIE plug-in. Other languages allow a transformation in WHY too, such as JAVA with the use of KRAKATOA. BSP-WHY could be used in the same way, as an intermediate between true parallel programs in C or other programming languages, and the WHY platform.

Informally, the input syntax of the VCG WHY is an intermediate and specific alias-free ML language dedicated to program verification. As a programming language, it is a ML language which (1) has limited side effects (only mutable variables that cannot be aliased), (2) provides no built-in data type, (3) proposes basic control statements (assignment, if, while, exceptions [105]) (4) uses program *labels* to refer to the values of variables at specific program points. The full syntax and semantics of WHY can be found in [101].

A WHY program is a set of functions, annotated with pre- and post-conditions. Those are written in a general purpose specification language (polymorphic multi-sorted first-order logic). This logic can be used to introduce abstract data types, by declaring new sorts, function symbols, predicates and axioms. The verification condition generation is based on a Weakest Precondition calculus completed by a functional interpretation of the imperative features [101], incorporating exceptional post-conditions and computation of effects over mutable variables. The WHY language also provides the possibility of defining axioms, pure logical assertions and parameters: primitives that have type definitions (with possibles side-effects and logical assertions) but no implementation.

In the case of alias-free programs, simpler proof obligations is the main goal of WHY: for parallel codes, our own tool can generate obligations that are “hard” to read; the simplest the obligations are for the sequential parts of a program, the less complex they are for the parallel parts.

In Fig 1.3, we give a simple but representative example of WHY-ML code — from the WHY distribution. It allows us to give an informal presentation of WHY. The example is traditional binary search for a value in a sorted array — searching for v in array t . First, we include the package of arrays. The type ‘a array is a built-in shortcut for ‘a farray ref, where ‘a farray is the following abstract type for purely applicative arrays. In fact, only the statement \leftarrow (“:=”) and parameters can modify in place data. For example in “array.why”, we find:

```
(* t[e] is syntactic sugar for (array_get t e) *)
parameter array_get : a:'a array → i:int → {0 ≤ i < array_length(a)} 'a reads a {result=access(a,i)}
(* t [e1] ← e2 is syntactic sugar for (array_set t e1 e2) *)
parameter array_set : a:'a array → i:int → v:'a → {0 ≤ i < array_length(a)} unit writes a {a=update(a@i,v)}
```

that is accesses to an element in an array at index i is only possible if i is in the bound of the array. A parameter gives a computation and can have thus side-effects (writes in an argument or raise an exception). A **predicate** (a syntactic sugar) or a **logic** give only logical properties and appear in logical annotations. A **logic** can also appear in the computations.

Then we have the value to search for and a function to compute the middle of a part of an array (with an axiom about this computation). Moreover, we give a predicate which gives the meaning of the presence of v in a part of the array. Finally, we take the array for the search and auxiliary variables. The search has the pre-condition that the array is sorted. The loop has an invariant which says that v is potentially present within the two index limits l (left) and u (up). Note the use of an **assert** to force

¹¹This also happens when modeling MPI [197]; it forces to use the smallest possible number of routines. But for MPI, this subset still largely bigger than BSP, a hundred of routines compare to approximatively ten of BSP.

```

(* Binary search *)
include "arrays.why"
logic v : int
function mean(x:int, y:int):int = (x+y)/2
axiom mean :  $\forall x,y:\text{int}. x \leq y \rightarrow x \leq \text{mean}(x,y) \leq y$ 
predicate Present(t:int farray, l:int, u:int) =  $\exists i:\text{int}. l \leq i \leq u \text{ and } t[i]=v$ 
parameter t : int array
parameter l,u,p,m : int ref

let binary_search () =
{ array_length(t)  $\geq 1$  and sorted_array(t,1,array_length(t)-1) }
begin
  l  $\leftarrow 1$ ; u  $\leftarrow$  (array_length_ t)-1; p  $\leftarrow 0$ ;
  while !l  $\leq$  !u do
    { invariant 1  $\leq$  l
      and u  $\leq$  array_length(t)-1
      and 0  $\leq$  p  $\leq$  array_length(t)-1
      and (p=0  $\rightarrow$  Present(t,1,array_length(t)-1)  $\rightarrow$  Present(t,l,u))
      and (p>0  $\rightarrow$  t[p]=v)
      variant 2+u-l }
    m  $\leftarrow$  (mean !l !u);
    assert { l  $\leq$  m and m  $\leq$  u };
    if t[!m] < v then l  $\leftarrow$  !m+1
    else if t[!m] > v then u  $\leftarrow$  !m-1
    else begin p  $\leftarrow$  !m; l  $\leftarrow$  !u+1 end
  done
end
{ (1  $\leq$  p  $\leq$  array_length(t)-1 and t[p]=v) or (p = 0 and  $\neg$ Present(t,1,array_length(t)-1)) }

```

Figure 1.3. A simple example of WHY-ML codes (from the WHY distribution): the binary search.

another property (for the rest of the obligations). At the end, we have the final post-condition which says that at index p of the array, we have found the search value; otherwise the index is 0 and v is not present in the array (in this example only, the values in the arrays are indexed from 1).

1.5 Outline

In Chapter 2, we present different libraries for imperative BSP programming (mainly for C and JAVA): the primitives and their informal semantics as well as their advantages and disadvantages. That allows us to show their differences and what they have in common, to extract the essential for our tool of deductive verification. Finally we present some choices for our core language, called BSP-WHY-ML, which tries to generalise the various imperative BSP libraries.

In Chapter 3, we show how BSP-WHY-ML can be used to specify a parallel program. We will then explain how our BSP-WHY tool works, that is how the transformation of a BSP-WHY-ML program into a sequential one is achieved. We also show how our model can be extended to prove the correction of programs with subgroup synchronisation.

In Chapter 4, we give several applications of BSP-WHY. We prove several classic parallel algorithms, such as prefix computation and parallel sorting. Finally, we give a mechanised proof of a more complex example, the parallel generation of the state-space in model-checking.

In Chapter 5, we will show the correctness of our approach. This is done by defining formally, mathematically and then mechanically in the COQ proof assistant the formal operational semantics of BSP-WHY-ML. We then give the proof that the transformation from BSP-WHY to WHY is correct with respect to the given semantics.

2 Imperative BSP Programming

Contents

2.1	Different Kinds of BSP Operations	15
2.1.1	Generalities	15
2.1.2	Description of these Routines	17
2.2	Some Details About Some BSP Libraries	20
2.2.1	The Message Passing Interface (MPI)	20
2.2.2	BSPLIB Like Libraries for C	21
2.2.3	BSP Programming over Multi-cores and GPUs	25
2.2.4	BSP Programming in Java	27
2.2.5	Other Libraries	30
2.2.6	Which Routines for BSP-Why	34
2.3	Common Errors Introduced in BSP Programming	35
2.3.1	Common Errors and Bugs	35
2.3.2	These Errors in BSP Programs	36
2.4	Related Works	38
2.4.1	Parallel Libraries and Imperative Programming	38
2.4.2	Other Parallel Paradigms	38

Since the design of the BSP model of execution there are different libraries and languages for BSP programming. Those libraries mainly exist as extensions of common sequential programming languages (C/C++, OCAML, PYTHON, JAVA, *etc.*) and some were designed for specific architectures. In this chapter, we study those extensions and compare them. That will be used later to show what we can process using our BSP-WHY-ML language (presented in the next chapter) and how it differs from the previous libraries.

2.1 Different Kinds of BSP Operations

Fig. 2.1 resumes and compares the considered libraries. We give below more details about those features. Like many other communications libraries, BSP libraries adopt a **Single Program Multiple Data** (SPMD) programming model. The task of writing an SPMD program will typically involve mapping a problem that manipulates a data structure of size N into p instances of a program that each manipulate an $\frac{N}{p}$ sized block of the original domain.

2.1.1 Generalities

Historically, the BSP library was the BSPLIB [157] [[23]] for the C programming language. Based on this library, for graph manipulation, CGMLIB [51] [[24]] has been developed. It also limits the BSP model by simplifying schemes of communications: only total exchanges are mainly authorised. The BSPLIB has also been re-implemented using MPI in BSPONMPI [[25]]. The BSPLIB has also been extended in the Paderborn University BSP library (PUB) [37] [[26]] by adding subset synchronisations, high-performance operations and migration of threads (only using TCP/IP). The PUB library has also been adapted to the JAVA language and grid computing in [36]. In [86], the authors have extended the BSPLIB to execute programs in heterogeneous systems.

For the JAVA language, different libraries exist. To our knowledge, the first one is [144]. There is also BSPONMULTICORE [287] (which also works for C [288]) [[27]] which aims to provide a BSP library for

Library	Language	BSMP	DRMA	Coll ops	H-P. ops	Oblivious sync	Subgroups sync	Thread mig	Hybrid archi
MPI	C/FORTRAN	yes	yes	yes	yes	yes	yes	not standard	yes**
BSPLIB	C	yes	yes	yes	yes	no	no	not standard	no
PUB	C	yes	yes	yes	yes	yes	yes	not portable	no
BSGP	C	no	yes	yes	no	no	no	yes	yes*
CT	C++	no	yes+	yes	no	no	yes	no	yes*
BSPONMULTICORE	C	yes	yes	no	yes	no	no	no	yes*
BSPONMULTICORE	JAVA	yes	yes	no	yes	no	no	not yet	yes*
NestStep	C	no	yes+	no	no	no	yes	yes	yes
NestStep	JAVA	no	yes+	no	no	no	yes	yes	yes
PUB-Web	JAVA	yes	no	yes	no	yes	no	yes	no
HAMA	JAVA	yes	no	no	no	no	no	yes	yes**
JBSF	JAVA	yes	no	no	no	no	no	no	no
jMigBSP	JAVA	yes	no	no	no	no	no	yes	no
BSML	OCAML	yes	no	no	no	no	no	no	no
BSP++	C++	yes	no	no	no	no	no	no	yes
BSP-PYTHON	PYTHON	yes	no	no	no	no	no	no	no
BSP-WHY	WhyML	yes	yes	yes	yes	no	not yet***	yes	no

with the following acronyms:

- BSMP \Rightarrow **M**essage **P**assing (send/received)
- DRMA \Rightarrow **D**irect **R**emote **M**emory **A**ccess
- Coll ops \Rightarrow **C**ollective operations/routines
- H-P. ops \Rightarrow **H**igh-**P**erformances operations (asynchronous/unbuffered sends)
- Oblivious sync \Rightarrow Oblivious synchronisation
- Subgroups sync \Rightarrow Subgroups/Subsets synchronisation
- Thread mig \Rightarrow Thread migration and automatic system balance of the computations over the processors
- Hybrid archi \Rightarrow Hybrid and hierarchical architecture (cluster of multi-cores)

and where:

- * : with the used of another library
- ** : optimisations in some implementations
- *** : easy to add but currently not yet implemented
- + : Work with shared data

Figure 2.1. A comparison of BSP libraries.

multi-core technologies. There is also the JMigBSP library [140] which also provides *code migration* — using the facilities of JAVA to serialise objects. More recently, this library, with scheduling and migration on grid environments of BSP threads, has been re-implemented in [75] — the scheduling is implicit but the migration can be explicit. But the most known library is HAMA [246] [[28]]. It allows BSP’s communications and the implementation is provided by a “MAPREDUCE like” framework of the foundation APACHE. A comparison with the Google’s MAPREDUCE language [80] is described in [227]. The author notes that some graph algorithms could not be efficiently implemented using MAPREDUCE. Notice that another Google’s language call PREGEL [205] [[29]] (without any public implementation) is also used by this company to perform BSP computations on graphs. We can also highlight the work of NESTSTEP [176] [[30]] which is C/JAVA library for BSP computing, which authorizes nested computations in case of a cluster of multi-cores but without any safety.

For the GPU architectures, a first BSP library was provided in [165] [[31]]. It is mainly the primitives to remote access memory of the BSPLIB/PUB. The BSML primitives (BSP for ML) [129] were adapted for C++ in BSP++ [148] [[32]]: this library provides nested computation in the case of a cluster of multi-cores (MPI + OPEN-MP). The BSML language also inspired BSP PYTHON [159] [[33]]. BSML also inspired BSP-HASKELL [214]. The library [131] provides collective operators in a BSP fashion for multi-core architectures.

MPI programs that only use *collective operations* are close to BSP ones [25]. [47] argues that most MPI programs can be thought as a sequence of collective patterns: most asynchronous communications are used to simulate a collective pattern that is not present in MPI. [211] proposes an automatic tool (not yet fully implemented) to transform asynchronous communications into some calls of collective operations.

2.1.2 Description of these Routines

(a) Sending Messages and One-sided Communication

Bulk Sending (BSMP). The first, simple and most frequent way for BSP programming is bulk sending. It is mainly a routine of the form “send(v,i)” to send a value to remote processor i. This differs from MPI send/received because the sending is non-blocking and the value is only available for processor i at the next super-step. It is the underlying system that decides when to truly send the value on the network — or a copy of the value in case of a shared memory architecture. The reception is performed using a routine of the form v=received(i,n) to read the nth value received from processor i during the past super-step. However some libraries do not give any order for the reception of messages but allow to “tag” messages to distinct them. The BSP libraries also give access to the number of received messages and/or get the tag of a message.

BSMP is more convenient than DRMA for computations where the volumes of data being communicated in super-steps are irregular and data dependent, and where the computation to be performed in each super-step depends on the quantity and form of sent data received at the start of this super-step.

Remote Data and Communication. Another way of communication is through DRMA (which stands for **D**irect **R**emote **M**emory **A**ccess): after every processor has registered a variable for direct access, all processors can read or write the value on other processors. This is usually called one-sided communication, because only one process needs to issue a send or receive call to achieve the communication. This scheme can simplify programming in cases where the memory locations that must be updated or interrogated are known on only one side of a communicating pair of processes. DRMA allows processes to specify shared memories and distant read/write in these memories.

The “put(x,v,i)” routine allow a processor to write the value v in the memory (variable v) of another processor i. It thus assumes that the source processor knows the memory location on the destination processor where the data must be put. The source processor is the initiator of the action, whereas the destination processor is passive. Thus, we assume implicitly that each processor allows all others to put data into its memory. To enable a processor to write into a remote variable, there must be a way to link the local name to the correct remote address. Linking is done by the registration primitive “push_reg(x)”. A variable is deregistered by a call to “pop_reg(x)”.

Note that for some case, using a “put” can be simpler than using a matching “send”/“receive” pair, as it is done in MPI-like algorithms: the program text of such an algorithm must contain additional if-statements to distinguish between sends and receives; careful checking is needed to make sure that pairs match in all possible executions of the program and even if every send has a matching receive, this does not guarantee correct communication as intended by the designer of the algorithm and a deadlock can occur. Such a problem cannot happen when using synchronous puts.

Sometimes, it may be necessary to let the destination processor initiate the communication. This may

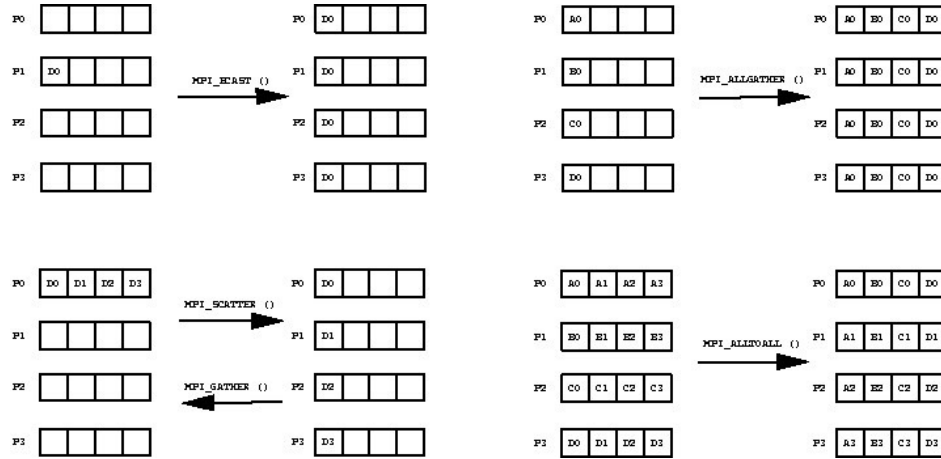


Figure 2.2. MPI collective operations.

happen in irregular computations, where the destination processor knows that it needs data, but the source processor is unaware of this need. In that case, the destination processor must fetch the data from the source processor. This is done by a statement of the form “get”.

Note that BSP put, get and sends operations do not block progress within their super-step: after a put or get is initiated, the program proceeds immediately. Most BSP libraries do not guarantee to exploit potential overlap of communication and computations. Instead, delaying all communication gives more scope for optimization, since this allows the system to combine different messages from the same source to the same destination and to reorder the messages with the aim of balancing the communication traffic.

(b) High-performance Routines and Oblivious Synchronisation

In some BSP libraries, message passing and DRMA routines have their high-performance counterpart. Those routines are less safe but can be more efficient. They are unbuffered and nonblocking that is when a processor executes `hp-send(v, i)` to send a value v to processor i , then it is unspecified when the value v will be sent. In this way, if the program changes the value of v , it is unknown which value will be received by processor i . In case of a `send(v, i)`, it is a copy of v that is sent and thus the program can then modify v for its own purpose without modifying the meaning of the sending. Note that unlike some “MPI send routines”, all “BSP’s send routines” are nonblocking in the sense that the sender does not wait until the receiver truly receives the value; the good reception is guaranteed by the BSP barrier, *i.e.* the end of the super-step. Note also that some “MPI send routines” are also asynchronous as well as reception: the famous `isend` and `irecv` where mainly a “wait” for a request is performed to know when the message is truly received. It is thus possible to program BSP algorithms using MPI but it is more difficult and error prone since one has to manage all the waiting of data.

The oblivious synchronization is the high-performance pending of the traditional BSP synchronisation. It should be used if the programmer knows the number of messages each processor will receive in a super-step. Thus, in the oblivious synchronization each processor waits until “nmsgs” are received. This type of synchronization is cheap and much faster than the other one because no additional communication is needed to determine that processors can move on to the next super-step.

To our knowledge, The PUB library has been the first to propose this kind of synchronisation.

(c) Collective Operations

A collective operation is a routine in which data are simultaneously sent to or received from many nodes. Common examples of collective operations (those of the standard MPI) are:

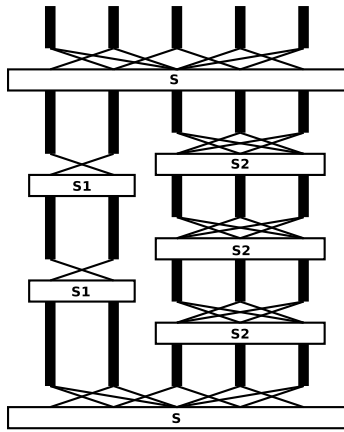
- Broadcast, where the same data is sent to all nodes;
- Gather, where data is collected from all nodes;
- Scatter, where a set of data is broken up into pieces, and each piece is sent to a different node;
- Allgather, where every node send a data to other nodes;
- Alltoall, where every node send a different value (even empty) to each node;
- Global reduces, where an associative operation is performed over each data of each node.

Fig 2.2 resumes the below operations. The advantage of this kind of operations is that their implementation can be optimised for some specific architectures since they provide a clear semantics. They also simplify the reading and understanding of the programs. Collective operations of BSP libraries generally work on the whole machine and are synchronous. MPI ones work on a communicator that is a subgroup of processors that have been registered to be within the communicator — using a specific routine. Another collective operation is the `sendrcv` in which two processors send (synchronously) each other a value.

Collective operations can be used to simulated the traditional BSP message passing and DRMA routines. This is the case of the BSPONMPI library and BSML where all the routines accumulate values in some proper structures and finally, to finish the super-step, an MPI's `alltoall` is performed.

(d) Subgroup Synchronisation

The BSP model is based on a global synchronisation. However, in some cases, a parallel algorithm may include problems that can be solved using only a subset of processors. Some libraries extend the basic BSP model, and allow the definition of *subgroups*, which are pairwise disjoint subsets of the set of processors. It is then possible to write a part of the parallel program with the subgroup acting as an independent BSP computer. A call to the `bsp_sync` function will then synchronise over the subgroup, instead of the whole parallel computer. We then speak of partitioned synchronisation.



In the left, we show an example of execution of subgroup synchronisation. In this example, the overall group of processors *S* is split into two subgroups (*S1* and *S2*) which run independent BSP computations. Finally the two subgroups are merged and the whole machine continue its work doing global barriers.

The idea of subgroup synchronisation is to allow the synchronisation to work on a subset of processors. This means that the communication procedures need to be able to tell in which group they are working. This is especially important for the synchronisation procedure. An additional argument is thus added to all the parallel procedures, representing a group of processors linked together. It is called a *communicator*. The MPI standard also allows to create sub-communicators: in this way, collective operations are performed only on a subset of the processors, those which participate to the communicator.

(e) Thread Migration

Some BSP libraries (mainly for JAVA) offer the possibility to have more threads of computations than true nodes/processors available in the machine. They provide object rescheduling by using two different solutions: (1) migration directives on the application code directly or (2) through automatic load balancing at middleware level. One can speak of migratable virtual processors. It was designed to work on grid/cloud computing environments since thread migration and rescheduling allow a thread remapping in response to application and infrastructure behaviour. This technique is thus useful to migrate entities for executing faster on lightly-loaded resources and/or approximating those ones that communicate frequently. The execution time of such a parallel program can be significantly improved because it is possible to migrate its processes at run-time to other hosts with currently offer more available computation power.

As for garbage collectors of high-level languages, the second solution provides both transparent and effortless mechanism of migration in the user's point of view. The first solution allows the user to find an appropriate rescheduling but with the drawback of missing a good mapping or rescheduling too often, which can induce too much communication and thus bad performances: the explicit rescheduling requires a developer with expertise in load balancing algorithms; in this way, the programmer may be required to collect data about processors' capacity and to load for decision making on process relocation manually.

(f) Hybrid and Hierarchical Architectures

Coupling SMP clusters (clusters of multi-cores) combine the packaging efficiencies of shared-memory multiprocessors with the scaling advantages of distributed-memory architectures. The result is a computer architecture that can scale more cost-effectively in size. Unfortunately, these systems come at the price of a more complex programming environment to deal with the two different modes of parallel execution. These architectures are called hybrid due to these two different modes and are also called hierarchical since the computational units are mainly organised as a tree. While tools exist for shared-memory systems

and for distributed-memory systems, solving problems on parallel computers with SMP nodes is not as simple as combining two tools.

But BSP is a flat parallel model and the view of a parallel machine as a set of communicating sequential machines remains true but is more than incomplete. This is why some libraries such as NestStep or PUB allow nested or subgroup synchronisation. Moreover, there are BSP libraries for different kinds of architectures such as multi-cores or GPUs. For example, BSP++ allows nested BSP computations and uses MPI over the network of the cluster and OPENMP to share memory multi-cores; it uses CUDA for GPUs. We will discuss this issue in the last chapter because currently, our work is based on a rather flat BSP model even if we can manage subgroup synchronisation.

2.2 Some Details About Some BSP Libraries

2.2.1 The Message Passing Interface (MPI)

The **Message Passing Interface** (MPI) [259] is a standard API for communication in distributed-memory parallel applications. There are numerous implementations of this API, both commercial and open source (*e.g.* OPEN-MPI, MPICH, *etc.*). MPI's goals are high performance, scalability, and portability. MPI is *de facto* the standard for high-performance computing today. And MPI is the main library used for implementing other parallel languages such as skeletons ones, BSP, *etc.* The MPI standard defines the informal semantics of routines useful to a wide range of users writing portable distributed programs in FORTRAN and C. However, there are many stubs codes for languages such as JAVA, OCAML, *etc.*

MPI contains a huge number of routines, thus it is not possible to present all of them. We thus only present the routines that are close to BSP such as collective operations and operations for the communicators manipulation¹. We also present only MPI-2 routines as, MPI-3 was released in September 2012 and is not yet fully implemented by any MPI library. Fig 2.3 gives the C types of these routines. The first are for initialize (or terminate) the MPI computation and know the number of processes or their “id” for a group of processor. There is an initial communicator `MPI_COMM_WORLD` which contains all the processes participating to the computation.

A collective communication is defined as communication that involves a group of processes, *i.e.* a communicator. All processes in the group identified by the communicator must call the collective routine. We do not give explanation of all the arguments but resume the most important ones for our purpose. In many cases, collective communication can occur “in place” for communicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, `MPI_IN_PLACE`, instead of the send buffer or the receive buffer argument, depending on the performed operation. The collective operations are for broadcasting, scattering, gathering and all-to-all exchanges. There is mainly the buffer to send (resp. to receive the data) with the number of sending data and their types — `MPI_INT`, `MPI_DOUBLE`, *etc.* to obtain portable programs. As many C routines, they return an integer that indicate if the operation has been well realized. The `int MPI_Alltoallw` routines is the most general form of all-to-all. For example, by making all processes have `sendcounts[i] = 0`, this achieves an `MPI_scatterw`. This is a powerful routine but hard to use. There is also a routine a bit different: `sendrecv`. This routine involves only two processors which exchange data in a synchronous way as a barrier. In fact, this is a sub-group of only two processors. Also, reducing operations involve an associative operator (`MPI_op`) on the data — `MPI_Datatype`. They enable to apply these operator on all the data in the distributed arrays.

Collective operations are not really BSP communications since they are not synchronous. For example, using a broadcast, if the emitter changes the buffer to send after the call of the routine, it is unspecified which value will be received by other processors. A call to a `MPI_barrier()` is needed to forbid this case. In practise, most MPI libraries implement collective operations in a synchronous way but this is not standard.

There are many routines for managing communicators — we count 35 in MPI-2. However, the main routine is `MPI_comm_split` which creates a new communicator by partitioning the group into disjoint subgroups using a set of colors one for each value of color. The colors are define in the API. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. This is a collective call, but each process is permitted to provide different values for color and key. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups and many routines, such as `MPI_comm_create`, are equivalent to a specific call to `MPI_comm_split`. Unlike the PUB (see thereafter) function, this MPI routine requires communications between the processors of

¹We certainly missed some, considering the amount of existing ones, but we present those that are the most used.

Tools:	
<code>int MPI_Init(int *argc, char ***argv)</code>	Initialize the MPI execution environment
<code>int MPI_Finalize(void)</code>	Terminates MPI execution environment
<code>int MPI_Comm_size(MPI_Comm comm, int *size)</code>	Determines the size of the group associated with a communicator
<code>int MPI_Comm_rank(MPI_Comm comm, int *rank)</code>	Determines the rank of the calling process in the communicator
Collective communicating operations:	
<code>int MPI_Barrier(MPI_Comm comm)</code>	
<code>int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)</code>	
<code>int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)</code>	
<code>int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</code>	
<code>int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)</code>	
<code>int MPI_Scatter(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</code>	
<code>int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)</code>	
<code>int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</code>	
<code>int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)</code>	
<code>int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</code>	
<code>int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)</code>	
<code>int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[], MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[], int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)</code>	
Collective reducing operations:	
<code>int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)</code>	
<code>int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</code>	
<code>int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</code>	
<code>int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</code>	
<code>int MPI_Exscan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</code>	
<code>int MPI_Reduce_scatter_block(void *sendbuf, void *recvbuf, int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</code>	
Communicator manipulation:	
<code>int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)</code>	Duplicates an existing communicator
<code>int MPI_Comm_free(MPI_Comm *comm)</code>	Deallocation of the communicator
<code>int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)</code>	Creates new communicators based on colors and keys
<code>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)</code>	Creates a communicator from two others
<code>int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)</code>	Creates a communicator from a communicator
Windows for DRMA:	
<code>int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)</code>	Creation of a window
<code>int MPI_Win_free(MPI_Win *win)</code>	Free a MPI window
<code>int MPI_Win_fence(int assert, MPI_Win win)</code>	Fence synchronisation
<code>int MPI_Win_complete(MPI_Win win)</code>	Complete a DRMA operations begun after a MPI_Win_start
<code>int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)</code>	Start DRMA accesses for MPI
<code>int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)</code>	Put data into a memory window on a remote process
<code>int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)</code>	Get data from a memory window on a remote process

Figure 2.3. The MPI primitives.

the group. In fact, exchanging the colors can be seen as a call to the `MPI_Allgather` collective operation. Communicators in MPI are thus both more flexible and complex. Unlike in PUB, there is almost no restriction to the way groups are formed. Processes can be in more than one group, they have an unique rank within each group. A traditional example of use of such communicators would be in linear algebra, where it is often useful to split processors by rows and by columns.

MPI also provides routines for asynchronous DRMA accesses, the windows. A programmer can create or free a window. It is a synchronous operation that every processors perform. After, the processors can write or read on remote processors. There is also a kind of barrier, the fence, for processors that have been registered to the window. And using the `Win_complete`, we can force a processor for waiting that all remote readings and writings have been performed on a window after a `Win_start`.

Furthermore, MPI obviously provides asynchronous send/received of data. These routines are outside the scope of this thesis.

2.2.2 BSPLIB Like Libraries for C

The BSPLIB [157] is a C-library of communication routines to support the development of parallel algorithms based on the BSP model. There is also the PUB [37] which provides additional features such as oblivious synchronization (where each processor waits until n messages are received), subgroup synchronisation (where only a part of the processors synchronise) and thread migration — not described in this document because too complex and too architecture dependant. Both libraries offer functions for both

message passing (BSMP) and remote memory access (DRMA). Both are implemented using TCP/IP or MPI. Some collective communication operations like broadcast are also provided in the PUB, which can easily be simulated by BSMP operations. Fig 2.4 gives the routines of the standard BSPLIB. Fig 2.6 gives the routines of PUB. Because they are the most used libraries for BSP programming, we now details these routines and their differences. Note that BSPONMPI is only a new implementation of the MPI implementation of BSPLIB.

(a) The Routines of the BSPLIB [157]

Tool Routine. As in standard MPI, we first need to initialise our parallel computation, using the function `bsp_init`, and then we can have different BSP computations, each beginning with `bsp_begin` and terminating with `bsp_end`. Now, within a BSP computation, we can query some information about the machine: `bsp_nprocs` returns the number of processors `p` and `bsp_pid` returns the processor “id” which is in the range $0, \dots, p - 1$.

Synchronisation. According to the BSP model, all messages are received during the synchronisation barrier and cannot be read before. The barrier is done using `bsp_sync` which blocks the node until all other nodes have called `bsp_sync` and all messages sent to it in the current super-step have been received.

Message Passing. In BSMP, a non-blocking send operation is provided that delivers (a packet of) messages to a system buffer associated with the destination process. The message is guaranteed to be in the destination buffer at the beginning of the subsequent super-step, and can be accessed by the destination process only during that super-step. If the message is not accessed during that super-step it is removed from the buffer. Using the BSPLIB, the destination buffer of a processor may therefore be viewed as a queue, where the incoming messages are enqueued in arbitrary order.

Sending a packet (in a buffering mode) is done using `bsp_send` and is based on the idea of two-part messages, a fixed-length part carrying tagging information that will help the receiver to interpret the message, and a variable-length part containing the main data payload. The length of the tag is required to be fixed during any particular super-step, but can vary between super-steps. Choosing a tag is done using `bsp_set_tagsize` where the user specifies the size of the fixed-length portion of every message in the subsequent super-steps. Allowing the user to set the tag size enables the use of tags that are appropriate for the communication requirements of each super-step. This should be particularly useful in the development of subroutines either in user programs or in libraries. The procedure must be called collectively by all processes. A change in tag size takes effect in the following super-step; it then becomes valid.

The programmer can know the number of received messages as well as the total size of received data (in bytes) using the routine `bsp_qsize`. This routine work on the queue of received messages. To receive a message, the user should use the procedures `bsp_get_tag` and `bsp_move`. The operation `bsp_get_tag` returns the tag of the first message in the queue and the size of the payload (status is -1 if the queue is empty). The operation `bsp_move` copies the payload of the first message in the system queue, *i.e.* the buffer call `payload`, and removes that message from the queue. Then, the system will advance to the next message.

DRMA Routines. Registering a variable or deleting it from global access is done using: `void bsp_push_reg(ident, size)` and `bsp_popregister(ident)`. Due to the SPMD structure of BSP programs, if `p` instances share the same name, they will not, in general, have the same physical address. To allow BSP programs to execute correctly, the BSPLIB provides a mechanism for relating these various addresses by creating associations called registrations. A registration is created when each process calls `void bsp_push_reg` and, respectively, provides the address and the extent of a local area of memory: registration takes effect at the next barrier synchronisation and newer registrations replace older ones. This scheme does not impose a strict nesting of push-pop pairs. For example:

On processor 0: <pre>void x[5], y[5]; bsp_push_reg(x, 5) bsp_push_reg(y, 5)</pre>	On processor 1: <pre>void x[5], y[5]; bsp_push_reg(y, 5) bsp_push_reg(x, 5)</pre>
--	--

The variable `x` on processor 0 would be associated with `y` on processor id 1. Note that this example is clearly not a good way of programming in BSP. In the same manner, a registration association is destroyed when each process calls `bsp_popregister` and provides the address of its local area participating in that registration. A run-time error will be raised if these addresses (*i.e.* one address per process) do

Tools:	
void bsp_init(void (*startproc)(void), int argc, char ** argv)	Initialise the BSPLIB system
void bsp_begin(int maxprocs)	Spawn a number of BSP processes
void bsp_end()	Terminate BSP processes (and free resources)
int bsp_nprocs()	Determine the total number of BSP processes
int bsp_pid()	Determines the process “id” of a BSP process
void bsp_sync()	BSP synchronization; end of a superstep
BSMP primitives:	
void bsp_set_tagsize(int *tag_bytes)	Tag size of a BSMP packet
void bsp_get_tag(int *status, void *tag)	Check the tag on a BSMP packet
void bsp_send(int pid, const void *tag, const void * payload, int payload_bytes)	Transmit a BSMP packet to a remote process
void bsp_qsize(int *packets, int *accum_nbytes)	Checks to see how many BSMP packets arrived
void bsp_move(void *payload, int reception_bytes)	Move a BSMP packet from the system queue
DRMA primitives:	
void bsp_push_reg (const void *ident, int size)	Register a data-structure as available
void bsp_popregister(const void *ident)	Remove the visibility of a data-structure
void bsp_get(int pid, const void *src, int offset, void *dst, int nbytes)	Copy data from a remote memory
void bsp_put(int pid, const void *src, void *dst, int offset, int nbytes)	Deposit data into a remote memory

Figure 2.4. The BSPLIB primitives.

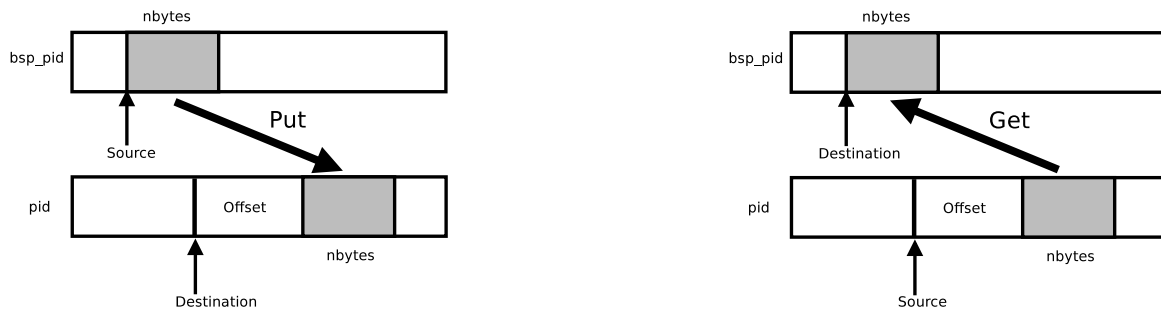


Figure 2.5. DRMA BSP operations: “put” and “get”.

not refer to the same registration association. Un-registration takes effect at the next synchronisation barrier. The two DRMA operations (illustrated in Fig 2.5) are the following:

1. **bsp_get** stands for global reading access. It copies *nbytes* to the local memory address *dst* from the variable *src* at offset *offset* of the remote processor *pid*;
2. **bsp_put** stands for global writing access. It copies *nbytes* bytes from local memory *src* to *dst* at offset *offset* on remote processor *pid*.

All get and put operations are executed during the synchronisation step and all get are served before a put overwrites a value. For example:

```
int array[100], privateArray[100];
src = 0; dest = 0; offset = 0;
bsp_push_reg (&bsp, array, sizeof(array));
bsp_get (&bsp, src, array, offset, privateArray, sizeof(array));
bsp_put (&bsp, dest, &privateArray[1], array, bsp_pid(bsp)*sizeof(int), sizeof(int));
bsp_sync (&bsp);
bsp_pop_reg (&bsp, array);
```

In this example every processor gets a copy of array on processor 0 in his own privateArray (**bsp_get**). After that each processor writes `privateArray[1]` into `array[pid]` on processor 0. You could also exchange the put and the get and you would get the same result since “gets” are served before “puts”.

(b) The Routines of the PUB [37]

The PUB’s routines are close to the ones of the BSPLIB. The true differences are the features for subgroup synchronisation and collective operations — not described here because trivially simulated by BSMP routines. During initialisation, the added parameter can be `BSPLIB_STDPARAMS` or a pointer to a `t_bsplib_params` that is initialized with `bsplib_params_init`. Is is as the `MPI_COMM_WORLD` of MPI. Now, let us describe the differences.

Tools:	
void bsplib_init(<i>t_bsplib_params*</i> parameter, <i>t_bsp*</i> bsp)	initialises the BSP computation
void bsplib_saveargs(<i>int*</i> argc, <i>char***</i> argv)	initialises the arguments in some architectures
void bsplib_params_init(<i>t_bsplib_params*</i> parameter)	initialises the BSP cost parameters
void bsplib_done()	exits and frees resources
int bsp_nprocs(<i>t_bsp*</i> bsp)	returns the number of processors in the group
int bsp_pid(<i>t_bsp*</i> bsp)	returns own processor-id in the group
void bsp_sync(<i>t_bsp*</i> bsp)	BSP synchronization of the group
BSMP primitives:	
void bsp_send(<i>t_bsp*</i> bsp, int dest, void* buffer, int size)	Bulk sending of a buffer
void bsp_sendmsg(<i>t_bsp*</i> bsp, int dest, <i>t_bspmsg*</i> msg, int size)	Bulk sending of a message
<i>t_bspmsg*</i> bsp_createmsg(<i>t_bsp*</i> bsp, int size)	create a message
int bsp_nmsgs(<i>t_bsp*</i> bsp)	the number of received messages or buffers
<i>t_bspmsg*</i> bsp_findmsg(<i>t_bsp*</i> bsp, int proc_id, int index)	find a message in the queue
<i>t_bspmsg*</i> bsp_getmsg(<i>t_bsp*</i> bsp, int index)	get a message in the queue
void* bspmsg_data(<i>t_bspmsg*</i> msg)	returns a pointer to the data of a message.
int bspmsg_size(<i>t_bspmsg*</i> msg)	returns the size of the message contain
int bspmsg_src(<i>t_bspmsg*</i> msg)	returns the node-id of the sender
DRMA primitives:	
void bsp_push_reg(<i>t_bsp*</i> bsp, void* ident, int size)	register a variable for remote access
void bsp_pop_reg(<i>t_bsp*</i> bsp, void* ident)	delete the registration of a variable
void bsp_put(<i>t_bsp*</i> bsp, int destPID, void* src, void* dest, int offset, int nbytes)	remote writing to another processor
void bsp_get(<i>t_bsp*</i> bsp, int srcPID, void* src, int offset, void* dest, int nbytes)	remote reading from another processor
Subgroup routines:	
void bsp_dup(<i>t_bsp*</i> bsp, <i>t_bsp*</i> dup)	a new group as a copy of the group
void bsp_partition(<i>t_bsp*</i> bsp, <i>t_bsp*</i> sub, int nr, int* partition)	creates a new subgroup
void bsp_done(<i>t_bsp*</i> bsp)	destroy a subgroup

Figure 2.6. The PUB primitives.

DRMA Routines. The DRMA routines are the same except that they work for a subgroup. Note that in the PUB library, if different variables have to be registered/unregistered, all processors must call the functions in the same order. The previous solution is simpler, thus we have currently considered only it. But at synchronisation, it is easy to compare the local lists of registered variables to test if variables are correctly registered or not.

Message Passing. Sending a single message can be done using `bsp_send` or `bsp_sendmsg` once created. The arguments are obvious. After calling one of these routines the buffer (or the message) may be overwritten or freed. Which sent operation should be used in which situation? If there is not much communication in your program or super-step then the routine `bsp_send` is preferred. If there is a lot of data in not continuous memory for the same destination processor then it is better to use the routine `bsp_sendmsg`. In the next super-step, each processor can access the received messages of type `t_bspmsg`. This can be done using `bsp_findmsg` where `proc_id` is the “id” of the source-node and `index` is the index of the message. To access to the message, we need `bspmsg_data` which returns a pointer to the sending block of data and `bspmsg_size` which returns its size. Also `bsp_nmsgs()` returns the number of messages and buffers received in the last super-step. Note that the messages of the last super-step are available until the next synchronisation call. At this point the memory used for these messages will be deallocated. As for DRMA operations, these routines work in the scope of a subgroup.

Subgroup Primitives. The function involved in the process are `bsp_dup`, `bsp_partition` and `bsp_done`. The `bsp_dup` function creates a subgroup that is a duplicate of the current BSP computer. This is useful to organize algorithms in a compositional manner, since a code working on the duplicate will not be affected by previous messages that were already in the queue, waiting to be sent. The synchronisation on the duplicate subgroup will only complete the communications that were requested with that subgroup. It is thus possible to define a function for a sub-algorithm, starting with a call to `bsp_dup` so that it can be called from anywhere in a parallel program, without interfering.

The `bsp_partition` function is the way to create proper subgroups. The subgroups are described as a partition of the current BSP computer in contiguous subsets. The starting indexes of these subsets are given in argument in the `partition` integer array. To be a correct partition, the array has to be sorted. Finally, the `bsp_done` function indicates that we have finished the work with the subgroup. In the PUB library, it is impossible to synchronize or create other subgroups from the parent object until the current subgroup is released with the routine `bsp_done`. It is however possible to create different new subgroups of the current subgroup. The organisation of subgroups is similar to a stack, with only the lowest sub-

group being allowed to create new subgroups. Here is an example of a program, in the PUB library, with subgroup synchronisation:

```
t_bsp subbsp;
int part[2];
part[0] = 2;
part[1] = bsp_nprocs(bsp);
bsp_partition (bsp, &subbsp, 2, part);
if(my_pid<2)
    {...;bsp_sync(&subbsp);...}
else {...}
bsp_done(&subbsp);
```

Communicators are thus present both in MPI and in the PUB library as group which are called BSP objects. Although they play a similar role, there are notable differences between both. The PUB library allows to partition the set of processors into pairwise disjoint subsets. Each subset acts like an independent BSP computer. Subgroups are created by a call to the `bsp_partition` function. A call to the function creates a new `t_bsp` object, that can be used in any BSP call. It is also still possible to use the parent `bsp` object. A call to `bsp_partition` does not require communications between the processors, since each processor has to explicitly give the whole partition table. However, there are limitations. It is not possible to create other subgroups of the parent `bsp` object, until the current subgroup `sub` has been disposed of using the `bsp_done` function. The subgroups work like a stack: it is only possible to create subgroups of the lowest subgroup. Another limitation comes from the way the subgroups are defined. The partition has to be made of subgroups of consecutive processors. It is not possible to combine the processors 1 and 3 in a group, and 2 and 4 in another one, for a parallel machine of four computers.

2.2.3 BSP Programming over Multi-cores and GPUs

(a) The Ct Library [131]

The Intel CT library is now part of an Intel framework for parallel computing called TBB [184]. TBB contains parallel libraries and compilers, tools for debugging, *etc.* CT is a deterministic nested data-parallel C++ library intended to leverage the best features of GPUs while fully exploiting multi-cores flexibility. The main goal for Intel is power efficiency because “*power consumption is the ultimate limiter to improving computational performance in silicon technology*” [131].

CT introduces a new (template-style) polymorphic type, called a TVEC. TVECs are write-once vectors that reside in a vector space segregated from native C++ types, *e.g.* `TVEC<F64>` stands for a vector of doubles. For manipulating these vectors, there are three CT operator categories:

1. Element-wise operators that support simple unary, binary and n-ary operators, such as addition, multiplication, *etc.* For example, `TVEC<F64> product = multiply(nonzeros,expv)` or as operator overloading `TVEC<F64> product = nonzeros*expv`;
2. Collective communication operations, such as reduction, prefix-sum or combining-send. For example, `TVEC<F64> innerproduct = addReduce(product,RowIdx)`;
3. Permutation operations which allow both structured and unstructured reordering and replication of data. For example, `TVEC<F64> expv = distribute(v,ColP)`.

CT operators are logically free of side effects, from the programmer’s perspective. That way, each CT operator logically returns a new TVEC. CT’s support for nested vectors is a generalization that allows a greater degree of flexibility. Vectors may be flat or regular multi-dimensional vectors. They also may be nested vectors of varying length, which allows for very expressive coding of irregular algorithms. Fig 2.7 gives an overview of the CT primitives. All the primitives are synchronous, even for collective communications. Thus, CT allows some kind of nested BSP computations. CT is implemented using owner Intel low-level libraries.

(b) BSPonMultiCore for C [288]

This library is a full implementation of the routines of BSPLIB but optimised for multi-cores architectures. The BSPLIB has a POSIX PThreading implementation, thus it works well for multi-core architectures. But there were no optimisations using shared caches and bus interfaces (from cores to the main memory) or current multi-core architectures. However, in the BSPLIB, these optimisations has been done for other (older) specific architectures such as CRAY or SGI ORIGIN massive computers.

Facilities:		
copyIn, copyin2D, copyin3D, copyout	Managed Vector/Native Space Copying	
cat, repeat, replicate, replace, index, copy, newVector	Vector Generators	
extract, copy, length	Vector Utilities	
newNestedVector, applyNesting, copyNesting, getNesting, <i>etc.</i>	Nested Vectors	
Element-wise:		
add, mul, div, equal, min, mod, greater, and, select, map, <i>etc.</i>	Vector-vector	
addVectorScalar, equalVectorScalar, andVectorScalar, <i>etc.</i>	Vector-Scalar	
abs, not, log, sin, cos, <i>etc.</i>	Unary	
Collective Communication:		
addReduce, mulReduce, andReduce, reduce, <i>etc.</i>	Reduction	
addScan, mulScan, andScan, scan, <i>etc.</i>	Scan/Prefix-Sum	
Permutation:		
pack, unpack		Pack/Unpack
scatter, gather		Scatter/Gather
leftShiftPermute, leftRotatePermute, shiftDefaultPermute, rotateDefaultPermute		Shift/Rotate
partition, unpartition		Partition
defaultPermute, omegaPermute, butterflyPermute, distribute		Miscellaneous

Figure 2.7. Some CT primitives.

(c) BSPGP: BSP programming over GPU Architectures [165]

BSGP is a BSP programming language for GPU. BSGP extends the C language with few primitives for spawning threads and performing barriers. The statements between two barriers are automatically deduced as a super-step and translated into a GPU kernel by the BSGP compiler. In a BSGP program, data dependencies are defined implicitly because local variables are visible and shared across super-steps.

As for GPU, the main goal of BSGP is stream processing and especially stream of pixels for image analysis. However The BSGP programming model does not directly match the GPU's stream processing architecture, and the BSGP compiler converts BSGP programs into efficient stream programs: (1) the compiler automatically adds context saving code making the barriers conform to the semantics and (2) due to the presence of shared variables across the super-steps (data dependencies between super-steps), the compiler uses a graph optimization to automatically allocate temporary streams to save local variable values and thus, for having a efficient codes, it minimizes the total number of temporary streams. The BSGP compiler generates C+CUDA codes. A current limitation is the inability to handle flow control across barriers such as **while** or **if** statements.

BSGP allows to: (1) thread creation, destruction (with operations **fork** and **kill**) and load balancing/reassignment of thread across the kernel units (the small units of computations) of a GPU; (2) remote variable access intrinsic for efficient communications between threads and (3) collective primitive operations (reduce, scan and sort). Fig 2.8 resumes the statements and primitives of BSGP. Here some details:

- A spawn statement executes a block of GPU code using the total number of threads as a parameter.
- **barrier(RANK REASSIGNED)** allows rank reassigning barriers. Since physical thread ranks cannot be changed within a kernel, a logical rank reassignment is performed by shuffling stored thread contexts at a barrier.
- In stream processing, some operations such as resource allocation and detailed kernel launch configuration are only available to the CPU (control processor); To address this issue, BSGP allow the control processor code to be inserted into BSGP source code as a **require** statement block. At run time, code inserted this way is executed before the containing super-step is launched.
- In case of independent super-steps, BSGP provides the **par** construct to let the programmer control the bundling behaviour by restructuring the code. The **par** construct specifies that all statements in the block are independent of each other; During compilation, the BSGP compiler aligns barriers in the statements and bundles the corresponding super-steps together. It is close to the superposition of BSML
- Create/destroy threads can improve load balancing as the application data is amplified/reduced. It is the main goal of **fork** and **kill** primitives.
- Using **Thread.split**, the rank is reassigned such that a thread with a false side has a smaller rank than a thread with a true side. Relative rank order is preserved among threads of the same side;
- In the same manner, **thread.sort** allows to reassigned thread on kernels depending on a specific order given by the array **key**

Thread manipulation:	
<code>spawn(int n) {...}</code>	Creates <i>n</i> threads on the GPU to execute the enclosed statements
<code>thread.rank</code>	The rank (id) of a thread
<code>thread.size</code>	The total number of threads
<code>barrier</code>	BSP barrier
<code>barrier(RANK_REASSIGNED)</code>	Barrier with load-balancing of alive threads
<code>require {...}</code>	memory of the GPU controller
<code>par {...}</code>	Reducing barriers
Collective data parallel primitives:	
<code>reduce(op, x)</code>	Reduction of <i>x</i> using associative operator <i>op</i>
<code>scan(op, x)</code>	Forward exclusive scan of <i>x</i> using associative; overwrites <i>x</i>
<code>compact(list, src, keep, flag)</code>	Stream compaction
<code>split(list, src, side, flag)</code>	Stream splitting
<code>sort_idx(data)</code>	Sorting
Supported rank adjusting primitives:	
<code>thread.kill(flag)</code>	Kill the calling thread if flag is true
<code>thread.fork(n)</code>	Fork <i>n</i> child threads
<code>thread.split(side)</code>	Split threads
<code>thread.sortby(key)</code>	Rank reassignment sorting.
DRMA operations:	
<code>thread.get</code>	as in BSPLIB (without the need of a push registering)
<code>thread.put</code>	as in BSPLIB

Figure 2.8. The BSGP statements and primitives.

Tools:	
<code>void bsp_begin()</code>	Start of SPMD code
<code>void bsp_end()</code>	End of SPMD code
<code>int bsp_nprocs()</code>	Returns the number of tasks
<code>int bsp_tid()</code>	Returns the task “id”
<code>void bsp_sync()</code>	BSP synchronization of tasks
BSMP primitives:	
<code>void bsp_send(int dest, int tag, Object v)</code>	Bulk sending of an object
<code>int bsp_get_tag()</code>	Get the tag of the object at the top of the queue
<code>Object bsp_move()</code>	Move the object of the queue

Figure 2.9. The main methods of JBSP.

2.2.4 BSP Programming in Java

There are some libraries for BSP programming using JAVA. The most known and promising is HAMA whereas the older, to our knowledge is JBSP. There are also libraries for multi-core architectures (BSPONMULTICORE) and for peer-to-peer/GRID environments (PUB-web and JMigBSP).

(a) Hama and JBSP

JBSP [144]. JBSP is mainly a BSPLIB for JAVA. It provides programmers with both explicit message passing and remote memory access routines. Due to the use of the JAVA’s object serialization, the author notes an overhead in communication compared to BSPLIB. The implementation relies to RMI and TCP/IP socket routines of the standard library of JAVA. A class that must perform BSP computations only need to extend `jbSPTask` and to overload the method `void run()`. Fig 2.9 resumes the routines. The library provides DRMA routines which are in fact BSMP routines but with a dedicated tag. We thus do not present them. BSMP is performed by sending a tagged object which can be serialized. Reception is performed by removing objects from the queue of received messages. As in the BSPLIB, there is no order for messages in the queue. Thus we can only have the tag of the message at the top of the queue.

Hama [246]. HAMA is an Apache project for a pure JAVA BSP computing framework on top of HDFS — **Hadoop Distributed File System**. Hadoop [[34]] is an open-source framework for distributed computing which is mainly a clone of the Google’s MapReduce framework. HAMA is thus a framework for cloud computing well adapted to search-engines and graph manipulations for data-intensive scientific applications such as machine learning, information retrieval, bioinformatics, and social network analysis. The HAMA’s run-time works using a master/peer paradigm where the master performs the barrier by sending messages to slaves and managing the input/output stream of data.

The way to create your own BSP computation is to create a class that extends the `BSP` class: `BSPPeer <K1, V1, K2, V2, M extends Writable> peer) throws ...` where *K1* (resp. *V1*) represent an input stream of

Tools:	
<code>Configuration getConfiguration()</code>	Get the Hadoop configuration
<code>void sync()</code>	Barrier
<code>int getPeerIndex()</code>	Return the index of this peer
<code>String[] getAllPeerNames()</code>	The names of all the peers
BSMP primitives:	
<code>M getCurrentMessage()</code>	A received message in the queue
<code>int getNumCurrentMessages()</code>	The number of messages in the queue
<code>void send(String peerName, M msg)</code>	Send a data to another peer
Stream I/O:	
<code>void write(K2 key, V2 value)</code>	Writes a key/value pair to the output collector
<code>boolean readNext(K1 key, V1 value)</code>	Test if there are no records to read anymore
<code>KeyValuePair<K1, V1> readNext()</code>	Reads the next key value pair

Figure 2.10. The main methods of HAMA.

keys (resp. values) and K2 (resp. V2) represent an output stream of keys (resp. values) and M messages that could be sent during the super-steps. Streams are like mappings as in the MapReduce paradigm. Then, in the main class, we need to initialise the distributed computation as an Hadoop program and which machine will be the master. Optionally, there are also `setup()` and `cleanup()` which will be called at the beginning of your computation, respectively at the end of the computation. For this, we just have to override these methods. Fig. 2.10 gives the main methods of the **BSP** class without the possible thrown exceptions. They are like the BSMP ones of the BSPLIB. There is also methods to write/read data in the streams.

As one of the goals of HAMA is the handling of large graphs, it also offers a special API for manipulating graphs *à la* PREGEL. We will present PREGEL thereafter.

(b) BSPonMultiCore [287]

BSPONMULTICORE is a JAVA library for BSP computing over multi-core architectures. It has been first developed for testing if BSP is applicable for shared-memory multi-core systems and can attain proper speedups. BSPONMULTICORE communication library is thus an object-oriented adaptation of BSPLIB, written in JAVA, and targeting only shared-memory systems. But stills a SPMD library. The implementation relies on the capacity of multi-threading of JAVA.

A generic BSP program is defined to be a class, **BSP_PROGRAM** having at least the functions defined in Fig 2.11. Any specific BSP algorithm is extended from this superclass, and must implement the two following virtual methods: (1) `main_part`, only executed by a single process to prepare data and can only call `bsp_begin(n)` for starting the parallel computation with `n` threads; (2) `parallel_part` which defines the code run in a SPMD fashion; once this method is reached, all BSP functions can be called, with the exception of `bsp_begin`.

BSPONMULTICORE allows sending or sharing an object by the use of an abstract class call **BSP_COMM**. Any sending or sharing objects will have to extend this class. Since an object has no meaning if not connected to a parallel program, the constructor must take a **BSP_PROGRAM** as parameter, thus linking the object and the parallel program it is used in. The methods are standard BSPLIB primitives with the use of a queue of received messages at the beginning of each super-step. Note the methods `bsp_unregister` frees up all memory it uses at all threads; this should never be called inside a super-step where the object is still used. As in the BSPLIB, there is also asynchronous (but unsafe) high-performance versions of the methods for benefit of the shared-memory of multi-cores architectures. However, the uses must use this routines carefully.

Each thread can also define a shared array. Such an array could be defined using a **BSP_REGISTER<T>** and standard JAVA array constructs. BSPONMULTICORE provides some subclasses of **BSP_COMM** dedicated for manipulating arrays of data. The DRMA methods have been modified to copy a number of length elements of the array in a single communication request (which is thus closer to the BSPLIB routines). The `getData` method, giving the programmer access to the underlying array. The main implementation of **BSP_COMM_ARR** is **BSP_ARRAY<T>** which makes available an array with elements of type `T` at each process. The type `T` must still support cloning. For efficiency (especially for numerical applications), when `T` is an `int` or a `double` primitive type, BSPONMULTICORE has defined the specialised **BSP_INT_ARRAY** and **BSP_DOUBLE_ARRAY** classes.

BSP_PROGRAM methods:	
virtual void main_part()	Sequential code
virtual void parallel_part()	Parallel code
public void start()	Starts the program
protected void bsp_begin(int n)	Starts the parallel execution
protected int bsp_nprocs()	Returns the number of threads executing this algorithm
protected int bsp_pid()	Returns the current, unique thread identification number
protected void bsp_sync()	Barrier
BSP_COMM methods:	
void bsp_put(int destination)	DRMA put of the object
void bsp_get(int source)	DRMA get from another object
void bsp_send(int destination)	BSMP sending of the object
int bsp_qsize()	Size of the queue
message bsp_move()	Moves an item from the queue
void bsp_unregister()	invalidate a shared variable
BSP_COMM_ARR adding methods:	
void bsp_put(int src, int src_offset, int dest_pid, int dest_offset, int length)	DRMA put
void bsp_get(int src, int src_pid, int src_offset, int dest_offset, int length)	DRMA get

Figure 2.11. The main methods of BSPONMULTICORE-JAVA.

(c) PUB-web [36]

PUB-web (PUBWCL) [[35]] is a BSP JAVA library and run-time environment that allows to execute tightly coupled, BSP algorithms on PCs distributed over the internet whose owners are willing to donate their unused computation power. PUB-web is realized as a peer-to-peer system and features migration and restoration of BSP processes executed on it. With the PUB-web client, any user can donate its unused computation power and also run its own parallel programs. As the unused computation power of the participating PCs in the web is unpredictable, PUB-web uses a dynamic strategy for load balancing the computations across the PCs. BSP-web implementation uses the JAVA-go RMI library for communicated objects and processes in a secure fashion. JAVA-go extends JAVA with these features: (1) migrations are performed using the keyword **go** and (2) all methods, inside which a migration may take place, have to be declared migratory.

Using PUB-web, in order to write a BSP and migratable program, the interface `BSPMigratableProgram` has to be implemented which means that the main method of the main class of the program has this signature: migratory **void** bspMain(`BSPMigratable bspLib`, `String[] args`) **throws** `AbortedException`, `NotifyGone`. Fig 2.12 gives the BSP routines. They are traditional BSMP send/received routines. In migratable programs, there is also a method available which may be called to mark additional points inside long super-steps where a migration is safe, *i.e.* no open files *etc.*.

One single process receiving little computation power can slow down the execution of the whole BSP program due to the barrier synchronization. Migrating these “slow” BSP processes can therefore significantly improve the execution time of the BSP program. In [36] the authors present and benchmark different heuristics for load-balancing/migrate processes.

(d) JMigBSP [75, 140]

jMigBSP is also a JAVA programming library *à la* BSPLIB but which offers object (threads) rescheduling. It is implemented using the ProActive framework [[36]]. JMigBSP was designed to work on grid computing environments in two different ways: (1) by using migration directives on the application code directly and (2) through automatic load balancing at middleware level. A distributed object must extend the `jMigBSP` class. In this way, the user can override the migration for its own purpose. The initial one allows migrating the caller object to a remote host. JMigBSP proposes automatic load balancing of objects at each barrier with a potential of migration function. For this, the user must override the `bsp_sync` with this code:

```
public void bsp_sync(){
    double[] vec_steps = new double[MAX_SUPERSTEPS];
    computeSuperstepTime(vec_steps);
    if (isSuperstepRescheduling()) {
        next_call = computeBalance(vec_steps);
        exchangeDataAmongSetManagers(computePM());
        machine = willMigrate();
        if (machine != null) bsp_migrate(machine);
    }
}
```

But he can also choose its own heuristics for migration. Fig 2.13 shows the primitives.

Facilities:	
int getPID ()	“Id” of the processes
int getNumberOfProcessors ()	The number of processes
void sync()	Barrier
migratory void syncMig()	Barrier with load-balancing (migration of threads)
migratory boolean mayMigrate()	Test if migration is possible
BSMP primitives:	
void send(int to, Serializable msg)	Sending a message
void send(int pidLow, int pidHigh, Serializable msg)	Broadcasting a message to a subset of processes
void send(int [] pids, Serializable msg)	Idem
int getNumberOfMessages()	The number of messages in the queue
Message getMessage(int index)	Get the nth received message
Message[] getAllMessages()	Getting all received messages
Message findMessage(int src, int index)	Get the nth message sent by a process
Message[] findAllMessages(int src)	Getting all received messages sent by a process
Serializable Message.getContent()	Getting the send object of a message
int Message.getSource()	Know which processor sent the message
Stream I/O:	
void printStdOut(String line)	Print on the stdout of the user client
void printStdErr(String line)	Print an error
InputStream getResourceAsStream(String name)	Access data from user's files

Figure 2.12. The main routines of PUB-web.

Tools:	
void sync()	Barrier
void run(int p)	Launch the BSP computation with p objects
int bsp_nprocs()	The number of objects
in bsp_pid()	The “id” of the object
BSMP primitives:	
void bsp_put(Object o, int dest)	Sending a value
Object bsp_get(int id)	Reading a sending value from object “id”
Migration:	
void bsp_migrate(int dest)	Migrate the caller object to another processor

Figure 2.13. The main methods of jMigBSP.

2.2.5 Other Libraries

(a) NestStep [176]

NestStep is a set of extensions to existing imperative programming languages like JAVA or C for programming BSP algorithms. It adds language constructs and run-time support for the explicit control of parallel program execution and the sharing of program objects: by default, shared scalar variables and objects are replicated across the processors; within a super-step only the local copies of shared variables are modified and the changes are committed to all remote copies at the end of the super-step. Also, NestStep introduces static and dynamic nesting of super-steps.

NestStep mainly provides the `step{...}` statement to denotes a super-step that is executed by entire groups of processors. Thus, a step statement always expects all processors of the current group to arrive at this program point — which is in the charge of the programmer. It implies a barrier at the beginning and the end of statement for performing communications (the combine phase) even if the implementation tries to avoid duplicate barriers. A step statement with parameters deactivates and splits the current group into disjoint subgroups that execute the step body independently of each other — in a nested manner. The parent group is reactivated and resumes when all subgroups have finished execution of the step body. For example, `step<k, #>=1>{...}` splits the current group into k subgroups where at least there one processor in each subgroup. For each group the run-time system holds on each processor its own class, group object. In particular, it contains the group size, the group “id”, and the processor’s rank within the group.

By default, objects (or basic variables) are private. Thus, they exist on every processor, executing their declarations. Sharing is explicitly specified by a type qualifier `sh type x` at declaration or for objects allocation with one of the following parameters:

- `sh<0>` declares a variable x of arbitrary **type** where the copy of the group leader (the processor with rank 0) is broadcast at the combine phase. All other local copies are ignored even if they have been written to.
- `sh<?>` denotes that an arbitrary updated copy is chosen and broadcast. If only one processor

updates the variable in a step, this is deterministic.

- `sh<=>`, the programmer asserts that he always assigned the same value to `x` on all processors of the declaring group; thus, combining is not necessary for `x`.

There is also a declaration for distributed array (block or cyclic distribution) in order to save space and to exploit locality where each access to a non-local element is automatically resolved by a blocking point-to-point communication with its owner, in order to guarantee a sequential consistency. The implementation uses TCP/IP or MPI.

(b) BSML, BSP++ and BSP-Python

BSP Programming in ML [129]. BSML is an extension of ML to code BSP algorithms. BSML uses a *small set of primitives* (over a parallel data structure, called parallel vector) which are currently implemented as a parallel library [[37]] for the ML programming language OBJECTIVE CAML, *i.e.* OCAML [[38]]. All communications in BSML are collective and deadlocks are avoided by a strict distinction between local and global computation. Two features of BSML are (1) its *deterministic* semantics; and (2) the property is ensured that for any BSML program, the sequential simulation (a top-level) gives the same results than the truly parallel run. BSML programs can mostly be read as OCAML ones, in particular, the execution order should not seem unexpected to a programmer used to OCAML, even though the program is parallel. That allows the parallelisation to be done incrementally from a sequential program. A parallel vector has type `'a par` and embeds `p` values of any type `'a` at each of the `p` different processors in the parallel machine. The nesting of parallel vectors is not allowed and a type system prevents this forbidden use of vectors.

We use the following notation to describe a parallel vector: $\langle x_0, x_1, \dots, x_{p-1} \rangle$ where `p` is the (constant) number of processors throughout the execution of the program. It can be accessed in BSML using the integer constant `bsp_p` — the other BSP parameters are also accessible as float values through constants `bsp_g`, `bsp_l` and `bsp_r`. We distinguish a parallel vector from an usual array of size `p` because the different values, that will be called *local*, are blind from each other; it is only possible to access the local value x_i in two cases: locally, on processor i (by the use of a specific syntax), or after some communications. In this way, the vector $\langle x_0, x_1, \dots, x_{p-1} \rangle$ holds the value x_i at processor i . Since a BSML program deals with a whole parallel machine and individual processors at the same time, a distinction between the levels of execution that take place will be needed:

- **Replicated** execution is the default. Code that does not involve BSML's parallel vectors is run by the parallel machine as it would be by a single processor. Replicated code is executed at the same time by every processor, and leads to the same result everywhere.
- **Local** execution is what happens inside parallel vectors, on each of their components: The processor uses its local data to do computation that may be different from the other's.
- **Global** execution concerns the set of all processors together, but as a whole and not as a single processor. Typical example is the use of communication primitives.

The distinction between local and replicated is strict. Hence, the replicated code can not depend on local information, and remains replicated. We say that we lose replicated consistency if an expression outside of a parallel vector holds different values at different processors. Fig. 2.14 subsumes the use of these primitives. Informally, it works as follows.

Let $\ll x \gg$ be the vector holding `x` everywhere — on each processor. The $\ll \gg$ indicates that we enter a local section and pass to the local level. Replicated information is available inside the vector. To access to local information, we add the syntax `x` to open the vector `x` and get the local value it contains, which can obviously be used only within local sections. The local *pid* can be accessed with `pid`. Thus $\ll \text{pid} \gg$ will contain i on the processor i . The **proj** primitive is the only way to extract a local value from a vector. Given a parallel vector, it returns a function such that applied to the *pid* of a processor, it returns the value of the vector at this processor. **proj** performs communications to make local results available globally within the returned function. Hence it establishes a meeting point for all processors and, in BSP terms, ends the current super-step. The **put** primitive allows any local value to be transferred to any other processor. As such, it is more flexible than **proj**. It is as well synchronous, and ends the current super-step. The parameter of **put** is a vector that, at each processor, holds a function of type $(\text{int} \rightarrow 'a)$ returning the data to be sent to processor j when applied to j . The result of **put** is another vector of functions: At a processor j the function, when applied to i , yields the value *received from processor i by processor j* . The last primitive allows the evaluation of two BSML expressions E_1 and E_2 as “super-threads”. From the programmer's point of view, the semantics of the **super** is the same as pairing *i.e.*, building the pair (E_1, E_2) but the evaluation of **super** $E_1 E_2$ is less costly because it merges

primitive	type	informal semantics
$\ll e \gg$	$t \text{ par (if } e : t)$	$\langle e, \dots, e \rangle$
pid (within a vector)	int	i on processor i
v (within a vector)	t (if $v : t \text{ par}$)	v_i on processor i (if $v = \langle v_0, \dots, v_{p-1} \rangle$)
proj	$'a \text{ par} \rightarrow (\text{int} \rightarrow 'a)$	$\langle x_0, \dots, x_{p-1} \rangle \mapsto (\text{fun } i \rightarrow x_i)$
put	$(\text{int} \rightarrow 'a) \text{ par} \rightarrow (\text{int} \rightarrow 'a) \text{ par}$	$\langle f_0, \dots, f_{p-1} \rangle \mapsto \langle (\text{fun } i \rightarrow f_i 0), \dots, (\text{fun } i \rightarrow f_i (p-1)) \rangle$
super	$(\text{unit} \rightarrow 'a) \rightarrow (\text{unit} \rightarrow 'a) \rightarrow 'a * b$	$f_a \mapsto f_b \mapsto (f_a (), f_b ())$

Figure 2.14. Summary of the BSML primitives.

the communication and the synchronization phases of E_1 and E_2 : using the superposition, the phases of asynchronous computation of E_1 and E_2 are run; then the communication phase of E_1 is merged with that of E_2 ; the messages are obtained by simple concatenation of the messages of each super-thread and only one barrier occurs; if the evaluation of E_1 needs more super-steps than that of E_2 then the evaluation of E_1 continues — and *vice versa*. BSML is implemented using MPI or TCP/IP.

The C++ Version: BSP++ [148]. The BSP++ interface is a template/object-oriented implementation of BSML for C++. As BSML, it uses parallel vectors which are now represented as a C++ class. Functional programming is done using templates. Any BSP computation/section must be started with the use of the `BSP_SECTION(){...}` macro where ... is a code where is possible to define parallel vectors and using the following routines, close to the ones of BSML:

- `par<T>` encapsulates the concept of parallel vector. This class can be built from a large selection of C++ constructions ranging from C-style array, C++ standard container, function or lambda-function. For example, `bsp::par<vector<double>>> v` creates a parallel vector of arrays of **double**. In this way, each processor contains an array of **double**. Local access to a parallel vector data is done through the traditional C++ dereferencing operators.
- `pid_` is a global parallel vector that evaluates to the “pid” of current processors.
- `sync` performs an explicit synchronization and ends the current super-step.
- `proj` returns a function object that maps the identifier of a processor to the contents of the parallel vector held by this processor and ends the current super-step.
- `put` allows the local values to be transferred to any other processor and ends the current super-step. The return of `put` is a parallel vector of function object of type `T(int)` that returns the data received from processor i when applied to i .

A special feature of BSP++ is the ability to nested parallel vector for hybrid architectures: the possibility to call a `BSP_SECTION` within a `BSP_SECTION`. The implementation thus uses MPI, Open-MP or CUDA depending of the architecture.

The Python Version: BSP-Python [159]. BSP-PYTHON is an adaptation of the BSML primitives for PYTHON. The implementation relies of MPI or of BSPLIB. Most scientists did not consider PYTHON’s programs sufficiently fast for number crunching. However, what make this language interesting in such applications was the idea of multiple language coding: the usually small parts of the code in which most of the CPU time is spent are written in a compiled language, usually FORTRAN, C, or C++, whereas the bulk of the code can be written in a high-level language. There are several ways to create global object (parallel vectors), corresponding to their typical uses:

- `ParConstant(v)`; a global object represents the constant v that is available on all processors.
- `ParData(lambda pid, nprocs: v)`; defines the local representation as a function of the processor number and the total number of processors to v .
- `ParSequence(v)`; distributes its argument v (which must be a PYTHON sequence) over the processors as evenly as possible.

BSP-PYTHON communication operations are defined as methods on global objects. An immediate consequence is that no communication is possible within local functions or methods of local classes. BSP-PYTHON propose a set of communication patterns implemented as methods in all of the global data classes. For example, we have:

- **put(pid list)** which sends the local value to all processors in **pid list** (a global object whose local value is a list of processor identifiers) and returns a global data object whose local value is a list of all the values received from other processors, in unspecified order;
- **fullExchange()** which sends the local value of each processor to all other processors and returns a global data object whose local value is a list of all the received values, in unspecified order;
- **accumulate(operator, zero)** which performs an accumulation with **operator** over the local values of all processors using **zero** as initial value and the result is a global data object whose local value on each processor is the reduction of the values from all processors with lower or equal number.

In the communication operations described until now, it is always the local value of the global data type that is sent, either to one or to several receiving processors. In some situations, it is necessary to send different values to different processors. In principle this can be achieved by a series of put operations, but a special communication operation is both more efficient and allows more readable code. For this purpose, BSP-PYTHON provides a specialized global data type called **ParMessages**. Its local values are lists (or sets) of data-processor identifier pairs. The method **exchange()** sends each data item to the corresponding processor and returns another **ParMessages** object storing the received data items stored in a list in an unspecified order, thus each processor can easily iterate this.

(c) BSP Large Data/Graph Manipulation: BSP over the Cloud

With the new opportunities brought by cloud computing, large-scale applications that were restricted before to large research centers can now be executed with modest investments on infrastructure and maintenance. The typical cloud applications generates and process huge quantities of data. One of the main challenges is now how to deal with problems on how to store, manipulate and analyse this huge amount of data. If the application presents a good data parallelism, MapReduce framework achieves very good performance. If not, some recent research have proposed more sophisticated ways to achieve better performances on cloud computing platforms mainly using the BSP model [132, 198, 236]. We here shortly present them.

Google's Pregel [205]. PREGEL is a powerful C++ dedicated parallel language to express graph algorithms and to run them efficiently on several clusters. The approach centers to the computations on the vertices of the graph. Graph algorithms are expressed by the use of specific BSP primitives. The implementation relies of low level Google's libraries with checkpoint at each barrier enable fault tolerant.

Each vertex of the graph has a unique "id", an associated value and a list of output weighted edges. The computation is organized using a master/slave architecture and the input data is arbitrarily partitioned and stored on a distributed storage system. The PREGEL system maintains the vertices and edges stored on the node that will perform the computation but the work is migrated to where the data are stored. On each super-step, each worker node invoke a procedure **Compute()** of each active vertex that is under its control. This procedure is responsible for the execution of the business rule of the algorithm and is allowed, among other actions, to invoke other methods, compute new values for the vertex, add or remove vertices and edges, and send messages to other vertices. These messages are exchanged directly among the vertices, even if the vertices are being executed on different machines of the platform. The messages are sent asynchronously in order to allow the overlapping of computation and communication, but are delivered to the destination vertex only on the beginning of the next superstep. If a vertex declares that all its processing was done, it sends a message informing all the other nodes and deactivates itself. The master node stops the execution of the application after receiving this message from all the participants.

An important feature required by graph algorithms is the ability of change the topology of the graph. For example, a clustering algorithm might replace each cluster with a single vertex, and a minimum spanning tree algorithm might keep only the tree edges. In order to avoid conflicting changes on the topology (for instance, multiple vertices could issue a command to create a vertex V with different initial values), Pregel uses two mechanisms to achieve determinism: (1) a partial ordering: removals are performed first, with edge removal before vertex removal and additions follow removals, with vertex addition before edge addition; (2) a handlers for user defined conflict-resolution policies. Any change on the topology is only performed on the next super-step, before the invocation of the **Compute** procedure. PREGEL is considered as the main reference of graph applications on the cloud. And it is use by Google for their own applications. Due its proprietary nature, it is not possible having the code source nor the full API. Fig. 2.15 gives the available methods of the class **Vertex**. For example, a Pregel implementation of the traditional Google's page-ranking is:

Facilities:		
const string& vertex_id() const	The “id” of a vertex	
int64 superstep() const	The total number of threads	
const VertexValue& GetValue()	Get the value of a vertex	
const int NumVertices()	Get the number of vertices	
const int NumEdges()	Get the number of edges	
VertexValue* MutableValue()	Get a pointer to a vertex enable changing the value	
BSP computations:		
virtual void Compute(MessageIterator* msgs);	Local computations	
OutEdgeIterator GetOutEdgeIterator()	Get the output edges of a vertex	
void SendMessageTo(const string& dest_vertex, const MessageValue& message)	Send a message (vertex to vertex)	
void VoteToHalt()	Barrier	

Figure 2.15. The PREGEL’s vertex methods.

```

class PageRankVertex: public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
        }
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        }
        else {VoteToHalt();}
    }
}

```

The example works as follow. The graph is initialized so that in superstep 0, the value of each vertex is $1/\text{NumVertices}()$. In each of the first super-steps, each vertex sends along each outgoing edge its tentative page-rank divided by the number of outgoing edges. Each vertex sums up the values arriving on messages into sum and sets its own tentative page-rank. Note that, a truly page-rank algorithm would run until a convergence was achieve.

Other frameworks. Several open source initiatives of PREGEL exist such as Giraph [[³⁹]] and GoldenOrb [[⁴⁰]]. Their implementations relies on Hadoop and their APIs are for JAVA.

HAMA has also a dedicated API for manipulating graph. It is mainly an implementation of some typical algorithms and parallel iterators over the distributed vertices or edges. The papers [132, 198, 236] give many other graph libraries but we have currently not yet had time to study them. And most of them are still in their beta version.

2.2.6 Which Routines for BSP-Why

After having seen all kinds of routines in the whole spectrum of libraries, we need to choose the most relevant ones to be treated in our tool BSP-WHY. One of the short term objectives of this research is to be able to prove with BSP-WHY: (1) MPI programs that are well structured and (2) most of the BSP programs from the above libraries. For (1), this can be done by working with MPI programs that only use collective operations. These ensure that the processors will synchronize to exchange their data, in a way similar to BSP. However, collective operations in MPI and some BSP libraries (*e.g.* the PUB) come with the notion of groups and communicators. It is thus necessary to extend our model to be able to work with programs that synchronise over a restricted group of processors. In BSP-WHY, we chose to take into account the approaches of both MPI and the PUB library to subgroup synchronisation.

Both BSPLIB and PUB have high performance versions of their primitives which are using unbuffered communications (the buffer is sent, not a copy) in a asynchronous fashion (at an unspecified time before the synchronisation) which means that if the buffer is modified, it is unspecified what will be sent. Those primitives are not studied here because they are (1) too complex; (2) and with unspecified interaction with traditional BSP primitives; (3) and too non-deterministic. For communicating routines, we have chosen:

- BSMP primitives *à la* PUB for just a historical reason (formal study of the semantics in [119, 124]). BSMP primitives using a queue for reading the received messages (in BSPLIB and JAVA libraries)

can trivially be added.

- DRMA primitives *à la* BSPLIB. The main reason is that the global registration seems cleaner than the constraint to have an order using the **push** primitives. Note that this method could be simulated by checking if the registration has the expected data.
- User defined collective operations. The user can define its own collective operations and thus its own schemes of communication. For example, in Chapter 4, we will use a kind of all-to-all collective operation.

BSML like and NestStep primitives have not been really taken into account. However, using the analysis (the block decomposition) of BSP-WHY, we will meet again the parallel vectors of BSML and the “steps” of NestStep because those structures are mainly used to distinct what are local computations (only for processors) and what is for the whole parallel machine. Also, BSP-WHY currently does not provide primitives for thread migration nor oblivious synchronisation nor nested computation (hybrid architectures). It is for future work.

2.3 Common Errors Introduced in BSP Programming

Proving the correct execution of parallel programs is actually significantly more difficult than proving that of sequential programs, because of several reasons. Let us talk a bit about the main problems that arise from the parallelisation of the computations. We thus try to enumerate the possible bugs that can be introduced by a faulty use of parallel/distributed routines. We then show how this possible errors can occur in BSP like computations.

2.3.1 Common Errors and Bugs

(a) Deadlocks

The possibility of a deadlock is probably the most well-known problem when dealing with the parallelism. A deadlock is a situation in which two or more processes are each waiting for the other to terminate, hence remaining in a blocked state. Let us give a basic MPI example of a deadlock situation. We consider a parallel machine with two processors, with the following parallel communication calls:

Process 0	Process 1
v' = Recv(1)	v' = Recv(0)
Send(1, v)	Send(0, v)

where **Send(i, v)** is for sending a value **v** to processor **i** and **Recv(j)** is for receiving a value from processor **j**. In this example, we assume that the **Recv** and **Send** procedures are *blocking*, meaning they await a result before the remaining of the program is executed. The consequence is that when executing this code, the processor 0 will be waiting for data from the processor 1, while the processor 1 will be waiting for data from the processor 0. The **Send** instruction can never be executed, and the two processes will wait in the deadlock state forever. A correct program could be for instance:

Process 0	Process 1
v' = Recv(1)	Send(0, v)
Send(1, v)	v' = Recv(0)

Here the **Send** and **Recv** operation are matched, so no deadlock occurs. Various models have been proposed to reduce the risk of deadlocks. In particular, using BSP or MPI collective operators or skeletons or high-level patterns, significantly decreases the risk. For instance, our example could be rewritten using the MPI **sendrecv** routine, which combines the send and receive operations.

Process 0	Process 1
v' = Sendrecv(1, v)	v' = Sendrecv(0, v)

(b) Interleaving of Computations (Data Race Condition)

When executing a sequential program, one can immediately see when reading the code the order in which the instructions will be executed. However, when analysing a parallel program where processors share data, one must take into account that some processors might progress faster than others. Let us give this simple example:

Process 0	Process 1
<code>x=0</code>	<code>x=0</code>
<code>x=x+1</code>	<code>x=x+1</code>
<code>y=hpget(x,1)</code>	<code>y=hpget(x,0)</code>
<code>print(y)</code>	<code>print(y)</code>

where `hpget(x,i)` returns (read) “immediately” the value of `x` at processor `i` — `hpget` is the high-performance DRMA communication routine explained below. In this example, every processor executes the same sequence of instructions. This is consistent with the SPMD form of parallelism, which will be the focus of our study. We can imagine that the expected behaviour of this program was to display the results of the computation of both processors. Indeed, if the processors execute always the same instruction simultaneously, they would both store in `x` the result of the computation, then get the result from the other processor, and then display both `y`. However, usually when looking at parallel programs it is impossible to ensure that all processors execute the same statement at the same time. In this example, if for instance the processor 0 finishes the computations faster than the processor 1, he might execute the next instruction, `y=hpget(x,1)`, while the second processor is still computing, and his `x` is still 0. The output on the processor 0 would then be 0, and 1 for the other.

In this example the problem was fairly obvious. However, such a bug can be more difficult to spot. The behaviour of such programs is often highly non-deterministic, which makes the bug even harder to anticipate and correct. Note that this problem does not occur using pure BSP routines since all communications are performed during the super-step’s phase of communication. Furthermore, this kind of bugs is generally the subject of concurrent analysis but can also appear in MPI programs with the use of asynchronous “window” — DRMA routines, also call “one-sided communication”.

2.3.2 These Errors in BSP Programs

(a) Deadlocks

Even if the BSP model is deadlock and data-race free in its principle and simplifies the writing of parallel codes, many errors, even deadlocks, can appear in BSP programs (or at least in MPI programs written in BSP style) in addition of the classical sequential programming errors (buffer and integer overflows, non terminating loops, *etc.*). Take for example the following C codes using BSP or MPI:

```
(* ***** BSP ***** *)
if (bsp_pid()==0) bsp_sync();
else asynchronous_computation();

(* ***** MPI ***** *)
comm=MPI_COMM_WORLD;
MPI_Comm_rank(comm,&me);
if (me==0) MPI_Barrier(comm);
else asynchronous_computation();
```

here, a deadlock can occur and the parallel machine would crash on some architectures. The same problem arises when: (1) one processor performs an oblivious synchronisation while other processors perform a global barrier; or (2) when a processor performs an oblivious synchronisation, waiting for instance for `n` messages and only `n-1` messages are received. Communication can also generate errors. In addition to sending a value to an “`id>nprocs`”, the reading of a bad number of values can induce bugs. For instance:

```
(* ***** BSP ***** *)
int me=bsp_pid();
(* All processors except 0 send a message to 0 *)
if (me!=0) bsp_send(0,(void*)me,sizeof(int));
bsp_sync();
(* processor 0 reads these messages *)
if (me==0)
    for(int i=0;i<=bsp_nprocs()-1;i++)
        y+=(int)bspmsg_data(bsp_findmsg(i,0));

(* ***** MPI ***** *)

comm=MPI_COMM_WORLD;
MPI_Comm_rank(comm,&me);
(* All processors except 0 send a message to 0 *)
if (me!=0) mpi_isend(0,(MPI_INT)me,MPI_INT,comm);
mpi_barrier(comm);
(* processor 0 reads these messages *)
```



```

if (me==0)
    for(int i=0;i<=mpi_comm_size()-1;i++)
        mpi_ircv(i,0,z,comm);
    y+=z

```

processor 0 will read a message from itself too, which does not exist. A segfault will occur for both the BSP program and a deadlock can occur for the MPI one.

(b) Out-of-bound and Non-deterministic Communication

Communication can be done outside bounds of buffers and thus bug. For instance:

```

(* ***** BSP ***** *)
int x[bsp_nprocs()];
bsp_push_reg((void *)x,bsp_nprocs()*sizeof(int));
bsp_sync();
(* All processors except 0 write to the x of processor 0 *)
if (bsp_pid()!=0)
    bsp_put(0,(void *)x,(void *)x,bsp_pid()+1,1*sizeof(int));
bsp_sync();

(* ***** MPI ***** *)

```

the last processor would write over the limits of x of processor 0 and a segfault will occur.

Subgroup synchronisation can also be a source of potential bugs. For example, using the PUB, the creation of subgroups can be nested, but they must be created and destroyed in a LIFO order. Otherwise a deadlock can occur or we can have unspecified (non portable) and non-deterministic results. Moreover, for both BSP and MPI, the subgroups must be disjoint when subgroups are created. Using MPI, a deadlock can occur when a processor is within two groups and each one performs a barrier. For instance:

```

(* ***** MPI ***** *)
MPI_Comm comm1,comm2;
MPI_Comm_rank(MPI_COMM_WORLD, &me);
color1 = me % 2;
color2 = me % 3;
MPI_Comm_split(MPI_COMM_WORLD, color1, me, &comm1);
MPI_Comm_split(MPI_COMM_WORLD, color2, me, &comm2);
if (me % 2==0) MPI_Barrier(comm1) else MPI_Barrier(comm2);

```

in this example, any processor that has an id (of MPI_COMM_WORLD) multiple of 2 and 3 (*e.g.* 6) is within the two subgroups (subcommunicators) comm1 and comm2. In this way, some processors would perform the barrier of communicator comm1, others that of communicator comm2. Those are within the two subcommunicators would generate a deadlock. Note that this problem can not happen using PUB because there is a nested decomposition and at each level, the routine `bsp_partition` tests if there is truly a creation of disjoint subgroups.

Our last example is not really an error since it does not crash the machine but it gives non-deterministic results and thus can disturb the meaning of a program. What happens when there are two distant writings (using the put primitive) of two different processors over the same area of memory? For instance:

```

int x[bsp_nprocs()];
bsp_push_reg((void *)x,bsp_nprocs()*sizeof(int));
bsp_sync();
bsp_put(0,(void *)x,(void *)x,0,1*sizeof(int));
bsp_sync();

```

Two solutions are possible. First, forbid this case by adding logical conditions for distant writings. Second, suppose an order of writing for the processors. We have currently chosen the second solution since we suppose a deterministic semantics of BSP programs. However, changing to the first solution would be trivial.

Many other errors can be cited: forgetting to register a variable, bad size/number of messages, forgetting a barrier and all errors with pointers of the messages that one can imagine. Proving that programs do not have these incorrect uses of the routines would increase confidence in the codes. This will be even better if you also formally prove the behaviour of your BSP programs — at least, the more important parts of your code.

2.4 Related Works

There are really many parallel languages or parallel extensions of a sequential language (functional, iterative, object-oriented, *etc.*). It would be too long to list all of them. We chose to point out those that were the most important to our mind. Notice that, except in [200], there is a lack of comparisons between parallel languages. But it is hard to compare them since many parameters have to be taken into account: efficiency, scalability, expressiveness, *etc.*

2.4.1 Parallel Libraries and Imperative Programming

There are many parallel libraries but the two most frequently used are certainly MPI and OPEN-MP. Both are working under C/FORTRAN — even if we can find bindings for JAVA, OCAML and other languages. MPI contains many operations such as asynchronous sending values, collective operations, parallel I/O accesses, *etc.* for distributed architectures. OPEN-MP has been designed for working on shared-memory architectures and allows the creation/synchronisation of threads. It mainly works using annotations in the code, *e.g.* indicates which loop can be achieved in parallel.

The “parallel loop paradigm” (where the compiler automatically distributes the computations) has given rise to many extensions of C and JAVA. We can cite SPLIT-C [74] [[41]], TITANIUM and UNIFIED PARALLEL C [95] [[42]], CILK [[43]], FORTRESS [[44]], *etc.* Mainly, the management of the processes and the implementation are the only differences for these languages.

ZPL [50] [[45]] (and its ancestor ORCA [11] [[46]]) is an array programming language. The programmer gives instructions (with specific primitives) to manipulate the arrays (*e.g.* to copy a value in the cells above) and the compiler automatically distributes the computations and the data. It is close to SAC but it is not a data-parallel language. This kind of operations can also be found in CO-ARRAY FORTRAN [[47]].

CILK [31] is an extension of C where the programmer can spawn a procedure like another thread of execution. Communications are performed using shared variables and synchronisations. OCCAM-PI [[48]] was designed following the Communicating Sequential Processes (CSP) process algebra and the π -calculus. The business code can be either HASKELL or C code. They appear in π -terms using a specific construction.

CHARM++ [173] [[49]] is an extension of C++ based on a migratable-objects programming model. The programmer decomposes the program into a large number of “chare”, which can migrate and interact with each other via asynchronous method invocations. The system maintains a “work-pool” consisting of seeds for new chares, and messages for existing chares.

All these languages/libraries have generally no semantics or are too asynchronous for the study of this thesis. However, they are still used by programmers. Simulated this kind of work is for future work.

2.4.2 Other Parallel Paradigms

(a) Parallel Functional Languages

Two nice introductions (with many references) to parallel functional programming can be found in [149] and [151]. They have been used as a basis for the following classification — with some updates.

The authors explain three reasons to use functional languages in the parallel programming. First, they ease the partition of a parallel program (task decomposition). Second, most of them are deadlock free: the value is independent of the evaluation order that is chosen and thus any program that delivers a value when run sequentially will deliver the same value when run in parallel. Third, they have a straightforward semantics which is great for debugging. Testing and debugging can be done on a sequential machine: functional programs have the same semantic value when evaluated in parallel as when evaluated sequentially.

The main drawbacks are: (1) their efficiency; and (2) the lack of libraries and tools; and (3) perhaps too hard to use for many programmers. It is still an old debate and it is not the subject of this thesis. BSP-WHY-ML is mostly an imperative language but there is some functional features.

(b) Algorithmic Skeletons and their Implementation

As described previously, skeletons are patterns of parallel computations. They can be seen as high-order functions that provide parallelism. They fall into the category of functional extensions following their semantics [1]. Most skeleton libraries extend a language (mostly JAVA, HASKELL, ML, C/C++) to provide those high level primitives. Currently, the most known library is the Google’s MAPREDUCE [80]. It is a framework to process embarrassingly parallel problems across huge data sets — originally for the page-ranking algorithm. Different implementations for JAVA or C exist. But only two skeletons are provided

which limit expressiveness. [98] provides a set of flow skeletons for C++. Templates are used to provide an efficient compilation of the programs: for each program, a graph of communicating processes is generated and is then transformed into a classical MPI program.

For JAVA, many libraries exist such as the ones of [3, 194] and [2]. The latter has been extended for multi-core architectures [60]. A study of how to type JAVA's skeletons has been done in [48]. The authors note that some libraries of data-flow skeletons use a unique generic type for the data (even if it is an integer or a string), which can cause a clash of the JVM. They explain how to avoid this problem, using a simple type system.

[64] describes how to add skeletons in MPI (the ESKEL library), as well as some experiments. It also gives convincing and pragmatic arguments to mix message passing libraries and skeleton programming. We think that using OCAML in parallel programming (HPC applications) is not a wrong choice, since the generated code is often very competitive with the C counterparts. Some benchmarks of an OCAML implementation of data-flow skeletons for a numerical problem are described in [62]. But, the implementation currently sucks to TCP/IP. The first study of how to integrate flow skeletons and data-parallel ones was in [182]. Implementing skeletons using a BSP library was first done in [289] and an application of the BSP cost prediction was also done in [152]. It seems possible to prove a BSP implementation of skeletons using BSP-WHY. It is for future work.

(c) Data-parallel Languages

To our knowledge, the first important *data-parallel* functional language was NESL [29] [[50]]. This language allows to create specific arrays and nested computations within these arrays. The abstract machine (or the compiler) is responsible for the distribution of the data and computations over the available processors. Two extensions of NESL for ML programming are NEPAL [[51]] and MANTICORE [[52]] [106]; the latter is clearly a mix between NESL and CONCURRENT-ML [238] [[53]] — CONCURRENT-ML is a language of the creation of asynchronous threads and send/received messages in ML.

We can also cite the following data-parallel languages. First, SAC (Single Assignment C) [142] [[54]] is a lazy functional language (with a syntax close to C) for array processing. Some higher-order operations on multi-dimensional arrays are provided and the compiler is responsible for generating an efficient parallel code. An extension of the famous lazy functional language HASKELL is data-parallel HASKELL [49] [[55]]. It allows to create data arrays that are distributed across the processors. And some specific operations permit to manipulate them. The main drawback of these languages is that cost analysis (for comparing algorithms) is hard to do since the system is responsible for the data distribution.

BSP-WHY is also a kind of data-parallel language. But we are currently not taken into account all their features. It is for future work.

3 The BSP-Why Tool

This chapter subsumes the work of [110].

Contents

3.1	The BSP-Why-ML Intermediate Language	41
3.1.1	Syntax of BSP-Why	42
3.1.2	The Parallel Memory Model in Why	46
3.2	Transformation of BSP-Why-ML Programs: the Inner Working	48
3.2.1	Generalities	48
3.2.2	Identification of Sequential Blocks	49
3.2.3	Block Tree Transformation	53
3.2.4	Translation of Sequential Blocks	54
3.2.5	Translation of Logic Assertions	58
3.2.6	Dealing with Exceptions	59
3.3	Dealing with Subgroup Synchronisation	61
3.3.1	Subgroup Definition	62
3.3.2	Transformation of Programs with the Subgroup Synchronisation	64
3.4	Related Work	67
3.4.1	Other Verification Condition Generators	67
3.4.2	Concurrent (Shared Memory) Programs	67
3.4.3	Distributed and MPI Programs	68
3.4.4	Proof of BSP Programs	69

In this chapter, we will present how our BSP-WHY tool works. In the first section, we will show the syntax of the language and of a core-calculus, the informal semantics and simple examples which will be used along this chapter. We will explain how the tool allows to ensure the safety and correctness of parallel (BSP) programs, mainly by transforming parallel programs (BSP-WHY-ML ones) into equivalent sequential ones (WHY-ML ones) that can be checked by the WHY tool.

Next, in the second section, we will explain the inner working of the BSP-WHY tool, and detail the transformation that is made to create a sequential WHY-ML program from the BSP-WHY-ML input. We thus give formally this transformation for a core-calculus.

Finally, we explain how to deal with subgroup synchronisation without much change.

3.1 The BSP-Why-ML Intermediate Language

As explain previously, we used a “sequentialisation” of the BSP-WHY-ML programs for their verification, that is we transform BSP-WHY-ML programs into WHY-ML ones and thus use the WHY¹ tools to generate adequate conditions for the correctness of the programs. The main idea is to extract the biggest blocks of code without synchronization (that is purely sequential) and then each block is transformed into a *for* loop: for each processor, we execute the block of code. Fig. 3.1 illustrates this idea.

The main idea of our approach is thus to simulate the execution of a BSP program on a parallel machine by a sequential execution which will simulate the entire parallel machine. This way we are able to use

¹Note that when we started writing our implementation of BSP-WHY, the only version of WHY was WHY2. Due to a lack of time, we did not adapt our work to WHY3. However, the modifications introduced by WHY3 are mostly syntactic or in the logic language, and it should not be too difficult to adapt our work in the future.

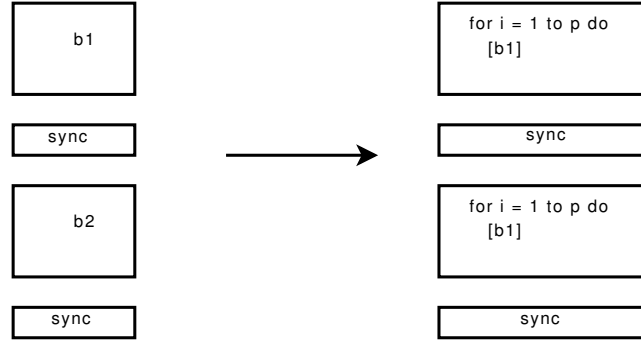


Figure 3.1. Main idea of the transformation.

the WHY tools. But in doing so, we need to simulate the memory (environment) of all the computers in a single computer. We also need to simulate the functioning of the communication operations.

The result is that each program written in BSP-WHY-ML, and then sequentialized into a WHY-ML program share the same structures: they use the same kind of environments to keep track of the parallel operations, the same data types (**p**-arrays, lists of messages, *etc.*), the same primitives to manipulate these environments. It is thus convenient to regroup all of these declarations in a separate file.

In the same way that WHY uses prelude files to define basic operations common to all WHY-ML programs, we use a `bspwhyprelude.mlw` file, which contains the common data types, the basic operations on these data types, the axiomatisation of the BSP operations, and of the memory model used.

We thus first present the syntax of a core-calculus BSP-WHY-ML that will be used to define the formal “sequentialisation”. We also describe how BSP-WHY-ML extends WHY-ML. We then describe the memory model used for the simulation, that is how we deal with **p** different values for each variable in a BSP-WHY-ML program — how we represent them in a single memory.

3.1.1 Syntax of BSP-Why

The core idea of our approach is to generate a sequential code in the WHY-ML language, so that we can re-use its powerful back-end. It is thus mandatory that we generate WHY-ML code. However, it would be in theory possible to have a totally different language as our parallel language.

Our set of communication primitives follows an idealised version of the standard BSPLib. We thus have DRMA primitives (`bsp_push` et `bsp_pop`, `bsp_put` and `bsp_get`), as well as the message passing style primitives (`bsp_send` and `bsp_findmsg`).

The so-called High-performance primitives of the BSPLib/PUB library or thread migration, however, are outside the scope of our approach. They introduce nondeterminism, and can break the BSP model if not used carefully. In addition, it is not needed to have them in our basic language; in [111], we showed how it is possible to first program using only the standard primitives, and then use a certified optimisation procedure to automatically use the high-performance routines when it is possible in a safe way — this work has currently not been updated for the full BSP-WHY-ML language.

The syntax of BSP-WHY-ML language is thus the one of WHY-ML with an additional syntax for parallel instructions, see Fig. 3.2 for the core-calculus. A program P is a list of declarations. A declaration d is either a program expression introduced with `let` or a declaration introduced with `parameter`, or an exception declaration. The full language also contains definitions: logic terms, axioms, parameters, *etc.*

(a) Programs Expressions

Program expressions mostly follows ML’s ones. Fig. 3.3 gives two simple examples that will be used through this chapter to illustrate the manipulation and transformations made by our BSP-WHY tool. Remark that `envCsendls(j, bsp_pid + 1, y, j, x)` is here a syntactic sugar (to simplify the reading of the example) to say that $\forall j \ y \leq j < \text{pid} + 1$ the j -th element of the environment (a list) of sending messages `envC` is the value of x . `envC` contains (at least) the necessary values for the prefix computation when $j = \mathbf{p}$. Fig. 3.4 gives an example of a more complete BSP-WHY-ML program without logical annotations (left) and with the full program expression (right). This example will be explained in details in Section 4.1.1.

Programs contain *pure terms* (t_e) made of constants (integers, booleans, `void`, *etc.*), variables, dereferences (written `!x`), application and application of function symbols from the logic to pure terms. A

Pure terms:		
t_e	$::=$	$c \mid x \mid !x \mid \phi(t_e, \dots, t_e)$
Expressions:		
e	$::=$	t_e term $\text{let } x = e \text{ in } e$ declaration $\text{let } x = \text{ref } e \text{ in } e$ variable $\text{if } e \text{ then } e \text{ else } e$ conditional $\text{loop } e$ infinite loop $\{\text{invariant } p \text{ variant } t_l\}$ $L:e$ label $\text{raise } (E \ e) : \tau$ exception $\text{try } e \text{ with } E \ x \rightarrow e \text{ end}$ catch it $\text{assert } \{p\}; e$ assertion test $e \ \{q\}$ condition test $\text{fun } (x : \tau) \rightarrow \{p\} \ e$ pure function $\text{rec } x \ (x : \tau) \dots (x : \tau) : \beta$ recursive one $\{\text{variant } t_l\} = \{p\} \ e$ $e \ e$ application $\text{bsp_push } x$ registering $\text{bsp_pop } x$ deregistering $\text{bsp_put } e \ x \ y$ DRMA writing $\text{bsp_get } e \ x \ y$ DRMA reading $\text{bsp_send } e \ e$ BSMP sending
Assertions:		
Logic terms:		
t_l	$::=$	$c \mid x \mid !x \mid \phi(t_l, \dots, t_l) \mid \text{old}(t_l)$ $\mid \text{at}(t_l, L) \mid t_l < i > \mid < x >$ $\mid \text{bsp_pid} \mid \text{nprocs}$
Predicates:		
p	$::=$	$A(t_l, \dots, t_l) \mid \forall x : \beta. p \mid p \Rightarrow p$ $\mid p \wedge p \mid \dots$
Types:		
τ	$::=$	$\beta \mid \beta \text{ ref} \mid (x : \tau) \rightarrow \kappa$
κ	$::=$	$\{p\} \tau \in \{q\}$
q	$::=$	$p; E \Rightarrow p; \dots; E \Rightarrow p$
ϵ	$::=$	$\text{reads } x, \dots, x \text{ writes } x,$ $\dots, x \text{ raises } E, \dots, E$ sync
Programs:		
P	$::=$	$\emptyset \mid d \ P$
d	$::=$	$\text{let } x = e$ $\mid \text{val } x : \tau$ $\mid \text{exception } E \text{ of } \beta$

Figure 3.2. Syntax of BSP-WHY: expressions (left), assertions, types and programs (right).

special constant nprocs (equal to \mathbf{p}) and a special variable bsp_pid (with range $0, \dots, \mathbf{p} - 1$) were added to WHY expressions. In pure terms (terms without possible side effects), we also have introduced the two special function symbols $\text{bsp_nmsg}(t)$ and $\text{bsp_findmsg } t_1 \ t_2$: the former corresponds to the number of messages received from a processor id t (C function $\text{bsp_nmsgs}(t)$) and the latter to get the t_2 -th message from processor t_1 (C function $\text{bsp_findmsg}(t_1, t_2)$).

$\text{ref } e$ introduces a new reference initialized with e . $\text{loop } e \ \{\text{invariant } p \text{ variant } t_l\}$ is an infinite loop of body e , invariant p and which termination is ensured by the variant t_l . The raise construct is annotated with a type τ since there is no polymorphism.

In the core-calculus, the five parallel operations are: (1) $\text{bsp_push } x$, registers a variable x for global access; (2) $\text{bsp_pop } x$, delete x from global access; (3) $\text{bsp_put } e \ x \ y$, distant writing of x to y of processor e ; (4) $\text{bsp_get } e \ x \ y$, distant reading from x to y ; (5) $\text{bsp_send } e_1 \ e_2$, sending value of e_1 to processor e_2 . In order to simplify the presentation of BSP-WHY, parallel operations of the core-calculus (notably DRMA primitives) take simple variables as argument, instead of buffers. In practice, BSP-WHY does manipulate buffers, and adds proof obligations to avoid buffer overflows.

There are two ways to insert proof obligations in programs: $\text{assert } \{p\}; e$ places an assertion p to be checked right before e and $e \ \{q\}$ places a post-condition q to be checked right after e .

(b) Logical Annotations and Types

One can remark that BSP-WHY-ML or WHY-ML programs are ML programs with logical annotations inside the brackets. As in WHY, programs can take parameters that correspond to external values. The prefix “global” corresponds to a value that would be available on each processor — and possibly different. The WHY language is typed using a simple monomorphic type system for program expressions and traditional polymorphic ML type system for pure expressions — logics and terms. This type system is also completed with effects: each expression is given a type together with the sets of possibly accessed and possibly modified variables and the set of possibly raised exceptions. BSP-WHY adds also a special effect to global parameters called “sync” which corresponds to the fact that the code behind the parameter performs at least a global synchronisation — this is obviously the case of the bsp_sync . That allows the user to define its own patterns of communications — as MPI collective operators or else. Note that a parameter with a “sync” effect suppose that all modifications of variables have been done and that the exception is only raised after the synchronisation.

In the core-calculus, annotations are written using a simple minimal first-order logic. A logic term t_l can be a constant c , a variable x , the contents of a reference x (written $!x$) or the application of a function symbol ϕ . Note that ϕ is a function symbol belonging to the logic which is thus not defined in the program. The construct $\text{old}(t_l)$ denotes the value of term t_l in the precondition state and the

<pre> let SimpleCode1 = (* a block of local computations and sending values *) while(!i<nprocs) do { invariant ... variant nprocs-i} bsp_send i m; i:= !i+1 done; (* end of the first super-step *) bsp_sync(); (* another block of local computations by reading a send value *) y:=bsp_findmsg 0; x:=!y+1 </pre>	<pre> let SimpleCode2 = (* doing p super-steps *) while !j<=nprocs do { invariant ... variant nprocs-j} bsp_sync(); j:=!j+1 done </pre>
---	--

Figure 3.3. Simple examples of BSP-WHY codes.

construct $\text{at}(t_i, L)$ denotes the value of the term t_i at the program point L .

From this simple minimal first-order logic, we add the construct $t\langle i \rangle$ which denotes the value of a term t at processor id i , and $\langle x \rangle$ denotes the \mathbf{p} -array x (a value on each processor) by opposition to the notation x which means the value of x on the current processor. We also add *farray* to logical term which is the abstract type for purely applicative arrays of size \mathbf{p} (with some obvious axioms) and *list*. They are used for the description of \mathbf{p} -values, one value per process and for the environment of communications. This will be explained later.

We assume the existence of a set of *pure types* (β) in the logical world, containing at least `unit`, `bool`, `int` and messages `value`. As in [101], predicates necessarily include conjunction, implication and universal quantification. An atomic predicate is the application of a predicate symbol A and is not interpreted.

As in [101], a value of type τ is either an immutable variable of a pure type (β), a reference containing a value of a pure type (β ref) or a function of type $(x : \tau) \rightarrow \{p\} \beta \in \{q\}$ mapping the formal parameter x to the specification of its body, that is a precondition p , the type τ for the returned value, an effect ϵ and a post-condition q . An effect is made of three lists of variables: the references possibly accessed (`reads`), the references possibly modified (`writes`) and the exceptions possibly raised (`raises`). And the effect `sync` is used to define functions that perform synchronisations. A post-condition q is made of several parts: one for the normal termination and one for each possibly raised exception (E stands for an exception name).

For synchronous routines (a global parameters), it can be useful, in order to prove the correctness of a program, to give a logic assertion just before the routine instruction: the routines holds a pre-condition. This assertion should describe the computations done during the previous superstep and define how are the environments of communications of the processes at the end of the previous super-step. In the transformation to the WHY code, this assertion is used in the invariant of the loop executing sequentially the code of each processor. If the assertion is void, it will not be possible to prove more than the *safety* of execution of the program, *i.e.* the fact that the program will terminate without failing by an array overflow, an illegal message read, *etc.* If the logic assertion has not been added, the BSP-WHY tool would search it automatically. This will be explain latter when speaking of the generation of invariants of “for loops”. We also explain in Section 3.3 how to deal with subgroup synchronisation.

(c) Syntactic Sugars

As the core-calculus used in this thesis (for the formal definition of the transformation) is missing commonly used statements, we can define some syntactic sugar. We consider the following classical syntax sugar:

$e_1; e_2 \equiv \text{let } _ = e_1 \text{ in } e_2$
 $\text{raise } E \equiv \text{raise } (E \text{ void}): \text{unit}$

and the traditional `while` is also a syntactic sugar for a combination of an infinite loop with the use of an exception *Exit* to exit the loop. That is, `while e_1 {invariant p variant t } do e_2 done` is interpreted as:

```

try
  loop
    if  $e_1$  then  $e_2$  else raise Exit
    {invariant  $p$  variant  $t$ }
  done
with Exit  $\rightarrow$  void end

```

We can note that the core calculus (and also the WHY tool) does not contain any assignment ($x := e$). This is due to the fact that it can be simulated by the following parameter with effect:

<pre> let prefixes () = let y = ref (bsp_pid void + 1) in while (!y < nprocs) do bsp_send !y (cast_int !x); y := !y + 1 done; bsp_sync; z := !x; let y = ref 0 in while (!y < bsp_pid void) do z := !z + uncast_int (bsp_findmsg !y 0); y := !y + 1 done </pre>	<pre> global parameter x: int ref global parameter z: int ref logic sigma_prefix : int farray → int → int axiom sigma_prefix_def1 : forall t:int farray. sigma_prefix(t,-1) = 0 axiom sigma_prefix_def2 : forall i:int. isproc(i) → forall t:int farray. sigma_prefix(t,i) = t<i> + sigma_prefix(t,i-1) let prefixes () = {} let y = ref (bsp_pid void + 1) in while (!y < nprocs) do { invariant envCsendls(j,bsp_pid + 1,y,j,x) variant nprocs - y } bsp_send !y (cast_int !x); y := !y + 1 done; bsp_sync; z := x; let y = ref 0 in while (!y < bsp_pid void) do { invariant z=x+sigma_prefix(<x>, y) variant bsp_pid - y } z := !z + uncast_int (bsp_findmsg !y 0); y := !y + 1 done {z=sigma_prefix(<x>, bsp_pid)} </pre>
---	---

Figure 3.4. BSP-WHY code of the direct prefix computation (left) and with its logical annotations (right).

parameter `ref_set` : $x : 'a \text{ ref} \rightarrow v : 'a \rightarrow \{\}$ unit writes $x \{ x = v \}$

In the same manner, the `bsp_sync` operation, barrier of synchronisation, can be simulated by:

global parameter `bsp_sync`: unit $\rightarrow \{\}$ unit writes ... **sync** {Do_the_Com}

that is a specific parameter that holds a **sync** effect and where “Do_the_Com” is for the modification of all the environments of communication. As explained above, that allows users to define their own patterns of communications. Such a parameter must have a write effect on every modified communication environment.

We also write τ as a syntactic sugar for the function specification $\{p\} \beta \in \{q\}$ when no precondition and no post-condition (both being **true**) and no effect (ϵ is made of three empty lists) are defined. Note that functions can be partially applied.

(d) Informal Semantics of the Primitives

Even if the formal semantics (defined in Chapter 5) contains many rules and many environments (due to the parallel routines), there is no surprise and it has to be read naturally. BSP programs are SPMD ones so an expression e is started \mathbf{p} times. We model this as a \mathbf{p} -vector of e with its environments. A final configuration is a value on all processors.

Basically, a primitive adds the corresponding message to the environment. Values to be sent and distant reading/writing are stored in environment of communications as simple lists of messages. There are thus six additional components in the environment, one per primitive that needs communications. Each of them is a special variable for the assertions. For DRMA primitives, there is also the registration \mathcal{T} which is described later (push and pop need communications for keeping the registration of each processor coherent). As with the BSPlib, DRMA variables are also registered using a registration mechanism that is each processor contains a registration \mathcal{T} which is a list of registered variables: the first one in the list of a processor i corresponds to the first one of the processor j .

To avoid confusion between a new reference and those that have been registered before, one could not declare a reference that has been created before. This is not a problem since WHY always forbids this case to only have alias-free programs.

3.1.2 The Parallel Memory Model in Why

(a) Returned Values

With BSP-WHY, we transform a parallel BSP program into a sequential code simulating its execution. At any given point in the execution of the program, a parallel program is likely to return p different values, on the p processors. For this reason, we chose to make the sequential expressions in the translated program return p -arrays of values.

For instance, let us say that we have a parallel function f defined in our parallel program, returning an integer value, that is guaranted to always be the processor id:

```
let f () = ... ; !x {return = bsp_pid}
```

Then the translated program will also be a function f , but instead of returning an integer value, it will return a p -array of integers:

```
let f () = ... ; !x { $\forall$  proc_i, isproc(proc_i)  $\rightarrow$  return[proc_i] = proc_i}
```

where `isproc` is a predicate to know if `proc_i` is a valid processor number, that is $0 \leq \text{proc_i} < p$. This transformation of the logical assertion into a `forall` statement will be explained in Section 3.2.5.

(b) Data Types

Several data types are used in the transformation, and are defined in the prelude file. The `farray` type, a functional array of length p , is used every time we need to have data on each processor. The `parray` type corresponds to the mutable array, which is a reference to the `farray`.

Lists are used in several ways for the communication environments, and are defined in this file too. Various other data types are defined, such as the `value` data type used to transmit any kind of data, and the `rvalue` type used to represent the values received with `send` messages.

(c) Communication Environments

As the semantics (defined in Chapter 5) suggest, three separates message queues, `send_queue` ($\mathcal{C}^{\text{send}}$), `put_queue` (\mathcal{C}^{put}), and `get_queue` (\mathcal{C}^{get}), are used to store the communication requests before synchronisation. Each queue is defined as a list, with the usual constructors `nil` and `cons`. Similar queues are used for the `push` ($\mathcal{C}^{\text{push}}$) and `pop` (\mathcal{C}^{pop}) mechanisms.

To be as close as possible to the semantics, the communication procedures `send`, `put`, `get`, and likewise the synchronisation `sync` are defined as parameters. As such, we only give the type of the procedure, and an axiomatization given by the post-condition, not the effective sequential code used: an actual sequential code would make more proofs, the additional verification conditions for this extra code.

In the *EnvR* section of the file, we describe \mathcal{R} which contains messages sent during the previous super-step. Since it is possible to send different types of value with the communication instructions, a generic type `value` is used, and one function of serialisation and one of deserialisation are needed for each data type used in the program. One axiom for each data type ensures that the composition of deserialisation and serialisation is the identity.

As an example, we show the different parts of the prelude file used to model the behaviour of BSMP communications. First, we define the type used to store the messages waiting to be sent, using the usual list definition (`nil` and `cons`):

```
type send_queue
logic add_send : value , send_queue  $\rightarrow$  send_queue
logic nil_send : send_queue
```

The logical function `add_send` (\oplus in the semantics) will be used to effectively add a send message in a `send_queue`. Each processor has p `send_queue`, containing the messages to be sent to the p processors of the parallel machine. Next, we define some useful operations on these lists, using an abstract logic definition, and an axiomatisation for each logic function. We give for example the axiomatisation of the `nsend` function, used to determine the number of messages waiting ($Size(s.\mathcal{R}, to)$ in the semantics):

```
logic nsend : send_queue  $\rightarrow$  int
logic in_send_n : send_queue, int, value  $\rightarrow$  prop
axiom nsend_nil : nsend(lfnil_send) = 0
axiom nsend_cons : forall q:send_queue. forall n:int. forall v:value. nsend(q) = n  $\rightarrow$  nsend(lfadd_send(v,q)) = n+1
axiom nsend_cons2 : forall q:send_queue. forall v:value. nsend(lfadd_send(v,q)) = nsend(q) + 1
```

<pre> let x = ref 0 in let y = bsp_pid in bsp_push x; bsp_sync (); bsp_put (1-bsp_pid) y x </pre>	<pre> type variable logic x : variable ... var_set x 0; let y = bsp_pid in bsp_push x; bsp_sync (); bsp_put (1-bsp_pid) y x </pre>
---	--

Figure 3.5. Use of a global array DRMA communication.

The `in_send_n` function is used to test the fact that a message is in the list. Lastly, we can define the variable used for the global environment. For each processor, we have a `parray` of `send_queues`, hence the final type, and the method `bsp_send` defined in the semantics. `isproc` is a useful predicate defined earlier in the prelude file, stating that an index is a valid processor id (*i.e.* $0 \leq i < p$):

parameter envCsend : send_queue farray farray **ref**

parameter bsp_send: dest0:int → v:value →
 { isproc(proc_i) }
 unit reads proc_i writes envCsend
 {envCsend=pupdate(envCsend@, proc_i, pupdate(paccess(envCsend@,proc_i),dest0,
 lfadd_send(v,(paccess(paccess(envCsend@,proc_i),dest0))))}}

We now define the environment used to store the values received during the previous synchronisation:

type rvaluet
logic rvalue_get : rvaluet, int, int → value

parameter envR : rvaluet farray **ref**

parameter bsp_findmsg: src:int → n:int → {} value reads proc_i,envR {result=rvalue_get(paccess(envR,proc_i),src,n)}

The logic function `rvalue_get` allows to retrieve the n -nth message sent by a processor `src`. `envR`, as previously, is defined as a `parray`. The `bsp_findmsg` is the corresponding parameter, and it can be used in the BSP-WHY programs.

(d) Global synchronisation: Example of a “sync” Effect

The only remaining part of the BSMP process is the synchronisation function, which is defined, as in the semantics, by the use of a *Comm* predicate. We give here the part of the predicate concerning the BSMP communications:

predicate comm_send(envCsend:send_queue farray farray,envCsend':send_queue farray farray,envR':rvaluet farray)
 = (forall i,j: int. isproc(i) → isproc(j) → (paccess(paccess(envCsend',i),j) = lfnil_send))
and (forall i: int. isproc(i) → (forall j:int. forall n:int. forall v:value.
 (rvalue_get(paccess(envR',i),j,n)=v) ↔ (in_send_n(paccess(paccess(envCsend,j),i),n,v))))

predicate comm(envCsend:send_queue farray farray,envCsend':send_queue farray farray,envR':rvaluet farray, ...)
 = comm_send(envCsend,envCsend',envR') **and** ...

parameter bsp_sync : unit → {} unit writes envCsend, envR, ... {comm(envCsend@, envCsend, envR, ...) }

The `comm_send` predicate is specific to the `send` messages, and is called from the `comm` predicate. Lastly, the `bsp_sync` parameter ensures that the `comm` predicate is true.

(e) Dealing with DRMA Primitives

The most complex part of the prelude file is the definition of the DRMA mechanism. A typical DRMA communication is something such as `bsp_put i x y`. The meaning of this function call is that at the next synchronisation, the current processor will write the content of its variable `x` (taken at the function call) in the variable `y` of the processor `i`. The queued message should thus contain:

- An integer value, representing the target processor identifier;
- A value (of any type), representing the data that will be sent;
- A *reference* to a variable, representing the place where the value should be written.

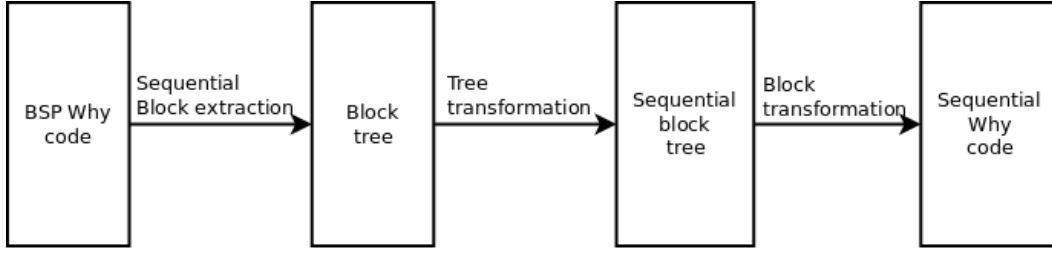


Figure 3.6. Scheme of transformation of BSP-WHY programs into WHY ones.

However, WHY does not allow the use of pointers, for the reasons explained in the introduction section on the Hoare logic and wp-calculus. It is thus impossible to simply put a reference to the target variable in a message queue. To solve this difficulty, we use a global two-dimensional array, named `global`, to store all the variables that need DRMA access. A special type is used to describe such variables, and for each variable x with DRMA in the program, a logical value x of type `variable` is added in the generated WHY file. This way, `global[x][i]` contains the value of variable x , on processor i .

To be in accordance with most BSPLIB-like libraries with DRMA routines, we define a registration \mathcal{T} . The `push` instruction can associate different variables on different processors. This is modelled using an additional array, which stores the association of the variables on different processors. For instance, if even processors push the variable x while odd processors push the variable y , with $p = 6$, the next `sync` operation will add a line $[x, y, x, y, x, y]$ in the association table. The index used in the `global` array is the variable on the first processor. For example, for the following code:

```
if(bsp_pid % 2 = 1) then bsp_push(x) else bsp_push(y)
```

we will have the following `global` array:

P1	P2	P3	P4	P5	P6
...
x	y	x	y	x	y

We also recall that in the PUB library, the order of `push` and `pop` should be the same on every processor, allowing to not use an additional synchronisation to check which block of memory (a variable) of a processor is linked to another block of another processor. It is easy (but currently not performed due to a lack of time) to deal with this memory model using the `global` array: at each synchronisation, we just have to check that all variables have been registered in the same order, that is appear in the same order in the lists of the `global` array.

3.2 Transformation of BSP-Why-ML Programs: the Inner Working

3.2.1 Generalities

Now that we have the necessary structures to simulate the environments and communication functions of the parallel machine (axiomatisation of the BSP routines), we can define the actual transformation of a BSP-WHY-ML program into a WHY-ML one, that will simulate its parallel execution. This transformation is composed of several steps, which we summarized in the scheme of Fig. 3.6. We also give the formal transformation for the core-calculus.

Note that the BSP-WHY tool was written in the OCAML language. Since we chose to keep the syntax of the language very close to WHY, we were able to re-use the open-source code of WHY for a large part of the program. It is the case in particular for the parsing and printing part of the program. The actual transformation however had to be written entirely from scratch.

Let us now detail these different steps of the transformation. A part of this transformation for the core-calculus will be machine-checked in Chapter 5.

(a) Notations

During the different steps of the transformation, we encounter different kinds of trees representing our expressions, which are naturally very close since they represent the same program, but subtly different from each other. In our implementation of BSP-WHY we used prefixes on each constructor to be able

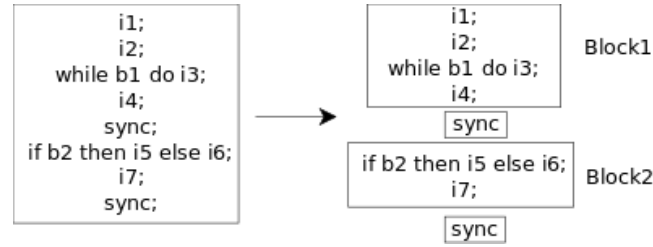


Figure 3.7. Main idea of the sequential block decomposition.

to differentiate the different types. For instance, a simple `if e1 then e2 else e3` might be successively represented with the following constructors:

- `bspIfThenElse(e1,e2,e3)` for the if-then-else as a BSP-WHY statement
- `blockIfThenElse(e1,e2,e3)` in the block tree
- `wblockIfThenElse(e1,e2,e3)` in the block tree in its sequential (WHY) form
- `whyIfThenElse(e1,e2,e3)`, finally, in the resulting WHY program

This was necessary in our OCAML program. However, in the mathematical notation of the transformation, we will keep the notations light and use the same keywords at the different stages of the transformation. Thus, we could write `[[if e1 then e2 else e3]] = if [[e1]] then [[e2]] else [[e3]]`, even if this means going from the parallel block tree to the sequential block tree. We will detail the transformation of the program expression and logic assertions in the following sections.

3.2.2 Identification of Sequential Blocks

The first step of the transformation is a decomposition of the program into blocks of sequential instructions, as Fig. 3.7 shows. The aim is to be able to simulate the execution of a sequential block consecutively for each processor executing it, in a sequential way, instead of the normal parallel execution on each processor at the “same time”. In order to obtain the best efficiency, we are trying to isolate the largest blocks of code that are purely sequential.

The result of this phase of the transformation is what we will call from now on a *block tree*, meaning an abstract syntax tree, but where the leaves of the tree are blocks of sequential code, instead of basic expressions. The block tree is constructed as the abstract syntax tree (AST), with the addition of a basic constructor for a block of non synchronising code. For this, we first must tag which parts of the code are purely sequential and which are not, that is tagging the parallel parts of the code. Then we can extract the blocks.

(a) Tagging of the Parallel Parts of the Code

In order to do the block decomposition correctly, we need to be able to tell if an instruction has a parallel effect or not. Two instructions potentially influence the parallelism of the program:

1. A BSP-WHY parameter defined with the synchronize effect (such as the `bsp_sync` instruction);
2. A function call, if the function body is determined to have a parallel code.

The first step of our transformation is to tag each node of the abstract syntax tree, with a boolean that says whether the subtree includes parallel code, or not. One might note that the following instructions will *not* be tagged as parallel:

- A function call, if the function body is entirely sequential;
- A call to a BSP procedure, such as `bsp_send`, `bsp_push`, *etc.* the reason is that the parallel communication associated with such functions are only in effect done during the next synchronisation.

The tagging algorithm is a recursive pattern matching of the BSP-WHY abstract syntax tree. It operates with side effects (instead of returning a new tree, we modify a mutable field in the current one), and also returns a boolean indicating whether the expression is parallel or not. An example of a generic case would be the `if-then-else` instruction: tagging the `if e1 then e2 else e3` node of the tree is done by recursively tagging the three subtrees `e1`, `e2` and `e3`; then, if at least one of the subexpressions was found to be parallel, we set the parallel tag of our node to *true*; else, we set it to *false*. For example, a part of the code of this tagging function is:

$$\begin{array}{c}
\frac{e_1 : t_1 \quad e_2 : t_2}{e_1 ; e_2 : \max(t_1, t_2)} \quad \frac{e_1 : t_1 \quad e_2 : t_2}{\text{let } x = e_1 \text{ in } e_2 : \max(t_1, t_2)} \quad \frac{e_1 : t_1 \quad e_2 : t_2}{\text{let } x = \text{ref } e_1 \text{ in } e_2 : \max(t_1, t_2)} \\
\\
\frac{e : t}{x := e : t} \quad \frac{e_1 : t_1 \quad e_2 : t_2 \quad e_3 : t_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \max(t_1, t_2, t_3)} \quad \frac{e_1 : t_1 \quad e_2 : t_2}{\text{while } e_1 \text{ do } e_2 \text{ done} : \max(t_1, t_2)} \\
\\
\frac{}{\text{bsp_sync} : \text{true}} \quad \frac{e : t}{\text{raise } (E \ e) : t} \quad \frac{e_1 : t_1 \quad e_2 : t_2 \quad e_3 : t_3}{\text{try } e_1 \text{ with } E \ e_2 \rightarrow e_3 \text{ end} : \max(t_1, t_2, t_3)} \\
\\
\frac{e_1 : t_1 \quad e_2 : t_2}{e_1 \ e_2 : \max(t_1, t_2)} \quad \frac{e : t}{l : e : t} \quad \frac{e : t}{\text{assert } \{p\}; e : t} \quad \frac{e : t}{e \ \{p\} : t}
\end{array}$$

Figure 3.8. Tagging of the parallel parts of the code of the core-calculus.

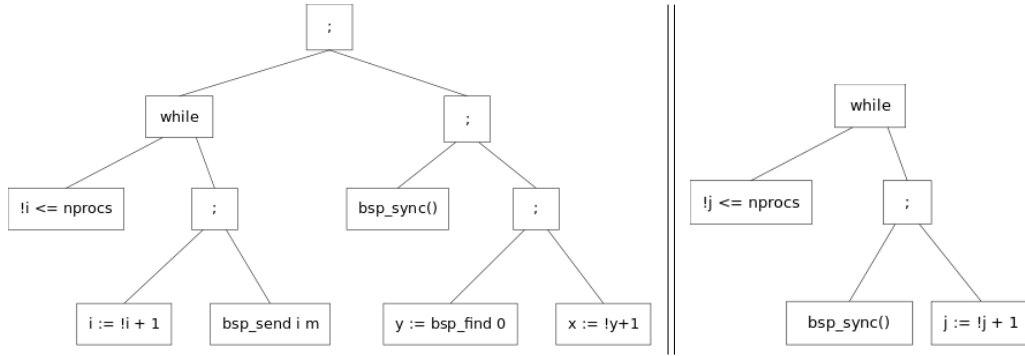


Figure 3.9. Abstract syntax trees of SimpleCode1 (left) and SimpleCode2 (right).

```

let rec tag_parallel(tree) = match tree with
| ... → ...
| bsplfThenElse(e1,e2,e3),tag → if(tag_parallel(e1) || tag_parallel(e2) || tag_parallel(e3))
    then tag←true
    else tag←false; !tag
| ... → ...
| Sync,tag → tag←true; !tag

```

Fig. 3.8 gives the formal definition of this tagging for the core-calculus. It is easy to read inductive rules where $\max(t_1, t_2)$ is *true* if either t_1 or t_2 are *true*, *false* otherwise (a boolean “or”). An expression tagged *true* is able to perform a synchronisation and thus could not appear into a sequential block.

Example: Tagging of Simple Expressions. We are going to illustrate the different steps of the transformation on the sample code given as example in Fig. 3.3. Since the transformation works on the abstract syntax trees, we start by giving the tree corresponding to the SimpleCode1 and SimpleCode2 expressions. This is done in Fig. 3.9.

We can now apply the tagging algorithm on the trees. The result is shown in Fig. 3.10. Here we can make a few remarks on the examples:

- The parallel exchanges come from the `bsp_sync()` statement. It is important to understand that the `bsp_send`, *etc.* instructions are only requests for communication. As such, they modify the local environment, but they do not lead to immediate parallel code. The communications will be done during the synchronisation. For this reason, the whole `while` loop in SimpleCode1 is tagged as being not parallel.
- In SimpleCode1, the only parallel nodes are sequence nodes. However, it is fully possible to have all of the control flow instructions tagged as parallel. If a `bsp_sync` appears inside a `while` loop, the loop will be tagged as parallel, *etc.* This is illustrated in SimpleCode2.

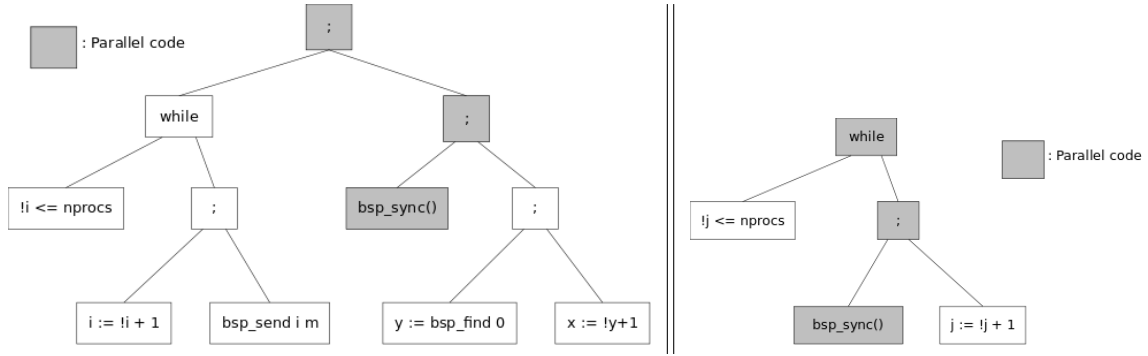


Figure 3.10. Application of the tagging algorithm for SimpleCode1 (left) and SimpleCode2 (right).

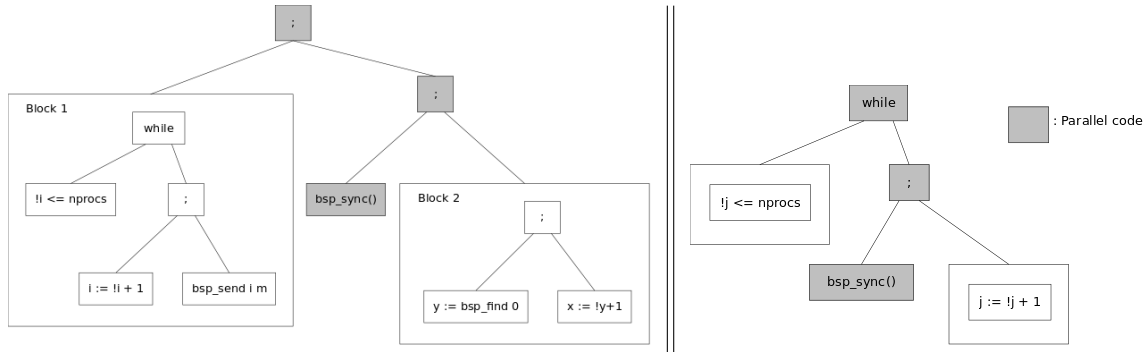


Figure 3.11. Application of the block decomposition for SimpleCode1 (left) and SimpleCode2 (right).

(b) Function of the Block Decomposition

Once we have tagged the syntax tree as described above, the transformation to the block tree is straightforward. It is simply a recursive algorithm on the syntax tree, with two cases at each node:

1. If the node is tagged as having no parallel effect, we simply return a newly constructed Block, with the sequential code being the subexpression at the node;
2. If the node is tagged as parallel, we return the corresponding block tree constructor, with his children obtained by recursively calling our algorithm on the children of the current node.

For example, a part of the code of this decomposition is:

```
let rec blocktree(tree) = match tree with
| ... → ...
| bspIfThenElse(e1,e2,e3),tag → if(!tag) then Block(tree) else blockIfThenElse(blocktree(e1),blocktree(e2),blocktree(e3))
| ... → ...
```

Fig. 3.11 gives the results of the simple examples of Fig. 3.3. Fig. 3.12 gives the formal definition of this block decomposition for the core-calculus. The first rules says that if the expression e is tagged as *false* then it can be a full sequential block which contain e . Otherwise, we build a tree like the expression.

(c) Maximising Sequential Blocks

It is interesting to note that for a program written mainly as a sequence of instructions, such as $A;B;C;D;E$, the order in which the sequences are parsed modifies the block decomposition. Let us take a look at our example SimpleCode1, assuming that the sequences are read in a different way. The corresponding tree is represented in Fig. 3.13. It appears immediately that one more node of the tree is considered to be a parallel code. Furthermore, there are now 3 isolated sequential sub-trees, which means we will get 3 sequential blocks in our transformation. This does not change the result of the execution in any way, so it could be ignored from a theoretical point of view. However, having 3 blocks instead of 2 also means that we will have 3 **for** loops in the resulting code instead of 2, so less readability, and more proof obligations.

$$\begin{aligned}
\langle e, false \rangle &= Block(e) \\
\langle e1; e2, true \rangle &= \langle e1 \rangle ; \langle e2 \rangle \\
\langle let\ x = e1\ in\ e2, true \rangle &= let\ x = \langle e1 \rangle\ in\ \langle e2 \rangle \\
\langle let\ x = ref\ e1\ in\ e2, true \rangle &= let\ x = ref\ \langle e1 \rangle\ in\ \langle e2 \rangle \\
\langle x := e, true \rangle &= x := \langle e \rangle \\
\langle if\ e1\ then\ e2\ else\ e3, true \rangle &= if\ \langle e1 \rangle\ then\ \langle e2 \rangle\ else\ \langle e3 \rangle \\
\langle while\ e1\ do\ e2\ done, true \rangle &= while\ \langle e1 \rangle\ do\ \langle e2 \rangle\ done \\
\langle raise\ (E\ e), true \rangle &= raise\ (E\ \langle e \rangle) \\
\langle try\ e1\ with\ E\ e2 \rightarrow e3\ end, true \rangle &= try\ \langle e1 \rangle\ with\ E\ \langle e2 \rangle \rightarrow \langle e3 \rangle\ end \\
\langle e1\ e2, true \rangle &= \langle e1 \rangle\ \langle e2 \rangle \\
\langle l:e, true \rangle &= l: \langle e \rangle \\
\langle assert\ \{p\};\ e, true \rangle &= assert\ \{p\};\ \langle e \rangle \\
\langle e\ \{p\}, true \rangle &= \langle e \rangle\ \{p\}
\end{aligned}$$

Figure 3.12. Formal definition of the block decomposition.

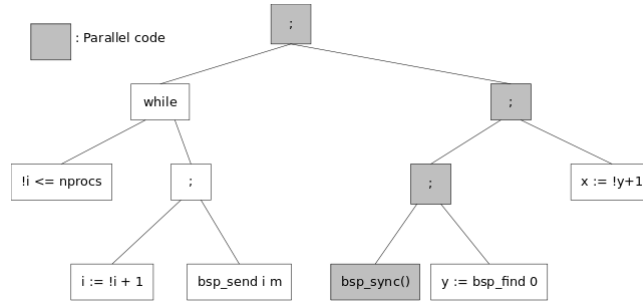


Figure 3.13. Example SimpleCode1, with a different parsing of the sequences.

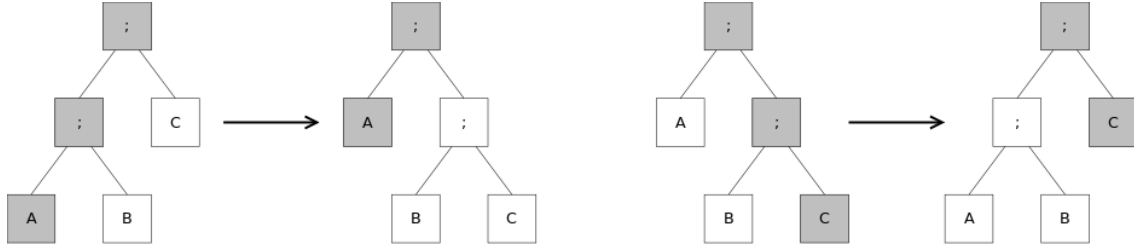


Figure 3.14. Re-organising sequences.

For this reason, we decided to maximise the blocks in sequences of instruction. This is done by making a simple observation. There are two elementary patterns in which the block decomposition is clearly not optimal, corresponding to the codes $A; (B; C)$ where C has synchronisations, and $(A; B); C$ where A has synchronisations. We can thus re-organise the code when we meet such a pattern, as described on Fig. 3.14. This is done by a simple recursive transformation on the tree:

```

let rec maxseq_tree(tree) = match tree with
| ... → ...
| bspSeq(e1,e2),t →
  begin
    match maxseq_tree(e1), maxseq_tree(e2) with
    | (a,false), (bspSeq((b,false),(c,true)),_) → bspSeq((bspSeq((a,false),(b,false)),false),(c,true)),true
    | (bspSeq((a,true),(b,false)),_), (c,false) → bspSeq((a,true), (bspSeq((b,false),(c,false)),false)), true
    | e'1, e'2 → bspSeq(e'1,e'2),t
  end
| ... → ...

```


(d) Limitation

This strict block decomposition does not accept all valid BSP programs. Take for example the following code:

```
if (bsp_pid >= !i) then
  begin
    computation1;
    bsp_sync();
    computation2;
  end
else
  computation3;
  bsp_sync;
  computation4;
```

which can be used for instance for a logarithmic computation (see Section 4.1). This is a case where our block decomposition fails: not all the processors run the same `bsp_sync` and our tool will generate unprovable assertions. But the program can be rewritten by factoring the two `bsp_sync`²:

```
if (bsp_pid >= !i) then computation1; else computation3;
bsp_sync();
if (bsp_pid >= !i) then computation2; else computation4;
```

In practice and by reading many BSP algorithms (those cited in Section 1.3.2), we only find this problem in reduction-like (logarithmic) loops where the code can clearly be re-factored. This does not seem too restrictive.

3.2.3 Block Tree Transformation

(a) Tree Transformation

After having regrouped the sequential parts of the program into blocks, the rest of the tree is just the structure of the parallel mechanisms, and can not be altered. Thus, the transformation on the block tree is made with a traversal of the tree where we apply recursively the transformation. We write $e' = [[e]]$ where e' is the result of the tree transformation.

We take a BSP-WHY block tree, and return a WHY block tree, which is structurally identical, but with different constructors. For example, a part of the code of this tree transformation is:

```
let rec whyblocktree(btree) = match btree with
| ... → ...
| blockLetin(v,e1,e2) → wblockLetin(v,whyblocktree(e1),whyblocktree(e2))
| ... → ...
| blockBlock(b1) → wblockBlock(block_transform(b1))
```

However, several cases need more attention. First, the base case in this recursive algorithm is the transformation of a sequential block. It is the next step of the global transformation, and will be detailed in Section 3.2.4. Then, the `if` and `while` statements needs an additional treatment: when transforming a `if` or `while` structure at the block tree level, there is a risk that a `bsp_sync` instruction might be executed on a processor and not on the other. We generate an assertion to forbid this case, ensuring that the condition associated with the instruction will always be true on every processor at the same time. For instance, if the source code is:

```
while ( (pow_int 2 !i) < bsp_nprocs ) do
  (...)
  bsp_sync void;
  (...)
  i:=!i + 1
done
```

then the assertion generated would be:

```
assert {forall proc_i, isproc(proc_i) →
  (((pow_int(2,paccess(i,proc_i))) < bsp_nprocs)
  ↔ (forall proc_j, isproc(proc_j) → (pow_int(2,paccess(i,proc_j))) < bsp_nprocs) ) };
```

If the condition is true for a processor (`proc_i`), then it must be true for any other processor.

²Note that doing this transformation automatically is perhaps possible in some specific cases but this is not the subject of this doctoral thesis.

$$\begin{aligned}
[[Block(e)]] &= forp([[e]]_i) \\
[[if\ e1\ then\ e2\ else\ e3]] &= if\ valid([[e1]])\ then\ [[e2]]\ else\ [[e3]] \\
[[while\ e1\ do\ e2\ done]] &= while\ valid([[e1]])\ do\ [[e2]]\ done \\
[[e1;\ e2]] &= [[e1]]; [[e2]] \\
[[let\ x = e1\ in\ e2]] &= let\ x = [[e1]]\ in\ [[e2]] \\
[[let\ x = ref\ e1\ in\ e2]] &= let\ x = ref\ [[e1]]\ in\ [[e2]] \\
[[x := e]] &= x := [[e]] \\
[[raise\ (E\ e)]] &= raise\ (E\ [[e]]) \\
[[try\ e1\ with\ E\ e2 \rightarrow e3\ end]] &= try\ [[e1]]\ with\ E\ [[e2]] \rightarrow [[e3]]\ end \\
[[e1\ e2]] &= [[e1]]\ [[e2]] \\
[[l:e]] &= l: [[e]] \\
[[assert\ \{p\};\ e]] &= assert\ \{[[p]]\};\ [[e]] \\
[[e\ \{p\}]] &= [[e]]\ \{[[p]]\}
\end{aligned}$$

Figure 3.15. Transformation of the block tree in the core-calculus.

The *valid* Parameter. In practice, instead of explicitly generating the assertion, we indirectly enforce it by a call to a WHY parameter, called **valid**. This parameter is called on the result of the boolean expression. For instance, the transformation of the `if e1 then e2 else e3` expression will be a WHY expression under the form: `if valid([[e1]]) then [[e2]] else [[e3]]`. The **valid** parameter is then defined in the file `bspwhy.prelude.mlw`. The WHY-ML code of its definition is as follows:

```

parameter valid : a:'a farray ref →
{ ∀i:int. ∀j:int. isproc(i) → isproc(j) → paccess(a,i) = paccess(a,j) }
'a reads a
{ ∀ i:int. isproc(i) → result = paccess(a,i) }

```

It takes a **p**-array of values in argument. A proof obligation will then be generated by WHY to ensure that the precondition is met. This translates into a proof obligation equivalent to the one described in the *assert* above. Fig. 3.15 gives the formal definition of this tree transformation for the core-calculus and Fig. 3.16 gives the results of the examples of Fig. 3.3. The first rule says that sequential block of code is transformed into a “for-loop” as described in Fig 3.1 that is we execute the code **p** times one per processor. Other rules are just inductive cases except for the **while** and **if** statements: we need to introduce the **valid** parameter to test if all processors go well and together inside the same sequential block.

3.2.4 Translation of Sequential Blocks

(a) Local Block Transformation

The last step of the transformation is the transformation of a block of sequential code. The idea of BSP-WHY is that for a block of instructions that would be run in parallel on each processor simultaneously, we simulate its parallel execution by the use of a “for loop” that will run the code sequentially, with one execution for each processor.

The transformation is thus the following: $[[Block(e)]] = forp([[e]]_i)$. The notation $forp(e)$ is a shortened notation for the “for loop”.

(b) Generation of the “for loop”

Main idea. Because WHY only includes the **while** statement, it actually corresponds to this code:

```

let proc_i = ref 0 in
loopstart:while(!proc_i < bsp_p) do
{ invariant ... variant nprocs - proc_i }
e;
proc_i ← (!proc_i) + 1
done

```

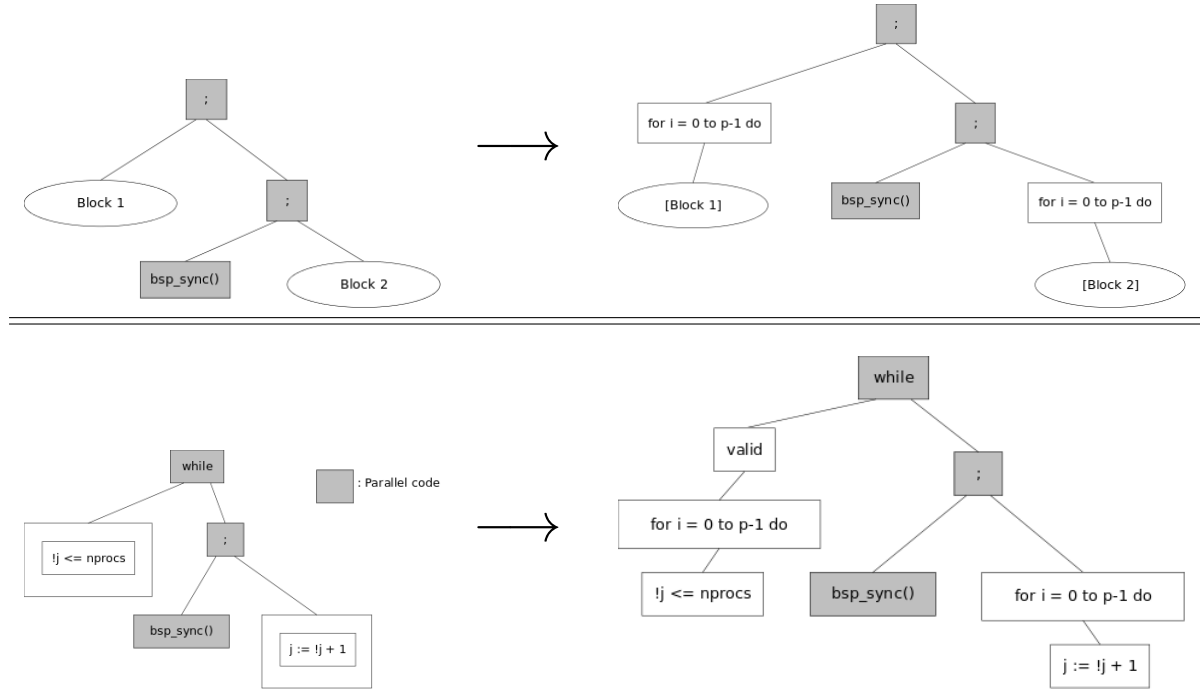


Figure 3.16. Application of the tree transformation for **SimpleCode1** (up) and **SimpleCode2** (down).

When making explicit the loop, one thing is immediately visible. it is necessary to give an invariant to the **while** loop, if we hope to prove anything about the program. Thankfully, the invariant can in general be inferred automatically which will be explained later. The loop consists of the independent execution of the sequential code e , simulated for the processors 0 to p . This means that one iteration of the loop will have executed the code e for one processor. Hence, if we know the post condition $post$ that we would like to ensure after the block e , the invariant at the i -th iteration should include: $\forall j:\text{int}. 0 \leq j < i \rightarrow post[j]$.

The meaning is that for all the processors $j < i$, the code e has already been executed, so the post condition $post$ at the processor j holds. However, because parallel variables are simulated with the use of arrays in the WHY sequential program, there needs to be another part in the invariant, ensuring that at the i th iteration of the loop, we have not modified the array for the processors i to $p - 1$ yet. In other words, for each variable v modified in the loop, we need to have: $\forall j:\text{int}. i \leq j < p \ v[j] = v[j]@loopstart$ that is the value of v for the processor j must still be the same as the value at the beginning of the loop, denoted by the label **@loopstart**.

Variables Accessed Within the Loop. Managing the variables that are accessed within the loop raises a few difficulties. The first difficulty comes from the fact that a variable x can be translated in three different ways depending on its use:

1. If the variable is declared locally, and is only used within the sequential block, it is simply translated in a similar variable x ;
2. If the variable is used outside of the block, it can have different values depending on the processor. If it is not used with a *push* instruction, it can simply be translated by an array of variables of the same type;
3. If the variable is used with a *push* instruction, it is more difficult to use directly an array, because it is not possible in WHY to transfer pointers to a variable, which would be necessary during the communications; in that case, we chose to use a bigger array, containing all the variables used in DRMA accesses; that way, we can transfer in the communications the index of the variable in the array, rather than the variable itself.

In our translation, we chose to solve both difficulties by adding an intermediate step inside the loop. At each iteration, before and after the translated code, we add lines of code to store temporarily the variables in their “natural” name.

$[[t]]_i$	$=$	t
$[[!x]]_i$	$=$	$!x$
$[[e1; e2]]_i$	$=$	$[[e1]]_i; [[e2]]_i$
$[[\text{let } x = e1 \text{ in } e2]]_i$	$=$	$\text{let } x = [[e1]]_i \text{ in } [[e2]]_i$
$[[\text{let } x = \text{ref } e1 \text{ in } e2]]_i$	$=$	$\text{let } x = \text{ref } [[e1]]_i \text{ in } [[e2]]_i$
$[[x := e]]_i$	$=$	$x := [[e]]_i$
$[[\text{if } e1 \text{ then } e2 \text{ else } e3]]_i$	$=$	$\text{if } [[e1]]_i \text{ then } [[e2]]_i \text{ else } [[e3]]_i$
$[[\text{while } e1 \text{ do } e2 \text{ done}]]_i$	$=$	$\text{while } [[e1]]_i \text{ do } [[e2]]_i \text{ done}$
$[[\text{raise } (E \ e)]]_i$	$=$	$\text{raise } (E \ [[e]]_i)$
$[[\text{try } e1 \text{ with } E \ e2 \rightarrow e3 \text{ end}]]_i$	$=$	$\text{try } [[e1]]_i \text{ with } E \ [[e2]]_i \rightarrow [[e3]]_i \text{ end}$
$[[e1 \ e2]]_i$	$=$	$[[e1]]_i \ [[e2]]_i$
$[[l:e]]_i$	$=$	$l: [[e]]_i$
$[[\text{assert } \{p\}; e]]_i$	$=$	$\text{assert } \{[[p]]\}; [[e]]_i$
$[[e \ \{p\}]]_i$	$=$	$[[e]]_i \ \{[[p]]\}$
$[[\text{BSP-WHY operator}]]_i$	$=$	use of the prelude file

Figure 3.17. Formal rules for the transformation of sequential blocks.

<pre> proc_i := 0 ; loop0: while (!proc_i < bsp_nprocs) do { invariant (proc_i >= 0) and ... variant bsp_nprocs - proc_i } let i = ref (pararray_get i_par !proc_i) in let m = ref (pararray_get m_par !proc_i) in localloop0: while(!i < bsp_nprocs) do { invariant ... variant bsp_nprocs-i } bsp_send !i !m; i := !i+1 done; pararray_set i_par !proc_i !i; pararray_set m_par !proc_i !m; proc_i := !proc_i + 1 done; bsp_sync (void) ; </pre>	<pre> proc_i := 0 ; loop1: while (!proc_i < bsp_nprocs) do { invariant (proc_i >= 0) and ... variant bsp_nprocs - proc_i } let x = ref (pararray_get x_par !proc_i) in let y = ref (pararray_get y_par !proc_i) in y := bsp_findmsg 0; x := !y+1; pararray_set x_par !proc_i !x; pararray_set y_par !proc_i !y; proc_i := !proc_i + 1 done </pre>
---	---

Figure 3.18. Full generated WHY program of SimpleCode1.

There are several advantages with this method. First, the translation itself is made easier. A statement with x in BSP-WHY can directly be translated into a statement with x too, in most cases. Perhaps more importantly, the translated code is easier to understand. If the source BSP-WHY program was talking about x , the resulting WHY program will be talking about x . It might not be so big an issue, since the programmer is not expected to read the WHY program anyway. But it also translates directly in the proof obligations generated by WHY. If an obligation is not discharged automatically by the provers, it is necessary that the user be able to look at it, understand what part of the code it comes from, and make adjustments to the program assertions if it is the correct action. This is much easier if the proof obligation includes references to x , and not `pararray_get x i`, when the original program defined a variable x .

Also when translating the logic expressions, it is necessary to translate the variable in the same way as previously. When it is necessary to refer to the variable x as an array $\langle x \rangle$, or to the variable on a different processor than the current one, $x\langle i \rangle$ is transformed in the access to the i -th component of x .

(c) Translation of the Local Code

Finally, we have to give the translation of a single block, denoted by $[[e]]_i$, to the code that can be executed within the “for” loop. The transformation will have the index i of the processor as a parameter, named `proc_i`. The transformation of control instructions is straightforward, in the same way as previously, by walking the tree recursively. Fig. 3.17 gives the formal definition of this block transformation for the core-calculus where the parallel instructions (*put*, *send*, *etc.*) are not directly translated in an equivalent sequential code but are replaced by calls to the parameters axiomatised in the prelude file. Fig. 3.18 gives the final results of this transformation for the simple examples of Fig. 3.3.

(d) Finding the Invariant of the “for loop”

When writing our implementation of BSP-WHY, we encountered an unplanned difficulty: generating the invariant of the “for loop”. We will discuss how we dealt with them, and a possible evolution of the BSP-WHY tool. It is relatively easy to find all the variables that are modified by the block, and generate the second part of the invariant for these variables. Guessing an appropriate post-condition for the sequential code is however a bit trickier. In BSP-WHY, we try to guess the most general post-condition by calling the WHY program on the block of code, and inferring the post-condition. In case the programmer wishes to utilise a different invariant, he can do so by giving explicitly a post-condition to the block of sequential code.

The Problem. In our planning of the BSP-WHY tool, we did not anticipate the difficulty of generating invariants for the loop executing a block of code on every processor successively. At first glance, it appears to be a trivial problem. The “for loops” are simple in their concept, they execute the same independent code on the processors of our parallel machine, from the processor 0 to the processor $p - 1$.

Since the processors execute independently from each other, the invariant form is really trivial: it just states that the code has been executed, for the processors 0 to i , where i is the current processor.

However, the difficulty comes from the logic expression representing the execution of the local block of code. With our approach, it is not easy to obtain it automatically. BSP-WHY is designed as a tool that transforms a program from one parallel language to a sequential program, and then trusting WHY to certify the correct execution of the sequential program. It is not designed to infer a correct postcondition from a block of code.

Past Solution. In our first version of BSP-WHY, we partly solved the problem by asking the programmer to give, for every block of local code, a postcondition describing the result of the execution of the block of code. In a program such as the prefix calculus presented before, this is not too much of a difficulty: there are only two blocks of code, one leading to the synchronisation, and one leading to the end of the function. A function always has a postcondition, and it did not seem unreasonable to ask the programmer to give an assertion describing the environment state just before a synchronisation.

Using these postconditions, we were able to generate the correct invariants for the for loops. However, this approach raises some major issues.

First of all, even with such a simple example, the programmer has to provide an assertion in the middle of the program. It might make sense because of the synchronisation that immediately follow, so the programmer should have a clear vision of what messages are about to be sent, and be able to describe it in a logic formula. For instance, the direct prefix example introduced before is in a way an “ideal” situation. The program is composed of only two blocks of code, both pretty apparent to the programmer, one before the synchronisation, and one after. However, there are many algorithms that would translate in BSP-WHY programs in a much less intuitive manner and that detracts from the usefulness of the WHY tool, where the idea is for a programmer to give a specification of the program, *i.e.* a precondition and postcondition for the function, and to obtain automatically the proof of all necessary proof obligations, as given by the wp-calculus. Having to give intermediate assertions at different points of the program was actually the main problem of the basic Hoare logic, and this is why the wp-calculus is used instead. For this reason, asking for the postcondition after every sequential block of code seems like a step in the wrong direction. Let us take a simple example:

```
let j = ref 0 in
  while(!j < nprocs) do
    { invariant 0 ≤ j ≤ nprocs variant nprocs - j }
    bsp_send(!j, 0); bsp_sync(); j := !j + 1
  done
```

This is a basic example, sending a message (containing 0) to every processor successively in a while loop, synchronising after each message. It might not be obvious for someone not familiar with the BSP-WHY internal functioning, but if we apply the basic transformation as defined in the previous sections, such a program would result in four distinct blocks of sequential code. This means that with the basic solution of asking the programmer a postcondition for each block, he would have to provide four logic assertions in such a code. It would clearly be totally unreasonable.

Proposed Solution. BSP-WHY does contain some optimisations to help improve the readability of the transformed code. In particular, the first block of sequential code of this example is simply the constant 0 given when initialising the variable j . For such a code, BSP-WHY will not go through the full transformation and execution of a “for loop”, since it is obviously not needed. So in this example, the programmer

$$\begin{aligned}
[[c]]_i &= c \\
[[x]]_i &= x[i] \\
[[!x]]_i &= x[i] \\
[[\Phi(t)]]_i &= \Phi([[t]]_i) \\
[[old(t)]]_i &= old([[t]]_i) \\
[[at(t, l)]]_i &= at([[t]]_i, l) \\
[[x < j >]]_i &= ([[x]]_i)[j] \\
[[< x >]]_i &= x
\end{aligned}$$

$$\begin{aligned}
[[A(t_1, \dots, t_n)]]_i &= A([[t_1]]_i, \dots, [[t_n]]_i) \\
[[\forall x : \beta, p]]_i &= \forall x : \beta, [[p]]_i \\
[[p_1 \rightarrow p_2]]_i &= [[p_1]]_i \rightarrow [[p_2]]_i \\
[[p_1 \wedge p_2]]_i &= [[p_1]]_i \wedge [[p_2]]_i
\end{aligned}$$

Figure 3.19. Rules of transformation of logic terms, and logic predicates.

would not actually need to give the full four postconditions. However, these optimisations would only reduce the scale of the problem, without solving it entirely. It would still be very inconvenient to have to give the logic assertions.

For this reason, the current implementation of BSP-WHY tries to guess the correct postcondition, by running the WHY tool on a specially crafted file, and inferring the postcondition from the proof obligations generated by WHY. This is not the cleanest but it was the only convenient way that we found to generate this invariant without changing totally the design choices of BSP-WHY.

It is important to understand that this issue is not really relevant to the basic logic of our transformation. The method of transforming a parallel program into a sequential program, and then applying a tool to prove sequential programs, is proved to be correct in our work. However, the limitations of the specific tool that we chose to use, WHY2, raise this implementation issue that complicated our work. If we used a sequential proving tool able to guess the invariants of some loops, we might not have encountered this issue at all. In addition, BSP-WHY still allows the user to specify a postcondition on a block of code for the case where it would not be inferred automatically correctly. So even if it is not perfect, the user will still be able to prove his parallel program.

Another important point is that even though this invariant generation is not formally proved in any way, this does not detract from the insurance that if WHY accepts the sequential program as being correct, then the BSP-WHY source program is correct. The invariant is used as a necessary step to be able to prove some of the generated proof obligation, but a wrong invariant would never allow to fully prove an incorrect program. It is also interesting to note that the use of WHY3 might allow us to solve this particular issue in a much more satisfying way. WHY2 was presented as an opaque tool, transforming a program into proof obligations that were sent to a set of provers. WHY3 is much more flexible, with the ability to use an API to manipulate WHY logic terms.

3.2.5 Translation of Logic Assertions

In the previous sections, we described how we generate a sequential program from the parallel program. The new program will have the same result as the parallel code, which means that if we prove that the sequential code is correct, then the parallel code is correct too. However, in order to be able to prove the correctness of the sequential code, we have to translate all of the logic annotations in your BSP-WHY program.

There are two parts in this translation: (1) generated annotations from those that concerning only the local memory of one processor; (2) annotations that can describing the whole parallel state of execution.

(a) Translation of Local Assertions

The syntax of logic assertions was described in Fig. 3.2. We write $[[t]]_i$ the translation of the logic term t , applied on the processor i . Similarly, we write $[[p]]_i$ the translation of the logic predicate p , applied on the processor i . The latter depends on the former, because logic predicates are constructed from logic terms. We give in Fig. 3.19 the rules of the transformation, on both logic terms and predicates where:

- Because of the underlying memory model, variables x and dereferencing $!x$ are both translated into a p -array access $x[i]$ (Section 3.1.2 described in more details the choices behind the memory model).
- In the case of a logic function call, the transformation is simply applied recursively to the arguments.
- For references with `old(t)` and `at(t, l)`, the translation will be the same reference, but on the translated term.
- Finally, we have the new notations introduced by the BSP-WHY language. $x<j>$ corresponds to the variable x on the processor j , so it is translated as a p -array access, but on the j -th component: $x[j]$. $<x>$ corresponds to all the values of x , seen as a p -array; because of the memory model, that is exactly x .
- The translation of logic formula is a simple recursive call to the components of the formula, translated logic terms in a function call as stated previously.

(b) Translation of Global Assertions

There are two ways to give a global assertion in BSP-WHY.

1. First, the \mathbf{x} notation allows to give a general statement about the x variables of all processes, seen as an array. For instance, one could have the statement `sorted(< \mathbf{x} >)` to say that the values found on the different processors are increasing along with the `pid`.
2. Another kind of global annotations is when a local annotation is implicitly seen as global. For instance, if the postcondition of a parallel function f is of the form $\mathbf{x}=\text{pid}$, what is meant is that on every processor, x will be equal to the processor identifier.

We could have had a separate translation for the two kinds of assertions. However, we found it easier to always translate in the same way. The translation of a logic predicate, in a parallel code, is defined as follows: $[[p]] \equiv \forall i, 0 \leq i < p \rightarrow [[p]]_i$ that is, the WHY code: $\forall \text{proc_i:int, isproc}(\text{proc_i}) \rightarrow P$ where P is the result of the local transformation $[[p]]_i$ as described in the previous paragraph. The generated predicate $[[p]]$ is true if and only if the translations of p on all processors i all hold true. Let us give a few examples:

- A parallel function f has for postcondition the predicate $\mathbf{x}=5$. It is a local predicate, seen as global by stating that it is true on every processor. The translation is: $\forall \text{proc_i:int, isproc}(\text{proc_i}) \rightarrow \text{pararray_get}(\mathbf{x}, \text{proc_i})=5$; This is exactly what we intended when we wrote the postcondition, so the translation works well.
- A sequential function g has for postcondition $\mathbf{x}=5$. Since the function is not tagged as parallel, its postcondition will be translated only with the local translation. The result is thus $\mathbf{x}=5$, once again as intended.

Fig 3.20 gives the full WHY code from the direct prefix example of Fig. 3.4.

3.2.6 Dealing with Exceptions

Problematic. In the previous sections, we have ignored a problem that comes from having exception handling in the language. To better understand the issue, let us consider this example that includes a `try` statement:

```
try ( (if pid=0 then raise E else void); bsp_sync() ) with E → ...
```

If we naively follow the transformation as described earlier, the first step would result in the block tree show in Fig. 3.21. The `if` statement would be tagged as not being parallel, so it would constitute a block. The main step (the transformation from the BSP-WHY block tree to the WHY block tree) is mainly concerned with the safety of `if` and `while` nodes of the block tree. Since there are none here, it would not change things. The problem comes to light with the final step, the transformation of the sequential

```

parameter x : int farray ref
parameter z : int farray ref

let prefixes () = { init_envCsend(envCsend) }
proc_i:=0;
loop0: while (!proc_i<nprocs) do
{ invariant (proc_i>=0)
  and (forall proc_j:int. 0<=proc_j<proc_i →
    (forall j:int. (proc_j + 1<=j<nprocs) → in_send_n(paccess(envCsend,proc_j),j-(proc_j+1),lcast_int (paccess(x,proc_j))))
    and nsend(paccess(envCsend,proc_j))=nprocs-(proc_j+1))
  and (forall proc_j:int. proc_i<=proc_j<nprocs → paccess(envCsend,proc_j) = paccess(envCsend@loop0,proc_j))
  variant nprocs - proc_i
}
let y = ref (bsp_pid (void))+1 in
while (!y < nprocs) do
{ invariant (forall j:int. (proc_i + 1<=j<y) → in_send_n(paccess(envCsend,proc_i),j-(proc_i+1),lcast_int (paccess(x,proc_i))))
  and nsend(paccess(envCsend,proc_i))=y-(proc_i+1)
  variant nprocs - y
}
bsp_send !y (cast_int (parray_get x !proc_i)) ;
y:=!y+1
done;
proc_i:=!proc_i + 1
done;
bsp_sync (void);
proc_i:=0;
loop1: while (!proc_i<nprocs) do
{ invariant (proc_i>=0)
  and (forall proc_j:int. 0<=proc_j<proc_i → paccess(z,proc_j)=sigma_prefix(x, proc_j))
  variant nprocs - proc_i
}
parray_set z !proc_i (parray_get x !proc_i);
let y = ref 0 in
while (!y < bsp_pid void) do
{ invariant paccess(z,proc_i)=paccess(x,proc_i)+sigma_prefix(x, y)
  variant proc_i - y
}
parray_set z (!proc_i) ((parray_get z !proc_i) + (uncast_int (bsp_findmsg !y 0)));
y:=!y+1
done;
proc_i:=!proc_i + 1
done
{forall proc_i:int. isproc(proc_i) → paccess(z,proc_i)=sigma_prefix(x, proc_i)}

```

Figure 3.20. Full prefix example of the generated WHY program.

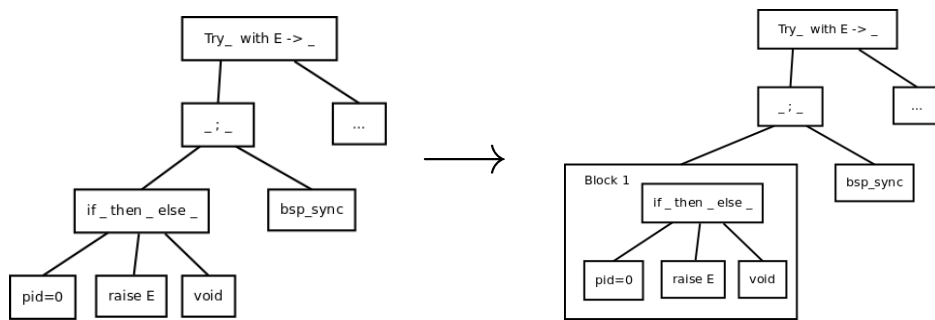


Figure 3.21. Illustration of the problem of a naive block decomposition with exceptions.

blocks. In this example, there is one, which would be wrapped inside a “for loop”. The resulting code would look like this:

```

try
let proc_i = ref 0 in
loopstart:while(!proc_i < bsp_p) do
{ invariant ... variant bsp_p - proc_i }
if !proc_i = 0 then raise E else void;
proc_i ← (!proc_i) + 1

```

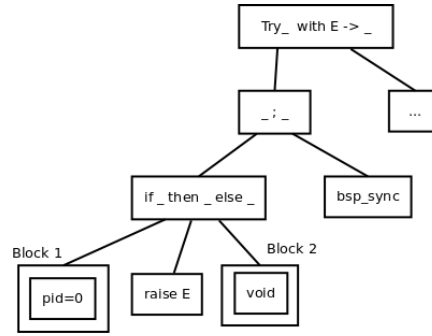



Figure 3.22. Solution to the block decomposition with exceptions.

```

done;
bsp_sync void
with E → ...

```

When running this code, the exception `E` will be raised in the first iteration of the “for loop”, and will actually stop the loop immediately. It means that the code of the other processors was not executed at all. This means the transformation is not valid for such a program without modification. Another point of note is that the parallel program itself is incorrect in this example, since it would result in the processor 0 alone not doing the synchronisation, and a deadlock would ensue.

Solution. To fix both of these issues, we modified the tagging of parallel code in the first step of the transformation, so that it recognizes that a `raise` instruction can have an effect on the parallelism of the program. However, tagging all of the `raise` instructions in this way would be too restrictive, since the `raise` can be in a purely local computation that does not need to be guarded like a parallel one — it is mainly the case for the `while` statement which is a syntactic sugar to an infinite loop adjunct to a `raise` of an exception when the loop is over.

A better solution is tagging a `raise` instruction if and only if the matching `try` subtree contains some other parallel code. It is the case in the example above, so the `raise` would be tagged as well. This is accomplished by modifying the `tag_parallel` algorithm, so that it takes an additional parameter, a list of exceptions that are to be tagged. Then, the program processes in two steps when it encounters a `try` statement. In a first step, the whole subtree is tagged without adding the exception in the list. If the result is a not-parallel subtree, it is enough. However, if the `try` subtree is tagged as parallel, then the matching `raise` statements will need to be considered as parallel too. This is done by tagging a second time the subtree, but this time with the exception added in the parameter. For example, a part of the code of this tagging function is:

```

let rec tag_parallel(tree,exc_l) = match tree with
| ... → ...
| bspTryWith(e1,exc,e2),tag → if(tag_parallel(e1,l) || tag_parallel(e2,l))
                             then (tag←true; tag_parallel(e1,exc::l))
                             else tag←false; !tag
| ... → ...
| bspRaise(exc),tag → tag←in(exc,exc_l); !tag

```

Applying this algorithm to our example gives the block tree of Fig. 3.22. For this example, the algorithm works as follow. First, the `raise` is recognized as having a parallel effect, so it will not be put inside a “for loop”. Second, as a result, the `if` statement becomes a node of the block tree, and will therefore be guarded with a proof obligation that the condition holds true for all the processors at the same time. In this example, it is not provable, so the deadlock is correctly detected.

3.3 Dealing with Subgroup Synchronisation

We have presented so far the transformation of programs in the strict BSP model, as defined in the BSPLIB standard. However, some extensions of this model are of a particular interest to us. In this section, we will show how to extend our transformation to include the possibility of synchronising over a subgroup of processors — mainly the possibility offered by the PUB library and MPI as introduced in Section 2.1.2.

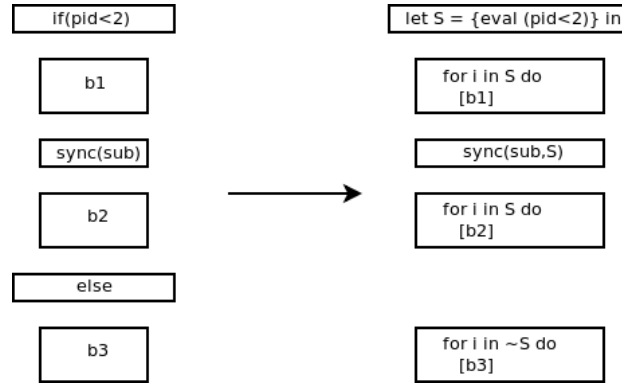


Figure 3.23. Main idea of the transformation with subgroup synchronisation.

Because this functionality can be seen as outside the strict BSP model, we have treated it later in our work and we give it a dedicated section. In this section we will thus discuss the various changes that were necessary in our transformation to be able to deal with subgroup synchronisation.

3.3.1 Subgroup Definition

(a) Extending the Operators of BSP-Why-ML

BSP-WHY is based on the BSP model. As thus, it seemed necessary to implement communicators in the BSP-PUB manner. However, BSP-WHY also aims at being able to model MPI programs that use only collective operations. For this reason, it is important to be able to simulate the use of communicators as it is done in MPI. We thus decided to provide the two ways of creating communicators in BSP-WHY:

1. The BSP-WHY parameter `bsp_partition` returns a communicator, without synchronisation, in the same way as it is done in the PUB library (this was detailed in Section 2.2), and with the same restrictions in the partition created.
2. The parameter `mpi_createcomm`, on the other hand, creates a communicator without any restriction on the processors in the groups.

In BSP-WHY, both parameters are without a synchronisation effect. However, it is possible to define a synchronising parameter that would simulate the `MPI_Comm_split` function using `mpi_createcomm`.

The BSP-WHY prelude file defines the datatypes used for the subgroup synchronisation. First, a subgroup of processors is defined as an array of booleans of size `p` (`bool farray`). This is easily interpreted: each processor of the BSP machine can either be part of the subgroup, or not. It is not possible, however, to simply assimilate a communicator with a subgroup. Several distinct communicators can match the same subset of processors. For this reason, communicators are stored in a list associating a communicator identifier and the corresponding subgroup of processors, with the `bsp_partition` and `mpi_createcomm` parameters returning such an identifier.

BSP-WHY does not as of now offer specific tools to write logic assertions about subgroups. A possible extension would be to provide, in addition to the existing `<x>i` and `<x>` notations, the possibility of referring to a variable on a specific processor inside a subgroup, or as the array of the values taken inside a subgroup.

(b) Main Idea of the Transformation

Fig. 3.23 illustrates the main idea of the simulation of a BSP program with subgroup synchronisation by a sequential program. The block decomposition remains the same as before. However, `if` statements in the parallel structure need another treatment because they allow the program to branch over different paths of execution depending on the subgroups.

In the abstract example of Fig. 3.23, a set of processors `S` is computed using “eval” to know which processors will effectively run the same branch of the `if`. Then, for each sequential block, we do not run the code for every processor, instead restricting to the processors in `S`, in this example the processors that have effectively their “id” lower than 2. Now the synchronisation is performed as above but we also check if `S` and the communicator `sub` match to forbid a deadlock: one processor does not synchronise

with other processors of the same group. This match is described below. Finally, for processors that are not in S we execute the **else** branch that is, for this example, for processors that have not their “id” less to 2.

In fact, the **bsp_pid(comm)** now depends of the communicator and has thus not the same value every time. In this way, a “true processor” can change of “id” depending on the communicator.

(c) New Pre-conditions to Primitives that Synchronise and Limitation

A New Pre-condition. It would be senseless to keep guarding the conditional statements as before, since it would only allow the synchronisation of all the processors. We already saw that in the BSP-WHY program, an additional argument is given to the synchronise primitive to tell which subgroup has to synchronise: the communicator. We thus need to verify in the execution of our WHY translated program that all the processors in the communicator are synchronising properly.

To do this, we now dynamically maintain a variable S during the execution of the translated program, that contains the set of the processors that are running the same branch of the code. To avoid deadlocks, for each **bsp_sync(comm,S)**, we check that all the processors of a subgroup will synchronise at the same time: **assert**(($\forall i:\text{int} \in S, \text{comm}[i] \subseteq S$) and ($\forall i:\text{int} \in S, \forall j:\text{int} \in \text{comm}[i], \text{comm}[i] = \text{comm}[j]$)).

That is **bsp_sync** (or every parameter with a synchronous effect) is now defined with this precondition. It ensures that it is called on a coherent set of processors at any time. For every processor in the set S , which is the set of the processors that will execute the call to **bsp_sync**, the subgroup that includes the processor is included in S . The subgroup of a processor i of S is denoted here by $\text{comm}[i]$, since it is an information contained in the communicator argument. The second part of the assertion states that if one processor synchronises over a communicator, then all the other processors of the communicator synchronises on it too. In the actual implementation, the pre-condition is harder to read, since WHY lacks the clarity of mathematical formula:

```
(forall i:int. isproc(i) → paccess(S,i) = true → incl(paccess(comm,i),S)) and
(forall i:int. isproc(i) → paccess(S,i) = true → forall j:int. isproc(j) →
  in_comm(j,paccess(comm,i)) → paccess(comm,i) = paccess(comm,j))
```

One might wonder why in the precondition of the **sync** statement, we write $\text{comm}[i] \subseteq S$, instead of for instance $\text{comm}[i]=S$. Let us consider again the simple example of Section 2.2. This is a program written in the PUB library that we would like to model with BSP-WHY. To understand better the second part of the invariant, we can take a simple example. Let us consider 3 processors synchronising over the subgroups (1,2), (2,3) and (1,3). Clearly, they would verify the first part of the assertion and still deadlock. It is thus mandatory to check that all the processors within a given subgroup are synchronising over this specific subgroup.

```
t_bsp subbsp;
int part[2];
part[0] = 2;
part[1] = bsp_nprocs(bsp);
bsp_partition (bsp, &subbsp, 2, part);
...
bsp_sync(&subbsp);
...
bsp_done (&subbsp);
```

The same program, in BSP-WHY, would look like this:

```
...
let subbsp = bsp_partition bsp 2 part in
...
bsp_sync(subbsp)
...
```

In this example, we do not have **if** statements to separate the behaviours of the disjoints subgroups. The **sync** operation is simultaneously executed by processors belonging to different groups. The important property is that inside every subgroups, all the processors do synchronise at the same time. It is true in this example, because the **&subbsp** are a partition of the bigger group.

In the precondition of the synchronisation, the variable S contains the set of the processors that execute the instruction. This variable is constructed dynamically by the WHY program, through additional instructions added during the transformation from BSP-WHY to WHY. We will present these modifications in the transformation when dealing with the subsynchronisation in the next section.

Limitation. So if one processor in a subgroup calls the synchronisation on that subgroup, every processors in the subgroup are executing it too. The restriction imposed by BSP-WHY is a bit more restrictive than what could be done in, for instance, the PUB library. But if a program can be executed normally in the BSP model, it is generally easy to transform it in an accepted BSP-WHY program. For example:

```
let S = {0,1} in
  if pid=0 then
    computation1; bsp_sync(S); computation2;
  else if pid=1
    then computation3; bsp_sync(S); computation4;
```

This program is correct in the BSP logic with subgroup synchronisation: the two processors of the subgroup S synchronise together, there is no deadlock. However, it is not a valid BSP-WHY program. The reason is that BSP-WHY sees that a `sync` operation is requested inside the `if`. To ensure that there can not be a deadlock, BSP-WHY asks that all the processors that enter the same branch of the `if` synchronise together. In the first branch, the processor 0 executes the `sync`, so all the processors that are with 0 in S must enter the same branch of the `if` conditional instruction. It is not true here (the processor 1 does not enter the same branch), so the pre-condition of the synchronisation will not be provable. In such cases, as previously, it is easy to rewrite the program, by factorising the synchronisation:

```
let S = {0,1} in
  if(pid in S) then
    begin
      if pid=0 then computation1; else if pid=1 then computation3;
      bsp_sync(S)
      if pid=0 then computation2; else if pid=1 then computation4;
    end
```

One can easily see that the result of the execution will be the same as for the previous program. However, this program is accepted by BSP-WHY. The first `if` instruction has a synchronisation instruction inside a branch, so BSP-WHY will check at that synchronisation that all the processors of a group enters the branch before synchronising, which is trivially true. On the opposite, the inside `if` instruction does not lead to a synchronisation, so it is simply handled inside a sequential block as seen previously, without additional requirements. The slight loss of efficiency due to the new `if` statement does not matter since the goal here is only to formally prove the algorithm, not to get the fastest program.

3.3.2 Transformation of Programs with the Subgroup Synchronisation

In the previous section, we presented parallel programs in the most usual BSP model, where all the processors of the parallel machine always synchronise together. The feature of subgroup synchronisation needs an extra treatment of the “sequentialisation” that is of the previous transformations.

Trivially, the first step of the transformation, the “decomposition into blocks”, remains almost unchanged: tagging of the parallel parts of the code stay identical and subgroup do not change this fact. The major changes come with the next steps of the transformation, that is the second step (“tree transformation”) and the third one (“local block transformation”).

First, the pre-condition of primitives of synchronisation need to be treated depending of the subgroup: not all the processors synchronise. Second, the “for loop” would only deal with a subgroup of the processors and we must keep the fact that other processors have no work or a different one depending of the program. As we say above, the main idea is to maintain the subgroups and to generate an appropriate code depending of these subgroups and to check where the flow of instructions depends on these subgroups: where to branch of the `if` statement diverge depending of the subgroups.

(a) A Tree Transformation that Deals with Subgroups

New rules. The main difference in this transformation, compared to the previously defined transformation without subgroup synchronisation, is that the transformation $[[t]]$, applied to the block tree, is now refined in a more precise transformation, $[[t]]_S$, which means that the block tree t will be executed on the processors of the set S . This is useful to handle the `if` statement, as we have seen in the example, and the `while` statement is similar.

We also define a parameter that will be used to find what processors have to execute a given code in the case of a `if` statement. `evalCond c S` returns the subset of the processors of S for which the condition c holds. Note that the rules for the other structures are exactly the same as without sub-synchronisation with the exception that they now carry the S parameter. We now give the new rules of the transformation using these definitions.

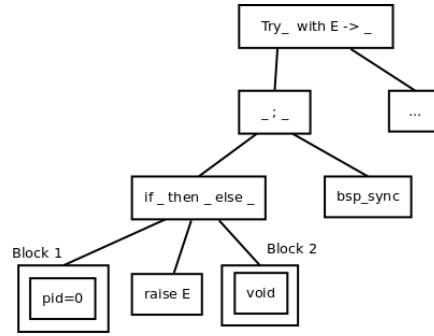


Figure 3.24. Block Decomposition with Exceptions.

$$[[\text{if } c_1 \text{ then } c_2 \text{ else } c_3]]_S \implies \begin{array}{l} \text{let } S_1 = \text{evalCond } c_1 \text{ } S \text{ in} \\ [[c_2]]_{S_1}; \\ [[c_3]]_{S \setminus S_1} \end{array}$$

Handling the Exceptions and the While Statement. As before, special care must be taken when considering the exception handling. Let us look again at the same example as in Section 3.2, the block decomposition of a simple `try` structure, with a `if` statement inside (Fig. 3.24).

The problem becomes apparent with the transformation of the `if` statement. With our new transformation, the generated code is supposed to be the succession of 3 steps:

1. we evaluate the condition ($pid = 0$), and the subset S of processors for which it is true ($\{0\}$);
2. we execute the `then` part of the code for all processors in S ;
3. we execute the `else` part of the code for all processors not in S .

However, in this case, the step 2 will actually raise an exception, that is only caught outside of the scope of the `if` statement. Thus the whole step 3 would be ignored, which is clearly not what we want.

There are several approaches to solve this problem. We chose to implement one that is not the most general, without having an overly restrictive impact on the scope of accepted programs. If we think back about the issue when we did not have sub-synchronisation, the problem was solved by the `valid` parameter on the `if` condition, ensuring that all the processors would raise the exception at the same time. We extend this idea by now imposing a pre-condition to the `raise` statement itself: it must be executed by all the processors that initiated the matching `try` statement. Thus, the transformation rules for the `try` and `raise` statements are now as follows:

$$\begin{aligned} [[\text{try } c_1 \text{ with } E \text{ } x \rightarrow c_2 \text{ end}]]_S^l &\implies \text{try } [[c_1]]_S^{(E,S)::l} \text{ with } E \text{ } x \rightarrow [[c_2]]_S^l \text{ end} \\ [[\text{raise } (E \text{ } c)]]_{S'}^{(E,S)::l} &\implies \text{assert } \{S = S'\}; \text{raise } (E \text{ } [[c]]_{S'}^{(E,S)::l}) \end{aligned}$$

The transformation takes one more argument, a list of the exceptions that are being caught, and the subgroup of processors that executed the `try`. With this information, it is easy to add an proof obligation for the `raise` statement, to ensure that all the processors in the subgroup do raise the exception.

The principal user of exceptions is of course the `while` loop, and it is directly affected by this choice. In practice, the restriction is that for a `while` loop in the parallel structure of the program (while loops inside purely sequential calculations are still free), all the processors that execute the loop must exit it at the same time. This is true of many parallel programs with subgroup synchronisation, but not all of them.

It would be possible to give a more general transformation that accounts for any kind of exception handling imaginable. This would significantly complicate the transformation, however, and would be of little interest since the programs we study almost never use exceptions in this way. In practice, only the `while` limitation really could be felt. In the actual transformation, BSP-WHY does not use the loop expansion of the `while` statement, preferring to translate it directly in its `while` form to generate a more readable program in output. Because of this, it is possible to give two variants of the while transformation; one following the choice that we made for the exception handling, and one variant more complex, but allowing some processors to leave the loop before others.

$$[[\text{while } c_1 \text{ do } c_2 \{ \text{invariant } i \text{ variant } v \}]]_S \implies \\ \text{while } \text{valid}([c_1]_S, S) \text{ do } [c_2]_S \{ \text{invariant } [i] \text{ variant } [v] \}$$

$$[[\text{while } c_1 \text{ do } c_2 \{ \text{invariant } i \text{ variant } v \}]]_S \implies \\ S' := S ; \text{while } S' := \text{evalCond } c_1 \text{ } S' \text{ do } [c_2]_{S'} \{ \text{invariant } [i] \text{ variant } [v] \}$$

The first option closely follows the previous transformation for the **while** loop: the valid parameter ensures that the condition remains true on every processor that executes the loop. The only difference is that now, the loop can be executed within a subgroup instead of all the processors. The invariant and variant are thus obtained in the same way as before.

In the second option, we enrich the loop by allowing processors to exit it while it progresses. Because of this, we need to update, at each iteration, the set of the processors that are currently executing the loop. This is done by introducing a variable S' , which is updated from the computation of the condition on each processor. While the invariant is still very much the same, the variant generation is a bit more tricky, since there is no way of knowing beforehand which processor will stay the longest in the loop. Instead, we need to provide some measure obtained from the variants on every processors, for instance their sum.

Because of the added complexity in the second option, both in code and with the proof obligations, we chose to provide the first option by default, with the possibility to request the generation of the more complex loop when necessary.

(b) A new Local Block Transformation that Deals with Subgroups

The idea of the transformation of a local block is similar to the one without subgroup synchronisation. However, instead of executing the block for all the processors successively, we only execute it for the processors that are running that part of the code. This is exactly what is denoted by the variable S . For this, we introduce a shortened notation: **for** i **in** S **do** c . The “for loop” means that we execute sequentially the instructions c for all the processors in S . This generates a code of the following form:

```
let i = ref 0 in
while !i < nprocs do
{
  invariant inv
  variant nprocs - i
}
if procIn i S then
  c;
  i ← !i + 1;
done
```

where the **procIn** functions just tells if i is in the set S — the treatment of the invariant **inv** is described above. With this notation, our transformation is now the following:

$$[[\text{Block}(e)]]_S = \text{for } i \text{ in } S \text{ do } [[e]]_{i,S}$$

(c) A new Generation of Invariants

The determination of the invariant of “for loop” can be made very similarly to what we did before. The difference is that for the processors that are not in S , we always need the second form of invariant to hold. That is: $\forall j:\text{int}. 0 \leq j < i \text{ and } j \in S \rightarrow \text{post}[j]$. And, as above, we also need to express that the computation has not yet been processed for some processors. This is done using the two following invariants

1. $\forall j:\text{int}. i \leq j < \text{nprocs} \rightarrow v[j] = v[j]@loopstart$, the computation has not been done for these processors;
2. $\forall j:\text{int}. 0 \leq j < i \text{ and } \neg(j \in S) \rightarrow v[j] = v[j]@loopstart$, processors not in S will never do the computation;

(d) Transformation of Local Code

When transforming local code from BSP-WHY to WHY, there are almost no modification compared to the standard BSP model. The main point here is that for synchronising parameters, we add the S argument containing the set of the processors that run the synchronisation. This is necessary to ensure the proper synchronisation. As explained before, the synchronisation parameters are defined with two arguments,

the array of the communicators and the set of the processors that execute the synchronisation. Both arguments are then used to define the correct precondition.

Similarly, an argument is added for all of the function calls, when the function contains parallel code. This is needed to transmit the information of S inside the functions, which could be needed for a synchronisation.

3.4 Related Work

3.4.1 Other Verification Condition Generators

We can cite WHO [175] [[56]] which is a derivation of WHY for ML programming with *polymorphic* functions, and in the same spirit the PANGOLIN system [237]. Both could be interesting for proofs of ML extensions of BSP [117]. But these tools are not yet stable enough to be used as basis for a parallel extension.

Close to WHY, we can cite the BOOGIE programming language (with the SPEC # annotation language) [17] [[57]]. The YNOT System [217] is an extension to the COQ proof assistant, able to reason about imperative higher-order programs, including functions with side effects as arguments, modular reasoning, while being able to reason about aliasing situations thanks to the separation logic.

3.4.2 Concurrent (Shared Memory) Programs

To our knowledge, the first work on deductive verification of concurrent programs is the famous Owicki-Gries proof system [226]. It is an extension of Hoare logic to parallel programs with shared-variable concurrency. It provides a methodology for breaking down correctness proofs into simpler pieces. First, the sequential *components* of the program are annotated with suitable assertions. Then, the proof reduces to showing that the annotation of each component is correct, and that each assertion of an annotation is invariant under the execution of the actions of the other components — the so-called *interference-freedom* of proof outlines. The main drawback of the Owicki-Gries method is that it is not compositional: to perform the interference-freedom tests for some component once requires information about the implementation of all other components. Another drawback is the massive use of *ghost* codes (auxiliary variables) needed for proving parallel programs using the Owicki-Gries method. For example, the number of interference-freedom tests is polynomial to the number of sequential components. Note that a formal analysis of what is provable without these ghost codes has been done in [207].

In order to remedy to the above problems, different solutions have been proposed. In [220, 222], the authors proposed a compositional extension of the Owicki-Gries's rules, a wp-calculus, each implemented in the theorem prover ISABELLE. The authors also applied their method for verifying a concurrent garbage collector [221]. Three advantages of the method are (1) the use of a theorem prover ensures a great confidence in the implementation; (2) the method allows verification of open systems, *i.e.* systems which interaction with the environment can be specified without knowing the precise implementation of the environment; this makes the method suitable for top-down design; (3) parametrised concurrent programs (programs with an unknown and unbound number of threads) can be directly verified in the system; this is achieved by modelling the parallel constructor such that its argument is a list of component programs; then, the length of the list can be fixed or left as a parameter. The main drawback (except that it only works for shared-memory programs) is still the number of generated conditions, that rely on “rely-guarantee” which seems to be not provable by automatic provers: they require extensive human guidance.

Notice also the work of [128] where annotations are used to prove safety of concurrent ADA programs, that is, they are deadlock free and without failure. Now, there are also some studies of proof obligations for concurrent programs; for example [223] presented a *Concurrent Separation Logic* as an extension of Separation Logic for reasoning about shared-memory concurrent programs with Dijkstra semaphores. [162] presents an operational semantics for Concurrent C minor which preserves as much sequentially as possible (coarse-grained spirit), by talking about permissions within a single thread instead of concurrency. This semantics is based on ideas from Concurrent Separation Logic: the resource invariant of each lock is an explicit part of the operational model. This model is well suited for correctness proofs done interactively in a proof assistant, or safety proofs done automatically by a shape analysis such as [138]. However, currently no tools for generating proof obligations are provided and it is not clear how hard the obligations would be. A first work in this direction has been done in [32].

[41] presents a type system that prevents data races and deadlocks (as in [162]) by enforcing that locks are acquired in a given locking order, and this order can be changed dynamically. This system supports thread-local objects and coarse-grained locking of shared objects. The type system allows concurrent

reading only for immutable objects.

In the same way, [190] presents a sound and modular verification methodology (implemented for an experimental language with some not-trivial examples) that can handle advanced concurrency patterns in multi-threaded, object-based programs. It prescribes the generation of verification conditions in first-order logic (well-suited for solvers). The language supports concepts such as multi-object monitor invariants, thread-local and shared objects, thread pre- and post-conditions, and deadlock prevention with a dynamically changeable locking order. [169] extends the BOOGIE framework for concurrent programs with a kind of locking strategy. In the same way, a VCG for concurrent C programs has been designed in [63] [[58]]. This VCG has been used for the development of the Microsoft Hyper-V hypervisor — see Chapter 6 for a discussion on hypervisors. The hypervisor is highly optimised for multi-core hardware and thus contains a number of custom concurrency control mechanisms and algorithms, mostly using fine-grained concurrency control. The model, with uniform treatment of objects and threads, is very similar to the one employed in Concurrent SPEC # and CHALICE [189]. There is also the VERIFAST tool [168, 233] for deductive verification of C and JAVA programs using separation logic [[59]]. Other VCG for separation logic are describe in [233]. All these tools are well defined for share-memory concurrency but not for distributed memory and HPC computing.

It is not clear if *locking strategies* are very suitable for high-performance applications [188]. BSP is by nature a coarse-grained and deadlock-free model which is used for high-performance problems and now in multi-core/GPU applications. Even if proof of concurrent programs is clearly useful (servers, *etc.*), parallel programming is not concurrent programming. High-performance programs are much simpler [188] (many time more coarse-grained) and BSP programs are even simpler. They can clearly be simulated by share-memory fork/lock concurrent programs by explicitly separating the local memories and allowing communications with copies of the data [272]. Global synchronisation would be implemented using a sufficient number of locks. But, that would not use the structural nature of the BSP programs and the understanding of the program to simplify the obligations.

3.4.3 Distributed and MPI Programs

A first work on distributed programs was done in [185]. It has served as a basis for the TLA^+ logic. But no work or a tool for VCG calculus was proposed.

In [79], the author proposes a data-parallel (SIMD, Single Instruction Multi Data) extension of C and its embedding into a verification environment based on ISABELLE. The data-parallel operations are mostly binary like operations on arrays. Another work on data-parallel programs is the one of [39]. The authors have designed a set of Hoare’s rules and a wp-calculus for proving data-parallel programs. Exchange of data is done using shared arrays. The two drawbacks are (1) there is a lack of mechanised proofs; (2) the programs need a lot of annotations (not only loop invariants) which can repel a user; (3) the method does not hold if a program modifies the variables in the assertions (a data-race). For the latter, auxiliary variables can be used but make the proofs slightly more complex.

MPI is the most used library for high-performance computing. It is therefore natural to study safety issues related to MPI programs. But this is very challenging due to the number of routines (more than one hundred), the concurrent nature of these primitives, the lack of formal specifications — even if works such as [197, 276] exist, some cases are not taken into account because of the lack of specification and too much dependence on the architecture. This enormous number of routines in the API makes difficult to trust any formal specification of a complete MPI *e.g.*, a non-trivial case could have been forgotten?

They are many works and tools dedicated to MPI. Surveys could be found in [85, 135, 180, 203, 250, 281] and in HPCBUGDATABASE [[60]]. These tools help to find some classical errors, but not all of them. For example, in [273], the authors propose to check dynamically if the parameters of collective operators are consistent — *e.g.* if the size of data to send is non-negative. Note that this kind of tools works well for many situations in development phases, but is not sufficient.

[284] presents a tool that directly *model-check* the MPI source code, executing its interleaving with the help of a verification scheduler (producing error traces of deadlocks and assertion violations). It allows an engineer to check its MPI code against deadlocks on a **p**-processors machine before running it on this machine. A large number of MPI routines (including DRMA ones [231]) are considered. This technique was also used to find *irrelevant barriers* [249]. The main idea (which is also the one of [251, 253]) is to analyse the code by *abstract interpretation* and to produce an output for model-checkers by eliminating all parts of the code which are not in the *communication schemes*. First implementations used the model-checker SPIN, but now specific and more efficient checkers are considered. The main advantage of these methods is to be “push-button”. The two main drawbacks are (1) they only consider deadlocks and assertion

violations (it is still an important problem for MPI programs) (2) programs are model-checked for a *predefined* number of processors which is less than 64 in [284]. That is clearly not a *scalable* approach. The BSP spirit is to write programs for any number of processors, so proofs of BSP programs should do the same thing. A last problem with model checking parallel programs is state explosion, *i.e.* the fact that the number of states of a program typically can grow dramatically with the number of processes.

As said above, these methods generally cannot scale beyond a relatively small number of processes, but defects, which usually appear only in large configurations, can often be detected in much smaller configurations using symbolic execution. This original solution is the TASS [256] [[61]] and FEVS tools [255] [[62]]. TASS uses *symbolic execution* and *explicit state enumeration* techniques to verify that safety properties of MPI programs hold for all possible executions within user-specified bounds. TASS has its own internal support for simplifying symbolic expressions, placing them into a canonical form, and dispatching some of the proof obligations; for those it cannot dispatch itself, it uses CVC3. TASS can also use comparative symbolic execution (in the spirit of FEVS) to verify that two programs are functionally equivalent (*i.e.* “input-output equivalent”). The basic idea is to construct a model in which the two programs run sequentially, one after the other, and at the end the outputs are compared. The state space of this model is then exhaustively explored. This technique is particularly useful in computational science, to compare a complex parallel version of a program (the “implementation”) to a simple, trusted sequential version (the “specification”). Techniques to verify program assertions using symbolic execution exhibit a significant limitation: they typically require to impose (small) bounds be imposed on the number of loop iterations. The use of *loop invariants* allows to overcome this limitation. In [257], the author proposes a solution using a new symbolic execution technique (with special “collective loop invariants”) that he uses to verify assertions in MPI programs with *unbounded loops*. Programs are then checked (a model-checking technique using POR reductions) for unknown sizes of data — but still for a fixed number of processor only. But the author note that discovering these collective loop invariants is currently to the charge of the programmer (as in our work) and these invariants are limited; for example, there is no way to express that the number of messages is invariant. The method still has the advantage of greatly reducing the number of necessary loop invariants.

The approaches of symbolic verification as well as VCG tools, suffer to the main limitation: as it now stands, models of the programs must be built by hand. This requires significant effort and a degree of skill from the user. The ideal situation would be to have tools that automatically extract the models from source code, at least for specific domains [114, 254].

Currently, we are either not aware of verification condition generators tools for MPI programs. We think that performing a sequential simulation (as done by our BSP-WHY tool) of any kind of MPI programs is not reasonable. Continuations would be necessary to simulate the interleaving of processes: that would generate unintelligible assertions. But collective MPI routines, which simplifies the life of parallel programming [262], can be seen as BSP programs, and certainly many MPI programs could be transformed into BSP ones. Automatically translating this class of programs is a possible way to analyse MPI programs [211]. We leave the aim of substantiating this claim for future work.

The only exception is the work of [282, 283] with their prototype tool call HEAP-HOP [[63]]. HEAP-HOP is a VCG for programs with synchronous and copyless sending. A concurrent separation logic is used for the assertions (pre- and post-conditions and loop invariants). As an example, the authors propose to verify a load balancing algorithm for binary trees for two processes. This tool is thus able to take into account some MPI programs. But it is still limited to a fixed number of processes – *e.g.* two for the load balancing algorithm of a consumer/producer algorithm. Using BSP-WHY, the user can prove its programs for any number of processors.

3.4.4 Proof of BSP Programs

Different approaches for proofs of BSP programs have been studied. In [117, 130, 271], functional BSP programs have been proved correct using Coq. The BSP primitives of extension of ML call BSML were axiomatised in Coq. In this way, one can use Coq for programming BSP algorithms and BSML programs can be extracted from Coq in the form of OCaml code. In [119], we presented the correctness of a classical numerical computation (the N-body problem) using a mechanised operational semantics. But, by using semantics inside COQ, proofs of correctness were too hard.

The derivation of imperative BSP programs using the Hoare’s axiom semantics [54] followed by the generation of correct C code [290] also exists. The two main drawbacks of this approach is a lack of an implementation of a dedicated tool for the logical derivation, which implies a lack of safety; users make hand proofs which are not machine checked; moreover, it is impossible to verified users existing codes.

The derivation of imperative BSP programs using the Hoare's axiom semantics has also been studied in [55, 170, 265, 266]. More recently, these works were extended for subgroup synchronisation in [264]. All of these approaches lack of mechanised proofs. Moreover, they are close to refinement “à la B” since they give logical rules (close to Hoare logic's axioms and inference rules) for derived algorithms from specifications — a wp calculus is also given in [264]. On the contrary, using deductive verification, we begin with a program and by adding logical assertions, we prove the correctness of the said program.

4 Case Studies

This chapter subsumes the works of [110] and [120, 123]

Contents

4.1 Simple BSP Algorithms	71
4.1.1 Parallel Prefix Reductions	71
4.1.2 Parallel Sorting Algorithm	73
4.1.3 Mechanical Proof	75
4.2 Parallel State-space Construction	75
4.2.1 Motivations and Background	75
4.2.2 Definitions and Verification of a Sequential State-space Algorithm	77
4.2.3 Verification of a Generic Distributed State-space Algorithm	79
4.2.4 Dedicated Algorithms for Security protocols	82
4.3 Related Work	87
4.3.1 Other Methods for Proving the Correctness of Model-checkers	87
4.3.2 Verification of Security Protocols	88
4.3.3 Distributed State-space Construction	89

In this chapter, we first give some simple examples of use of BSP-WHY and mainly show how many proof obligations are discharged by automatic provers. Then, we focus on using BSP-WHY, for the verification of state-space construction algorithms which is the basis of model-checking. We mainly study three algorithms (one sequential, one distributed and another one dedicated to the state-space of security protocols) as a first step towards mechanically-assisted deductive verification of model-checkers. Because of a lack of time, we are currently not able to provide an example that uses the subgroup synchronisation capabilities of BSP-WHY.

4.1 Simple BSP Algorithms

All codes which are not given in the thesis are available as examples at the BSP-WHY web page¹. In this section, we first present two examples: parallel prefix calculations (Section 4.1.1), and a parallel sorting algorithm (Section 4.1.2). We then give the results obtained with automatic provers in Section 4.1.3.

4.1.1 Parallel Prefix Reductions

(a) Direct Method

Our first example is a simple one-step parallel prefix reduction, that is obtaining the $\oplus_{i=0}^k v_i$ on each processor k , with processor i initially holding v_i (this is the classical `MPI_SCAN`) for an operation \oplus . Here, we used integers and addition for \oplus but a polymorphic program can be considered. Using BSMP routines, we can give the BSP-WHY code of Fig. 4.1 (left).

The program starts with a distributed parameter x (with the notation `parameterg`), which contains the initial values, with one value on each processor. The prefixes are computed by the program in the z parameter. We use the user-defined logic term `sigma_prefix(X, n1, n2)` to describe the partial sums, that is $\sum_{i=n_1}^{n_2} X[i]$. The programs is mainly composed of two `while` loops. In the first loop, each processor sends its value in a message to each processor with a greater `pid` than itself. The instruction `bsp_sync`

¹<http://www.lacl.fr/fortin/BSP-Why/>

<pre> parameterg x: int ref parameterg z: int ref logic sigma_prefix : int farray,int → int axiom sigma_prefix1 : forall t:int farray. sigma_prefix(t,-1) = 0 axiom sigma_prefix2 : forall i:int. isproc(i) → forall t:int farray. sigma_prefix(t,i) = t<i> + sigma_prefix(t,i-1) let prefixes () = {comm_empty (<envCsend>)} (let y = ref (bsp_pid void + 1) in while(!y < bsp_nprocs) do { invariant (y > (bsp_pid)) and envCsendIs(j,bsp_pid + 1,y,lcast_int(x)) variant bsp_nprocs - y } bsp_send !y (cast_int !x); y := !y + 1 done); { envCsendIs(j,bsp_pid + 1,bsp_nprocs,lcast_int(x)) } bsp_sync; (z:=x; let y = ref 0 in while(!y < bsp_pid void) do { invariant (0 <= y <= bsp_pid) and z=x+sigma_prefix(<x>, y-1) variant bsp_pid - y } z := !z + uncast_int (bsp_findmsg !y 0); y := !y + 1 done) { z=sigma_prefix(<x>, bsp_pid)} </pre>	<pre> parameter x : int farray ref parameter z : int farray ref logic sigma_prefix : int farray,int → int axiom sigma_prefix1 : forall t:int farray. sigma_prefix(t,-1) = 0 axiom sigma_prefix2 : forall i:int. isproc(i) → forall t:int farray. sigma_prefix(t,i) = paccess(t,i) + sigma_prefix(t,i-1) let prefixes () = { comm_empty(envCsend) } proc_i := 0 ; loop0: while (!proc_i < bsp_nprocs) do { invariant (proc_i >= 0) and (...) variant bsp_nprocs - proc_i } let y = ref (bsp_pid (void))+1 in while (!y < bsp_nprocs) do { invariant (...) variant bsp_nprocs - y } bsp_send !y (cast_int (parray_get x !proc_i)) ; y:=!y+1 done ; proc_i:=!proc_i + 1 done; bsp_sync (void) ; proc_i := 0 ; loop1: while (!proc_i < bsp_nprocs) do { invariant (proc_i >= 0) and (...) variant bsp_nprocs - proc_i } parray_set z !proc_i (parray_get x !proc_i); let y = ref 0 in while (!y < bsp_pid void) do { invariant paccess(z,proc_i)=paccess(x,proc_i) + sigma_prefix(x, y-1) and (...) variant proc_i - y } parray_set z (!proc_i) ((parray_get z !proc_i) + (uncast_int (bsp_findmsg !y 0))) ; y:=!y+1 done ; proc_i:=!proc_i + 1 done { forall proc_i:int. isproc(proc_i) → paccess(z,proc_i)=sigma_prefix(x, proc_i)} </pre>
--	--

Figure 4.1. BSP-WHY code of the direct prefix computation (left) and its WHY transform (right).

then executes the synchronisation barrier. In the second loop, each processor computes the sum of all the received values.

Note the use of our notations in the program: x designs the value on the current processor, $\langle x \rangle$ refers to the global array and $x \langle i \rangle$ refers to the value of x at processor i . `envCsendIs` is a syntactic sugar to describe the communication environment, without having to use the internal list description and its associated functions. The `bsp_send` and `bsp_findmsg` functions can be used to transfer any type of data. For this reason, we use the `cast_int` and `uncast_int` functions, that encapsulates the data in a generic *value* data type.

The generated WHY code is in Fig. 4.1 (right). The BSP-WHY engine has, as expected, separated the program into two sequential blocks, linked by the synchronisation operation. Around those two blocks, a `while` loop has been constructed, so that the code is executed sequentially for each processor `proc_i`. As described in Section 3.2.4, the invariant generated for the `while` loops are quite complex, and for a better readability we omitted large parts of them. We can note that the distributed variables, such as x and z , are translated into arrays of size \mathbf{p} , using the type $\mathbf{p} - \text{array}$. Reading or writing such a variable is done with the `parray_get` and `parray_set` functions, or in the logic world their counterparts `paccess` and `pupdate`. Local variables, with a lifespan within a sequential block do not need to be translated into an array. For instance, an access to y will remain the same. Note that the WHY source code generated by BSP-WHY is actually not supposed to be manipulated by the end-user, and is in general significantly less readable by a human. It is now possible to use the generated code, and feed it to the WHY program, in order to generate the proof obligations for any supported back-end. The results will be presented in Section 4.1.3.

(b) Logarithmic Reduction

The above reduction does not make use of parallelism and we may prefer to reduce in a multi-step manner, the classical logarithmic way, doing the combinations locally. In our second example, the algorithm combines the values of processors i and $i + 2^n$ at processor $i + 2^n$ for every step n from 0 to $\lceil \log_2 \mathbf{p} \rceil$.

<pre> let scan () = let Xin = ref vundef in let X' = ref (cast_int !X) in let i = ref 0 in begin push(X',1); {envCpush=cons(X',nil)} bsp_sync; init: while ((pow_int 2 !i) < bsp_nprocs) do { invariant (0 <= pow_int(2,i) <= bsp_nprocs) and X' = sigma_prefix(<X'@init>, bsp_pid - pow_int(2,i-1), bsp_pid) } </pre>	<pre> variant bsp_nprocs-i } if (bsp_pid >= (pow_int 2 !i)) then get (bsp_pid - (pow_int 2 !i)) X' 0 Xin sizeof_int; bsp_sync; if (bsp_pid >= (pow_int 2 !i)) then X' := cast_int((uncast_int !Xin) + (uncast_int !X')); i := !i + 1 done {X' = sigma_prefix(<X'@init>, 0, bsp_pid)} end </pre>
---	---

Figure 4.2. BSP-WHY code of the logarithmic reduction.

One can write the main loop as:

```

while ( (pow_int 2 !i) < bsp_nprocs ) do
  if (bsp_pid >= (pow_int 2 !i)) then
    begin
      bsp_get (bsp_pid - (pow_int 2 !i)) X' 0 Xin;
      bsp_sync void;
      X' := cast_int((uncast_int !Xin) + (uncast_int !X'));
    end
  else
    bsp_sync;
  i := !i + 1
done

```

This is the typical case where our block decomposition fails: not all the processors run the same `bsp_sync` and our tool will generate unprovable assertions. But the program can be rewritten by factoring the two `bsp_sync`:

```

  if (bsp_pid >= (pow_int 2 !i)) then
    bsp_get (bsp_pid - (pow_int 2 !i)) X' 0;
    bsp_sync;
  if (bsp_pid >= (pow_int 2 !i)) then
    X' := cast_int((uncast_int !Xin) + (uncast_int !X'));

```

Fig 4.2 gives the full code of the logarithmic reduction with assertions.

4.1.2 Parallel Sorting Algorithm

Parallel Sorting by Regular Sampling Algorithm. Our last example is the **Parallel Sorting by Regular Sampling** algorithm (PSRS) of Schaeffer in its BSP version [272]. The goal is to have data locally sorted and that processor i have smaller elements than those of processor $i + 1$. Data also need to be well balanced. We assume n elements to sort where $\mathbf{p}^3 \leq n$ and elements are well distributed over the processors — each processor have $\frac{n}{\mathbf{p}}$ elements.

The PSRS algorithm proceeds as follows. First, the local lists of the processors are sorted independently with a sequential sort algorithm. The problem now consists of merging the \mathbf{p} sorted lists. Each process selects from its list $\mathbf{p} + 1$ elements for the primary sample and there is a total exchange of these values. In the second super-step, each process reads the $\mathbf{p} \times (\mathbf{p} + 1)$ primary samples, sorts them and selects \mathbf{p} secondary (main) samples. Note that the main sample is thus the same on each processor. That allows a global choice of how to remap the data. In the third super-step, each processor picks a secondary block and gathers elements that do belong to the assigned secondary block. In order to do this, each processor i sends to processor j all its elements that may intersect with the assigned secondary blocks of processor j . To simplify we suppose elements of the “same size”. The BSP cost of the first super-step is thus:

$$\frac{n}{\mathbf{p}} \times \log\left(\frac{n}{\mathbf{p}}\right) \times c_e + \frac{n}{\mathbf{p}} + (\mathbf{p} \times (\mathbf{p} + 1) \times s_e) \times \mathbf{g} + \mathbf{L}$$

where c_e is the time to compare two elements and s_e size of an element. In [272] it is shown that each processor receives at most $\frac{3n}{\mathbf{p}}$ elements. The BSP cost of the second super-step is thus:

$$\frac{n}{\mathbf{p}^2} \times \log\left(\frac{n}{\mathbf{p}^2}\right) \times c_c + \frac{n}{\mathbf{p}^2} + \frac{3n}{\mathbf{p}} \times s_e \times \mathbf{g} + \mathbf{L} + \text{time}_{\text{fusion}}$$

```

1 let psrs (a:int array) =
2 {
3   (* size of a > p+1 on each processor *)
4   array_length(a)>p+1
5 }
6 (* local sort *)
7 local_sort__(a);
8 (* first sampling *)
9 let samp1 = sampling(!a) in
10 (* create the buffer of reception *)
11 let total_samp1 = ref (cast_int_farray
12   (make_array_int (bsp_nprocs*(bsp_nprocs+1)) 0)) in
13 let i = ref 0 in
14 push(total_samp1,bsp_nprocs*(bsp_nprocs+1));
15 { sorted_array(a, 0, array_length(a)-1)
16   and envCpush=cons(total_samp1,nil)
17 }
18 bsp_sync;
19 (* total exchange of the sampling *)
20 while (!i<bsp_nprocs) do
21 {
22   invariant i ≥ 0
23   and envCputls(bsp_pid,k,0,i,put(k,(cast_int_farray samp1),
24     total_samp1,(bsp_nprocs+1)*k,(bsp_nprocs+1)))
25   variant bsp_nprocs-i
26 }
27 put !i (cast_int_farray samp1) total_samp1 ((bsp_nprocs+1)*i) (bsp_nprocs+1);
28 i←i+1
29 done;
30 {envCputls(bsp_pid,k,0,bsp_nprocs,put(k,(cast_int_farray(samp1)),
31   total_samp1,(bsp_nprocs+1)*k,(bsp_nprocs+1)))}
32 bsp_sync;
33 (* local sort of the sampling *)
34 local_sort__(uncast_int_farray !total_samp1);
35 (* second sampling *)
36 let samp2 = sampling(uncast_int_farray !total_samp1) in
37 Second_step i←0;
38 (* same samp2 on each processor *)
39 assert {∀ pid1,pid2:int. isproc(pid1)→ isproc(pid2)
40   → eq_array(samp2<pid1>,samp2<pid2>)}
41 while (!i<bsp_nprocs) do
42 {
43   invariant i ≥ 0
44   and envCsendls(bsp_pid,k,0,i,
45     send(cast_int_farray (lselection(samp2,a,k)))
46   variant bsp_nprocs-i
47 }
48 send (cast_int_farray (selection samp2 !a !i)) !i;
49 i←i+1
50 done;
51 (* sending data within the samplings *)
52 { envCsendls(bsp_pid,k,0,bsp_nprocs,send(cast_int_farray (lselection(samp2,a,k)))
53   and sorted_array(total_samp1, 0, array_length(total_samp1)-1)
54   and (∀ pid1,pid2:int. isproc(pid1)→ isproc(pid2)
55     → eq_array(samp2<pid1>,samp2<pid2>))
56 }
57 bsp_sync;
58 assert {∀ pid:int.
59   (0 ≤ pid<bsp_nprocs)→ (bsp_findmsg pid 1)=(lselection(samp2,a,pid))}
60 Third_step a← (make_array_int 0 0);
61 i←0;
62 (* merge the data *)
63 while (!i<bsp_nprocs) do
64 {
65   invariant i ≥ 0
66   and a=sigma_merge(envR,0,i)
67   variant bsp_nprocs-i
68 }
69 a←merge_arrays !a (uncast_int_farray (bsp_findmsg !i 1));
70 i←i+1
71 done
72 {
73   (* array locally sorted and between processors and same global length
74     and same elements as in the beginning
75   *)
76   sorted_array(a, 0, array_length(a)-1)
77   and (∀ pid:int. (0 ≤ pid<bsp_nprocs-1) → greatest(a<pid>) ≤ smallest(a<pid+1>))
78   and global_permutation(<a>,<a@>)
79   and (sigma_size <a> = sigma_size <a@>)
80 }

```

Figure 4.3. Parallel sorting algorithm.

where the time of merging elements (in a sorting way) is of order of n/p . The code is given in Fig 4.3. We currently make the hypothesis that arrays merged in this way give the expected result. A proof of this fact needs to be done in COQ in the future.

The precondition (line 3) ensures that each processor has enough data. The BSP-WHY program is then made of four super-steps, since in addition to the three super-steps of the algorithm, a first super-step is necessary to register the variables for DRMA access. The code and its logical assertions are as follow:

- The first super-step of the algorithm comprises the lines 2 to 17. On each processor, the local data is first sorted (line 7), then a sampling is created. The super-step ends with the registration of a variable for the DRMA accesses.
- In the second super-step (lines 19 to 31), the samplings are exchanged between the processors, using the **put** routine. The invariant of the loop is similar to the one in the first loop of our prefix example: the loop is simply adding one message in the **put** message queue. The operation done during the loop is actually a total exchange of the sampling: each processor puts it on every other processor. As such, we could have instead used a global parameter with a synchronising effect.
- The third super-step of the program (lines 33 to 56) starts by a local sort on all the received sample values. A new sample is then extracted from the sample values. Because the sample array is the same on every processor and the sampling function is deterministic, the new sample will also be the same on every processor; this is stated in the assertion line 39. In the following **while** loop, a processor will send data to every other processor, according to the partition described by the second sampling. The selection of the values of **a** according to the sampling is a purely sequential process, and is thus abstracted by a parameter called **selection**. The communication of these data is done with a **send** operation, line 45. Once again, the invariant of the loop is straightforward, since each iteration only adds one send message in the communication environment.
- Finally, in the last super-step (lines 58 to 71), we start by resetting the array **a**. We then insert all the data received, starting from the processor 0. This is done by a call to the sequential parameter **merge_arrays** that merges two sorted arrays. The invariant for this loop is that after *i* iterations, **a** contains the merged data from the received messages from processor 0 to the processor *i*.

The post-condition of the function (lines 76 to 79) states that the array **a** is locally sorted, with the values on a processor *i* lower than the values on the next processor, and that it is globally the same array (same values) as the original array.

4.1.3 Mechanical Proof

In this table, we can show how many verification conditions are generated for the above examples. We also show this number when no assertions are given for the correctness of the programs (it is just to have safe execution of the programs without buffer overflow or out-of-bound read of messages *etc.*). We also show (noted AP) the number of obligations that are automatically discharged by automatic procedures. We used the following automatic provers: ALT-ERGO (0.9), SIMPLIFY (1.5.4), Z3 (2.6), YICES (1.0.27), GAPP (0.13.0) and CVC3 (2.2).

Prog	Correctness/AP	Safety/AP
Direct Prefix	37/37	19/19
Log Prefix	41/37	21/19
PSRS	51/45	27/27

For a simple example such as the direct prefix, all the proof obligations are automatically discharged (proved) by automatic provers. For more complex examples, a few proof obligations are not automatically discharged yet. But safety (no deadlock, no buffer overflow, no out-of-the-bound sending messages *etc.*) is automatically ensured for all examples (except the log prefix) which is an interesting first result.

Not having all the properties given automatically is sad since the generated proof obligations are generally hard to read for WHY and even more for BSP-WHY: this is due to the use of loops over **p** for local computations. That also generated harder proof obligations for the provers. Furthermore, automatic provers are also a work in progress. For example, logarithm's (and power-of-two) properties are not currently well interpreted by any automatic prover and thus they fail to prove check bounds accesses in a logarithmic loop.

The key to evaluating the promise of a translation-based technique is in studying the effort needed to prove the generated proof obligations. Currently, many of them are automatically proved and it is thus an encouraging result regarding that it also happens for sequential computations and that our work is to our knowledge the first of its kind. Since these examples are not too difficult, we also believe that by giving more axioms (*e.g.* for log, sqrt, sort, *etc.* which are currently given to the minimum in the WHY library) all the proof obligations can be automatically proved. Furthermore, the complete proof of the proof obligations generated for these examples is still a work in progress.

4.2 Parallel State-space Construction

As any software, model-checkers are subject to bugs. They can thus report false negatives or validate a model that they should not. Different methods, such as theorem provers or **Proof-Carrying Code** [218] (PCC for short) have been used to gain more confidence in the results of model-checkers. In this section, we focus on using the verification condition generator WHY to study two generic² algorithms (one sequential and one distributed) of state-space construction and one distributed and dedicated to the state-space of security protocols. This work is a first step towards mechanically-assisted deductive verification of model-checkers.

4.2.1 Motivations and Background

Model-checkers (MCs for short) are often used to verify safety-critical systems. The correctness of their answers is thus vital: many MCs produce the answer “yes” or generate a counterexample computation (if a property of the model fails), which forces, in the two cases, to assume that the algorithm and its implementation are both correct.

But MCs, like any software are subject to bugs and there exist surprisingly few attempts to prove them correct. Three main reasons can explain this fact [216]: (1) MCs involve complicated logics, algorithms and sophisticated state reduction techniques; (2) because efficiency is essential, MCs are often highly optimised, which implies that they may not be designed to be proved correct; (3) MCs are often updated. But there is a more and more pressing need from the industrial community, as well as from national authorities, to get not just a boolean answer, but also a formal proof — which could be checked by an established tool such as the theorem prover Coq. This is required in Common Criteria certification of computer products at the highest assurance level EAL 7 — <http://www.commoncriteriaportal.org/>. And hand proofs are not sufficient for EAL 7, mechanical proofs are needed. The author of [247] summarizes the

²In the sense of independent of the considered model.

problem: *Quis custodiet ipsos custodes ?* (Who will watch the watchmen? that is, who will verify the verifier?). We want to be able to trust the results of model-checkers with a high degree of confidence.

(a) Different Solutions for Verifying Model-checkers

For verifying model-checkers, different solutions have been proposed. The first one is to prove MCs inside theorem provers and use the extraction facilities to get pure functional machine-checked programs such as in the works of [261] and [96]. The second and more common approach, in the spirit of PCC, is to generate a “certificate” during the execution of the MC that can be checked later or on-the-fly by a dedicated tool or a theorem prover. This is the so-called “certifying model-checking” [216]. In this way, users can re-execute the certificate/trace and have some safety guarantees because even if the MC is buggy, its results can be checked by a trustworthy tool.

But, any explicit MC may enumerate a very large state-space (the famous state-space explosion problem), and mimicking this enumeration with proof rules inside any theorem prover (or with PCCs) would be foolish, even if specific techniques and optimisations of the abstract machine of theorem provers [5] are used. Note that this problem does not arise when finding a refutation of the logical formula (the trace is generally short) but when the answer is “yes” since the entire explicit state-space (or at least a symbolic representation) needs to verify the checked properties. In this way, certificate generation could also hamstring both the functionality and the efficiency of the automation that can be built from theorem provers (functional programs can be too memory consuming) and PCC tools (too big certificates) [247]. Because model-checking can cause *memory crashing* on single or multiple processor systems, it has led to consider exploiting the larger memory space available in *distributed* systems [116], which also gives the opportunity to reduce the overall execution time. Paralleling the state-space construction on several machines is thus done in order to benefit from each machine’s complete storage and computing resources. Only efficient, imperative and distributed programs can override the state-space explosion problem.

And MCs, especially distributed ones [116], like any complex software are subject to *bugs*. And generating distributed certificates to be later machine-checked using a theorem prover is thus currently not reasonable since provers are critical softwares that can not be altered without much attention. For this purpose, we proposed to prove the *correctness* of distributed MCs themselves and not their results as “certifying MC” [216] generally does.

Another solution, proposed in [268] for a MC call PAT, is to use coding assumptions directly in the source code. They indeed use Spec# and a check of the object invariants (the contracts) is generated. Nevertheless, they cannot completely verify the correctness of PAT and they thus focus on some safety properties (as no overflows, no deadlocks) of the underlying data structures of PAT (which can run on a multi-core architecture) and check if some options may conflict with each other.

(b) The Proposed Solution

Our contribution follows the approach of [268] but by using WHY and by extending the verification to the correctness of the final result: has the full state-space been well computed without adding unknown states?

Since the WHY-ML language is not immediately executable but a higher-level algorithmic language, we only focus on algorithms. We can thus focus on which formal properties need to be preserved and not be obstructed by problems specific to a particular programming language. Even if most of the bugs in MCs will not be due to wrong algorithms but rather due to subtle errors in the implementation of some complex data structures and bad interactions between these structures and compression aspects, we must first check the algorithms to get an idea of the amount of work necessary to verify a true model-checker.

Our goal is then a mechanically-assisted proof that these annotated algorithms terminate and indeed compute the expected finite state-space. This is an interesting first step before verifying MCs themselves: it allows to test if this approach is doable or not. This is also challenging due to the nature of model-checking (critical system) and to the algorithmic complexity. The main contribution of this section is to demonstrate the ability of a VCG such as WHY to tackle the wide range of verification issues involved in the proof of correctness of imperative codes of MCs.

(c) Application to Security Protocols

Security protocols are small and standard components of systems that communicate over untrusted networks. Their relatively small size, combined with their critical role, makes them a suitable target for formal analysis [65]. But verifying secure protocols is a challenging problem. In spite of their apparent simplicity, they are notoriously error-prone. *Attacks* exploit *weaknesses* in the protocol that are due to the

complex and unexpected *interleaving* of different protocol sessions generated by an *intruder* (malicious) which *resides in the network*. The intruder is assumed to have *complete network control* and to be powerful enough to perform potentially *dangerous actions* such as intercepting messages flowing over the network, or replacing them by new ones using the knowledge he has previously gained [89]. Unfortunately, the question of whether a protocol achieves its security requirements or not is, in the general case, undecidable [93] or NP-complet in case of bounded number of agents [241].

Model-checking (MC) is common solution to find flaws [6]. By focusing on verification of a *bounded number of sessions*, MC of a protocol can be done by simply *enumerating* and *exploring all traces* of the execution of the protocol and looking for a violation of some of the requirements. Verification through *model-checking* thus consists in defining a formal model of the system to be analysed and then using automated tools to check whether the expected properties are met or not on the state-space of the model that is having compute all the different configurations of the execution of the agents evolving in the protocol. Ideally, we would also like to have a proof of the protocol's correctness or of the attack found: generate a "certificate" [216] that can be checked later by a trusted theorem prover (*e.g.* COQ).

But the greatest problem with explicit model-checking in general (and for security protocols in particular) is the so-called state explosion: the fact that the number of states typically grows dramatically with the number of agents. This is especially true when complex data-structures are used in the model such as the knowledge of an intruder in a security protocol. The same problem applies for the generation of large discrete state spaces of some non traditional protocols [244] (especially when complex data-structures are used by the agents such as lists of trusted servers *etc.*)

As said above, we will use deductive verification of state-space algorithms. For security protocols, we will verify a specialised distributed algorithm for security protocols designed in [121].

(d) Definition of Security Protocols

Security protocols specify an exchange of *cryptographic messages* between *principals*, *i.e.* the agents (*e.g.*, users or servers) participating in the protocol. Messages are sent over open *networks*, such as the Internet, that are not secured. As a consequence, protocols should be designed to work fine even if messages may be eavesdropped or tampered with by an *intruder* — *e.g.*, a dishonest or careless agent. Each protocol is aimed to provide security guarantees, such as authentication or secrecy of some pieces of information.

We assume the use of keys sufficiently long of the best-known cryptographic algorithms to prevent a brute force attack in a reasonable time. This is the well-known *perfect cryptography* assumption. The idea is that an encrypted message can be decrypted only by using the appropriate decryption key, *i.e.*, it is possible to retrieve M from the message encrypted $\{M\}_K$ only by using K^{-1} as decryption key and it is hopeless to compute K^{-1} from K .

The abilities of the attackers and relationship between participants and attackers together constitute a threat model and the almost exclusively used threat model is the one proposed by Dolev and Yao [89]. The Dolev/Yao threat model is a worst-case model in the sense that the network, over which the participants communicate (agents perform "ping-pong" data exchanges), is thought as being totally *controlled* by an *omnipotent* intruder. In this thesis, we thus consider a Dolev/Yao attacker that resides on the network. An execution of such a model is thus a series of message exchanges as follows. (1) An agent sends a message on the network. (2) This message is captured by the attacker that tries to learn from it by recursively decomposing the message or decrypting it when the key to do so is known. Then, the attacker forges all possible messages from newly as well as previously learnt information (*i.e.*, attacker's knowledge). Finally, these messages (including the original one) are made available on the network. (3) The agents waiting for a message reception accept some of the messages forged by the attacker, according to the protocol rules.

4.2.2 Definitions and Verification of a Sequential State-space Algorithm

We now describe how we model the state-space, and present the verification of a well-known algorithm. The annotated source codes are available at <http://laci.fr/gava/cert-mc.tar.gz>.

(a) Definition of the Finite State-space

Let us recall that the finite state-space construction problem is computing the explicit graph representation (also known as *Kripke structure*) of a given model from the implicit one. This graph is constructed by exploring all the states reachable through a successor function `succ` (which returns a set of states)

<pre> 1 let seq_algo () = 2 let known = ref {} in 3 let todo = ref {s0} in 4 while todo ≠ {} do 5 let s = pick todo in 6 known ← !known ⊕ s; 7 todo ← !todo ∪ (succ(s) \ !known) 8 done; 9 !known </pre>	<pre> 1 let seq_algo () = 2 let known = ref {} in 3 let todo = ref {s0} in 4 while todo ≠ {} do 5 { 6 invariant (1) (known ∪ todo) ⊆ StSpace 7 and (2) (known ∩ todo) = {} 8 and (3) s0 ∈ (known ∪ todo) 9 and (4) (∀ e:state. e ∈ known → succ(e) ⊆ (known ∪ todo)) 10 variant StSpace \ known 11 } 12 let s = pick todo in 13 known ← !known ⊕ s; 14 todo ← !todo ∪ (succ(s) \ !known) 15 done; 16 !known 17 {result=StSpace} (* result is the value of known*) </pre>
--	--

Figure 4.4. Sequential WHY-ML algorithm.

from an initial state s_0 . Generally, during this operation, all the explored states must be kept in memory in order to avoid multiple explorations of a same state.

In this thesis, all algorithms only compute the state-space, noted StSpace . This is done without loss of generality and it is a trivial extension to compute the full Kripke structure — usually preferred for checking temporal logic formulas. To represent StSpace in the logic of WHY, we used the following axiom `contain_state_space` (for consistency, it has been proved in COQ using an inductive definition of the state-space, also available in the source code):

```

logic s0: state
logic succ: state → state set
logic StSpace: state set
axiom contain_state_space: ∀ss:state set. StSpace ⊆ ss ↔ (s0 ∈ ss and (∀ s:state. s ∈ ss → s ∈ StSpace → succ(s) ⊆ ss))

```

i.e. defines which sets can contain the state-space. Now ss is the state-space ($\text{ss}=\text{StSpace}$) if and only if, the two following properties holds: (A) $\text{ss} \subseteq \text{StSpace}$ and (B) $\text{StSpace} \subseteq \text{ss}$; that is equality of sets using extensionality. Note that using this first-order definition makes the automatic (mainly SMT) solvers prove more proof obligations than using an inductive definition for the state-space.

(b) Sequential Algorithms for State-space Construction

Fig. 4.4 gives the common *sequential* (random walk) algorithm in WHY-ML using an appropriate syntax for set operations. In left, without adding proof annotations for a best reading. All computations in this algorithm are set operations where a set called `known` contains all the states that have been processed and would finally contain StSpace . It involves a set of states `todo` that is used to hold all the states whose successors have not been constructed yet; each state s from `todo` is processed in turn (lines 4–5) and added to `known` (line 6) while its successors are added to `todo` unless they are known already — line 7. Note this algorithm can be made strictly depth-first by choosing the most-recently discovered state (*i.e.* `todo` as a stack), and breadth-first by choosing the least-recently discovered state. This has not been studied here.

For correctness, all our algorithms need three properties: (1) they do not fail (no rule of reduction) \equiv safety; (2) they indeed compute the state-space; (3) and they terminate. The first property is immediate for the sequential algorithm since the only operation that could fail is `pick` (where the precondition is “not take any element from an empty set”) and this is assured by the guard of the **while** loop. The four invariants are: (1) `known` and `todo` are subsets of StSpace ; at the end, (3) and (4) `known` is a subset of StSpace and has the “same” inductive property; and when `todo` will be empty, then `known` contains StSpace — property (B).

The termination is ensured by the following variant: $|\text{StSpace} \setminus \text{known}|$ since, at each iteration of the loop, the algorithm only adds a new state s because $(\text{known} \cap \text{todo}) = \emptyset$.

All the obligations produced by the VCG of WHY are automatically discharged by a combination of automatic provers. For each prover, we give a timeout of 10 seconds — otherwise some obligations are not proved. In the following table, we give the number of generated obligations (column Total) and then how many are discharged by the provers:

algo/Solvers	Total	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
Normal	11	2	10	11	7	3	3

One could notice that the SMT solvers SIMPLIFY and Z3 give the best results. In practice, we mostly used them. SIMPLIFY is the faster and Z3 sometime verified some obligations that had not be discharged by

<pre> 1 let naive_state_space () = 2 let total = ref 1 in 3 let known = ref [] in 4 let todo = ref [] in 5 let pastsend = ref [] in 6 if cpu(s0) = bsp_pid then 7 todo ← !todo ⊕ s0; 8 while total > 0 do 9 let tosend = (local_successors 10 known todo pastsend) in 11 exchange todo total !known 12 pastsend !tosend 13 done; 14 !known 15 \end{whyprog} 16 \begin{whyprog} 17 let exchange (...) = 18 rcv, total ← BSP_EXCHANGE(tosend) 19 todo ← rcv \ known 20 pastsend ← !pastsend ∪ (tosend) </pre>	<pre> 1 let local_successors (...) = 2 let tosend = ref (init_send []) in 3 while todo ≠ [] do 4 let s = pick todo in 5 known ← !known ⊕ s; 6 let new_states = ref ((succ s) \ !known \ !pastsend) in 7 while new_states ≠ [] do 8 let new_s = pick new_states in 9 let tgt = cpu(new_s) in 10 if tgt = bsp_pid 11 then todo ← !todo ⊕ new_s 12 else tosend[tgt] ← tosend[tgt] ⊕ new_s 13 done 14 done; 15 !tosend </pre>
---	---

Figure 4.5. Parallel (distributed) BSP-WHY-ML algorithm for state-space construction.

SIMPLIFY. We have worked with the provers as black-boxes, without detailed knowledge of their inner working, and we have thus no explanation for this fact. Proof obligations are as expected when working with a VCG such as WHY.

4.2.3 Verification of a Generic Distributed State-space Algorithm

Parallelize the construction of the state-space on several machines is a standard method [116]. In this section, we give an example of how to verify a generic (in the sense of independent of the function of successor *succ*) distributed (BSP) algorithm and show that it is more challenging but feasible.

(a) A Generic BSP Algorithm for State-space Construction

The above algorithm can be parallelised using a partition function *cpu* (a hashing) that returns for each state a processor id, *i.e.*, the processor numbered *cpu(s)* is the owner of *s*: **logic** *cpu*: state → int and **axiom** *cpu_range*: ∀s:state. 0 ≤ *cpu(s)* < *nprocs*

The idea is that each process computes the successors for only the states it owns. This is rendered as the BSP algorithm of Fig. 4.5. This is thus a SPMD algorithm so that each processor executes it. Initially, only state *s*₀ is known and only its owner puts it in its *todo* set. This is performed in lines 6–7, where *bsp_pid* evaluates locally to each processor to its own identifier.

Sets *known* and *todo* are still used but become local to each processor and thus provide only a partial view on the ongoing computation.

Function *local_successors* computes the successors of the states in *todo* where each computed state that is not owned by the local processor is recorded in the array (of size *nprocs*) of sets of states *tosend* depending of its owner number. The set *pastsend* contains all the states that have been sent during the past super-steps — the past exchanges. This prevents returning a state already sent by the processor: this feature is not necessary for correctness and consumes more memory but it is generally more efficient mostly when the state-space contains many cycles.

Then, function *exchange* is responsible for performing the actual communications between processors. It assigns to *todo* the set of received states that are not yet known locally together with the new value of *total*. The primitive *BSP_EXCHANGE* performs a global (collective) synchronisation *barrier* which makes data available for the next super-step so that all the processors are now synchronised. The synchronous routine *BSP_EXCHANGE* sends each state *s* from the set *tosend*[*i*] to the processor *i* and returns the set of states received from the other processors, together with the total number of exchanged states — it is mainly the MPI's *alltoall* primitive. Notice that, by *postponing* communication, this function allows buffered sending and forbids sending several times the same state. More formally, at processor *myid*:

$$\text{BSP_EXCHANGE}(\text{tosend}) = \begin{cases} \text{total} = \sum_{k=0}^{nprocs-1} \sum_{i=0}^{nprocs-1} |\text{tosend}[k][i]| \\ \text{rcv} = \bigcup_{i=0}^{nprocs-1} \text{tosend}[i][\text{myid}] \end{cases}$$

where *tosend* [*i*] represents the array *tosend* at processor *i*.

To ensure termination of the algorithm, we use the additional variable *total* in which we count the total number of sent states. We have thus not used any complicated methods as the ones presented in [16]. It

can be noted that the value of **total** may be greater than the intended count of states in **todo** sets. Indeed, it may happen that two processors compute a same state owned by a third processor, in which case two states are exchanged but then only one is kept upon reception. In the worst case, the termination requires one more super-step during which all the processors will process an empty **todo**, resulting in an empty exchange and thus **total**=0 on every processor, yielding the termination.

(b) Verification of this BSP Algorithm

For care of brevity, we do not present **exchange** which is technical and without really interesting properties and still available in the source code: the exchange procedure is only a permutation of the states that is, from a global point of view, only states in **arrays** have moved and there is no loss of states and a state has not magically appeared during the communications. Fig. 4.6 gives the annotated main parallel loop of the algorithm. We also use the following predicates:

- $\text{isproc}(i)$ is defined what is a valid processor id that is $0 \leq i < \text{nprocs}$;
- $\bigcup(\langle p_set \rangle)$ is the union of the sets of the **p**-value **p_set** that is $\bigcup_{i=0}^P p_set \langle i \rangle$;
- $\text{GoodPart}(\langle p_set \rangle)$ is used to indicate that each processor only contains the states it owns that is $\forall i:\text{int. isproc}(i) \rightarrow \forall s:\text{state. } s \in p_set \langle i \rangle \rightarrow \text{cpu}(s)=i$;

As before, we need to prove that (1) the code does not fail; (2) indeed computes the entire state-space and (3) terminates. The first property follows immediately since only the routine **pick** is used as before; and to also prove that the code is deadlock free (the loop contains **exchange** which implies a global synchronisation of all the processors), we can maintain that **total** (which gives the condition for termination) has the same value on all the processors during the entire execution of the algorithm. Let us now focus on the two other properties.

Correctness of the Main Parallel Loop (Fig. 4.6). The invariants (lines 9 – 18) of the main parallel loop work as follows: (1) as in sequential algorithm, we need to maintain that **known** (even distributed) is a subset of **StSpace** which finally ensures (A) when **todo** is empty; (2) as usual, the states to be treated are not already known; (3) our sets are well distributed (there is no duplicate state that is, each state is only kept in a unique processor); (4) **total** is a global variable, we thus ensure that it has the same value on each processor; (5) ensures that no state remains in **todo** (to be processed) when leaving the loop since **total** is at least as big as the cardinality of **todo**, **total** is an over-approximation of the number of sent states; (6–8), as usual, ensure property (B); (9) past sending states are in the state-space; (10) **pastsend** only contains states that are not owned by the processor and (11) all these states, that were sent, are finally received and stored by a processor.

In the post-condition (line 26), we can also ensures that the result is well distributed: the state-space is complete and each processor only contains the states it owns according to the partition function **cpu**.

Termination of the Main Parallel Loop (Fig. 4.6). The main loop is more subtle: **total** is an over-approximation and thus could be greater to 0 whereas **todo** is empty. This happens when all the received states are already in **known**. The termination has thus two cases: (a) in general the set **known** globally (that is, from a global point of view, of all processors) grows and we have thus the cardinality of **StSpace** minus **known** which is strictly decreasing; (b) if there is no state in any **todo** of a processor (case of the last super-step), no new states would be computed and thus **total** would be equal to 0 in the last stage of the main loop.

We thus used a lexicographic order (this is well-founded ensuring termination) on the two values: sum of **known** across all processors; and **total** (which is the same on all processors) when no new states are computed and thus when no state would be sent during the next super-step. At least, one processor cannot received any state during a super-step. We thus need an invariant in the **local_successors** for maintaining the fact that the set **known** potentially grows with at least the states of **todo**. We also maintain that if **todo** is empty then no state would be sent (in **local_successors**) and received, making **total** equal to 0 — in **exchange**. This is due to the fact that eventually, in the last super-step, no new state are computed or added to **known** by any processor.

Local Computations. The termination is ensured as in the sequential algorithm since **known** can only grow when entering the loop. Fig 4.7 gives the annotated sequential part of the generic BSP algorithm.

```

1 let naive_state_space () =
2   let known = ref ∅ in let todo = ref ∅ in
3   let pastsend = ref ∅ in let total = ref 1 in
4   if cpu(s0) = bsp_pid then
5     todo ← s0 ⊕ !todo;
6     while total > 0 do
7       {
8         invariant
9         (1)  $\bigcup(<\text{known}>) \cup \bigcup(<\text{todo}>) \subseteq \text{StSpace}$ 
10        and (2)  $(\bigcup(<\text{known}>) \cap \bigcup(<\text{todo}>)) = \emptyset$ 
11        and (3)  $\text{GoodPart}(<\text{known}>) \text{ and } \text{GoodPart}(<\text{todo}>)$ 
12        and (4)  $(\forall i,j:\text{int}. \text{isproc}(i) \rightarrow \text{isproc}(j) \rightarrow \text{total}<i> = \text{total}<j>)$ 
13        and (5)  $\text{total}<0> \geq |\bigcup(<\text{todo}>)|$ 
14        and (6)  $s0 \in (\bigcup(<\text{known}>) \cup \bigcup(<\text{todo}>))$ 
15        and (7)  $(\forall e:\text{state}. e \in \bigcup(<\text{known}>) \rightarrow \text{succ}(e) \subseteq (\bigcup(<\text{known}>) \cup \bigcup(<\text{todo}>)))$ 
16        and (8)  $(\forall e:\text{state}. \forall i:\text{int}. \text{isproc}(i) \rightarrow e \in \text{known}<i> \rightarrow \text{succ}(e) \subseteq (\text{known}<i> \cup \text{pastsend}<i>))$ 
17        and (9)  $\bigcup(<\text{pastsend}>) \subseteq \text{StSpace}$ 
18        and (10)  $(\forall i:\text{int}. \text{isproc}(i) \rightarrow \forall e:\text{state}. e \in \text{pastsend}<i> \rightarrow \text{cpu}(e) \neq i)$ 
19        and (11)  $\bigcup(<\text{pastsend}>) \subseteq (\bigcup(<\text{known}>) \cup \bigcup(<\text{todo}>))$ 
20        variant pair( $\text{total}<0>, |S \setminus \bigcup(\text{known})|$ ) for lexico_order
21      }
22      let tosend = (local_successors known todo pastsend) in
23      exchange todo total !known !tosend
24    done;
25    !known
26    {StSpace =  $\bigcup(<\text{result}>)$  and  $\text{GoodPart}(<\text{result}>)$ }

```

Figure 4.6. Annotated BSP generic algorithm for state-space construction.

Note that the inner loop needs to define a ghost³ variable `ghost_diff` which represents the set of states of the initial `new_states` that have been processed since `new_state` decreases and we lack this information for proving that all states have been really well processed.

The annotations work as follow. First, in the pre-condition to `local_successors`, we have that all manipulated sets of states are subset of the `StSpace` (property to prove (A)); `known` and `todo` are disjoint sets; all states are owned by the right processor and finally that any state in `known` has its successor in `known` or have been previously sent in a past super-step.

Second, the nine invariants of the outer loop are the following: (1–3) as in the pre-condition; (4) any new state to be send has not been sent in a past super-step; (5) the sets are in `tosend` (to be send) as `cpu` needs it; (6) the processor does not send states to itself; (7) any state in `known` has its successors to be processed, to be send or still known (this is for property (B) since `todo` would be finally empty as well sets of states to be send; `known` has the “same” inductive property than `StSpace`); (8) `known` grows with at least the states from `todo`; (9) if there is no state to process then there will be no state to send nor to process (it is useful for the termination property of the main parallel loop).

Third, the inner loop maintains the same kind of invariants but with also the set `new_states`, the states that have been computed from one state of `todo`. We have also the fact (invariants (e) and (f)) that all states of `new_states` are processed in the loop (we do not miss a state).

Fourth, the post-conditions are a mix of the pre-conditions and of the above invariants. Mainly, we have that all manipulated sets of states are subset of the `StSpace`; states still been well partitioned; if there is no state to process then there will be no state to send.

Mechanical Proof. With some obvious axioms on the above predicates (such as $\bigcup \langle \emptyset, \dots, \emptyset \rangle = \emptyset$) so that solvers can handle the predicates, all the produced obligations are automatically discharged by a combination of the solvers. In the following table, for each part of the parallel algorithm, we give the number of obligations and how many are discharged by the provers (some proof obligations require long timeouts *e.g.* 10 mins):

part/Solvers	Total	ALT-ERGO	SIMPLIFY	Z3	CVC3	YICES	VAMPIRE
main	106	74	98	101	0	54	78
successor	46	16	42	41	24	14	32
exchange	24	20	22	23	0	16	15

Now the combination of all provers is needed since none of them is able to prove all the obligations. This is certainly due to their different heuristics. We also note that `SIMPLIFY` and `Z3` remain the most efficient.

³Additional codes not participating in the computation but accessing the program data and allowing the verification of the original code.

```

1 let local_successors (known: state set ref) (todo:state set ref) (pastsend:state set ref) =
2 {
3   (known ∪ todo ∪ pastsend) ⊆ StSpace
4   and (known ∩ todo) = ∅
5   and (∀ s:state. s ∈ (known ∪ todo) → cpu(s)=bsp_pid)
6   and (∀ e:state. e ∈ known → succ(e) ⊆ (known ∪ pastsend))
7 }
8 init:let tosend = ref (init_send ∅) in
9 while todo ≠ ∅ do
10 {
11   invariant (1) (known ∪ todo ∪ ⋃(tosend)) ⊆ StSpace
12   and (2) (known ∩ todo)=∅
13   and (3) (∀ s:state. s ∈ (known ∪ todo) → cpu(s)=bsp_pid)
14   and (4) (⋃(tosend) ∩ pastsend)=∅
15   and (5) (∀ i:int. isproc(i) → ∀e:state. e ∈ tosend[i] → cpu(e)=i)
16   and (6) tosend[bsp_pid]=∅
17   and (7) (∀ e:state. e ∈ known → succ(e) ⊆ (known ∪ todo ∪ ⋃(tosend) ∪ pastsend))
18   and (8) (todo@init ∪ known@init) ⊆ (todo ∪ known)
19   and (9) (todo@init=∅) → (todo=∅ and ⋃(tosend)=∅)
20   variant |StSpace \ known|
21 }
22 let s = pick todo in
23 known ← !known ⊕ s;
24 let new_states = ref ((succ s) \ !known \ !pastsend) in
25 let ghost_diff=ref ∅ in (* does not participate in the computation; only use for proof ! *)
26 L:while new_states ≠ ∅ do
27 {
28   invariant (1) and (2) and (3) and (4) and (5) and (6) and (8)
29   and (a) (known ∩ new_states)=∅
30   and (b) (new_states ∩ pastsend)=∅
31   and (c) new_states ⊆ StSpace
32   and (d) (∀ e:state. e ∈ known → succ(e) ⊆ (known ∪ todo ∪ ⋃(tosend) ∪ pastsend ∪ new_states))
33   and (e) new_states@L=(ghost_diff ∪ new_states)
34   and (f) (ghost_diff ∩ new_states)=∅
35   variant |new_states|
36 }
37 let new_s = pick new_states in
38 let tgt=cpu(new_s) in
39 ghost_diff ← (⊕ new_s !ghost_diff); (* only use for proof ! *)
40 if tgt=bsp_pid
41 then todo ← !todo ⊕ new_s
42 else tosend[tgt] ← tosend[tgt] ⊕ new_s
43 done
44 done;
45 !tosend (* result is the value of the array tosend *)
46 {
47   (known ∪ ⋃(result) ∪ pastsend) ⊆ StSpace
48   and todo=∅
49   and (∀ s:state. s ∈ known → cpu(s)=bsp_pid)
50   and (known@ ∪ todo@) ⊆ known
51   and (⋃(result) ∩ pastsend)=∅
52   and (∀ i:int. isproc(i) → ∀e:state. e ∈ result[i] → cpu(e)=i)
53   and result[my_pid]=∅
54   and (∀ e:state. e ∈ known → succ(e) ⊆ (known ∪ ⋃(result) ∪ pastsend))
55   and (⋃(result) ∩ known)=∅
56   and (todo@=∅) → ⋃(result)=∅
57 }

```

Figure 4.7. Local computations of the naive parallel algorithm.

Some obligations are hard to follow due to the parallel computations. But reading them carefully, we can find the good annotations.

4.2.4 Dedicated Algorithms for Security protocols

(a) Specific Properties of Security Protocols [121]

As said before, we model security protocols as an explicit set of states (or Kripke structure) such that any state can be represented by a function from a set of *locations* to an arbitrary data domain. For instance, locations may correspond to local variables of agents, buffers, *etc.* Our approach is largely independent of the chosen formalism and it is enough to assume that the four following properties hold:

- (P1) The function successor succ can be *partitioned* into two successor $\text{succ}_{\mathcal{R}}$ and $\text{succ}_{\mathcal{L}}$ (\mathcal{R} and \mathcal{L} form a partition of the locations) that correspond respectively to execute the transitions upon which

an agent (except the intruder) receives information (and stores it), and to execute all the other transitions;

- (P2) There is an initial state s_0 and there exists a function slice from states to natural numbers (a *measure*) such that if $s' \in \text{succ}_{\mathcal{R}}(s)$ then there is no path from s' to any state s'' such that $\text{slice}(s) \leq \text{slice}(s'')$ and $\text{slice}(s') = \text{slice}(s) + 1$ (it is often call a *sweep-line* progression);
- (P3) There exists also a function $\text{cpu}_{\mathcal{R}}$ from states to natural numbers such that for all state s if $s' \in \text{succ}_{\mathcal{L}}(s)$ then $\text{cpu}_{\mathcal{R}}(s) = \text{cpu}_{\mathcal{R}}(s')$; mainly, the knowledge of the intruder is not taken into account to compute the hash of a state;
- (P4) If $s_1, s_2 \in \text{succ}_{\mathcal{R}}(s)$ and $\text{cpu}_{\mathcal{R}}(s_1) \neq \text{cpu}_{\mathcal{R}}(s_2)$ then there is no possible path from s_1 to s_2 and *vice versa*.

More precisely: for all state s and all $s' \in \text{succ}(s)$, if $s'|_{\mathcal{R}} = s|_{\mathcal{R}}$ then $s' \in \text{succ}_{\mathcal{L}}(s)$, else $s' \in \text{succ}_{\mathcal{R}}(s)$; where $s|_{\mathcal{R}}$ denotes the state s whose domain is restricted to the locations in \mathcal{R} . Intuitively, $\text{succ}_{\mathcal{R}}$ corresponds to transitions upon which an agent receives information and stores it, and \mathcal{R} are the locations where the agents (except the attacker) store the information they receive.

On concrete models, it is generally easy to distinguish *syntactically* the transitions that correspond to a message reception in the protocol with information storage. Thus, is it easy to partition succ as above and, for most protocol models, it is also easy to check that the above properties are satisfied. Note also that our approach is independent of using partial order reductions as in [108] where the main idea is that the knowledge of the intruder *always grows* and thus it is safe to *prioritise* the sending transitions with respect to receptions and local computations of agents. A simple modification of the successors functions is sufficient.

It also to notice that some security properties such as secret (confidentiality), authentication, integrity, anonymity can be expressed only using a state-space computation since these properties force to a *reachability analysis*, *i.e.*, finding a state that breaks one on the above properties. However, more complex property usually need to resort to temporal logics.

We can give (part) of these properties in the WHY's logic using the following:

```

logic slice: state  $\rightarrow$  int
(* Two successors functions *)
logic succ $_{\mathcal{L}}$ : state  $\rightarrow$  state set
logic succ $_{\mathcal{R}}$ : state  $\rightarrow$  state set

(* P1 *)
axiom def_cut_succ_1:  $\forall s:\text{state}. \text{succ}_{\mathcal{L}}(s) \cap \text{succ}_{\mathcal{R}}(s) = \emptyset$ 
axiom def_cut_succ_2:  $\forall s:\text{state}. \text{succ}(s) = \text{succ}_{\mathcal{L}}(s) \cup \text{succ}_{\mathcal{R}}(s)$ 

(* P2 *)
axiom slice_s0: slice(s0)=0
axiom succ_L_slice:  $\forall s:\text{state}. \forall s':\text{state}. s' \in \text{succ}_{\mathcal{L}}(s) \rightarrow \text{slice}(s')=\text{slice}(s)$ 
axiom succ_R_slice:  $\forall s:\text{state}. \forall s':\text{state}. s' \in \text{succ}_{\mathcal{R}}(s) \rightarrow \text{slice}(s')=\text{slice}(s)+1$ 
axiom succ_R_slice2:  $\forall s,s':\text{state}. \text{slice}(s')>\text{slice}(s) \rightarrow \neg(s \in \text{succ}(s'))$ 

```

(b) Algorithms for State-space Construction of Security Protocols [121]

Based on the above properties, we have designed, in an *incremental manner*, BSP algorithms for efficiently computing the state-space of security protocols. Successive *improvements* of the generic BSP algorithm will result in a parallel algorithm that remains quite *simple* in its expression (and efficient [122]) but that actually relies on a precise use of a consistent set of observations and algorithmic modifications. Only the functions `local_successors` and `exchange` have been really modified in the distributed algorithms.

Increasing Local Computation Time. Using the naive parallel algorithm, function `cpu` distributes evenly the states over the processors. However, each super-step is likely to compute few states because only too few computed successors are locally owned. This results in a bad balance of the time spent in computation with respect to the time spent in communication. If more states can be computed locally, this balance improves but also the total communication time decreases because more states are computed during each call to `local_successors`.

```

1 (* An exploration to improve local computations *)
2 let local_successors (...) =
3   (* tosend is a map for the balance algorithm *)
4   let tosend = ref (init_send ()) in
5   while todo ≠ ∅ do
6     let s = pick todo in
7     known ← !known ⊕ s;
8     todo ← !todo ∪ ((succL s) \ !known)
9     (* "\ !known" is now not needed for sweep-line algorithm *)
10    let send_states = ref ((succR s) \ !known) in
11    while send_states ≠ ∅ do
12      let new_s = pick send_states in
13      (* cpuB for the balance algorithm *)
14      let tgt = cpuR(new_s) in
15      tosend[tgt] ← tosend[tgt] ⊕ new_s
16    done
17  done;
18  !tosend

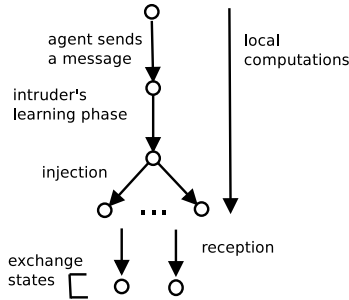
```

```

(* Sweep-line implementation and balance *)
let exchange (...) =
  dump known
  todo, total ← BSP_EXCHANGE(balance(tosend))

```

Figure 4.8. Dedicated BSP algorithms for state-space construction of security protocols.



To achieve this result, we consider a peculiarity of the models we are analysing (see the scheme on the left). The *learning phase* (2) of the attacker is computationally *expensive*, in particular when a message can be actually decomposed, which leads to recompose a lot of new messages. Among the many forged messages, only a (usually) small proportion are accepted for reception by agents. Each such reception gives rise to a new state. This whole process can be kept local to the processor and so without cross-transitions. To do so, we need to design our partition function $\text{cpu}_{\mathcal{R}}$ such that, for all states s_1 and s_2 , if $s_1|_{\mathcal{R}} = s_2|_{\mathcal{R}}$ then $\text{cpu}_{\mathcal{R}}(s_1) = \text{cpu}_{\mathcal{R}}(s_2)$. For instance, this can be obtained by computing

a hash (modulo the number of processors) using only the locations from \mathcal{R} , *i.e.*, the locations where the honest agents store received information.

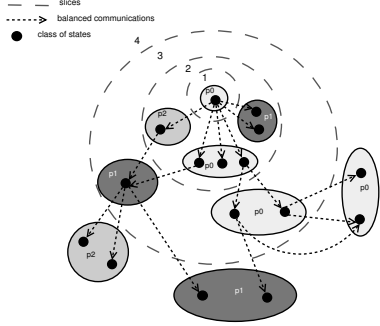
In this first improvement, when `local_successors` is called, then all new states from $\text{succ}_{\mathcal{L}}$ are added in `todo` (states to be proceeded) and states from $\text{succ}_{\mathcal{R}}$ are sent to be treated at the next super-step, enforcing an order of exploration of the state-space that matches the progression of the protocol in *slices*. Another difference is that no state could be sent twice due to this order. Fig. 4.8 gives the new function `local_successors`. The main loop of the parallel algorithm still unchanged. We call this new algorithm “Incr”. Another difference is the forgotten variable “pastsend” since no state could be sent twice due to this order. With respect to the “Naive” algorithm, this one splits the local computations, avoiding calls to $\text{cpu}_{\mathcal{R}}$ when they are not required. This may yield a performance improvement, both because $\text{cpu}_{\mathcal{R}}$ is likely to be faster than cpu and because we only call it when necessary. But the main benefits in the use of $\text{cpu}_{\mathcal{R}}$ instead of cpu is to generate less cross transitions since less states are need to be sent. Finally, notice that, on some states, $\text{cpu}_{\mathcal{R}}$ may return the number of the local processor, in which case the computation of the successors for such states will occur in the next super-step. We now show how this can be exploited.

Decreasing Local Storage. One can observe that the structure of the computations now *matching* the structure of the protocol execution: each super-step computes the executions of the protocol until a message is received. As a consequence, from the states exchanged at the end of a super-step, it is not possible to *reach* states computed in any previous super-step.

This kind of progression in a model execution is the basis of the *sweep-line* method [58] that aims at reducing the memory footprint of a state-space computation by exploring states in an order compatible with *progression*. It thus becomes possible to regularly dump from the main memory all the states that cannot be reached anymore — a disk-based backup can also be done if necessary for restoring the trace of a forbidden computation. Thus, statement `dump(known)` resets `known` to an empty set, possibly saving its content to disk if this is desirable. The rest of function `exchange` is simplified accordingly.

Enforcing such an exploration order is usually made by defining on states a measure of progression. In our case, such a measure is not needed because of the match between the protocol progression and the super-steps succession. So we can apply the sweep-line method by making a simple modification of the above algorithm: we empty `known` at the beginning of each super-step and $\setminus !\text{known}$ is now not needed after the successor function $\text{succ}_{\mathcal{R}}$ because all these states need to be sent for the next super-step/slice. Fig. 4.8 gives the new function `exchange`. We call this new algorithm “Sweep”.

Balancing the Computations. During our benchmark, we have found that using $\text{cpu}_{\mathcal{R}}$ can introduce a bad balance of the computations due to a lack of information when hashing only on \mathcal{R} . Thus, the final optimisation step aims at balancing the workload. To do so, we exploit the following observation: for all the protocols we have studied so far, the number of computed states during a super-step is usually closely related to the number of states received at the beginning of the super-step. So, before to exchange the states themselves, we can first exchange information about how many states each processor has to send and how they will be spread onto the other processors. Using this information, we can *anticipate* and compensate balancing problems.



To compute the balancing information, we use a new partition function cpu_B that is equivalent to $\text{cpu}_{\mathcal{R}}$ without modulo. This function defines classes of states for which cpu_B returns the same value. Those classes are like a “bag-of-tasks” [167] that can be distributed over the processors independently (see scheme on the left). To do so, we compute a *histogram* of these classes on each processor, which summarises how $\text{cpu}_{\mathcal{R}}$ would dispatch the states. This information is then globally exchanged, yielding a global histogram that is exploited to compute on each processor a better dispatching of the states it has to send. This is made by placing the classes according to a simple *heuristic for the bin packing problem*: the largest class is placed onto

the less charged processor, which is repeated until all the classes have been placed.

It is worth noting that this placement is computed with respect to the global histogram, but then, each processor dispatches only the states it actually holds, using this global placement. Moreover, if several processors compute a same state, these identical states will be in the same class and so every processor that holds such states will send them to the same target. So there is no possibility of duplicated computation because of dynamic states remapping. We call this algorithm “Balance”. Classes of states (consistent with partition function cpu_B) are grouped on processors so there is no possibility of duplicated computation. This “Balance” algorithm gives better performances than a naive distributed one for security protocols [122].

(c) Verification of these Dedicated Parallel Algorithms

For all these algorithms, termination and safety are proved as the naive one.

Algorithm “Incr”. The invariants are the same of Fig. 4.6 but with these changes. First, we need to forget all the behaviour about `pastsend` in the invariants that is invariants (9 – 11) since we no longer use this variable — due to the slice progression. Second, we introduce these two new invariants:

- (12) and $(\forall e:\text{state}. e \in \bigcup \langle \text{known} \rangle) \rightarrow \text{slice}(e) < \text{ghost_slice}$
 (13) and $(\forall e:\text{state}. e \in \bigcup \langle \text{todo} \rangle) \rightarrow \text{slice}(e) = \text{ghost_slice}$

It is easier for the proofs of correctness to introduce the ghost variable `ghost_slice` which is incremented at each super-step and thus corresponds to the measure of progression of the protocol. Invariant (12) is needed to prove that the set `known` contains only states of the past slices and invariant (13) proves that `todo` contains only states of the current slice. The assertions of the local computations need also to be changed. Most of them are as for the “Naive” algorithm. They are available in Fig 4.9.

We only detail the true changes. First, In the pre-conditions and in all the annotations, we now used $\text{cpu}_{\mathcal{R}}$; Second, any state in `known` has its slice measure less than the current slice (super-step) since it is a state that has been previously computed; but any state in `todo` has its slice equal to the current slice. We maintain this fact in the invariants (10–12) where the slice measure of states to be send are equal to the current slice plus one (for the next slice/super-step). Finally, the post-conditions are as above and with the properties of the slice measure.

Algorithm “Sweep”. When dumping `known` at each begin of super-step/slice, we can thus no longer use `known` as the variable which contains the full state-space. We thus introduce another ghost variable called `ghost_known` which will grow at each super-step by recovering all the states of `known`. In this way and for having the correctness of this algorithm, in all the previous invariants, `ghost_known` replaces `known`. The rest is unchanged. Finally, it is thus an easy modification (for the algorithm and its correctness) but memory efficient.

```

1 let local_successors (known: state set ref) (todo:state set ref) (ghost_slice:int ref) =
2 { (known ∪ todo) ⊆ StSpace
3   and (known ∩ todo)=∅
4   and (∀ s:state. s ∈ (known ∪ todo) → cpuR(s)=bsp_pid)
5   and (∀ e:state. e ∈ (known ∪ todo) → succ(e) ⊆ (known ∪ todo))
6   and (∀ s:state. e ∈ (s,known) → slice(s) < ghost_slice)
7   and (∀ s:state. e ∈ (s,todo) → slice(s)=ghost_slice) }
8 init:let tosend = ref (init_send ∅) in
9 while todo ≠ ∅ do
10 {
11   invariant (1) (known ∪ todo ∪ ⋃(tosend)) ⊆ StSpace
12   and (2) (known ∩ todo)=∅
13   and (3) (∀ s:state. s ∈ (known ∪ todo) → cpuR(s)=bsp_pid)
14   and (4) ⋃(tosend) ∩ known=∅
15   and (5) (∀ i:int. isproc(i) → ∀ e:state. e ∈ tosend[i] → cpuR(e)=i)
16   and (6) (∀ e:state. e ∈ known → succ(e) ⊆ (known ∪ todo ∪ ⋃(tosend)))
17   and (7) ⊆ (⋃ (todo@init,known@init), ⋃ (todo,known))
18   and (8) (todo@init ∪ known@init) ⊆ (todo ∪ known)
19   and (9) todo@init=∅ → (todo=∅ and ⋃(tosend)=∅)
20   and (10) (∀ e:state. e ∈ known → slice(e) ≤ ghost_slice)
21   and (11) (∀ e:state. e ∈ ⋃(tosend) → slice(e)=ghost_slice+1)
22   and (12) (∀ e:state. e ∈ todo → slice(e)=ghost_slice)
23   variant |StSpace \ known|
24 }
25
26 let s = pick todo in
27 known ← !known ⊕ s;
28 todo ← !todo ∪ ((succC s) \ !known)
29 let send_states = ref ((succR s) \ !known) in
30 let ghost_diff=ref ∅ in
31 L:while send_states ≠ ∅ do
32 {
33   invariant (1) and (2) and (3) and (4) and (5) and (6) and (10) and (11) and (12)
34   and (known ∩ send_states)=∅
35   and send_states@L = (ghost_diff ∪ send_states)
36   and (ghost_diff ∩ send_states)=∅
37   and (todo@init ∪ known@init) ⊆ (todo ∪ known)
38   and (∀ e:state. e ∈ send_states → slice(e)=ghost_slice+1)
39   and (todo@L ∪ known@L) ⊆ (todo ∪ known)
40   variant |send_states|
41 }
42 let new_s = pick send_states in
43 let tgt=cpuR(new_s) in
44 ghost_diff ← !ghost_diff ⊕ new_s
45 tosend[tgt] ← tosend[tgt] ⊕ new_s
46 done
47 done;
48 !tosend
49 { (known ∪ ⋃(result)) ⊆ StSpace
50   and todo=∅
51   and (∀ s:state. e ∈ (s,known) → cpuR(s)=bsp_pid)
52   and (known@ ∪ todo@) ⊆ known
53   and (∀ i:int. isproc(i) → ∀ e:state. e ∈ result[i] → cpuR(e)=i)
54   and (∀ e:state. e ∈ known → succ(e) ⊆ (known ∪ ⋃(result)))
55   and ⋃(result) ∩ known = ∅
56   and (∀ e:state. e ∈ known → slice(e) ≤ ghost_slice)
57   and (∀ e:state. e ∈ ⋃(result) → slice(e)=ghost_slice+1)
58   and (todo@=∅) → (⋃(result)=∅)
59   and (∀ i:int. isproc(i) → ∀ e:state. e ∈ result[i] → succ(e) ⊆ StSpace) }

```

Figure 4.9. Annotated computations of the “Incr” algorithm.

Algorithm “Balance”. Using another partition function that is cpu_B which partitioned (at each slice) the state-space into independent classes of states, we cannot remain of the cpu_R partition. In this way, tosend is not a fix (size nprocs) array of sets of states, but rather a mapping from the hashing representing the classes (integer compute using cpu_B) to sets of states (the states of the classes). For the correctness of the algorithm, we thus need a new predicate $\text{class}(e,e')$ that logically define that two states belong to the same class. We also need to redefine the predicate $\text{GoodPart}(\langle p_set \rangle)$ as follow:

$$\forall i,j:\text{int. isproc}(i) \rightarrow \text{isproc}(j) \rightarrow i \neq j \rightarrow \forall s,s':\text{state. } s \in p_set\langle i \rangle \rightarrow s' \in p_set\langle j \rangle \rightarrow \neg \text{class}(e,e')$$

which denotes that two states that belong to two different processors are not in the same class. We also need to assert that after the computation of the balance (currently axiomatised since it is a heuristic of a NP-problem), sent states respect the predicate GoodPart . We also need the following invariant:

$$(14) \text{ and } (\forall i:\text{int. isproc}(i) \rightarrow \forall e,e':\text{state. } e \in \text{ghost_known}\langle i \rangle \rightarrow \text{class}(e,e') \rightarrow e' \in \text{ghost_known}\langle i \rangle)$$

which denotes that known states respect the fact that all states in a class belong to the same slice at the same processor. The `local_successor` function should verify this fact in its invariants which is rather very simple to do. The complete annotated codes are also available in the source of this paper.

Proof Obligations. In the following table, for each part of each parallel algorithm, we give the number of obligations and how many are discharged by the provers:

Algorithm	Part	Total	ALT-ERGO	SIMPLIFY	z3	CVC3	YICES	VAMPIRE
Incr								
	main	109	50	93	85	0	0	85
	successor	105	55	102	101	77	0	73
	exchange	32	15	28	22	19	0	27
Sweep								
	main	129	62	114	109	0	0	92
	successor	107	58	103	102	81	0	78
	exchange	31	14	29	23	21	0	28
Balance								
	main	135	71	123	119	0	0	102
	successor	113	62	111	108	87	0	81
	exchange	38	16	31	29	22	0	29

As above, only the combination of all provers is able to prove all the obligations. And few of them need that provers run minutes. SIMPLIFY and z3 still remain the most efficient. An interesting point is that this work has been partially done by a master student: he was able to perform the job (annotate these parallel algorithms) in three months. Based on this fact, it seems conceivable that a more seasoned team in formal methods can tackle more substantial algorithms (of model-checking) in a real programming language.

4.3 Related Work

4.3.1 Other Methods for Proving the Correctness of Model-checkers

Fig. 4.10 summarises different methods that have been used for verifying MCs where each arrow corresponds to a proof of correctness (using a theorem prover or a PCC approach) and the papers related to the work. The state-space explosion can be a problem for MCs extracted from theorem provers. They are pure functional programs such as the ones of [96, 261]. They certainly would be too slow for big models even if there work on obtaining imperative programs from extracted (pure) functional programs. A sequential state-space algorithm (with a partial order reduction) has been checked in B in [275].

The “certifying model-checking” is an established research field [230, 269]. But, the performance issue of PCC is discussed in [280] and [239] where the authors present developments (and model-checking benchmarks) of BDDs and tree automata using theorem provers: BDDs are common data-structures used by MCs and tree automata is an approach for having a formal successor function. PCC only focuses on the generation of independently-checkable evidences as the compiled code satisfies a simple behavioural specification such as memory safety; the evidence can then be checked efficiently. Using PCC for state-space is the same as computing it a “second time”. In fact, the drawback of proof certificates is that verification tools have to be instrumented to generate them, and the size of fully expanded proofs may be too large. Authors of [239, 280] conclude that PCCs are here inadequate and we can conclude that MCs themselves need to be proved. It is also the conclusion of [109] where the authors note that “to avoid the inefficiency of fully expansive proof generations, a number of researchers have advocated the verification of decision procedures”.

In [109, 248], the authors have done a mechanical verification (using the theorem prover PVS) of automatic solvers methods: they note that the inefficiency of fully expansive proof generation can be avoided through verifying the decision procedures. One of the authors argues that trust need not be achieved at the expense of automation, and outlines a lightweight approach where the results of untrusted verifiers are checked by a trusted offline checker. The trusted checker is a verified reference kernel that contains a satisfiability solver to support the robust and efficient checking of untrusted tools. He summarizes the problem: *Quis custodiet ipsos custodes?* (Who will watch the watchmen? that is, who will verify the verifier?). We want to be able to trust the results of provers/model-checkers with a high degree of confidence. But currently, only an approach using functional programs is presented with the same main drawback as above: less efficiency.

In our work, we also only use automatic solvers for proving the generated goals of the VCG WHY and thus we do not use any “elaborate” theorem prover such as COQ. The correctness of our results depends on the

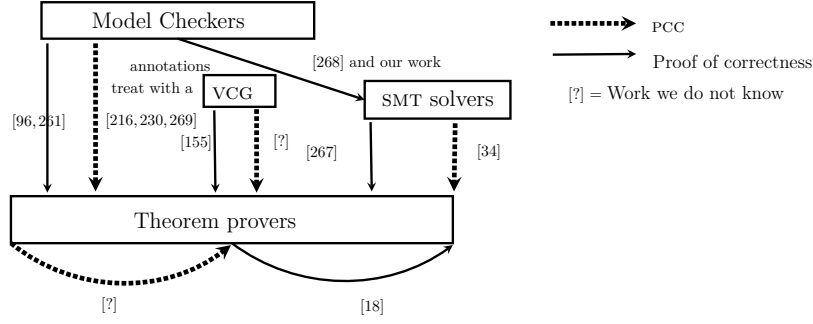


Figure 4.10. Different ways for proving model-checking algorithms.

correctness of (1) the WHY tool (correct generation of goals) and (2) the results of the solvers. Relying on modules like SMT solvers has the advantage that these tools would certainly be verified in a close future. The work of [155] is a first approach for (1) and the work of [34] is a PCC approach for (2). Moreover, a SMT solver has been proved using a theorem prover [267]. In a close future, we can hope to achieve the same confidence in our codes as the MCs extracted from [96, 261], as well as better performances since our codes are realistic imperative codes — and not functional ones from theorem provers. Finally, we think that using annotations (and a VCG tool) has the advantage of being “easy”. And we can prove the correctness of programs or limit the work to some safety properties if the full correctness is too difficult to obtain. And it extends to parallel programs which is not easy using PCCs or theorem provers.

A mechanically assisted proof using Isabelle of how LTL formulae can be transformed into Büchi automata is presented in [245]. CTL* temporal logic is also available in COQ [274] (and LTL in [69]). All these works are interesting since logical theories may be axiomatised in WHY.

Model compilation is one of the numerous techniques to speedup model-checking: it relies on generating source code (then compiled into machine code) to produce a high-performance implementation of the state-space exploration primitives, mainly the successor function. In [113], authors propose a way to prove the correctness of such an approach. More precisely, they focus on generated **Low-Level Virtual Machine** (LLVM) code from high-level Petri nets and manually prove that the object computed by its execution is a representation of the compiled model. If such a work can be redone using a theorem prover, we will have a machine-checked successor function which is currently axiomatised in WHY.

4.3.2 Verification of Security Protocols

Gavin Lowe has discovered the now well-known attack on the Needham-Schroeder public-key protocol using the model-checker FDR [201]. In spite of this, over the last two decades, a wide variety of security protocol analysis tools have been developed that are able to detect attacks on protocols or, in some cases, establish their correctness. We distinguish three classes: tools that attempt *verification* (proving a protocol correct), those that attempt *falsification* (finding flaws, *i.e.*, counterexamples), and hybrids that attempt to provide both proofs and counterexamples. In the first category, we find the use of theorem provers [229] and dedicated tools such as PROVERIF [27] [[64]] or SCYTHER [71] [[65]], ATHENA [260], *etc.*, falsification is the domain of model-checking [6, 7], and the latter of model-checker with lazy intruders such as AVISPA [8] [[66]].

Papers [66, 72, 77] present different cases study of verifying security protocols with various standard tools. To summarise, there is currently no tool that provides all the expected requirements. Most of them limit possible kinds of attacks or limit in their model language how addresses of agents can be manipulated in ad-hoc protocols (using arithmetic operations). Paper [77] presents different cases study of verifying security protocols with various standard tools. To summarise, there is currently no tool that provides all the expected requirements. Model-checking is the one we choose in this work.

Model-checking security protocols is not new [6, 20, 196, 202]. In the work of [208], a POR reduction of security protocols is used and in the work of [6], a lazy intruder was used — both are sequential algorithms. Our work has the advantage to take into account distributed architectures. In [215], the authors have used the MURPHI modelling language and different parallel programs for MURPHI now exist. But the algorithm [263] uses a naive random hash function. Notice also the work [30] for the μ CRL checker. For finite checking scenarios (and enumerate state-space construction), the most well known tool is certainly AVISPA [8]. We believe that our observations and the subsequent *optimisations* are general enough to be

adapted to the tools dedicated to protocol verification: we worked in a general setting of Kripke structure, defined by an initial state and a successor function. Our only requirements are the four simple conditions (P1 to P4) that can be easily fulfilled within most concrete modelling formalisms.

[115] allows to verify some properties about some classes of protocols for an infinite number of sessions and with some possibility of replay using a process algebra. Nevertheless, each time a new property is needed, a new theorem has to be proved. That could lead to a complex maintenance of the method. Furthermore, the method cannot be applied to some protocols, *e.g.*, the Yahalom one. On the contrary, our approach is based on explicit state-space construction, that is not tied to any particular application domain. This is a well-desired feature for checking P2P security protocols as in [52].

To our knowledge, there are three existing approaches to automatically generate machine-checked protocol security proofs. The first approach is in [139] where a protocol and its properties are modelled as a set of Horn-clauses and where the certificate is machine-checked in COQ. The second [43] uses the theorem prover ISABELLE and computes a *fixpoint* of an *abstraction* of the transition relations of the protocol of interest — this fixpoint over-approximates the set of reachable states of the protocol. The latter [19, 213] also uses COQ or ISABELLE and invariants are derived from an operational semantics of the protocols. We see three main drawbacks to these approaches. First, they *limit* (reasonably) protocols and properties that can be checked. Second, *each time*, the proof of the tested property of the protocol needs to be machine-checked; in our approach, the results of the MC are correct by construction. Third, there is currently no possibility of distributed computations for larger protocols — hours of computations are needed for some protocols. But they have one evident advantage: only the final result has to be correct (intermediate computations do not need to be verified) and it saves time for correct design (machine-checked) of algorithms. All these methods are also based on perfect cryptography. The author of [83] annotated cryptographic algorithms to mechanically prove their correctness.

4.3.3 Distributed State-space Construction

The main idea of most known approaches to the distributed memory state space generation is similar to the naive algorithm [116].

Close to our hashing technique, [232] presents a hashing function that ensures that most of the successors are local: the partition function is computed by a round-robin on the successor states. This improves the locality of the computations but can duplicate states. Moreover, it works well only when network communication is substantially slower than computation, which is not the case on modern architectures for explicit model-checking. For load balancing, [199] presents a new dynamic partition function scheme that builds a dynamic remapping, based on the fact that the state space is partitioned into more pieces than the number of involved machines. When load on a machine is too high, the machine releases one of the partitions it is assigned and if it is the last machine owning the partition it sends the partition to a less loaded machine.

In [191], a state-space exploration algorithm derived from the SPIN model-checker is implemented using a master/slave model of computation. Several SPIN-specific partition functions are experimented, the most advantageous one being a function that takes into account only a fraction of the state vector, similarly to our function cpu_R . The algorithm performs well on homogeneous networks of machines, but it does not outperform the standard implementation except for problems that do not fit into the main memory of a single machine. Moreover, no clue is provided about how to correctly choose the fraction of states to consider for hashing, while we have relied on reception locations from $\mathcal{L} - R$.

In [228] various technics from the literature are extended in order to avoid sending a state away from the current processor if its 2nd-generation successors are local. This is complemented with a mechanism that prevents re-sending already sent states. The idea is to compute the missing states when they become necessary for model-checking, which can be faster than sending it. That clearly improves communications but our method achieves similar goals, in a much simpler way, without ignoring any state.

There also exist approaches, such as [178], in which parallelization is applied to “partial verification”, *i.e.* state enumeration in which some states can be omitted with a low *probability*. In our project, we only address *exact, exhaustive* verification issues. For the partition function, different technics have been used. In [116] authors used of a prime number of virtual processors and map them to real processor. This improves load balancing but has no real impact on cross transitions. In [225], a user defined *abstract interpretation* is used to reduce the size of the state space and then it allows to distribute the abstract graph; the concrete graphs is then computed in parallel for each part of the distributed abstract graph. In contrast, our distribution method is fully automated and does not require input from the user.

To our knowledge, the first BSP algorithm for state-space construction is the one of [209]. The authors

limit their study to dead/livelock detection of CSP expressions. Termination is ensuring by an additional global “reduction operation” to discover the total number of pending states which remain in the system. They also used fixed size buckets and processors computed until their own buckets are full which induce a greater number of super-steps but a better balancing.

5 Formal Semantics

This chapter subsumes the work of [119, 124].

Contents

5.1	Big-steps Semantics	92
5.1.1	Semantics Rules	92
5.1.2	Co-inductive Semantics	93
5.1.3	Adding the Subgroup Synchronisation	95
5.2	Small-steps Semantics	97
5.2.1	Semantics Rules	97
5.2.2	Co-inductive Semantics	99
5.2.3	Equivalence Between the Semantics	99
5.2.4	Adding the Subgroup Synchronisation	100
5.3	Semantics in Coq	101
5.3.1	Mechanized Semantics in Coq	101
5.3.2	Memory Model	103
5.3.3	Environment of Execution	103
5.3.4	Semantics Rules	104
5.3.5	Adding the Subgroup Synchronisation	106
5.4	Proof of the Translation	108
5.4.1	Program Correctness	108
5.4.2	Equivalence Between Elements of the Semantics	110
5.4.3	Elements of Proof in Coq	114
5.5	Related Work	116

In chapter 3, we presented a transformation from a BSP-WHY-ML program to a WHY-ML program, and explained how it was thus possible to use the WHY tools to prove the correctness of the original program. However, one can only trust the proof obtained in this way as much as the BSP-WHY tool is trusted. In this chapter, we present the tools needed to acquire this trust.

First, we define formal semantics of the BSP-WHY-ML language, which is necessary to be able to reason formally about the programs and the transformation. We start by giving a mathematical definition of the semantics in Section 5.1. A natural big-steps operational semantics is presented, close to the intuition of the language. We also give co-inductive infinite semantics, allowing us to study programs that are running indefinitely. Those big-steps semantics can be seen as a formal specification. In the Section 5.2, we give a more refined small-step semantics, using continuations. This semantics, although less intuitive, is better suited for complex proofs on BSP-WHY-ML programs. To keep a high level of trust, the semantics is proved equivalent to the big-step semantics, the specification. The challenges added if we allow the subgroup synchronisation, and the corresponding semantics, are also studied along these sections.



In Section 5.3, we show more in details how those operational semantics are formalised (machine-checked) in COQ using a set of (co-)inductive definitions. In this chapter, all the proofs that have been machine-checked in COQ, are concluded with a symbol of a rooster.

Finally, in Section 5.4, we will present the proof of the transformation done by BSP-WHY. The structure of the proof was also checked by the machine in COQ, though not all the steps are currently proved in COQ because of a lack of time.

5.1 Big-steps Semantics

5.1.1 Semantics Rules

(a) Problematic

Values to be sent and distant reading/writing are stored in environments of communications as simple list of messages. There are thus six additional components in the environment ($\mathcal{R}, \mathcal{C}^{\text{send}}, \mathcal{C}^{\text{put}}, \mathcal{C}^{\text{get}}, \mathcal{C}^{\text{pop}}, \mathcal{C}^{\text{push}}$), one per primitive that needs communications. Each is a simple list of messages. For DRMA primitives, there is also the registration \mathcal{T} as described in Section. 3.2 (push and pop need communications to keep the registration of every processor coherent).

In [102], J.-C. Filliâtre gives the formal definitions and rules for a big-step semantics of the WHY-ML language. The notions of values v and states for BSP-WHY-ML are similar to the WHY-ML semantics with the additional possible value $\text{SYNC}(e)$, which describes the awaiting of a synchronisation with e as program to be executed after the global synchronisation: $v ::= c \mid (Ec) \mid \text{SYNC}(e)$. A value v can be a constant (integer, boolean, *etc.*), an exception E carrying a constant c , or the synchronisation state.

We note s for the environment of a processor. It is a 8-tuple $\mathcal{E}, \mathcal{T}, \mathcal{R}, \mathcal{C}^{\text{send}}, \mathcal{C}^{\text{put}}, \mathcal{C}^{\text{get}}, \mathcal{C}^{\text{pop}}, \mathcal{C}^{\text{push}}$. We note $s.X$ to access to the component X of the environment s , \oplus the update of a component of an environment without modifying other components and \in to test the presence of an item in the component.

As the BSPLIB, DRMA variables are registered using a registration mechanism that is each processor contains a registration \mathcal{T} which is a list of registered variables: the first one in the list of a processor i corresponds to first one of the processor j .

(b) Local Rules

We first have semantics rules for the local execution of a program, on a processor i . We note $s, e \Downarrow^i s', v$ for these local reductions rules (*e.g.* one at each processor i): e is the program to be executed, v is the value after execution, s is the environment before the execution, s' the environment after the execution.

Rules for the local control flows are fully defined in Fig. 5.2. For each control instruction, it is necessary to give several rules, depending on the result of the execution of the different sub-instructions: one when an execution leads to a synchronisation (when processors finish a super-step), and one if it returns directly a value. We have thus to memorise as a value the next instructions of each processor. These intermediate local configurations are noted $\text{SYNC}(e)$. To avoid confusion between a new reference and those that have been registered before, one can not declare a reference that has been created before. This is not a problem since WHY already forbids this case. Rules of the BSP operations are given in Fig. 5.1 (executed on a single processor i). Basically, a primitives adds the corresponding message to the environment.

(c) Parallel Rules and Results

BSP programs are SPMD ones so an expression e is started \mathbf{p} times. We model this as a \mathbf{p} -vector of e with its environments. A final configuration is a value on all processors. We note \Downarrow for this evaluation and the two needed rules are given in Fig. 5.3.

First rule gives the base case, when each processor i executes a local (sequential) evaluation \Downarrow^i to a final value. The second rule describes the synchronisation process when all processors execute to a $\text{SYNC}(c)$ state: the communication are effectively done during the synchronisation phase and the current super-step is finished. The **AllComm** function models the exchanges of messages and thus specifies the order of the messages. It modifies the environment of each processor i . For the sake of brevity, we do not present this function which is a little painful to read and is just a reordering of the \mathbf{p} environments.

Note that if at least one processor finishes his execution while others are waiting for a synchronisation, a deadlock will occur. Finally we have the following results.

Lemma 1

$\forall i \quad \Downarrow^i$ is deterministic.

Proof. Trivially proved by induction on the evaluation.



Lemma 2

\Downarrow is deterministic.

$$\begin{array}{c}
\frac{}{s, c \Downarrow^i s, c} \\
\\
\frac{s, e_1 \Downarrow^i s', v \quad s'[x \leftarrow v], e_2 \Downarrow^i s'', o}{s, \text{let } x = e_1 \text{ in } e_2 \Downarrow^i s'', o} \quad \frac{s, e_1 \Downarrow^i s', E(v)}{s, \text{let } x = e_1 \text{ in } e_2 \Downarrow^i s', E(v)} \\
\\
\frac{s, e_1 \Downarrow^i s', \text{SYNC}(e')}{s, \text{let } x = e_1 \text{ in } e_2 \Downarrow^i s', \text{SYNC}(\text{let } x = e' \text{ in } e_2)} \\
\\
\frac{s, e_1 \Downarrow^i s', v \quad s'[x \leftarrow v], e_2 \Downarrow^i s'', o}{s, \text{let } x = \text{ref } e_1 \text{ in } e_2 \Downarrow^i s'', o} \quad \frac{s, e_1 \Downarrow^i s', E(v)}{s, \text{let } x = \text{ref } e_1 \text{ in } e_2 \Downarrow^i s', E(v)} \\
\\
\frac{s, e_1 \Downarrow^i s', \text{SYNC}(e')}{s, \text{let } x = \text{ref } e_1 \text{ in } e_2 \Downarrow^i s', \text{SYNC}(\text{let } x = \text{ref } e' \text{ in } e_2)} \\
\\
\frac{s, e \Downarrow^i s', v}{s, x := e \Downarrow^i s'[x \leftarrow v], \text{void}} \quad \frac{s, e \Downarrow^i s', E(v)}{s, x := e \Downarrow^i s', E(v)} \quad \frac{s, e \Downarrow^i s', \text{SYNC}(e')}{s, x := e \Downarrow^i s', \text{SYNC}(x := e')} \\
\\
\frac{s, e_1 \Downarrow^i s', \text{true} \quad s', e_2 \Downarrow^i s'', o}{s, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow^i s'', o} \quad \frac{s, e_1 \Downarrow^i s', \text{false} \quad s', e_3 \Downarrow^i s'', o}{s, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow^i s'', o} \\
\\
\frac{s, e_1 \Downarrow^i s', E(v)}{s, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow^i s', E(v)} \quad \frac{s, e_1 \Downarrow^i s', \text{SYNC}(e')}{s, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow^i s', \text{SYNC}(\text{if } e' \text{ then } e_2 \text{ else } e_3)} \\
\\
\frac{s, e; \text{loop } e \{ \text{invariant } i \text{ variant } v \} \Downarrow^i s', o}{s, \text{loop } e \{ \text{invariant } i \text{ variant } v \} \Downarrow^i s', o} \\
\\
\frac{s, e \Downarrow^i s', v}{s, \text{raise } (E e) \Downarrow^i s', E(v)} \quad \frac{s, e \Downarrow^i s', E'(v)}{s, \text{raise } (E e) \Downarrow^i s', E'(v)} \quad \frac{s, e \Downarrow^i s', \text{SYNC}(e')}{s, \text{raise } (E e) \Downarrow^i s', \text{SYNC}(\text{raise } (E e'))} \\
\\
\frac{s, e_1 \Downarrow^i s', v}{s, \text{try } e_1 \text{ with } E x \rightarrow e_2 \text{ end } \Downarrow^i s', v} \quad \frac{s, e_1 \Downarrow^i s', E(v) \quad s'[x \leftarrow v], e_2 \Downarrow^i s'', o}{s, \text{try } e_1 \text{ with } E x \rightarrow e_2 \text{ end } \Downarrow^i s'', o} \\
\\
\frac{s, e_1 \Downarrow^i s', \text{SYNC}(e')}{s, \text{try } e_1 \text{ with } E x \rightarrow e_2 \text{ end } \Downarrow^i s', \text{SYNC}(\text{try } e' \text{ with } E x \rightarrow e_2 \text{ end})} \\
\\
\frac{s, e \Downarrow^i s', o}{s, \text{assert } \{p\}; e \Downarrow^i s', o} \quad \frac{s, e \Downarrow^i s', o}{s, e \{p\} \Downarrow^i s', o} \quad \frac{s, e \Downarrow^i s', o}{s, e : l \Downarrow^i s', o}
\end{array}$$

Figure 5.1. Big-step semantics: local sequential operations.

Proof. Trivially proved by induction on the evaluation.



5.1.2 Co-inductive Semantics

In addition to the standard big-step operational semantics, it is often useful to define co-inductive (or infinite) semantics. They allow to characterize the behaviour of a program that runs indefinitely.

Defining divergence (infinite evaluations) is also done using inference rules but interpreted coinductively. More precisely, the relation is the greatest fixpoint of the rules, or, equivalently, the conclusions of infinite derivation trees built from these rules [193]. Throughout this thesis, double horizontal lines in inference rules denote inference rules that are to be interpreted coinductively; single horizontal lines denote the inductive interpretation.

$$\begin{array}{c}
\frac{}{s, \mathbf{bsp_sync} \Downarrow^i s, \mathbf{SYNC}(\mathbf{void})} \quad \frac{s' = s.\mathcal{C}^{\text{push}} \oplus x}{s, \mathbf{bsp_push} x \Downarrow^i s', \mathbf{void}} \quad \frac{s' = s.\mathcal{C}^{\text{pop}} \oplus x}{s, \mathbf{bsp_pop} x \Downarrow^i s', \mathbf{void}} \\
\\
\frac{}{s, nprocs \Downarrow^i s, \mathbf{p}} \quad \frac{}{s, pid \Downarrow^i s, i} \\
\\
\frac{s, e \Downarrow^i s', to \quad 0 \leq to < \mathbf{p} \quad \{x \mapsto c\} \in s'.\mathcal{E} \quad n^{\text{th}}(s'.\mathcal{T}, y) = n \quad s'' = s'.\mathcal{C}^{\text{put}} \oplus \{to, c, n\}}{s, \mathbf{bsp_put} e x y \Downarrow^i s'', \mathbf{void}} \\
\\
\frac{s, e \Downarrow^i s', \mathbf{SYNC}(e')}{s, \mathbf{bsp_put} e x y \Downarrow^i s', \mathbf{SYNC}(\mathbf{bsp_put} e' x y)} \quad \frac{s, e \Downarrow^i s', \mathbf{SYNC}(e')}{s, \mathbf{bsp_get} e x y \Downarrow^i s', \mathbf{SYNC}(\mathbf{bsp_get} e' x y)} \\
\\
\frac{s, e \Downarrow^i s', \mathbf{SYNC}(e')}{s, \mathbf{bsp_send} x e \Downarrow^i s', \mathbf{SYNC}(\mathbf{bsp_send} x e')} \\
\\
\frac{s, e \Downarrow^i s', to \quad 0 \leq to < \mathbf{p} \quad \{x \mapsto c\} \in s'.\mathcal{E} \quad n^{\text{th}}(s'.\mathcal{T}, y) = n \quad s'' = s'.\mathcal{C}^{\text{get}} \oplus \{to, n, x\}}{s, \mathbf{bsp_get} e x y \Downarrow^i s'', \mathbf{void}} \\
\\
\frac{s, e \Downarrow^i s', to \quad 0 \leq to < \mathbf{p} \quad \{x \mapsto c\} \in s'.\mathcal{E} \quad s'' = s.\mathcal{C}^{\text{send}} \oplus \{to, c\}}{s, \mathbf{bsp_send} x e \Downarrow^i s'', \mathbf{void}} \\
\\
\frac{s, t_1 \Downarrow^i s', to \quad 0 \leq to < \mathbf{p} \quad s', t_2 \Downarrow^i s'', n \quad \{to, n, c\} \in s''.\mathcal{R}}{s, \mathbf{bsp_findmsg} t_1 t_2 \Downarrow^i s'', c} \\
\\
\frac{s, t \Downarrow^i s, to \quad 0 \leq to < \mathbf{p} \quad n = \text{Size}(s.\mathcal{R}, to)}{s, \mathbf{bsp_nmsg}(t) \Downarrow^i s, n}
\end{array}$$

Figure 5.2. Big-step semantics: local BSP operations.

$$\begin{array}{c}
\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, v_i}{(s_0, c_0), \dots (s_{\mathbf{p}-1}, c_{\mathbf{p}-1}) \Downarrow (s'_0, v_0), \dots (s'_{\mathbf{p}-1}, v_{\mathbf{p}-1})} \\
\\
\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, \mathbf{SYNC}(e'_i) \quad \mathbf{AllComm}\{(s'_0, e'_0), \dots (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1})\} \Downarrow (s''_0, v_0), \dots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}{(s_0, e_0), \dots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow (s''_0, v_0), \dots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}
\end{array}$$

Figure 5.3. Big-step semantics: global reductions.

The co-inductive rules for the local control flow are as expected. They can easily be inferred from the regular big-step semantics. We give as an example one of the co-inductive rules for the **if** instruction:

$$\frac{s, e_1 \Downarrow^i s_1, \mathbf{true} \quad s_1, e_2 \Downarrow_{\infty}^i}{s, \mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3 \Downarrow_{\infty}^i}$$


The rule can be read as follow: “If e_1 evaluates to *true*, and if e_2 runs infinitely, then the program **if** e_1 **then** e_2 **else** e_3 will run infinitely”. Local BSP operation (**push**, **send**, *etc.*) always terminates, so the co-inductive are as expected. On the other hand, and more interestingly, we define the co-inductive rules for the parallel operations:

$$\begin{array}{c}
\frac{\frac{\exists i \quad s_i, e_i \Downarrow_{\infty}^i}{\langle (s_0, e_0), \dots, (s_{p-1}, e_{p-1}) \rangle \Downarrow_{\infty}}}{\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, \text{SYNC}(e'_i) \quad \text{AllComm}\{\langle (s'_0, e'_0), \dots, (s'_{p-1}, e'_{p-1}) \rangle\} \Downarrow_{\infty}}{\langle (s_0, e_0), \dots, (s_{p-1}, e_{p-1}) \rangle \Downarrow_{\infty}}}
\end{array}$$

The first rule states that if one of the processors runs infinitely, then the parallel program will run infinitely. In the second rule, it is said that if all the processors reach a synchronisation barrier, and if the parallel program runs infinitely starting from the resulting state, then the parallel program runs infinitely. Some results linking the two semantics are easily proved:

Lemma 3

\Downarrow and \Downarrow_{∞} are mutually exclusive.

Proof. Trivially proved by induction and co-induction on the evaluations. 

Co-inductive semantics is also deterministic in the sense that the constructed infinite tree will always be the same. But as we do not currently give the execution traces, this property is not interesting and we felt non-provable in COQ.

5.1.3 Adding the Subgroup Synchronisation

In Section 3.3, we introduced a variation of BSP-WHY, that is not strictly dealing with the pure BSP model, instead allowing processors to synchronise within a *subgroup*. This gives us the possibility to work on parallel programs written for the BSP PUB library, which allows for some limited subgroup synchronisation, but also on some families of MPI programs.

However, the subgroup synchronisation also introduces a slightly more complex language and parallel model, thus it is necessary to give the corresponding semantics. Let us describe the modifications that are needed.

(a) Local Operations

The semantics of local control flow operations of BSP-WHY-ML is not modified when introducing the subgroup synchronisation. There are some modifications in the way conditional instructions and loops are handled by BSP-WHY, but the language still follows the same semantics. Thus the rules defined in Fig. 5.2 are still valid.

With the subgroup synchronisation model, BSP operations are done in the scope of a *communicator*. Every BSP call thus takes an additional argument, this communicator, which describes the subset of processors in which the communications are done. However, apart from this addition, everything else remains the same. For example, the inductive rule for $\mathcal{C}^{\text{send}}$ is now:

$$\frac{s, e \Downarrow^i s', to \quad cmt \in s' \quad 0 \leq to < p_{cmt} \quad \{x \mapsto v\} \in s'.\mathcal{E} \quad s'' = s'.\mathcal{C}_{cmt}^{\text{send}} \oplus \{to, v\}}{s, \text{bsp_send } x \text{ e } cmt \Downarrow^i s'', \text{void}}$$

where “*cmt*” is a valid communicator in the environment. The message to processor “*to*” of the communicator “*cmt*” is added to the queue of message.

(b) Global Reduction Rules

The major changes in the semantics are located within the parallel rules. Instead of having all the **p** processors synchronise together, and communicate together, it is now possible for a subgroup to synchronize together and make the needed communications. Several subgroups can also work independently from each other, and synchronise at the same time. There are two major options for the semantics in this situation:

1. All processors execute their code locally, until they reach a synchronisation state (or they terminate). We execute all of the possible subgroup synchronisations, and then start again the local computations. We call this options **AllSub**.
2. A subgroup of processors execute their code locally, until they reach their synchronisation. We execute the synchronisation and the associated communications, then start again. We call this options **Diamond**.

With the subgroup synchronisation, the BSP notion of *superstep* is less clearly defined, and the two options could be seen as two different definitions of a superstep in this model.

There are *pros and cons* to both formulations. In the **AllSub** option, the rule is complex to write, even more so in the COQ proof assistant. However, it is perhaps the rule matching most closely the execution of a parallel program where all processors compute in parallel. The **Diamond** definition is much easier to write and understand, but artificially gives priority to one subgroup over another one. This ordering of the subgroups execution leads to another issue: with this definition, the semantics loses its determinism, since when several subgroup are synchronising, it is possible to choose any subgroup to execute first.

We first give the formulation of **Allsub**:

$$\frac{\{0, \dots, p-1\} = N \oplus C_1 \oplus \dots \oplus C_k \quad \forall i \in C_j \quad s_i, e_i \Downarrow^i s'_i, \text{SYNC}(C_j, e'_i) \quad \forall i \in N \quad s_i, e_i \Downarrow^i s'_i, v_i}{\text{AllCommSimul}\{C_1 \dots C_k, v, (s'_0, e'_0), \dots (s'_{p-1}, e'_{p-1})\} \Downarrow_{All} (s''_0, v_0), \dots (s''_{p-1}, v_{p-1})} \\ (s_0, e_0), \dots (s_{p-1}, e_{p-1}) \Downarrow_{All} (s''_0, v_0), \dots (s''_{p-1}, v_{p-1})$$

In this rule, we first partition the set of processors in k subsets that will synchronise, plus a subset N of processors that do not synchronise. **AllCommSimul** is then similar to the **AllComm** function defined before. However, because there can be several subgroups synchronizing, its exact definition is more complicated.

- It accepts as arguments the set of the communicators used in the synchronisation.
- In addition, it accepts as argument the array of the final values v_i already reached in the superstep. For $i \in N$, the i -th processor terminates without synchronisation with the value v_i , so the i -th component of the result of **AllCommSimul** will be the couple (s'_i, v_i) .
- For every set of processors matching a communicator C_j , all the communications corresponding to the communicator are done.
- Among the messages of these processors, it only considers the ones that were sent within the matching communicator.

We now give the formulation of **Diamond**:

$$\frac{\forall i \in C \quad s_i, e_i \Downarrow^i s'_i, \text{SYNC}(C, e'_i) \quad \text{AllCommC}\{C, (s'_0, e'_0), \dots (s'_{p-1}, e'_{p-1})\} \Downarrow_{Diamond} (s''_0, v_0), \dots (s''_{p-1}, v_{p-1})}{(s_0, e_0), \dots (s_{p-1}, e_{p-1}) \Downarrow_{Diamond} (s''_0, v_0), \dots (s''_{p-1}, v_{p-1})}$$


Here, **AllCommC** is similar to the **AllComm** function defined before, with the following differences:

- It accepts a second argument (a communicator).
- It only modifies the environments of the processors in the range of the communicator.
- Among the messages of these processors, it only considers the ones that were sent within the matching communicator.

It is easy to see that even though this semantics leads to non-determinism, it is still confluent. The reason is that the only source of non-determinism is the communication rules, for which any matching communicator can be chosen. However, at any given point, the communications between two communicators are independent, since each processor leads to a synchronisation in one communicator only. Thus, the diamond property holds. We justify our use of the second, shortest rule with the following lemma:

Property 1

The **Diamond** semantics is confluent.

Proof. It is a direct consequence of the diamond property. 

Lemma 4

Both semantics are equivalent.

Proof. By induction on the number k of communicators used in a **AllSub** super-step, we can show that there is a **Diamond** derivation, formed by k synchronizations, that simulates the global **AllSub** synchronization rule.

Since the **Diamond** semantics is confluent, any derivation will lead to the same result. We thus have the following:

$$s, e \Downarrow_{Diamond} s', v \wedge s, e \Downarrow_{All} s'', v' \Rightarrow s' = s'' \wedge v = v'$$



5.2 Small-steps Semantics

Big-step semantics offers rules that are both easily readable and intuitive. They are thus most adapted for a specification of the BSP-WHY language. For basic properties (such as the termination of a given program, *etc.*), these semantics allow for a natural proof. However, in order to formally prove more complex results, it is often necessary to reason precisely on the parallel execution of a program. Having a semantics that describes in the closest possible way the execution, including for instance the interleaving of the computations on the different processors, is needed. Proving the correctness of the BSP-WHY tool, *i.e.* that the generated sequential program will be correct if and only if the parallel program is correct, is such a result that benefits from additional control in the semantics. For this reason, we provide a small-step operational semantics for the BSP-WHY language. In this section, we first define the small-step semantics mathematically, then explain its definition in COQ. We then prove the equivalence between the small-step semantics, and the big-step semantics that was given as the definition of the language. This proof is entirely checked by the COQ proof assistant.

5.2.1 Semantics Rules

Small-step semantics specify the execution of a program, one step at a time. A set of rules is repeatedly applied on program states (or configurations), until a final state is reached. If rules can be applied infinitely, it means the program diverges. If at one point in the execution there is no rule to apply, it is a faulty program.

In our parallel case, we will have two kinds of one-step reductions: local ones (on each processor) noted \xrightarrow{i} and global ones (for the whole parallel machine) noted \rightarrow . The whole evaluation \rightarrow^* of a program is the transitive and reflexive closure of \rightarrow . For diverging programs, we note the whole (co-inductive) reduction $\xrightarrow{\infty}$. All of our semantics are thus a set of “rewriting” rules. As we will see, the small-step semantic is harder to define than the big-step one.

(a) Problematic

As for the big-step semantic, most of rules are commons ones. Synchronisation is the only problem. A naive solution is the following global rule:

$$\langle (s_0, \mathbf{bsp_sync}; e_0), \dots, (s_{p-1}, \mathbf{bsp_sync}; e_{p-1}) \rangle \rightarrow \langle (s'_0, e_0), \dots, (s'_{p-1}, e_{p-1}) \rangle$$

that is all processors are waiting for a global synchronisation and then each processor executes what remains to be done. The problem with this rule is that it cannot evaluate a synchronisation inside a control instruction *e.g.* `if e_1 then $\mathbf{bsp_sync}$ else e_3` . Different solutions exist:

- Adding specific global rules for the synchronisation inside each control instruction; the drawback is that this implies too much rules;
- Using a global rule with “contexts” (a context is an expression with a hole): the `$\mathbf{bsp_sync}$` instruction replaces the hole within a context on each processor; the drawback is that the use of contexts is not friendly when using a theorem prover such as COQ;
- In [270] the authors propose the following rule: $s, \mathbf{bsp_sync} \xrightarrow{i} s, \mathbf{Wait}(\mathbf{skip})$ in adjunction with rules to propagate this waiting (as the ones of the big-step semantic) and the following rule

$$\langle (s_0, \mathbf{Wait}(e_0)), \dots, (s_{p-1}, \mathbf{Wait}(e_{p-1})) \rangle \rightarrow \langle (s_0, e_0), \dots, (s_{p-1}, e_{p-1}) \rangle$$

but two subtleties persist: (1) the rules add a *skip* instruction that complicates the proofs; (2) in their work, $(e_1; \mathbf{bsp_sync}); e_2$ cannot be evaluated, only $e_1; (\mathbf{bsp_sync}; e_2)$ can.

To remedy to the latter problem, in [124], we choose to add the congruence (equivalence) $(e_1; \mathbf{bsp_sync}); e_2 \equiv e_1; (\mathbf{bsp_sync}; e_2)$ but that also complicates the proofs. Another solution would consist in having only lists of instructions (and not $e_1; e_2$) but that complicates the proofs too.

(b) Local Rules

The solution we propose is the use of a “continuation semantics”, in the spirit of the semantics described in [4]¹. This semantics mainly allows a uniform representation of configurations that facilitates the design of lemmas.

¹Using this semantics we also get for free the evaluation of control structures in C (*e.g.* break and continue in loops) if we want to move to a realistic programming language such as C.

$$\begin{aligned}
& s, nprocs . \kappa \xrightarrow{i} s, \mathbf{p} . \kappa \\
& s, pid . \kappa \xrightarrow{i} s, i . \kappa \\
& s, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 . \kappa \xrightarrow{i} s, e_1 . (\text{if } _ \text{ then } e_2 \text{ else } e_3) . \kappa \\
& \quad s, \text{true} . (\text{if } _ \text{ then } e_2 \text{ else } e_3) . \kappa \xrightarrow{i} s, e_2 . \kappa \\
& \quad s, \text{false} . (\text{if } _ \text{ then } e_2 \text{ else } e_3) . \kappa \xrightarrow{i} s, e_3 . \kappa \\
& s, \text{let } x = e_1 \text{ in } e_2 . \kappa \xrightarrow{i} s, e_1 . (\text{let } x = _ \text{ in } e_2) . \kappa \\
& \quad s, v . (\text{let } x = _ \text{ in } e_2) . \kappa \xrightarrow{i} s[x \leftarrow v], e_2 . \kappa \\
& s, \text{let } x = \text{ref } e_1 \text{ in } e_2 . \kappa \xrightarrow{i} s, e_1 . (\text{let } x = \text{ref } _ \text{ in } e_2) . \kappa \\
& \quad s, v . (\text{let } x = \text{ref } _ \text{ in } e_2) . \kappa \xrightarrow{i} s[x \leftarrow v], e_2 . \kappa \\
& \quad s, x := e . \kappa \xrightarrow{i} s, e . (x := _) . \kappa \\
& \quad s, v . (x := _) . \kappa \xrightarrow{i} s[x \leftarrow v], \text{void} . \kappa \\
& s, \text{loop } e \{ \text{invariant } i \text{ variant } v \} . \kappa \xrightarrow{i} s, e; \text{loop } e \{ \text{invariant } i \text{ variant } v \} . \kappa \\
& \quad s, \text{raise } (E \ e) . \kappa \xrightarrow{i} s, e . (\text{raise } (E \ _)) . \kappa \\
& \quad s, v . (\text{raise } (E \ _)) . \kappa \xrightarrow{i} s, E \ v . \kappa \\
& s, \text{try } e_1 \text{ with } E \ x \rightarrow e_2 \text{ end} . \kappa \xrightarrow{i} s, e_1 . (\text{try } _ \text{ with } E \ x \rightarrow e_2 \text{ end}) . \kappa \\
& \quad s, v . (\text{try } _ \text{ with } E \ x \rightarrow e_2 \text{ end}) . \kappa \xrightarrow{i} s, v . \kappa \\
& \quad s, E' \ v . (\text{try } _ \text{ with } E \ x \rightarrow e_2 \text{ end}) . \kappa \xrightarrow{i} s, E' \ v . \kappa \\
& \quad s, E \ v . (\text{try } _ \text{ with } E \ x \rightarrow e_2 \text{ end}) . \kappa \xrightarrow{i} s[x \leftarrow v], e_2 . \kappa \\
& \quad s, E \ v . k . \kappa \xrightarrow{i} s, E \ v . \kappa, k \neq \text{try } _ \text{ with } _ \rightarrow _ \text{ end} \\
& \quad s, \text{assert } \{p\}; e . \kappa \xrightarrow{i} s, e . \kappa \\
& \quad s, e \{p\} . \kappa \xrightarrow{i} s, e . \kappa \\
& \quad s, e:l . \kappa \xrightarrow{i} s, e . \kappa
\end{aligned}$$

Figure 5.4. Small-step semantics: local sequential operations.

A configuration is completed with a control stack κ . The final configuration is $(s, \text{void} . \epsilon)$, an empty control stack. The control stack represents what has not been executed. There are sequential control operators to handle local control flow. This is close to an abstract machine. In Fig. 5.4 we give the local rules of control flow and in Fig. 5.5 those for the local BSP operations. Note that currently, no rule is needed for the **bsp_sync** instruction. In the control stack we find expressions with holes. Each hole represents the sub-expression that is currently evaluated.

Most instructions are dealt with by several rules. Generally, the first rule simply puts the instruction in the continuation stack, and sets the first basic element of the instruction to compute as the main program. Then, one or more rules will match the possible results of this execution, and perform the necessary operations for the control instruction. For instance, with the first rule of the “if” statement, the “if” continuation is put in the control stack and, depending on the result of e_1 , the control stack gives the evaluation of e_2 or e_3 with the rest of the stack. A communication primitive consists in simply adding a new value in the environment and in practice all rules of BSP primitives are merged into a generic one.

The **raise** and **try** rules behave a bit differently, and deserve an explanation. The first **raise** rule will compute the argument as usual, but the second rule will simply give an “error state”, with the given error type and the computed value. The error states then go through the control stack, ignoring everything but the **try** continuations. The **try** rules try to match such an error state with the exception being caught. If it is another exception, the try will simply be removed from the stack; however, if it is the matching exception, the error treatment will be started, with the variable taking the value given in the error state.

(c) Global Rules

As in the big-step semantics, global rules are mainly used to call local ones and **p** configurations have to be reduced. Figure 5.6 gives those rules.

First, the global reduction calls a local one. This represents a reduction by a single processor, which thus introduces an interleaving of computations. Communications and BSP synchronisation are done

$$\begin{aligned}
& s, \text{bsp_send } e \ x \ . \ \kappa \xrightarrow{i} s, e \ . \ \text{bsp_send } _ \ x \ . \ \kappa \\
& s, to \ . \ \text{bsp_send } e \ x \ . \ \kappa \xrightarrow{i} s', \kappa \quad \text{if } 0 \leq to < \mathbf{p} \text{ and } s' = s.C^{\text{send}} \oplus \{to, x\} \\
& s, \text{bsp_put } e \ x \ y \ . \ \kappa \xrightarrow{i} s, e \ . \ \text{bsp_put } _ \ x \ y \ . \ \kappa \\
& s, to \ . \ \text{bsp_put } _ \ x \ y \ . \ \kappa \xrightarrow{i} s', \kappa \quad \text{if } 0 \leq to < \mathbf{p} \text{ and } s' = s.C^{\text{put}} \oplus \{to, x, y\} \\
& s, \text{bsp_get } e \ x \ y \ . \ \kappa \xrightarrow{i} s, e \ . \ \text{bsp_get } _ \ x \ y \ . \ \kappa \\
& s, to \ . \ \text{bsp_get } _ \ x \ y \ . \ \kappa \xrightarrow{i} s', \kappa \quad \text{if } 0 \leq to < \mathbf{p} \text{ and } s' = s.C^{\text{get}} \oplus \{to, x, y\} \\
& s, \text{bsp_push } x \ . \ \kappa \xrightarrow{i} s', \kappa \quad \text{if } s' = s.C^{\text{push}} \oplus \{x\} \\
& s, \text{bsp_pop } x \ . \ \kappa \xrightarrow{i} s', \kappa \quad \text{if } s' = s.C^{\text{pop}} \oplus \{x\}
\end{aligned}$$

Figure 5.5. Small-step semantics: local BSP operations.

$$\begin{aligned}
& \frac{s_i, e_i \ . \ \kappa_i \xrightarrow{i} s'_i, e'_i \ . \ \kappa'_i}{\langle (s_0, e_0 \ . \ \kappa_0), \dots, (s_i, e_i \ . \ \kappa_i), \dots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1} \ . \ \kappa_{\mathbf{p}-1}) \rangle \rightarrow \langle (s_0, e_0 \ . \ \kappa_0), \dots, (s'_i, e'_i \ . \ \kappa'_i), \dots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle} \\
& \langle (s_0, \text{bsp_sync} \ . \ \kappa_0), \dots, (s_{\mathbf{p}-1}, \text{bsp_sync} \ . \ \kappa_{\mathbf{p}-1}) \rangle \rightarrow \mathbf{AllComm}\{\langle (s_0, \kappa_0), \dots, (s_{\mathbf{p}-1}, \kappa_{\mathbf{p}-1}) \rangle\}
\end{aligned}$$

Figure 5.6. Small-step semantics: global reductions.

with the second rule: each processor is in the case of a **bsp_sync** with its control stack. The **AllComm** function computes the communications, and returns the new environments. Then what remains to be executed is only the control stacks since the synchronisation has been performed — with communications. Finally, the whole evaluation \rightarrow^* is inductively define as follow:

$$\frac{s_1, e_1 \rightarrow s_2, e_2 \quad s_2, e_2 \rightarrow^* s_3, e_3}{s_1, e_1 \rightarrow^* s_3, e_3} \quad \frac{}{s, e \rightarrow^* s, e}$$

where (s, e) represents globally the \mathbf{p} local configurations.

5.2.2 Co-inductive Semantics

Co-inductive semantics are much easier to define with the small-step semantics. A program runs indefinitely if it has an infinite sequence of small-step reductions. Thus, the definition of \rightarrow_∞ is as follow:

$$\frac{s, e \rightarrow s', e' \quad s', e' \rightarrow_\infty}{s, e \rightarrow_\infty}$$


5.2.3 Equivalence Between the Semantics

The small-step semantics have less rules than the big-step ones. But finding them is much harder and it is thus less a formal specification of the language. We give now some results of equivalence.

Lemma 5

\rightarrow is confluent.

Proof. The small-step semantics verifies the diamond property: if from one state two steps are possible, they can only be local executions on two different processors. Since the executions are independent within a super-step, we can reach a common state by executing the other computation. The confluence follows

as a direct consequence. 

Lemma 6

\rightarrow^* and \Downarrow (resp. \rightarrow_∞ and \Downarrow_∞) are equivalent.

Proof. The proof has some common ground with classic big-step to small-step equivalences, with two main difficulties.

- The small-step semantics allows operations to execute on the different processors in any order, while the big-step semantics fixes the order.
- The continuations, coupled with the synchronisation, introduce the need for a notion of equivalence between a program and a continuation.

The first implication (big-step to small-step) is done by induction. However, an induction directly on the stated theorem would not be enough, and we need to generalize the result by defining a notion of equivalence between pairs (program, continuation). This is because after a synchronisation, in the big-step semantics we still have a program to execute, while in the small-step semantics it is a continuation. It is then a straightforward induction on the derivation of the big-step execution.

For the second implication, we rely on the previous lemma. Since the small-step semantics is confluent, we can order the local executions in any order, in particular the order chosen with the big-step semantics (execution on the first processor first until the synchronisation barrier, then the second, *etc.*) The proof is then done by induction on the derivation.



Lemma 7

\vdash and \vdash_∞ are mutually exclusive.

Proof. By induction and co-induction.



5.2.4 Adding the Subgroup Synchronisation

Small-step semantics with subgroup synchronisation follow the same ideas seen with the big-step semantics.

(a) Local Operations

As for the big-step semantics, the BSP operations take an additional argument, the communicator, and the rules are modified accordingly.

For instance, the rules for the *send* operation will be as follow:

$$\begin{aligned} & s, \mathbf{bsp_send} \ cmt \ e \ x \ . \ \kappa \xrightarrow{i} s, e \ . \ \mathbf{bsp_send} \ cmt \ _ \ x \ . \ \kappa \\ & s, to \ . \ \mathbf{bsp_send} \ cmt \ e \ x \ . \ \kappa \xrightarrow{i} s', \kappa \quad \text{if } 0 \leq to < \mathbf{p}_{cmt} \text{ and } s' = s.C^{\mathbf{send}}_{cmt} \oplus \{to, x\} \end{aligned}$$

(b) Global Reduction Rules

Unlike in the big-step semantics, defining the global reduction rules in the small-step semantics is relatively straightforward. The small-step semantics aims at providing elementary reductions, so there is no reason to provide a rule matching the big-step **AllComm**. Instead, the **Diamond** rule can be directly translated to the small-step language.

The global rules are now as follows:

$$\begin{aligned} & \frac{s_i, e_i \ . \ \kappa_i \xrightarrow{i} s'_i, e'_i \ . \ \kappa'_i}{\langle (s_0, e_0 \ . \ \kappa_0), \dots, (s_i, e_i \ . \ \kappa_i), \dots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1} \ . \ \kappa_{\mathbf{p}-1}) \rangle \rightarrow \langle (s_0, e_0 \ . \ \kappa_0), \dots, (s'_i, e'_i \ . \ \kappa'_i), \dots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle} \\ & \frac{\forall i \in C, O_i = \mathbf{bsp_sync} \ C \ . \ \kappa_i}{\langle (s_0, O_0), \dots, (s_{\mathbf{p}-1}, O_{\mathbf{p}-1}) \rangle \rightarrow \mathbf{AllCommC}\{C, \langle (s_0, O_0), \dots, (s_{\mathbf{p}-1}, O_{\mathbf{p}-1}) \rangle\}} \end{aligned}$$

In the second rule, O_i is used to represent an outcome of a local execution, and C a communicator. The rule, similar to the big-step **Diamond** rule, states that if all the processor inside a communicator reach a synchronisation state on this communicator, then the communications are done along the **AllCommC** function. This function is defined in almost the same way as for the big-step semantics:

- It accepts a second argument (a communicator).
- It only modifies the environments of the processors in the range of the communicator.
- Among the messages of these processors, it only considers the ones that were sent within the matching communicator.

With this definition of the small-step semantics, we can state the same lemma of equivalence with the big-step semantics.

Lemma 8

$\vdash \multimap^*$ and \Downarrow are equivalent.

Proof. The proof follows the same structure as the proof without sub-synchronisation. \square

5.3 Semantics in Coq

All of the semantics presented in this chapter were formally defined in the COQ proof assistant. The whole COQ development is available at <http://www.lacl.fr/fortin/dev-coq.tar.gz>. In this section, rather than giving the full code (which is several thousands of lines long), we will focus on key points of the development, explaining the main choices we took in the course of the modeling. We first present the semantics in COQ without the subgroup synchronisation and then show how to add this feature.

5.3.1 Mechanized Semantics in Coq

The COQ system (see [22] for a nice introduction to this theorem-proving system) is a proof assistant based on a logic which is a variant of type theory, following the “propositions-as-types, proofs-as-terms” paradigm, enriched with built-in support for inductive and co-inductive definitions of predicates and data types. From a user’s perspective, COQ offers a rich specification language to define problems and state theorems about them. This language includes (1) constructive logic with all the usual connectives and quantifiers; (2) inductive definitions *via* inference rules and axioms; (3) a pure functional programming language with structural recursion. Proofs are developed interactively using tactics that build incrementally the proof term behind the scene. These tactics range from the trivial (**intro**, which adds an abstraction to the proof term) to rather complex decision procedures (**omega** for Presburger arithmetic).

For example, let us define in COQ a toy language of numerical expressions $e := n \mid e_1 + e_2 \mid \frac{e_1}{e_2} \mid \text{inf}$ where “inf” is an infinite computation and $\frac{e_1}{e_2}$ is possible only if $e_2 \neq 0$ otherwise it is an infinite computation. More examples of semantics on simple languages (λ -calculus, small imperative language, abstract machine, etc.) can be found in [193]. Now let us define our expressions using an inductive:

```
Inductive expr:Set := num:Z → expr
  | plus:expr → expr → expr
  | div: expr → expr → expr
  | inf: expr
```

(integers are noted **Z**). Now, its natural (big-step) semantics *i.e.* abstract finite evaluation to integers would be defined by the following inductive:

```
Inductive eval: expr → Z → Prop :=
  eval_num: ∀n:Z, eval (num n) n
| eval_plus: ∀e1 e2 n1 n2, eval e1 n1 → eval e2 n2 → eval (plus e1 e2) (n1+n2)
| eval_div: ∀e1 e2 n1 n2, eval e1 n1 → eval e2 n2 → n2≠0 → eval (div e1 e2) (n1/n2)
```

and we noted (eval_expr e n) the evaluation of expression e to integer n . Now the determinism of the semantics is expressed with the following lemma and its proof:

Lemma eval_deterministic: $\forall e \ n1, \text{eval } e \ n1 \rightarrow \forall n2, \text{eval } e \ n2 \rightarrow n1=n2$.

Proof.

```
induction 1; intros.
  (* case Num *)
  inversion H; auto.
  (* Case Plus *)
  inversion H1; subst.
  generalize (IHeval1 n3 H4); intro.
  generalize (IHeval2 n4 H6); intro.
  auto.
  (* Case Div *)
  inversion H2; subst.
  generalize (IHeval1 n3 H5); intro.
```

generalize (IHeval2 n4 H6); **intro.**
auto.
Qed.

where we have performed a proof by induction on the evaluation of “e” and for each case, a proof by inversion² on the second evaluation.

The co-inductive for infinite computations would be defined as follows:

CoInductive eval_inf: expr → **Prop** :=
 evalinf_inf: eval_inf inf
 | evalinf_plus1: ∀e1 e2 n1, eval e1 n1 → evalinf e2 → eval_inf (plus e1 e2)
 | evalinf_plus2: ∀e1 e2 n2, evalinf e1 → eval e2 n2 → eval_inf (plus e1 e2)
 | evalinf_div1: ∀e1 e2 n1, eval e1 n1 → evalinf e2 → eval_inf (div e1 e2)
 | evalinf_div2: ∀e1 e2 n2, evalinf e1 → eval e2 n2 → n2 ≠ 0 → eval_inf (div e1 e2).
 | evalinf_div3: ∀e1 e2 n1 n2, eval e1 n1 → eval e2 n2 → n2 = 0 → eval_inf (div e1 e2)

that is “inf” is an infinite computation and if one sub-expression of $e_1 + e_2$ or $\frac{e_1}{e_2}$ is “inf” then it is an infinite computation. Now the fact that the semantics are mutually exclusive is expressed and proved as follows:

Lemma eval_eval_inf_exclusive: ∀a v, eval a v → eval_inf a → False.

Proof.

induction 1; **intros.**

(* Case Num *)

inversion H.

(* Case Plus *)

inversion H1; **auto.**

(* Case Div *)

inversion H2; **auto.**

(* case a/b with b=0 *)

subst.

generalize (eval_deterministic _ _ H0 _ H6); **intro.**

absurd (n2=0); **auto.**

Qed.

that is by induction on the evaluation of “a” and proving by case that it is not possible to have the same co-inductive evaluation. Note the use of the deterministic result of “eval” to prove that it is impossible to have both $n_2 = 0$ and $n_2 \neq 0$ to split an infinite evaluation to a finite one.

For a small-step semantics, we will first define a one-step reduction with a left-to-right order for sub-expressions:

Inductive one_step: expr → expr → **Prop** :=
 (* plus *)
 | plus_left: ∀ e1 e'1 e2, one_step e1 e'1 → one_step (plus e1 e2) (plus e'1 e2)
 | plus_right: ∀ n1 e2 e'2, one_step e2 e'2 → one_step (plus (num n1) e2) (plus (num n1) e'2)
 | plus_sum: ∀ n1 n2, one_step (plus (num n1) (num n2)) (num (n1+n2))
 (* div *)
 | div_left: ∀ e1 e'1 e2, one_step e1 e'1 → one_step (div e1 e2) (div e'1 e2)
 | div_right: ∀ n1 e2 e'2, one_step e2 e'2 → one_step (div (num n1) e2) (div (num n1) e'2)
 | div_div: ∀ n1 n2, n2 ≠ 0 → one_step (div (num n1) (num n2)) (num (n1/n2))

and then the transitive and reflexive closure of one_step or the infinite evaluation

Inductive step_star: expr → expr → **Prop** :=
 | sos_refl: ∀e, step_star e e
 | sos_trans: ∀e1 e2 e3, one_step e1 e2
 → step_star e2 e3 → step_star e1 e3

CoInductive step_inf: expr → **Prop** :=
 | sos_inf: ∀a b, one_step a b → step_inf b → step_inf a.

The small-step semantics *i.e.* abstract evaluation of expression e to integer n is noted (step_star e (num n)). Now we can have the following results. First the equivalence between small-step semantics and big-step ones:

(* finite evaluations *)

Lemma step_star_to_eval: ∀a v, step_star a (num v) ↔ eval a v.

(* infinite evaluations *)

Lemma step_int_to_eval_int: ∀a, step_inf a ↔ eval_inf a.

²That is, we derive for each possible constructor of an evaluation, all the necessary conditions that should hold for the evaluation to be proved by the constructor.

Now that the small-step semantics is deterministic (very easy to prove using above lemmas):

Lemma `step_star_determinist`: $\forall e, n1, \text{step_star } e \text{ (num } n1) \rightarrow \forall n2, \text{step_star } e \text{ (num } n2) \rightarrow n1 = n2$.

And to finish, that also for small-step semantics, finite and infinite reductions are mutually exclusive:

Definition `notred` (a : `expr`) : **Prop** := $\forall b, \sim(\text{one_step } a \ b)$.

Lemma `step_star_or_step_inf`: $\forall a, (\text{exists } b, \text{step_star } a \ b \wedge \text{notred } b) \vee \text{step_inf } a$

5.3.2 Memory Model

In BSP-WHY, all variables exchanged contain data of the generic type `value`, which can represent any elementary type. We define a corresponding type in COQ:

Parameter `value` : **Type**.

A few special values are also defined, such as `null`, `void`, `true`, `false`, *etc.* The memory is then defined as a function from memory blocks to values. The blocks are numbered according to numbers:

(* Type of memory blocks *)

Definition `mblock` : **Type** := **Z**.

Parameter `null_mblock` : `mblock`.

(* Represents the memory *)

Definition `mem_t` := `mblock` \rightarrow `value`.

Thus the initial empty memory is the function that always returns the null value:

Definition `empty_mem` : `mem_t` := **fun** `b` \Rightarrow `null_value`.

It is then easy to define a memory modification:

Definition `update_mem` (`m` : `mem_t`) (`b` : `mblock`) (`v` : `value`) : `mem_t` :=
fun (`x` : `mblock`) \Rightarrow **if** (`Z_eq_dec` `x` `b`) **then** `v` **else** (`m` `x`).

We then need to define variables. For this, we represent identifiers as positive numbers:

Definition `ident` := `positive`.

The link between a variable and its memory block is then stored as a part of the execution environment, which we will detail more in the next section:

Definition `envE_t` := `ident` \rightarrow `mblock`.

5.3.3 Environment of Execution

We first define the environment for a single processor. The environment is composed of 8 parts, which are regrouped into a record type.

(* Record for the environment on a fixed pid *)

Record `env` : **Type** := `mkenv` { `envE` : `envE_t`; `envR` : `envR_t`; `envCs` : `envCs_t`; `envCp` : `envCp_t`;
`envCg` : `envCg_t`; `envCpu` : `envCpu_t`; `envCpo` : `envCpo_t`; `mem` : `mem_t` }

where:

- The first component, `envE`, was described in the previous section. It is the part of the environment that describes the variables in the program, and give their associated memory block.
- The last component, `mem`, is the memory itself, which was also described in the previous section.
- The remaining components of the record all deal with the parallel communication model of BSP-WHY. `envR` is used to store the values that were received during the previous super-step, and that can be accessed through the `bsp_findmsg` function. It is defined as a function, which takes in argument the `pid` and the number of the message, and returns its associated value.

Definition `envR_t` := `pid` \rightarrow `positive` \rightarrow `value`.

- Finally, `envCs`, `envCp`, `envCg`, `envCpu` and `envCpo` all share the same role and structure. They are lists keeping track of the parallel messages that will be sent during the next synchronisation. One list is used per type of message: *send*, *put*, *get*, *push*, and *pop*:

Definition `envCs_t` := `list msg_send` **and** `envCp_t` := `list msg_put` **and** `envCg_t` := `list msg_get`
and `envCpu_t` := `list msg_push` **and** `envCpo_t` := `list msg_pop`

The types for representing the different kind of messages are themselves record type. Here is for instance the definition of the *send* messages:

Record msg_send : **Type** := mkmsg_send { ms_dest : pid; ms_value : value }

Of course, BSP programs run on parallel machines, so we have to define the parallel environment. As usual, we do this by the use of a function from the *pid* to the local environments.

Definition par_env := pid → env.

5.3.4 Semantics Rules

It is now possible to define our semantics in COQ, according to the rules given in previous sections. For the big-step operational semantics, there are two parts in the semantics. First, we will define the local reduction rules, which represent the evaluation on a single processor, and then we will give the parallel reduction rules.

(a) Big-step Semantics

Local Reduction Rules. As is usual in COQ, the semantics rules are given as an inductive predicate. `eval_expr i e a e'` defines the evaluation of the expression `a` in the environment `e` on the processor `i`.

Variable `i`: pid.

Inductive eval_expr: env → expr → env → outcome → **Prop**

The definition of an outcome directly follows the definition given in Section 5.1: there are three possible outcomes, either the computation returns a value, raises an exception, or requests a synchronisation, with another expression remaining to be executed.

Inductive outcome :=

| Outval : value → outcome
 | Outexn : exn → value → outcome
 | Outsync : ident → expr → outcome (* first argument is the type of synchronization *)

The definition of the `eval_expr` predicate closely matches the mathematical definition of the semantics given in Section 5.1. We thus have several rules for each language instruction, depending on the kind of outcome obtained during the evaluation of the sub-expressions. There is typically one rule for a value outcome, one rule for an exception outcome, one rule for a synchronisation outcome, *etc.*. The complete rules are naturally available in the COQ development. Let us focus of a few representative rules in the reduction.

The evaluation of a term is the most basic rule. It never returns an exception or a synchronisation request, and does not change the environment, thus it is the simplest rule:

| eval_Eterm : ∀(e:env) (t:term), eval_expr e (Eterm t) e (Outval (evalterm i t e))

We can see that the rule depends on a simple call to the evaluation of the term with the function `evalterm`. This matches the mathematical rule given before.

For most language instructions, we have at least three rules, as explained before. In the COQ development, we distinguish the rules for when an exception occurs with a `_e` suffix in its name, and `_s` when a synchronisation happens. For instance, here are the three rules for the `let ... in` structure:

| eval_Elet : ∀x a1 a2 e e' e'' v o,
 eval_expr e a1 e' (Outval v) →
 eval_expr (update e' x v) a2 e'' o →
 eval_expr e (Elet x a1 a2) e'' o
 | eval_Elet_e : ∀x a1 a2 e e' v ex,
 eval_expr e a1 e' (Outexn ex v) →
 eval_expr e (Elet x a1 a2) e' (Outexn ex v)
 | eval_Elet_s : ∀x a1 a2 a' sy e e',
 eval_expr e a1 e' (Outsync sy a') →
 eval_expr e (Elet x a1 a2) e' (Outsync sy (Elet x a' a2))

Once again, these rules closely match the mathematical rules given before. The rule for the parameter application is a special case.

| eval_Epapp : ∀e p e' a e'' v v',
 eval_expr e a e' (Outval v) →
 eval_param e' p v e'' v' →
 eval_expr e (Epapp p a) e'' (Outval v')

A parameter does not strictly have a code in WHY (or BSP-WHY). Thus, it is impossible to give a semantics execution of it with only the parameter specification. For this reason, we write the semantics assuming that we have a predicate, **eval_param**, that tells us how the parameters are evaluated. We can then say that a program is correct if its execution is correct for any such **eval_param** matching the parameters specification. Lastly, the rule for the synchronisation is as follow:

| eval_Esync : $\forall e, \text{eval_expr } e \text{ (Esync) } e \text{ (Outsync sync Evoid)}$

that is it does not modify the environment, simply returning a request for synchronisation.

Parallel Rules. We had two rules in the mathematical definition of the semantics, so we have two matching rules in our COQ development.

(* Parallel evaluation of expressions. *)

Inductive eval_par: par_env \rightarrow par_expr \rightarrow par_env \rightarrow par_outcome \rightarrow **Prop** :=

| evalpar_nosync : $\forall pe \text{ pa } pe' \text{ po}, \text{final_outcome po} \rightarrow$
 $\forall i:\text{pid}, \text{eval_expr } i \text{ (pe } i) \text{ (pa } i) \text{ (pe' } i) \text{ (po } i) \rightarrow$
 $\text{eval_par } pe \text{ pa } pe' \text{ po}$
| evalpar_sync : $\forall pe \text{ pa } pe' \text{ po } po' \text{ pec } pe'',$
 $\forall i:\text{pid}, \text{eval_expr } i \text{ (pe } i) \text{ (pa } i) \text{ (pe' } i) \text{ (po } i) \rightarrow$
 $\text{sync_outcome po} \rightarrow \text{pec} = \text{comm(pe)} \rightarrow$
 $\text{eval_par } pec \text{ pa } pe'' \text{ po'} \rightarrow$
 $\text{eval_par } pe \text{ pa } pe'' \text{ po'}$

The **final_outcome** and **sync_outcome** are predicates describing outcomes where all processors reach a value, or a synchronisation request, respectively.

(b) Small-Step Semantics

The small-step semantics use the same environments and memory model as the big-step semantics.

Continuations. However, the semantics are significantly different. First of all, since we chose to use continuation semantics, we need to define the continuations. For each statement in the language, zero, one, or several continuations will be defined, depending on the number of computations that are done sequentially in the statement. For instance, labels and asserts will not have any continuation, but a “if” statement has one. The continuation stack is defined inductively, in a similar fashion as a list:

Inductive cont : **Type** :=

| Kempty : cont
| Kif : expr \rightarrow expr \rightarrow cont \rightarrow cont
| Klet : ident \rightarrow expr \rightarrow cont \rightarrow cont
| Kassign : ident \rightarrow cont \rightarrow cont
| Kraise : exn \rightarrow cont \rightarrow cont
| Ktry : exn \rightarrow ident \rightarrow expr \rightarrow cont \rightarrow cont

Kempty is the empty continuation, and every other continuation is linked to a previous continuation.

Local States. A major difference with the big-step semantics is the notion of program execution “state”. For the natural semantics, a state was simply the association of a program to execute and an environment. In the small-step semantics however, we define four kinds of states.

1. A “normal” state is similar to the notion of state in the natural semantics.
2. Result states are the values returned when a computation is finished.
3. An Error state is characterized by an error type, and a parameter value.
4. Finally, the synchronisation state is the result of a call to a synchronising parameter.

States are thus naturally defined as an inductive type, with four constructors.

Inductive state : **Type** :=

| State (a : expr) (e : env) (k : cont) : state
| ResState (v : value) (e : env) (k : cont) : state
| ErrState (ex : exn) (v : value) (e : env) (k : cont) : state
| SyncState (sp : pident) (e : env) (k : cont) : state.

There is always a continuation in a state, but it can be the empty continuation.

Local Steps. A step in the semantics is defined as an inductive from states to states. The definition closely matches the rules that were given in Fig. 5.4. As an illustration, here are the two rules for the “let” construction:

Inductive $\text{step} : \text{state} \rightarrow \text{state} \rightarrow \mathbf{Prop} :=$

```
...
| step_let : ∀ k x a1 a2, step (State (Elet x a1 a2) e k)
              (State a1 e (Klet x a2 k))

| step_let1 : ∀ e k x a2 v, step (ResState v e (Klet x a2 k))
              (State a2 (update e x v) k)
...
```

Parallel Steps. The parallel reduction is defined on parallel states (that is, a function from the pid to the local states, and a global environment):

Inductive $\text{pstep} : \text{pstate} \rightarrow \text{pstate} \rightarrow \mathbf{Prop} :=$

```
| pstep_lstep : ∀ i s s' ge, step i (s i) (s' i) →
                ∀ j, j ≠ i → (s j) = (s' j) →
                pstep (PState s ge) (PState s' ge)
| pstep_sync : ∀ s ge ge' k e e' sp, all_sync s sp e k →
                comm2 sp e ge e' ge' →
                pstep (PState s ge)
                    (PState (fun i ⇒ (ResState void (e' i) (k i))) ge').
```

From there, the transitive closure pstar is defined in a standard manner.

5.3.5 Adding the Subgroup Synchronisation

In order to allow the subgroup synchronisation, it is necessary to make some changes to the definitions of the parallel model.

First, a subgroup is defined as a function from the processor identifiers to the booleans.

Definition $\text{subgroup} := \text{pid} \rightarrow \text{bool}$.

As we explained in in Section 3.3, a communicator can not be simply seen as a subgroup. Instead, we define a communicator as a unique positive number, as we did for other identifiers.

Definition $\text{comm} := \text{positive}$.

The next change concerns the communication environments. We saw that a message is now sent within the context of a communicator, and it needs to be reflected in the definition of the environment.

This is simply done by modifying the envC definition: it is now a function from the communicators to the previous envC definition.

```
Record env : Type := mkenv {
  envE : envE__t;
  envR : envR__t;
  envC : comm → envC__t;
  mem : mem__t
}.
```

Finally, we modify the definition of a BSP-WHY expression, to

Inductive $\text{expr} : \mathbf{Type} :=$

```
...
| Esend : comm → expr → expr → expr
| Eput : comm → expr → ident → ident → expr
| Eget : comm → expr → ident → ident → expr
| Epush : comm → ident → expr
| Epop : comm → ident → expr
| Esync : comm → expr
...
```

The only difference here is that every BSP primitive takes an additional argument, the communicator.

(a) Big-step Semantics with Subgroup Synchronisation

With these modifications, we can now change the semantics rules. The send rule changes as follows:

Inductive $\text{pstep} : \text{pstate} \rightarrow \text{pstate} \rightarrow \mathbf{Prop} :=$
 $| \text{pstep_lstep} : \forall i \ s \ s' \ \text{ge}, \text{step } i \ (s \ i) \ (s' \ i) \rightarrow$
 $\quad \forall j, j \neq i \rightarrow (s \ j) = (s' \ j) \rightarrow$
 $\quad \text{pstep} (\text{PState } s \ \text{ge}) (\text{PState } s' \ \text{ge})$
 $| \text{pstep_sync} : \forall \text{cm } s \ \text{ge} \ \text{ge}' \ k \ e \ e' \ \text{sp}, \text{all_sync_cm } \text{cm } s \ \text{sp } e \ k \rightarrow$
 $\quad \text{comm2_cm } \text{cm } \text{sp } e \ \text{ge} \ e' \ \text{ge}' \rightarrow$
 $\quad \text{pstep} (\text{PState } s \ \text{ge})$
 $\quad (\text{mkpstate_cm } \text{cm } e \ e' \ s \ k) \ \text{ge}'$.

5.4 Proof of the Translation

Our BSP-WHY tool transforms a parallel program, written in BSP-WHY, into a sequential WHY program. We explained that the WHY program simulates the execution of the parallel program, thus allowing to reason on the sequential program, with the existing tools and provers, instead of the parallel program.

However, doubts could be raised on the validity of the BSP-WHY transformation. In this section, we alleviate those doubts by proving formally that BSP-WHY does, indeed, properly simulates the parallel program. Our main result will be as follow: For a given parallel program P , if the transformation of P by BSP-WHY is *correct*, then P itself is *correct*. The proof of this theorem is detailed in this section.

5.4.1 Program Correctness

First, we need to formally define the notion of program *correctness*.

(a) An Unsatisfactory Solution: the Hoare Correctness

A WHY (or BSP-WHY) function (a program) is defined with a pre-condition and post-condition. As such, it can be seen as a Hoare triple, as defined in Section 1.2. With the standard Hoare logic, the meaning of a triple $\{p\} e \{q\}$ is: if e is executed in a state satisfying its precondition p , then if it terminates, the resulting state satisfies its postcondition q . In [110], we gave a mathematical proof of the correctness of BSP-WHY, based on this definition. However, there are several limitations with this approach:

- The first drawback is that with the standard Hoare logic, only *partial* correctness is proved. Termination must be ensured separately. It is however possible to extend the Hoare logic so that it includes the termination, by adding a variant in the *while* rule.
- Such a definition of the correctness only ensures the validity of the global Hoare triple defined by a function. If there are internal assertions in the definition of a BSP-WHY program, their correctness is not guaranteed by such a proof.
- Perhaps most importantly, in our proof based on the Hoare correctness, we had to assume that the BSP-WHY program was structurally sound, in regard to the synchronisations, for the transformation to be valid. The reason is linked to the previous point. In the BSP-WHY transformation, in order to ensure that there will be no deadlocks, an internal assertion is added whenever an instruction might be problematic. The assertion then generates a proof obligation, which is ensured to be correct by running the WHY tool on the sequential program. However, if we only assume that the sequential program is correct with the standard definition of a Hoare triple, then nothing guarantees that the assertion will be true.

The heart of the problem here is that the standard Hoare correctness does not give us all the information that we get with a WHY proof of the sequential program. WHY ensures both the total correctness, and the fact that every internal assertion will hold true. A definition of the correctness that includes all the information is needed to get a satisfactory proof of the BSP-WHY transformation.

(b) Correctness with Blocking Semantics

We achieve a more satisfactory definition of the correctness with the use of refined semantics for the BSP-WHY and WHY languages. Starting from the operational semantics that we defined in Section 5.1, we modify the rules so that in addition of describing the execution of a program, the semantics checks for the validity of the logic assertions that are present inside the program. In Fig. 5.7, we give the big-step blocking semantics for WHY. Only three rules are changed compared to the basic big-step semantics. They are the rules with a logical assertion:

$$\begin{array}{c}
\frac{}{s, c \Downarrow s, c} \quad \frac{s, e_1 \Downarrow s', v \quad s'[x \leftarrow v], e_2 \Downarrow s'', o}{s, \text{let } x = e_1 \text{ in } e_2 \Downarrow s'', o} \quad \frac{s, e_1 \Downarrow s', v \quad s'[x \leftarrow v], e_2 \Downarrow s'', o}{s, \text{let } x = \text{ref } e_1 \text{ in } e_2 \Downarrow s'', o} \\
\\
\frac{s, e \Downarrow s', v}{s, x := e \Downarrow s'[x \leftarrow v], \text{void}} \quad \frac{s, e_1 \Downarrow s', \text{true} \quad s', e_2 \Downarrow s'', o}{s, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow s'', o} \quad \frac{s, e_1 \Downarrow s', \text{false} \quad s', e_3 \Downarrow s'', o}{s, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow s'', o} \\
\\
\frac{s \models i \quad s, e; \text{loop } e \{ \text{invariant } i \text{ variant } v \} \Downarrow s', o}{s, \text{loop } e \{ \text{invariant } i \text{ variant } v \} \Downarrow s', o} \quad \frac{s, e \Downarrow s', v}{s, \text{raise } (E \ e) \Downarrow s', E(v)} \\
\\
\frac{s, e_1 \Downarrow s', v}{s, \text{try } e_1 \text{ with } E \ x \rightarrow e_2 \text{ end } \Downarrow s', v} \quad \frac{s, e_1 \Downarrow s', E(v) \quad s'[x \leftarrow v], e_2 \Downarrow s'', o}{s, \text{try } e_1 \text{ with } E \ x \rightarrow e_2 \text{ end } \Downarrow s'', o} \\
\\
\frac{s \models p \quad s, e \Downarrow s', o}{s, \text{assert } \{p\}; e \Downarrow s', o} \quad \frac{s, e \Downarrow s', o \quad s' \models p}{s, e \{p\} \Downarrow s', o} \quad \frac{s, e \Downarrow s', o}{s, l : e \Downarrow s', o}
\end{array}$$

Figure 5.7. Big-step Blocking Semantics for WHY.

$$\begin{aligned}
s \models \forall x : \beta, p & \text{ iff } \forall v : \beta, s \oplus \{x = v\} \models p \\
s \models p \wedge p' & \text{ iff } s \models p \text{ and } s \models p' \\
s \models p \vee p' & \text{ iff } s \models p \text{ or } s \models p' \\
& \dots
\end{aligned}$$

Figure 5.8. Semantics of the WHY Logic Language (Predicates).

$$\begin{aligned}
\mathcal{V}(s, v) &= v \\
\mathcal{V}(s, x) &= s.\mathcal{E}(x) \\
\mathcal{V}(s, !x) &= s.\mathcal{E}(x) \\
\mathcal{V}(s, f(t)) &= [[f]](\mathcal{V}(s, t)) \\
&\dots
\end{aligned}$$

Figure 5.9. Semantics of the WHY Logic Language (Terms).

- For the assert rule, a premise is added to the inference rule, stating that the assertion is valid in the environment *before* the execution.
- For the post-condition rule, a premise is added to the inference rule, stating that the assertion is valid in the environment *after* the execution.
- For the loop rule, a premise is added to the inference rule, stating that the invariant is valid in the environment *before* the execution of an iteration.

The semantics does not check if the variant decreases. Instead, the termination will be as before an immediate consequence of any derivation $s, e \Downarrow s', r$. The rules for the BSP-WHY semantics are modified in a similar fashion. Only the local control flow rules are modified; the local BSP operations and global rules do not include logical assertions. Of course, we also need to define formally the $s \models p$ relation. The BSP-WHY logic language was defined in Section 3.1, as an extension of the WHY logic language. Logic formulas are defined from *predicates* and *logic terms*. The rules for the predicates are given in Fig. 5.8. Logic terms are evaluated according to the environment, as described in Fig. 5.9, where the evaluation of the term t in the state s is written $\mathcal{V}(s, t)$. We remind that the notation $s.\mathcal{E}(x)$ is used to denote the value of x in the state s .

Definition 1 (A correct function).

A function f (for WHY or BSP-WHY), defined syntactically by a pre-condition p , a body b and a post-condition q , is said to be correct if and only if for any state s in which p holds, then there exists a derivation according to the blocking semantics, towards a state s' , in which q holds, and with a result r :

$$(f \equiv (p, b, q) \text{ is correct}) \Leftrightarrow (\forall s, s \models p \Rightarrow \exists s', r \text{ such as } (s, b) \Downarrow (s', r) \wedge s' \models q)$$

5.4.2 Equivalence Between Elements of the Semantics**(a) Equivalence Between Environments**

We define an equivalence relation between parallel environments and sequential environments. It corresponds to the model that was chosen in BSP-WHY to simulate a parallel execution with a sequential program.

Definition 2.

We say that a parallel environment s and a sequential environment s' are equivalent, and we write $s \sim s'$, if and only if:

- For every variable x defined in one of the \mathcal{E} of s , a variable x is defined in $s'.\mathcal{E}$, which is a \mathbf{p} -array of the values of x on the different processors. If x is not defined in one of the $s_i.\mathcal{E}$, then the i -th component of the \mathbf{p} -array is undefined.
- The messages found in the \mathcal{C}^{send} , \mathcal{C}^{put} , \mathcal{C}^{get} , \mathcal{C}^{push} and \mathcal{C}^{pop} parts of s correspond to the messages stored in the WHY variables `send_queue`, `put_queue`, `get_queue`, `push_queue` and `pop_queue` in s' (as described in Section 3.1).
- The received values of the different $s_i.\mathcal{R}$ matches the values stored in the `envR` variable of s' (also described in Section 3.1).

Such a definition allows us to make links between the state of a parallel environment, and a sequential one. In particular, we can formulate a lemma concerning the evaluation of terms (which only depends on the environment):

Lemma 9

If s and s' are equivalent, then the result of the evaluation of a BSP-WHY term t in s on the processor i is the same as the result of the evaluation of the WHY term $[[t]]_i$ in s' :

$$s \sim s' \Rightarrow eval(s, t, i) = evalwhy(s', [[t]]_i)$$

Proof. By induction on the term t .

- If t is a constant, its translation is the same constant.
- If t is a variable x , its translation is the access to the \mathbf{p} -array variable x in WHY. By definition of the equivalence $s \sim s'$, we have the equality.
- If t is a call to the `bsp_pid` function, its translation is simply the variable i , which gives the same value in both cases.
- The other cases do not raise more difficulty.

□

In addition to the equivalence between environments, we also define the (much simpler) equivalence between outcomes.

Definition 3.

We say that a parallel outcome o and a sequential outcome o' are equivalent, and we write $o \sim o'$, if and only if:

- all the local outcomes $o(0) \dots o(\mathbf{p} - 1)$ are values, and o' is the \mathbf{p} -array of those values;
- or all the local outcomes $o(0) \dots o(\mathbf{p} - 1)$ are exceptions, and o' is the corresponding exception, with argument the \mathbf{p} -array of the arguments in o .

Also, with this definition of equivalence, our main lemma can be written as follows:

Lemma 10

Let a be a BSP-WHY expression, and $a' = [[a]]$ its transformation into WHY. If from a state s'_1 , a' executes to a state s'_2 with an outcome o' , and if s'_1 is equivalent to a parallel state s_1 , then a will execute to a state s_2 and an outcome o , with s_2 equivalent to s_1 and o equivalent to o' :

$$s_1 \sim s'_1 \Rightarrow s'_1, [[a]] \Downarrow s'_2, o' \Rightarrow \exists s_2, o \text{ such as } (s_1, a \Downarrow s_2, o) \wedge (s_2 \sim s'_2) \wedge (o \sim o')$$

Proof. Detailed in the next pages. □

(b) Equivalence Between Semantics Rules

As we described in Section 3.1, the BSP-WHY transformation is a complex process, with several distinct stages. Attempting to prove the whole transformation in one big induction would most likely be unsuccessful. A better approach consists in splitting the transformation, and proving that we maintain an equivalence for the different stages. In the course of the BSP-WHY transformation, the program is part of several data-types:

- At the beginning, we have a parallel program in the BSP-WHY language.
- It is then transformed by the *block decomposition*, where the biggest sequential blocks of codes are isolated. The result of this step is a *block tree*, which corresponds to a parallel program control flow tree.
- The leaves of the block tree are *blocks*, a purely sequential code, without synchronisation.
- Finally, a sequential WHY program is generated.

In order to show that the equivalence is maintained during the transformation, we need to give a semantics for all of these data-types. We already defined the blocking semantics for BSP-WHY parallel programs, as well as for WHY sequential code. The semantics of a *block* of sequential code can be seen as a subset of the \Downarrow^i semantics: apart from the fact that there are no synchronisation (and thus the result is either a value, or an exception), it is identical. The only thing missing is thus to give a semantics for the execution of a *block tree*. We will start by defining such a semantics.

Block Semantics. The *Block Semantics* (\Downarrow^b) (not to be confused with *blocking* semantics) describes a theoretical execution of a parallel program, when it is in the block tree form. The block tree function is to extract the essential parallel structure of a program, while putting aside (in the so-called *blocks*) the sequential computations. As we explained in Section 3.1, we require that all processors always follow the same control flow inside the block tree. If a processor enters a branch of a *if* statement that affects the parallelism (for instance because there is a synchronisation inside), then all the other processors must do the same. Because of this, the block semantics does not allow processors to advance independently in the execution of the program. Instead, it requires a simultaneous progression in the control flow for all processors. The full rules for the block semantics are given in Fig. 5.10.

Let us explain more in detail the key rules:

- The first rule applies to the leaves of the tree, the blocks. In this case, the execution consists simply in executing the block of code on all processors simultaneously. Their execution is independant, and can not lead to a synchronisation.
- The **bsp_sync** rule is significantly different from what we have seen before. Here, it simply has to do all the communications to end the super-step, then there is nothing more to do.
- For a generic rule, such as the instruction **let** $x = t_1$ **in** t_2 (where t_1 and t_2 are both block trees), t_1 is first executed, returning a parallel outcome. If this outcome is made of p values (the vector \vec{v} , then we continue with the execution of t_2 , after updating the environment s so that for any processor i , x is now the i -th component of the parallel outcome returned by t_1 .
- In the semantics, $E(\vec{v})$ is used to denote a parallel outcome where all the individual outcomes are the exception E , with a value v_i .
- In the **if** rule, we enforce that all processors must always choose the same branch of a conditional instruction on the parallel level. For this reason, the parallel outcome must be either the vector with all **true**, or all **false**.
- Finally, the semantics is also blocking in the **loop**, **assert** and **post** rules. The assertions must be true on every processor for the rule to be applied.

$$\begin{array}{c}
\frac{s_1, B \Downarrow^i s'_1, r_1 \quad \dots \quad s_p, B \Downarrow^i s'_p, r_p}{s, B \Downarrow^b s', r} \quad \frac{}{s, \text{bsp_sync} \Downarrow^b \text{AllComm}(s), \text{void}} \\
\\
\frac{s, t_1 \Downarrow^b s', \vec{v} \quad s'[x \leftarrow v], t_2 \Downarrow^b s'', o}{s, \text{let } x = t_1 \text{ in } t_2 \Downarrow^b s'', o} \quad \frac{s, t_1 \Downarrow^b s', E(\vec{v})}{s, \text{let } x = t_1 \text{ in } t_2 \Downarrow^b s', E(\vec{v})} \\
\\
\frac{s, t \Downarrow^b s', \vec{v}}{s, x := t \Downarrow^b s'[x \leftarrow v], \text{void}} \quad \frac{s, t \Downarrow^b s', E(\vec{v})}{s, x := t \Downarrow^b s', E(\vec{v})} \\
\\
\frac{s, t_1 \Downarrow^b s', \vec{true} \quad s', t_2 \Downarrow^b s'', o}{s, \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow^b s'', o} \quad \frac{s, t_1 \Downarrow^b s', \vec{false} \quad s', t_3 \Downarrow^b s'', o}{s, \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow^b s'', o} \\
\\
\frac{s, t_1 \Downarrow^b s', E(\vec{v})}{s, \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow^b s', E(\vec{v})} \\
\\
\frac{\forall k, s_k \models i \quad s, t; \text{loop } t \{ \text{invariant } i \text{ variant } v \} \Downarrow^b s', o}{s, \text{loop } t \{ \text{invariant } i \text{ variant } v \} \Downarrow^b s', o} \\
\\
\frac{s, t \Downarrow^b s', \vec{v}}{s, \text{raise } (E t) \Downarrow^b s', E(\vec{v})} \quad \frac{s, t \Downarrow^b s', E'(\vec{v})}{s, \text{raise } (E t) \Downarrow^b s', E'(\vec{v})} \\
\\
\frac{s, t_1 \Downarrow^b s', \vec{v}}{s, \text{try } t_1 \text{ with } E x \rightarrow t_2 \text{ end} \Downarrow^b s', \vec{v}} \quad \frac{s, t_1 \Downarrow^b s', E(\vec{v}) \quad s'[x \leftarrow v], t_2 \Downarrow^b s'', o}{s, \text{try } t_1 \text{ with } E x \rightarrow t_2 \text{ end} \Downarrow^b s'', o} \\
\\
\frac{\forall i, s_i \models p \quad s, t \Downarrow^b s', o}{s, \text{assert } \{p\}; t \Downarrow^b s', o} \quad \frac{s, t \Downarrow^b s', o \quad \forall i, s'_i \models p}{s, t \{p\} \Downarrow^b s', o} \quad \frac{s, t \Downarrow^b s', o}{s, l:t \Downarrow^b s', o}
\end{array}$$


Figure 5.10. Big-step block semantics.

Logic Translation. We need to ensure that the BSP-WHY transformation keeps the meaning of a logical assertion.

Lemma 11

If e and e' are equivalent, then p holds in e if and only if the transformation of p holds in e' :

$$e \sim e' \Rightarrow (e' \models [[p]] \Leftrightarrow e \models p)$$

Proof. By induction on the predicate p . 

Translation of a Sequential Block. Now that we have semantics for all steps in the transformation, we can focus on proving the different steps independently.

Lemma 12

Let b be a BSP-WHY sequential block, and i a processor. Let $b'_i = [[b]]_i$ the transformation of b into WHY. If from a state s'_1 , b'_i executes to a state s'_2 with an outcome o' , and if s'_1 is equivalent to a parallel state s_1 , then the execution of b on a processor i will lead to a state s_2 and an outcome o , with s_2 equivalent to s_1 and o equivalent to o' :

$$s_1 \sim s'_1 \Rightarrow s'_1, [[b]]_i \Downarrow s'_2, o' \Rightarrow \exists s_2 \text{ } o \text{ such as } (s_1, b \Downarrow^i s_2, o) \wedge (s_2 \sim s'_2) \wedge (o \sim o')$$

Proof. By induction of the sequential code b . It does not contain parallel code, so synchronisation and communications are not a concern here. We will not detail all the induction cases here, but give the relevant ones.

- If b is a term, Lemma 9 ensures that the outcome is the same for both derivations.
- A generic induction case would be the **if-then-else** statement. Let us assume $b = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$. The translation is $[[b]]_i = \text{if } [[e_1]]_i \text{ then } [[e_2]]_i \text{ else } [[e_3]]_i$. Several semantics rules might have been applied in the WHY derivation, depending on the result of the execution of e_1 . For $s'_1, [[e_1]]_i \Downarrow s'_0, o'_0$, we have by induction s_0 and o_0 such as $s_0 \sim s'_0$ and $o_0 \sim o'_0$. In the hypothesis that $o'_0 = \text{true}$, the result of the execution of $[[b]]_i$ is obtained by executing $[[e_2]]_i$. By induction, we then have a result equivalent for the execution of e_2 , and thus b has a result equivalent to $[[b]]_i$. The same holds true when $o'_0 = \text{false}$ or if an exception is raised.
- Other cases are similar.

□

Lemma 13

Let b be a BSP-WHY sequential block, and $b' = [[b]]$ its transformation into WHY. If from a state s'_1 , b' executes to a state s'_2 with an outcome o' , and if s'_1 is equivalent to a parallel state s_1 , then b will execute to a state s_2 and an outcome o , with s_2 equivalent to s_1 and o equivalent to o' .

$$s_1 \sim s'_1 \Rightarrow s'_1, [[b]] \Downarrow s'_2, o' \Rightarrow \exists s_2, o \text{ such as } (s_1, b \Downarrow^b s_2, o) \wedge (s_2 \sim s'_2) \wedge (o \sim o')$$

Proof. Here, $[[b]]$ is a for loop, executing $[[b]]_i$ for $i = 0$ to $\mathbf{p} - 1$. The proof is thus an induction on \mathbf{p} , by applying repeatedly the Lemma 12. □

Block Tree Translation.**Lemma 14**

Let t be a BSP-WHY block tree, and $a = [[t]]$ its transformation into WHY. If from a state s'_1 , a executes to a state s'_2 with an outcome o' , and if s'_1 is equivalent to a parallel state s_1 , then t will execute to a state s_2 and an outcome o , with s_2 equivalent to s'_2 and o equivalent to o' :

$$s_1 \sim s'_1 \Rightarrow s'_1, [[t]] \Downarrow s'_2, o' \Rightarrow \exists s_2, o \text{ such as } (s_1, t \Downarrow^b s_2, o) \wedge (s_2 \sim s'_2) \wedge (o \sim o')$$

Proof. By induction on t .

- If t is a block, Lemma 13 gives the result directly.
- For $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$. The translation is $a = \text{if } \text{valid}([[t_1]]) \text{ then } [[t_2]] \text{ else } [[t_3]]$. The use of the **valid** parameter is the key point that allows the transformation to work. Since a has a derivation, the only rule that can apply is a rule in which $\text{valid}([[t_1]])$ has a derivation. Because we are using a blocking semantics, this means that the pre-condition of **valid** holds, hence either $[[e_1]]$ returns a \mathbf{p} -array with all *true*, or all *false*. By the induction hypothesis, this means we have a derivation $s_1, t_1 \Downarrow^b s_2, o$ with o the parallel outcome with all values *true*. We can thus apply the block semantics rule for the *if* instruction. By applying the induction hypothesis once more with either t_2 or t_3 , we get the result.
- The other induction cases are similar to the *if*, without the difficulty introduced by the **valid**.

□

Block Decomposition.**Lemma 15**

Let a be a BSP-WHY expression, and $t = \langle a \rangle$ the block tree result of the block decomposition. If from a state s_1 , t executes to a state s_2 with an outcome o , then a will execute to the same state s_2 and the same outcome o :

$$s_1, \langle a \rangle \Downarrow^b s_2, o \Rightarrow s_1, a \Downarrow s_2, o$$

Proof. By induction on the block tree.

- The general induction case is straightforward. For $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$, inverting the block tree decomposition gives us that $a = \text{if } a_1 \text{ then } a_2 \text{ else } a_3$, with $t_i = \langle a_i \rangle$. Inverting the derivation of b gives us three possible rules with the **if** statement. Let us consider the example

of the *true* rule. t_1 evaluate to the vector of *true* values, and t_2 evaluates to o . By induction, a_1 evaluates to the same *true* vector, and a_2 evaluates to o . Thus a evaluates to o .

- The interesting case here is when t is a block ($t = \text{Block}(b)$). By the block tree decomposition rules, we know that $a = b$ is an expression, tagged as having no parallel effects. The block semantics rules gives us that for every processor i , b is executed locally to the outcome o_i . Thus the global rule of the semantics can be used, and we have $s_1, a \Downarrow s_2, o$.

□

(c) Putting Pieces Back Together

We have now all the tools needed to prove our main lemma.

Lemma 16

Let a be a BSP-WHY expression, and $a' = [[a]]$ its transformation into WHY. If from a state s'_1 , a' executes to a state s'_2 with an outcome o' , and if s'_1 is equivalent to a parallel state s_1 , then a will execute to a state s_2 and an outcome o , with s_2 equivalent to s'_2 and o equivalent to o' :

$$s_1 \sim s'_1 \Rightarrow s'_1, [[a]] \Downarrow s'_2, o' \Rightarrow \exists s_2, o \text{ such as } (s_1, a \Downarrow s_2, o) \wedge (s_2 \sim s'_2) \wedge (o \sim o')$$

Proof. $[[a]]$ is the composition of the block decomposition and the block tree transformation:

$[[a]] = [[\langle a \rangle]]$. By Lemma 14, we have s_2 and o such as $s_1, \langle a \rangle \Downarrow^b s_2, o$. By Lemma 15, $s_1, a \Downarrow s_2, o$.



We can finally prove the theorem that we announced in the beginning of the section.

Theorem 1

For a given parallel program P , if the transformation of P by BSP-WHY is correct, then P itself is correct.

Proof. Let (p, b, q) be the pre-condition, body and post-condition of P . Let s_1 be a parallel environment that satisfies the pre-condition of P ($s_1 \models p$). We want to prove that P is correct, *i.e.* there exists a derivation for the expression b in the environment s_1 . Let us consider the sequential environment s'_1 canonically constructed from s_1 (we have $s_1 \sim s'_1$). Since $s_1 \models p$, the Lemma 11 gives us that $s'_1 \models [[p]]$. By hypothesis, $[[P]]$ is correct, so there exists a derivation of $[[b]]$, starting from the environment s'_1 , resulting in an environment s'_2 and an outcome o . We can thus apply the Lemma 16 to get a derivation of the parallel program, resulting in an environment $s_2 \sim s'_2$. $s'_2 \models [[q]]$, thus by Lemma 11, s_2 satisfies

the post-condition q .



(d) Correctness of Infinite Programs

In this thesis, we have only proved the correctness of the transformation for terminating programs. However, it is also possible to define the correctness of an infinite program, by using the co-inductive semantics seen in Section. 5.1. It is easy to modify these semantics to include blocking rules when dealing with logic assertions. A correct infinite program would then be a program that accepts an infinite derivation, according to the semantics. For instance, one could check the correctness of a server program made of an infinite loop, with an assertion in the loop body ensuring the desired property. The proof of the BSP-WHY transformation in this case would follow a similar reasoning, but has not been done because of a lack of time.

5.4.3 Elements of Proof in Coq

We did not have enough time to provide a full correctness proof in COQ in the scope of this thesis. However, we give an outline of the proof, with the necessary structures implemented in COQ.

(a) Semantics Definitions

We start by defining the evaluation of a logical term, and the satisfaction of a logical formula. The evaluation of a term is a straightforward recursive function:

Fixpoint lteval (e:wenv) (t:wlterm) : value := **match** t **with**
 | LTconst v \Rightarrow v
 | LTvar x \Rightarrow valueof e x
 | LTlet x t1 t2 \Rightarrow lteval (wupdate e x (lteval e t1)) t2
 ...
end.

Similarly, we define the satisfaction of a logical formula by a recursive definition. It is important to note that sat returns a **Prop**:

Fixpoint sat (e:wenv) (p:wprop) : **Prop** := **match** p **with**
 | Ptrue \Rightarrow True
 | Pand p1 p2 \Rightarrow (sat e p1) \wedge (sat e p2)
 | Pimply p1 p2 \Rightarrow (sat e p1) \rightarrow (sat e p2)
 | Plet x t p \Rightarrow sat (wupdate e x (lteval e t)) p
 | PForall x p \Rightarrow $\forall v$: value, sat (wupdate e x v) p
 | Peq t1 t2 \Rightarrow lteval e t1 = lteval e t2
 ...
end.

The semantics are then modified to include the blocking rules. For instance, the WHY big-step semantics now has the following three rules with logic assertions:

Inductive eval_wexpr: wenv \rightarrow wexpr \rightarrow wenv \rightarrow woutcome \rightarrow **Prop** :=

...
 | eval_WElloop : $\forall a e e' o i v$, sat e i \rightarrow eval_wexpr e (WEseq a (WEloop i v a)) e' o \rightarrow eval_wexpr e (WEloop i v a) e' o
 | eval_WEassert : $\forall e e' p a r$, sat e p \rightarrow eval_wexpr e a e' r \rightarrow eval_wexpr e (WEassert p a) e' r
 | eval_WEpost : $\forall e e' p a r$, eval_wexpr e a e' r \rightarrow sat e' p \rightarrow eval_wexpr e (WEpost p a) e' r
 ...

We also define the new *Block Semantics*:

Inductive eval_bexpr: par_env \rightarrow bexpr \rightarrow par_env \rightarrow par_outcome \rightarrow **Prop** :=

(* evaluation of a block *)
 | eval_BEblock : $\forall b e e' o$, eval_block e b e' o \rightarrow eval_bexpr e (BEblock b) e' o
 ...
 | eval_BEif1 : $\forall a1 a2 a3 e e' v o$, eval_bexpr e a1 e' v \rightarrow
 all_true v \rightarrow eval_bexpr e' a2 e' o \rightarrow eval_bexpr e (BEif a1 a2 a3) e' o
 ...
 | eval_BEsync : $\forall e$, eval_bexpr e (BESync) (comm e) po_void

(b) Correctness

Next, we define the notion of correctness, matching the blocking semantics definition:

Definition wcorrect prog := $\forall e1, e1 \models wpre \text{ prog} \rightarrow \text{exists } e2 o,$
 eval_wexpr e1 (wbody prog) e2 o $\wedge e2 \models wpost \text{ prog}$.

(c) Proof Skeleton

The four lemmas used to separate the proof (Lemma 12, Lemma 13, 14 and 15) are given next:

Lemma block1 : $\forall i e1 e'1 e'2 o' a$, bij_env e1 e'1 \rightarrow
 eval_wexpr e'1 (seqexpr_transformation a i) e'2 o' \rightarrow
 (exists e2 o, bij_env e2 e'2 \wedge bij_outl o o' \wedge eval_expr i (e1 i) a (e2 i) o).

Lemma block2 : $\forall e1 e'1 e'2 o' a$, bij_env e1 e'1 \rightarrow
 eval_wexpr e'1 (block_transformation a) e'2 o' \rightarrow
 (exists e2 o, bij_env e2 e'2 \wedge bij_out o o' \wedge eval_block e1 a e2 o).

Lemma blocktree : $\forall e1 e'1 e'2 o' a$, bij_env e1 e'1 \rightarrow
 eval_wexpr e'1 (tree_transformation a) e'2 o' \rightarrow
 (exists e2 o, bij_env e2 e'2 \wedge bij_out o o' \wedge eval_bexpr e1 a e2 o).

Lemma blockdecomp : $\forall e1 e2 a o$, eval_bexpr e1 (block_decomposition a) e2 o \rightarrow
 eval_par e1 (mkpar a) e2 o.

The main Lemma 16 is easily proved with the last two lemmas.

Lemma mainlemma : $\forall a e1 e'1 e'2 o'$, bij_env e1 e'1 \rightarrow
 eval_wexpr e'1 (trans_expr a) e'2 o' \rightarrow
 (exists e2 o, bij_env e2 e'2 \wedge bij_out o o' \wedge eval_par e1 (mkpar a) e2 o).

Finally, we can state the Theorem 1:

Theorem `correct_transformation` : $\forall \text{prog}, \text{wcorrect} (\text{bspwhy prog}) \rightarrow \text{correct prog}.$

5.5 Related Work

To our knowledge, the first work on a formal operational semantic of BSP is [187]: the author gives a small-step semantic using its own primitives of its own core language. Neither mechanised work nor applications have been done. The interests and examples of the use of mechanised semantics for certified program verifiers are given in [57]. In [155], the author gives a mechanised proof of the results of the weakest preconditions calculus used in WHY. A mechanised big-step semantic of WHY were given. The author used massively dependent types whereas we choose a simple model of the language in the spirit of [192]. But our work on the proof of the transformation (of BSP-WHY codes to WHY ones) and the results of [155] can clearly be associated to gain more confidence in the outputs of BSP-WHY.

A work on proving determinism, using assertions in the code, of multi-threaded JAVA programs with barriers can be found in [44]. The authors note that *there seemed to be no obvious simpler, traditional assertions that would aid in catching non-deterministic parallelism*. In our case of BSP programs, this work is simple — but still limited to BSP programs.

Another work on concurrent threading with barriers is [163]. The authors have developed and proved sound a concurrent separation logic for barriers of threads. An interesting point is that the proofs are machine-checked in COQ. The authors also showcase a program verification toolset that automatically applies the logic rules (Hoare logic) and discharges the associated proof obligations. It is thus a work for derivation of formal specification into correct parallel programs. The drawback (as in [283] and partly in [220]) is that only programs with a predefined constant number of threads (*e.g.* two for a producer-consumer problem) can be considered. For HPC, we prefer to have correct programs for an unknown number of processors in a data-parallel fashion. Another interesting work is [206] in which, using the theorem prover COQ, the authors give a mechanised deductive verification of shared-memory concurrent algorithms for software barriers (of synchronisation) of multi-threaded programs. But assertions are hard to understand and especially the number of threads is still bounded.

6 Conclusion

It is time to conclude and sketch possible improvements and future work. We first briefly summarize our contributions and then we discuss how close we have come to the initial goal of the thesis, namely to be able to prove parallel (BSP) programs. Finally, we discuss remaining challenges and future work.

6.1 Summary of the Contributions

Parallel/distributed programs have applications in many different areas because they offer the possibility of computing data much faster than sequential programs. However, even for small parallel programs, error is more often the rule, not the exception. That seems to create a need for *verification* of parallel programs. Verification is the rigorous demonstration of the correctness of a program with respect to a specification of its intended behaviour. And *formal methods* are increasingly being applied for this goal which also create a strong need for on-machine (mechanised) formalisation and verification of the used tools — programming language semantics, compilers, static/dynamic analysis, *etc.*

Since the seminal paper “Goto Statement Considered Harmful”, *structured* sequential programming is the norm. And there are now many relevant methods and tools for the verification of sequential programs. But it is not the case for parallel programming. Beside a true zoo of parallel paradigms (models and languages), the main reason is that programmers have kept the habit to use low-level parallel routines (such as asynchronous and unbuffered send/receive of *e.g.* MPI) [136] or concurrent languages [188]. They more or less manage the interleaving of the instructions adjuncting with the communications with the usual problems of *data race*, *deadlocks* and *non-determinism*. That makes the verification of parallel programs harder. By consequence, programmers cannot focus on the correctness of their algorithms in addition to the detection of typical bugs — *e.g.* such as buffer overflow. This makes the design of robust tools for the verification of parallel programs an important area of research.

In this thesis, we have presented an intermediate language aimed at the verification of BSP programs. It is an extension of the language of the verification tool WHY which is designed for sequential programs. We summarize the contributions done throughout this thesis thereafter and in Fig. 6.2.

Chapter 2: A comparison of different BSP libraries. In this chapter, we have tried to present the common routines for imperative BSP programming as well as more exotic BSP and dedicated languages. We mainly found buffered sending, DRMA primitives and collective operations. We also present common extensions (such as subgroup synchronisation, high-performance send/DRMA routines and thread migration) of the BSP model with their advantages and inconveniences. We have finished this chapter with a presentation of the common bugs, like deadlocks, that can be found in distributed/parallel computations and mainly in BSP programs.

Chapter 3: A tool for deductive verification of BSP programs. In this chapter, we have presented a SPMD BSP extension of the WHY-ML language. Then, we gave a block transformation of the BSP-WHY-ML language into the WHY-ML one, featuring obligation generation for BSP programs — for the correctness. It is the main work of the tool BSP-WHY. The output programs rely on a generic library of logical axioms and definitions. The transformation uses a particularity of the BSP model: since BSP programs are decomposed in *super-steps* (local computations separated with global barrier of synchronisation), parallelism can be removed by replacing a portion of code between barriers with a loop to repeat that portion for every process. The BSP-WHY transformation is formally given for a core-calculus, mainly using inductive rules. We also gave some simple examples of the transformation to illustrate its working.

The BSP-WHY transformation processes as follow. First, decomposing the program into its blocks of local computations; then adding some piece of logical information for maintaining that every processor would execute the block properly; finally transforming each block by detecting which variable is

transformed into an array of \mathbf{p} -values (where \mathbf{p} is the number of processors) which also arise for logical assertions. WHY processed by applying a wp-calculus (for generating proof obligations) whereas BSP-WHY “sequentialise” (simulate) the parallelism by directly transforming the parallel program (with its logical annotations) into a sequential one. And then a wp-calculus can be applied. We think that building upon an existing tool for program verification (and not do this work from scratch) is quite appealing since generated proof obligations need a lot of work — in theory and in practice.

Finally, we have also presented how this transformation can be adapted to an extension of the BSP model, the subgroup synchronisation. We have shown that some modifications are needed as well as additional new logical assertions in the generated sequential WHY programs in order to achieve safety of execution and absence of deadlock. We believe that the restriction of the block decomposition still is an efficient heuristic for proving BSP algorithms.

Chapter 4: Applications of the BSP-Why tool. The BSP-WHY has first been applied to some classical BSP algorithm examples, namely reductions and sorting. We showed the results of our tool on these examples by counting how many generated proof obligations were automatically discharged. As might be expected, not all the goals are discharged and some still remain to be proved, using COQ for example. Nevertheless, those for safety are discharged which is an encouraging first step — we have the proof for any number of processors. Thus, in view of this ratio “number to prove/proved”, we believe our tool is far from perfect (notably for correctness) but nevertheless, it can rapidly increase the confidence to be placed in the code since at least the safety properties are massively proved automatically.

We have then used BSP-WHY for the formal analysis of model-checking (restricted to state-space generation) algorithms — with one dedicated to security protocols because model checkers are specialised software, using sophisticated algorithms, whose correctness is vital. For example, verifying security protocols is complex and often error prone: various attacks are reported in the literature to protocols thought to be “correct” for many years. There are now many tools that check the security of cryptographic protocols and model-checking is one of the solutions [65]. But model checkers could miss a state which can be an unknown attack of the security protocol.

In this work, we focus on correctness of a well-known sequential algorithm for finite state-space construction (which is the basis for explicit model-checking) and distributed ones. We annotated the algorithms for finite set operations (available in COQ) to obtain goals that were entirely checked by automatic solvers. These goals ensure the termination of the algorithms as well as their correctness for any successor function — assumed correct and generating a finite state-space. We thus gained more confidence in the code. We also hope to have convinced the reader that this approach is humanly feasible and applicable to real (parallel or sequential) model-checking algorithms.

Chapter 5: Mechanized semantics of a parallel core calculus. In this chapter, we have presented mechanized operational semantics for BSP programs using COQ for a core-calculus of our BSP-WHY-ML language. Fig 6.2 outlines the scheme of the results. We have given a big-step semantics as well as a small-step one and co-inductive rules for both semantics in order to reason on infinite computations. We have prove that they are both deterministic and equivalent. We have then extended this work for subgroup synchronisation — but the complete mechanised proof of equivalence of the semantics must still be done. Using the semantics, we also have sketched the proof of correctness of the BSP-WHY transformation (“sequentialisation”): transform (under some conditions) a BSP code with annotations into an equivalent sequential one.

The big-steps semantics uses two kinds of rules: one for local (sequential) computation and one for the whole parallel machine. There are also different kinds of returning values depending of the context: returning a truly value, an exception or a call to synchronisation. The last one is used in the parallel rules to finish the super-step and perform the communications. For this, the small-steps semantics has been written in a continuation style: the continuation keeps all the computations that still remain to perform. And there is no local rule to reduce a call to synchronisation: only a global rule, with all the continuations, can terminate the BSP super-step. Adding subgroup synchronisation does not really change the rules, except that only a group of processors synchronises which induces a “diamond” property. The proofs of confluence are fairly standard inductions. The proof of equivalence is also standard but required more efforts due to the nature of the semantics: continuations and a distributed memory model. Our mechanised semantics do not use at all dependent types (in contrary of [155]) for the simple reason that we do not manage them in our proofs — even for \mathbf{p} -values where \mathbf{p} is a fixed integer and thus we could have \mathbf{p} -vectors. Our semantics are thus close (in their design) to the ones of [192]: we want to extend this work for BSP programs so it seems natural to consider the same kind of semantics.

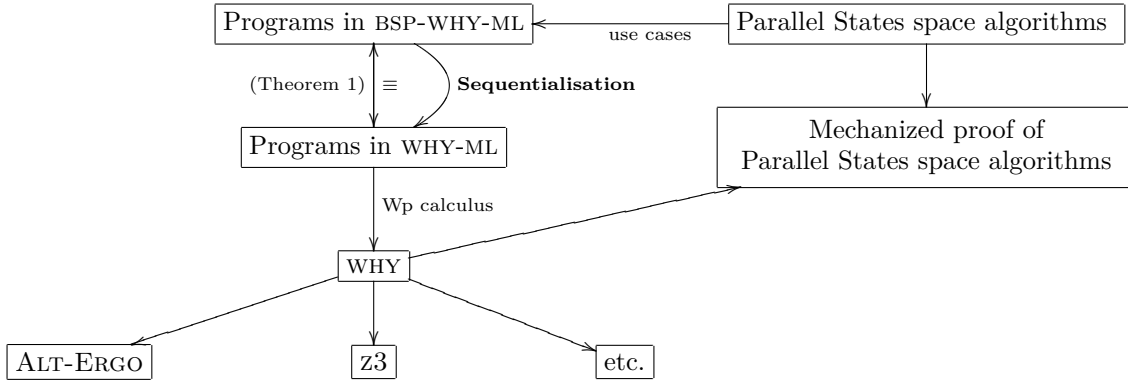


Figure 6.1. Scheme of the main results of the Chapters 3, 4 and 5.

Up to now, most (all?) of the tools for verification of parallel programs have never been formally proved to be sound. The verification tools needs thus to be trusted, even if it can become quite complex, in order to gain confidence in their results — the correctness of the verified programs. One can say that verification tools do not (always) respect the “de Bruijn criterion” that is having the correctness of the system depending of a trusted small kernel¹ — such as COQ. Given the difficulties and the amount of time needed to perform the soundness proof of the sequentialisation, one may wonder whether it would have been easier to avoid this semantics preservation proof altogether by proving the soundness of a wp-calculus directly on BSP-WHY-ML without going through an intermediate language as WHY-ML. At least, this would incite focusing on the real issues of a wp-calculus [155, 156, 285] and implicit block decomposition (to avoid deadlock) would have remained as hard. An “originality” of this chapter is that the semantics of the language as well as the transformation have been written in the specification language of COQ. A part of the proof of observational semantic equivalence between the source and the generated code has been partially machine-checked using COQ which ensures a better trust in the results. It also used an intermediate semantics. An executable compiler can be obtained by automatic extraction of executable OCAML code from COQ — as in [155, 156, 285]. This work is our first experiment to create a certified software for deductive verification. The main goal of this work is an environment where programmers could prove correctness of their BSP programs in a trusted way.

To finish, Drawbacks and Lessons Learnt from this Work. Our BSP-WHY tool extends WHY for BSP algorithms and has been partially mechanically proved. We are not aware of such an environment (with such a confidence in the results) for parallel programs and for an unbound number of processors. It in part meets the request of a proof tool presented in the introduction. We can enumerate the four distinct lacks:

1. The user still needs to annotate himself the programs. This can be tedious and needs a great degree of skill of the user. The ideal situation would be to have tools that automatically extract some properties (the full specification is not decidable), and indeed a great deal of research on this subject has been carried out, at least for some specific domains such as numerical computations [114, 254].
2. Currently, BSP-WHY does not work for C or JAVA programs;
3. There are no “error messages” available (except syntax error) which does not make it a really useful and diffusable tool;
4. The sequentialisation induces too many goals and those that are not automatically proved are too hard to read; thus, BSP-WHY does not truly scale for big programs.

BSP-WHY is our first experiment about the design of a tool for deductive verification of parallel programs. It was rewarding and showed us unexpected difficulties. First, the invariants for the “for loop” were hard to find — a technical difficulty which did not imply a fundamental change of the tool. Second, testing the implementation were hard due to the big number of generated goals and the fact that automatic solvers could not prove anything even for trivial cases. Third, the small-step semantics with continuations is not a thing easily handled in COQ and the sketch of the semantics preservation proof

¹It is generally call “**Trusted Code Base**” (TCB) the part of the tool on which the soundness depends on. For example, COQ have a very small TCB.

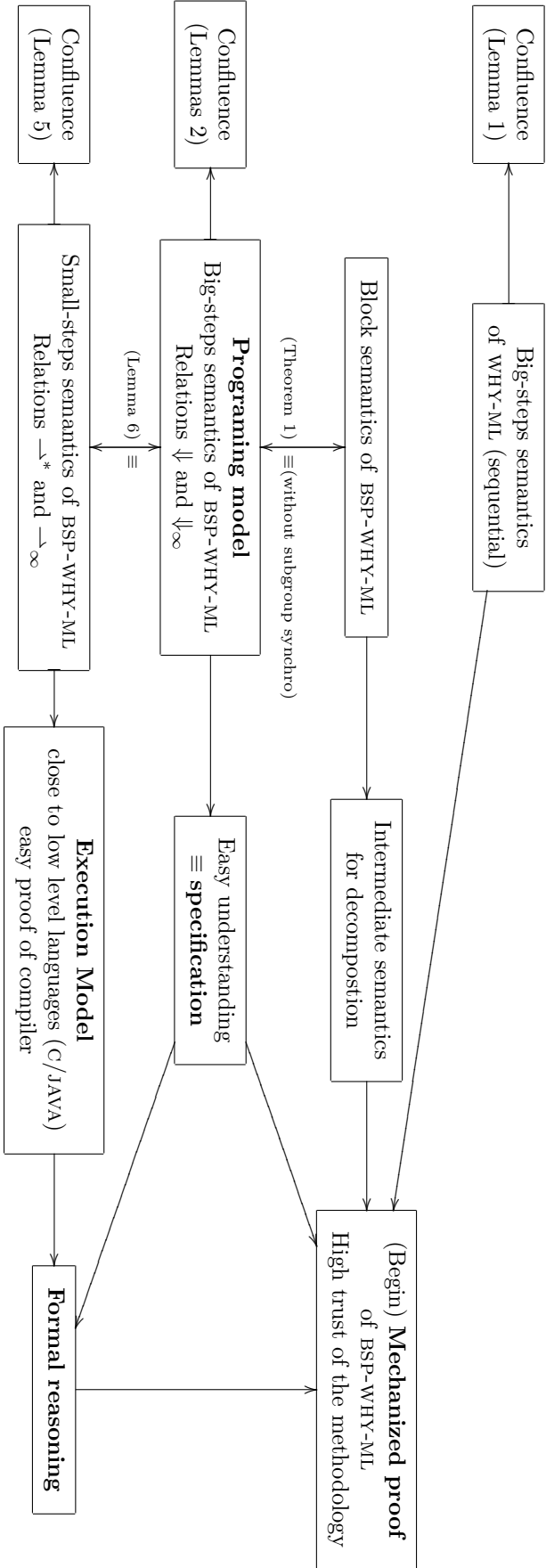


Figure 6.2. Scheme of the contributions of the Chapter 5 — mechanized semantics investigation.

were hard to find because of a vicious circle in the proof: to prove the additional logical assertions due to the block decomposition, we need the fact that the sequential program is correct, but to prove it we need to prove that the block decomposition had done its job. Nevertheless, we have learnt many things.

First, creating your own tool for deductive verification needs a great carefulness and having machine-checked proofs is not only a way to do well in the academic: it is during the design of the mechanised semantics that we highlighted some issues with the translation. Thus, by formalising our semantics and a part of the methods of BSP-WHY in COQ we have gained a full understanding of the difficulty involved in designing correct proof methods for the verification of parallel programs. The level of detail required for such a formal study naturally leads to approaching each step of the formalisation with a critical eye, considering first to investigate alternatives that could simplify the formalisation. However, to understand the theorem proving techniques, which are involved in many formalisations, requires a great deal of time, effort and mistakes.

Second, an interesting part of a verification process is to understand the program and to find an intuitive proof of the correctness. However, the projection of this intuition into assertions implies, in general, changing and tuning the assertions too many times and a great deal of effort is expended to get the details right. For example, the generated conditions of BSP-WHY are really unreadable and only automatic provers could manage them. But automatic provers regularly give strange results: sometimes they found the result and sometimes not without any comprehensible reason; sometimes they failed due to the adding of a new axiom that has nothing to do with the current job. This is tiring when experiments are performed. BSP-WHY does not directly help in finding the right annotations but at least automatizes the iterative process of changing assertions and checking the proof again. From our experience, we do not recommend trusting paper and pencil proofs of correctness of parallel algorithms and programs. Especially if the invariants of the loops are implicitly given — which is traditionally done in the field of parallel model-checking.

Finally, we hope that the modest contributions presented in this thesis will convince the reader that the BSP-WHY tool will be viable for the verification of parallel programs for a large number of areas. And we will now present some remaining challenges, how to improve BSP-WHY and expected future work.

6.2 Future Work and Perspectives

6.2.1 Close Future Work

(a) For the BSP-Why Tool

We need to test our method on realistic BSP computations even if the results of our examples are encouraging. Programs of [25, 81] (LU decomposition, fast Fourier transform, *etc.*) will be used. The current prototype implementation of BSP-WHY is still limited. We plan to extend it in several ways.

First, we intend to add a companion tool for C programs as in [105] and for JAVA programs. The tool for C programming is a plug-in called JESSIE for the FRAMA-C framework — note that another plug-in call WP is also available but it directly generates conditions without using WHY. JESSIE generates WHY codes from C ones. For JAVA programs, the KRAKATOA tool will be considered — it also generates WHY codes. JAVA programs which use HAMA [246]) could be considered as HAMA's communication primitives are close to those of BSP-WHY.

Second, conditions generated by WHY from BSP-WHY programs are not friendly, even for theorem provers (especially for them?), and a lot of them are not proved automatically. This is mainly due to the massive use of **p**-loops generated to simulate the **p**-asynchronous computations. Special tactics are needed to simplify the analysis. Also, many parts of the programs work in the same manner and only differ by the manipulated data. For example, variables such as counters for loops, local sorting of data. Finding these parts to factorize them and generate more friendly conditions would simplify the work of provers. In the same manner, syntactic sugar to manipulate the environment of communications (list of messages) is needed to facilitate the writing of logical assertions.

Third, they are still bugs in the current implementation. We need to correct them. And we need to deal with error messages to facilitate the use of the tool. Fourth, we currently work on WHY-2 whereas WHY-3 now exists — close to WHY-2 except for the logical parts and some new features such as pattern-matching of union types. We need to adapt our work to the new syntax which should mainly be only a technical work. Fifth, we are currently not taking into account oblivious synchronisation (`obl_sync(n)`) but it seems feasible quickly by counting the number of received message after the communication and adding this new assertion: `n` strictly equal to this counting and every processor calls this routines. This

will induce a safe use of oblivious synchronisation.

(b) Mechanised Semantics

First, we need to finish the proofs of soundness of the sequentialisation, of the equivalence of semantics for subgroups synchronisation and of the soundness of the sequentialisation with subgroups. It will be again mostly a technical work but needs to be done for a gain of confidence in our methodology.

We also plan to extend our semantics studies to the semantics of the aforementioned languages as in [28, 156] for the CompCert (C compiler certified in COQ) [192]. In [156], the author gives (and proves using COQ) an equivalence of the big-step semantics of Clight with the semantics of a transformation of Clight into “WHY”. The semantics were close to ours, and we believe that we could achieve this goal for BSP-WHY.

(c) Machine-checked Model-checking

We are currently proving state-space algorithms and not the effective code. Regarding the code structure, this is not really an issue and translating the resulting proof into a verification tool for true programs should be straightforward, if high level data-structures are used. Also, machine-checked model-checkers would certainly be less efficient than traditional ones. But they could be used in addition when it seems necessary to give greater confidence in the results. We also believe that another interesting application of a verified tool (such as we are envisaging it) would be to serve as a *reference* implementation that is used to compare the results of an efficient implementation over a set of *benchmark* problems.

6.2.2 Long Term Perspectives

(a) For the BSP-Why Tool

BSP is an interesting model because it features a realistic cost model for an estimation of the execution time of its programs. Formally giving these costs by extended pre-, post-condition and invariants is an interesting challenge: one could speak of a cost certification. In fact, many scientific algorithms (numeric computations such as matrix ones) do not have too complicated complexities: it is often a polynomial number of super-steps. In the case of cloud-computing [9], we can imagine a scheduler server that distributes the parallel programs depending on the cost certificates to optimise power consumption. Our current case studies prove that BSP-WHY is an interesting possibility for this topics. For example, we have currently (not presented here) proven the BSP cost of the prefix computation by traditionally adding ghost variables for counting: (1) the operations; (2) the loop iterations; (3) and the number of super-steps. For communication we have tested: (1) manipulating a matrix which represents the counting of communication (a transposition allows to count the BSP communication) but automatic solvers were not able to prove anything; (2) adding a “heuristic” where during the “for loop” a processor increases the number of received messages of the other processors; automatic provers would be more efficient in this case but we are not yet convinced that we can deal enough algorithms in this way. Worst-case, average case, amortised worst-case, complexity, *etc.* give different meanings and thus would need different studies.

Furthermore, in a previous work [111], we have mechanically proven a simple optimization of the source code that transforms traditional BSP routines to their high-performance versions. These routines are proposed to programmers by most BSP libraries for improved speedup of their programs even if they are unsafe: they are unbuffered and do not really follow the safe BSP model of execution. Replacing BSP routines by their high-performance pendants remains to the responsibility of the programmer or of a non-formally verified compiler analyser such as in [78]. By using the logical assertions (and those of the block decomposition), we believe that we will have more information to optimise the programs. Using our semantics, this new function of optimisation could (and should) be done using COQ as in [111].

Last, there are many more MPI programs than BSP ones. Our tool is not intended to manage all MPI programs. It cannot be used to model asynchronous send/receive ones — with possible deadlocks depending on the MPI scheduler [251, 284]. Only programs which are BSP-like (*e.g.* MPI’s collective primitives) [47] will be considered. Analysing MPI programs to find out which ones are BSP-like and to interpret them in BSP-WHY is a great challenge, as in [211].

(b) Machine-checked Model Checking

Future goals are clear. First, we would like to adapt our work for true model-checking algorithms — as those for LTL/CTL* [23]. Model-checking algorithms are mainly Tarjan like algorithms or NDFS

(Nested Depth-First Searches) for considered Strongly Connected Components (SCC) in graphs. This is challenging in general but using an appropriate tool, we believe that a team could “quickly” do it. Second, the successor function (computation of the transitions of the state-space) is currently an abstract function. We think about proving the work of [113] in a mechanically-assisted way in order to compensate this deficiency. Third, compression aspects like symmetry, partial order, *etc.* must be studied since they can generate wrong algorithms. The work of [275] which use the B method could be a good basis.

And to finish, all these approaches require mathematical foundations (CTL* as in [274], SCC of proof-graphs for [23], *etc.*) which need to be machine-checked. The effort for such a project and thus for verifying the whole stack of Fig. 4.10 is not at all within the reach of a single team. But our guess is that each of these single steps is largely feasible. Also, as for the problematic of compilers [192], machine-checked model-checkers would certainly be less efficient than traditional ones. But they could be used in addition when it is desired to give greater confidence in the results.

(c) Hybrid and Hierarchical Computing

The flat view of a parallel machine as a set of communicating sequential machines (as BSP reminds us) remains useful, but it is nowadays incomplete [12]. For example, GPU processors have a master-worker architecture and clusters of multi-cores make the core share a few network cards, which implies a bottling and a congestion of the communications. Moreover, we can observe that heterogeneous chip multi-processors (as CELL and GPU’s feature) present unique opportunities for improving system throughput and reducing processor power: the trend towards green-computing puts even more pressure on the optimal use of architectures that are not only highly scalable but hierarchical and non-homogeneous. We do not know how to design algorithms for nested level systems with the original flat BSP model. The hierarchical architectures share communication resources inside every level but not between different levels. The flat BSP cost model is not thus suitable for this kind of architectures which are the future of parallel and high-performance computing.

With these issues in mind, the authors of [195] proposed a language called SGL (for scatter/gather language) which is a master-worker language based on the multi-BSP model [279]: a multi-level variant of the BSP model — to our knowledge, the first multi-BSP model was proposed in [286]. The one of [279] is more realistic for “cluster of clusters of multi-cores with GPU processors”. This recursive model abstracts parallel architectures as trees of processes (each branch is a master and a child, except for leaves) and thus needs recursive algorithms. But SGL does not seem appropriate for more complicated algorithms such as the ones dedicated to model-checking. By allowing only scatter and gather operations, it is not possible to have point-to-point communications. In [148], the authors propose an explicit call to nested parallel computations: the nested BSP computation has just a pre-function to scatter the data and a post-function to gather them. We should continue such approaches.

Currently, BSP-WHY can only deal with the flat parallelism. But, as in [264], BSP-WHY can manage subgroup synchronisation. Subgroups could be the cores of the processors; another group could be the whole machine. In this way, we can deal with hybrid/hierarchical programs. But that is too static and explicit nested parallel computations should be considered in order to take into account this future architectures or programs. Finally, one could wish to mix BSP computations and less structured models. Future reflections will deal with how to mix deductive verification of different models for the same program.

Bibliography

- [1] M. Aldinucci and M. Danelutto. Skeleton-based Parallel Programming: Functional and Parallel Semantics in a Single Shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, 2007. Pages 8 and 38.
- [2] M. Aldinucci, M. Danelutto, and P. Dazzi. MUSKEL: An Expandable Skeleton Environment. *Scientific International Journal for Parallel and Distributed Computing*, 8:325–341, 2007. Pages 8 and 39.
- [3] M. Alt, H. Bischof, and S. Gorlatch. Algorithm Design and Performance Prediction in a JAVA-Based Grid System with Skeletons. In *EUROPAR*, volume 2790 of *LNCS*, pages 899–906. Springer, 2003. Pages 8 and 39.
- [4] A. W. Appel and S. Blazy. Separation Logic for Small-Step Cminor. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *LNCS*, pages 5–21. Springer, 2007. Page 97.
- [5] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending COQ with Imperative Features and Its Application to SAT Verification. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010. Page 76.
- [6] A. Armando, R. Carbone, and L. Compagna. LTL Model Checking for Security Protocols. *Applied Non-Classical Logics*, 19(4):403–429, 2009. Pages 77 and 88.
- [7] A. Armando and L. Compagna. SAT-based Model-checking for Security Protocols Analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008. Page 88.
- [8] A. Armando and *et al.* The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005. Page 88.
- [9] M. Armbrust, A. Fox, R. Griffith, and *al.* Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. Pages 9 and 122.
- [10] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: a Structured High-level Parallel Language, and its Structure Support. *Concurrency: Practice and Experiences*, 7(3):225–255, 1995. Page 8.
- [11] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992. Page 38.
- [12] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. L. Lusk, R. Thakur, and J. L. Träff. MPI on Millions of Cores. *Parallel Processing Letters*, 21(1):45–60, 2011. Pages 1 and 123.
- [13] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003. Page 10.
- [14] M. Bamha and G. Hains. Frequency-adaptive Join for Shared Nothing Machines. *Parallel and Distributed Computing Practices*, 2(3):333–345, 1999. Page 10.
- [15] M. Bamha and G. Hains. An Efficient Equi-semi-join Algorithm for Distributed Architectures. In V. Sunderam, D. van Albada, and J. Dongarra, editors, *International Conference on Computational Science (ICCS)*, LNCS. Springer, 2005. Page 10.
- [16] J. Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Faculty of Informatics Masaryk University Brno, 2004. Page 79.
- [17] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and Verification: The Spec# Experience. *Commun. ACM*, 54(6):81–91, 2011. Page 67.
- [18] B. Barras and B. Werner. COQ in COQ. Technical report, INRIA, 1997. Page 88.
- [19] G. Barthe, B. Grégoire, and S. Z. Béguelin. Computer-Aided Cryptographic Proofs. In A. Miné and D. Schmidt, editors, *International Symposium on Static Analysis (SAS)*, volume 7460 of *LNCS*, pages 1–2. Springer, 2012. Page 89.
- [20] D. A. Basin, S. Mödersheim, and L. Viganò. An On-the-Fly Model-Checker for Security Protocol Analysis. In E. Snekenes and D. Gollmann, editors, *European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *LNCS*, pages 253–270. Springer, 2003. Page 88.
- [21] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming Distributed Memory Systems Using OpenMP. In *IPDPS*, pages 1–8, 2007. Page 7.
- [22] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. Pages 4 and 101.
- [23] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Logic in Computer Science (LICS)*, pages 388–398. IEEE Computer Society, 1995. Pages 122 and 123.
- [24] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP versus LogP. *Algorithmica*, 24:405–422, 1999. Pages 10 and 11.

- [25] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004. Pages 8, 9, 10, 17 and 121.
- [26] R. H. Bisseling and W. F. McColl. Scientific Computing on Bulk Synchronous Parallel Architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing*, volume 51 of *IFIP Transactions*, pages 509–514. Elsevier, 1994. Page 10.
- [27] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Computer Security Foundations Symposium (CSFW)*. IEEE Computer Society, 2001. Page 88.
- [28] S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. Page 122.
- [29] G. E. Blelloch. NESL. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1278–1283. Springer, 2011. Page 39.
- [30] S. Blom, J. F. Groote, S. Mauw, and A. Serebrenik. Analysing the BKE-security Protocol with μ CRL. *Electr. Notes Theor. Comput. Sci.*, 139(1):49–90, 2005. Page 88.
- [31] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Principles and Practice of Parallel Programming (POPL)*. ACM Press, 1995. Page 38.
- [32] F. Bobot and J.-C. Filliâtre. Separation Predicates: a Taste of Separation Logic in First-Order Logic. In *Formal Engineering Methods (ICFEM)*, volume 7635 of *LNCS*. Springer, 2012. Page 67.
- [33] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd Your Herd of Provers. In *International Workshop on Intermediate Verification Languages (Boogie)*, 2011. Page 13.
- [34] S. Böhme and T. Weber. Fast LCF-Style Proof Reconstruction for Z3. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010. Page 88.
- [35] O. Bonorden. *Versatility of Bulk-Synchronous Parallel Computing: From the Heterogeneous Cluster to the System on Chip*. PhD thesis, University of Paderborn, 2008. Page 10.
- [36] O. Bonorden, J. Gehweiler, and F. M. auf der Heide. A Web Computing Environment for Parallel Algorithms in JAVA. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006. Pages 15 and 29.
- [37] O. Bonorden, B. Judoiink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003. Pages 15, 21 and 23.
- [38] O. Bonorden, J. von zur Gathen, J. Gerhard, O. Muller, and M. Nocker. Factoring a Binary Polynomial of Degree over one Million. *ACM SIGSAM Bulletin*, 35(1):16–18, 2001. Page 10.
- [39] L. Bougé, D. Cachera, Y. L. Guyadec, G. Utard, and B. Virot. Formal Validation of Data-Parallel Programs: a Two-Component Assertional Proof System for a Simple Language. *Theoretical Computer Science*, 189(1-2):71–107, 1997. Page 68.
- [40] W. Bousdira, F. Loulergue, and J. Tesson. A verified library of algorithmic skeletons on evenly distributed arrays. In Y. Xiang, I. Stojmenovic, B. O. Apduhan, G. Wang, K. Nakano, and A. Y. Zomaya, editors, *Algorithms and Architectures for Parallel Processing (ICA3PP)*, volume 7439 of *LNCS*, pages 218–232. Springer, 2012. Page 12.
- [41] C. Boyapati, R. Lee, and M. C. Rinard. Ownership Types for Safe Programming: Preventing Data-races and Deadlocks. In *OOPSLA*, pages 211–230. ACM, 2002. Page 67.
- [42] A. Braud and C. Vrain. A Parallel Genetic Algorithm based on the BSP Model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, 1999. Page 10.
- [43] A. D. Brucker and S. Mödersheim. Integrating Automated and Interactive Protocol Verification. In *Formal Aspects in Security and Trust (FAST)*, volume 5983 of *LNCS*, pages 248–262. Springer, 2009. Page 89.
- [44] J. Burnim and K. Sen. Asserting and Checking Determinism for Multithreaded Programs. *Commun. ACM*, 53(6):97–105, 2010. Page 116.
- [45] R. Calinescu. Bulk Synchronous Parallel Algorithms for Optimistic Discrete Event Simulation. Technical Report PRG-TR-8-96, Programming Research Group, Oxford University Computing Laboratory, 1996. Page 10.
- [46] F. Cappello, P. Fraigniaud, B. Mans, and A. L. Rosenberg. An Algorithmic Model for Heterogeneous Hyper-clusters: Rationale and Experience. *Int. J. Found. Comput. Sci.*, 16(2):195–215, 2005. Pages 10 and 11.
- [47] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *Computer Communications and Networks (ICCCN)*, pages 1–8. IEEE, 2010. Pages 10, 17 and 122.
- [48] D. Caromel, L. Henrio, and M. Leyton. Type Safe Algorithmic Skeletons. In *Parallel, Distributed and Network-Based Processing (PDP)*, pages 45–53. IEEE, 2008. Pages 8 and 39.
- [49] M. M. T. Chakravarty, R. Leshchinskiy, S. L. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In N. Glew and G. E. Blelloch, editors, *Declarative Aspects of Multicore Programming (DAMP)*, part of *POPL*, pages 10–18. ACM, 2007. Page 39.
- [50] B. L. Chamberlain. ZPL. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 2161–2166. Springer, 2011. Page 38.
- [51] A. Chan, F. Dehne, and R. Taylor. Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *Journal of High Performance Computing Applications*, 2005. Pages 2, 10, 11 and 15.
- [52] S. Chaou, G. Utard, and F. Pommereau. Evaluating a Peer-to-peer Storage System in Presence of Malicious Peers. In W. W. Smari and J. P. McIntire, editors, *High Performance Computing and Simulation (HPCS)*, pages 419–426. IEEE, 2011. Page 89.
- [53] B. Chapman. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2008. Page 7.

- [54] Y. Chen and J. Sanders. Logic of Global Synchrony. *ACM Transactions on Programming Languages and Systems*, 26(2):221–262, 2004. Page 69.
- [55] Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3):389–400, 2003. Page 70.
- [56] B. Cheng, Y. Jiang, and W. Tong. Hierarchical Resource Load Balancing Based on Multi-Agent in Service BSP Model. *International Journal of Communications, Network and System Sciences (IJCNSS)*, 3(1):59–65, 2010. Page 10.
- [57] A. Chlipala. Modular Development of Certified Program Verifiers with a Proof Assistant. *J. Funct. Program.*, 18(5-6):599–647, 2008. Page 116.
- [58] S. Christensen, L. M. Kristensen, and T. Mailund. A Sweep-Line Method for State-space Exploration. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001. Page 84.
- [59] C.-K. Chui. The LogP and MLogP Models for Parallel Image Processing with Multi-core Microprocessor. In *Symposium on Information and Communication Technology (SoICT)*, pages 23–27. ACM, 2010. Page 10.
- [60] P. Ciechanowicz and H. Kuchen. Enhancing Muesli’s Data Parallel Skeletons for Multi-core Computer Architectures. In *HPCC*, pages 108–113. IEEE, 2010. Pages 8 and 39.
- [61] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000. Page 3.
- [62] F. Clément, V. Martin, A. Vodicka, R. D. Cosmo, and P. Weis. Domain Decomposition and Skeleton Programming with OCamlP3L. *Parallel Computing*, 32:539–550, 2006. Pages 8 and 39.
- [63] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. Page 68.
- [64] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. Pages 8 and 39.
- [65] H. Comon-Lundh and V. Cortier. How to Prove Security of Communication Protocols? A Discussion on the Soundness of Formal Models w.r.t. Computational Ones. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 29–44, 2011. Pages 76 and 118.
- [66] V. Cortier, S. Kremer, and B. Warinschi. A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems. *J. Autom. Reasoning*, 46(3-4):225–259, 2011. Page 88.
- [67] R. D. Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OCamlP3L 2.0. *Parallel Processing Letters*, 18(1):149–164, 2008. Page 8.
- [68] V. G. Costa and M. Marín. A Parallel Search Engine with BSP. In *Latin American Web Congress (LA-WEB)*, pages 259–268. IEEE, 2005. Page 10.
- [69] S. Coupet-Grimal. An Axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions. *Logic and Computation*, 13(6):801–813, 2003. Page 88.
- [70] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of programming languages (POPL)*, pages 238–252. ACM, 1977. Page 4.
- [71] C. J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2006. Page 88.
- [72] J. F. Cremers, P. Lafourcade, and P. Nadeau. Comparing State-spaces in Automatic Security Protocol Analysis. In *Formal to Practical Security*, volume 5458 of *LNCS*, pages 70–94. Springer, 2009. Page 88.
- [73] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Not.*, 28:1–12, 1993. Pages 10 and 11.
- [74] D. E. Culler, A. Dusseau, S. C. Goldstein, and A. Krishnamurthy. Parallel Programming in Split-C. In *SuperComputing (SC)*, 1993. Page 38.
- [75] R. da Rosa Righi, L. L. Pilla, A. Carissimi, P. O. A. Navaux, and H.-U. Heiss. MigBSP: A Novel Migration Model for Bulk-Synchronous Parallel Processes Rescheduling. In *High Performance Computing and Communications (HPCC)*, pages 585–590. IEEE, 2009. Pages 10, 17 and 29.
- [76] R. da Rosa Righi, L. L. Pilla, N. Maillard, A. Carissimi, and P. O. A. Navaux. Observing the Impact of Multiple Metrics and Runtime Adaptations on BSP Process Rescheduling. *Parallel Processing Letters*, 20(2):123–144, 2010. Page 10.
- [77] N. Dalal, J. Shah, K. Hisaria, and D. Jinwala. A Comparative Analysis of Tools for Verification of Security Protocols. *Int. J. Communications, Network and System Sciences*, 3:779–787, 2010. Page 88.
- [78] A. Danalis, L. Pollock, and M. Swamy. Automatic MPI Application Transformation with ASPhALT. In *Performance Optimization for High-Level Languages and Libraries (POHLL)*, in conjunction with *IPDPS*, 2007. Page 122.
- [79] M. Daum. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*, 2007. Page 68.
- [80] J. Dean and S. Ghemawat. MapReduce: a Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, 2010. Pages 8, 17 and 38.
- [81] F. Dehne. Special Issue on Coarse-grained Parallel Algorithms. *Algorithmica*, 14:173–421, 1999. Pages 9 and 121.
- [82] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Computational Geometry for Coarse Grained Multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996. Page 10.
- [83] J. den Hartog. Towards Mechanized Correctness Proofs for Cryptographic Algorithms: Axiomatization of a Probabilistic Hoare Style Logic. *Sci. Comput. Program.*, 74(1–2):52–63, 2008. Page 89.

- [84] N. Deo and P. Micikevicius. Coarse-Grained Parallelization of Distance-Bound Smoothing for the Molecular Conformation Problem. In S. K. Das and S. Bhattacharya, editors, *Distributed Computing, Mobile and Wireless Computing (IWDC)*, volume 2571 of *LNCS*, pages 55–66. Springer, 2002. Page 10.
- [85] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, Scalable Debugging of MPI programs with Intel® Message Checker. In *Software Engineering for High Performance Computing System Applications (SE-HPCS)*, pages 78–82. ACM, 2005. Pages 11 and 68.
- [86] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Parallel architectures and compilation techniques (PACT)*, pages 353–364. ACM, 2010. Page 15.
- [87] E. W. Dijkstra. Letters to the Editor: Goto Statement Considered Harmful. *Commun. ACM*, 11(3):147–148, 1968. Page 1.
- [88] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975. Page 5.
- [89] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. Page 77.
- [90] D. C. Dracopoulos and S. Kent. Speeding up Genetic Programming: A Parallel BSP Implementation. In *Genetic Programming (GP)*. MIT Press, 1996. Page 10.
- [91] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OPENMP Parallelization Models on SMP Clusters. In *Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–15, 2004. Page 8.
- [92] R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2), 1990. Page 6.
- [93] N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of Bounded Security Protocols. In *Formal Methods and Security Protocols (FMSP)*, part of *FLOC conference*, 1999. Page 77.
- [94] J. Eisenbiegler, W. Löwe, and W. Zimmermann. BSP, LogP, and oblivious programs. In *Euro-ParConference on Parallel Processing*, pages 865–874. Springer, 1998. Page 10.
- [95] T. A. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *High Performance Networking and Computing (SC)*, page 27. ACM Press, 2006. Page 38.
- [96] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A Fully Verified Executable LTL Model Checker. In *Computer Aided Verification (CAV)*, 2013. to appear. Pages 76, 87 and 88.
- [97] J. Falcou. Parallel Programming with Skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009. Page 8.
- [98] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste. QUAFF : Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, 2006. Pages 8 and 39.
- [99] P. Ferragina and F. Luccio. String Search in Coarse-Grained Parallel Computers. *Algorithmica*, 24(3):177–194, 1999. Page 10.
- [100] A. Ferreira, I. Guérin-Lassous, K. Marcus, and A. Rau-Chauplin. Parallel Computation on Interval Graphs: Algorithms and Experiments. *Concurrency and Computation: Practice and Experience*, 14(11):885–910, 2002. Page 10.
- [101] F.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4), 2003. Pages 13 and 44.
- [102] J.-C. Filliâtre. Program Verification using Coq. Introduction to the WHY tool, Aug. 2005. Lecture Notes, TYPES Summer School 2005 (Goteborg, Sweden). Page 92.
- [103] J.-C. Filliâtre. Verifying Two Lines of C with Why3: an Exercise in Program Verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, 2012. Page 13.
- [104] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004. Page 13.
- [105] J.-C. Filliâtre and C. Marché. The why/Krakatoa/Caduceus Platform for Deductive Program Verification. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, LNCS. Springer, 2007. Pages 13 and 121.
- [106] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded Parallelism in Manticore. *SIGPLAN Not.*, 43(9):119–130, 2008. Page 39.
- [107] M. Flynn. Some Computer Organizations and their Effectiveness. In *Trans. on Computers*, volume C-21(9), pages 948–960. IEEE, 1972. Page 6.
- [108] W. Fokink, M. T. Dashti, and A. Wijs. Partial Order Reduction for Branching Security Protocols. In L. Gomes, V. Khomenko, and J. M. Fernandes, editors, *Conference on Application of Concurrency to System Design (ACSD)*, pages 191–200. IEEE Computer Society, 2010. Page 83.
- [109] J. Ford and N. Shankar. Formal Verification of a Combination Decision Procedure. In A. Voronkov, editor, *Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 347–362. Springer, 2002. Page 87.
- [110] J. Fortin and F. Gava. BSP-why: An Intermediate Language for Deductive Verification of BSP Programs. In *High-Level Parallel Programming and Applications (HLPP)*, pages 35–44. ACM, 2010. Pages 41, 71 and 108.
- [111] J. Fortin and F. Gava. From BSP Routines to High-performance Ones: Formal Verification of a Transformation Case. *Procedia Computer Science*, 1(1):155–164, 2010. Pages 42 and 122.
- [112] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 276–287, 1997. Page 11.
- [113] L. Fronc and F. Pommereau. Towards a Certified Petri Net Model-Checker. In H. Yang, editor, *Programming Languages and Systems (APLAS)*, volume 7078 of *LNCS*, pages 322–336. Springer, 2011. Pages 88 and 123.

- [114] C. A. Furia and B. Meyer. Inferring Loop Invariants Using Postconditions. In A. Blass, N. Dershowitz, and W. Reisig, editors, *Fields of Logic and Computation*, volume 6300 of *LNCS*, pages 277–300. Springer, 2010. Pages 69 and 119.
- [115] H. Gao. *Analysis of Security Protocols by Annotations*. PhD thesis, Technical University of Denmark, 2008. Page 89.
- [116] H. Garavel, R. Mateescu, and I. M. Smarandache. Parallel State Space Construction for Model-Checking. In M. B. Dwyer, editor, *Proceedings of SPIN*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001. Pages 76, 79 and 89.
- [117] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003. Pages 12, 67 and 69.
- [118] F. Gava. Une Bibliothèque Certifiée de Programmes Fonctionnels BSP. *Technique et Science Informatiques*, 25(10):1261–1280, 2006. Page 12.
- [119] F. Gava and J. Fortin. Formal Semantics of a Subset of the Paderborn’s BSPLIB. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE Press, 2008. Pages 34, 69 and 91.
- [120] F. Gava, J. Fortin, and M. Guedj. Deductive Verification of State-space Algorithms. In *integrated Formal Methods (iFM)*, LNCS. Springer, 2013. to appear. Page 71.
- [121] F. Gava, M. Guedj, and F. Pommereau. A BSP Algorithm for the State Space Construction of Security Protocols. In *Parallel and Distributed Methods in veriFiCation (PDMC)*, pages 37–44. IEEE Computer Society, 2010. Pages 77, 82 and 83.
- [122] F. Gava, M. Guedj, and F. Pommereau. Performance Evaluations of a BSP Algorithm for State-space Construction of Security Protocols. In R. Stotzka, M. Schiffers, and Y. Cotronis, editors, *Parallel, Distributed and Network-Based Processing (PDP)*, pages 170–174. IEEE, 2012. Pages 83 and 85.
- [123] F. Gava, A. Hidalgo, and J. Fortin. Mechanised Verification of Distributed State-space Algorithms for Security Protocols. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE Press, 2012. Page 71.
- [124] F. Gava and J. Fortin. Two Formal Semantics of a Subset of the Paderborn University BSPLIB. In *Parallel, Distributed and Network-Based Processing (PDP)*. IEEE Press, 2009. Pages 34, 91 and 97.
- [125] A. V. Gerbessiotis. *Topics in Parallel and Distributed Computation*. PhD thesis, Harvard University, 1993. Page 10.
- [126] A. V. Gerbessiotis, C. J. Siniolakis, and A. Tiskin. Parallel priority queue and list contraction: The BSP approach. *Computing and Informatics*, 21:59–90, 2002. Page 10.
- [127] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994. Page 10.
- [128] R. Gerth and W. P. de Roever. A Proof System for Concurrent ADA Programs. *Sci. Comput. Program.*, 4(2):159–204, 1984. Page 67.
- [129] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski. Bulk Synchronous Parallel ML with Exceptions. *Future Generation Computer Systems*, 26:486–490, 2010. Pages 17 and 31.
- [130] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE, 2010. Pages 12 and 69.
- [131] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical report, Intel Research, 2007. Pages 8, 17 and 25.
- [132] A. Goldman, D. Cordeiro, and A. Kraemer. The suitability of BSP/CGM model for Clouds. In *High Performance Computing, Grids and Clouds (HPC)*, 2012. Pages 33 and 34.
- [133] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Software, Practice & Experience*, 40(12):1135–1160, 2010. Page 8.
- [134] G. Gopalakrishnan and R. M. Kirby. Runtime verification methods for MPI. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–5. IEEE, 2008. Page 11.
- [135] G. Gopalakrishnan, R. M. Kirby, S. F. Siegel, R. Thakur, W. Gropp, E. L. Lusk, B. R. de Supinski, M. Schulz, and G. Bronevetsky. Formal Analysis of MPI-based Parallel Programs: Present and Future. *Commun. ACM*, 54(12):82–91, 2011. Pages 2, 11 and 68.
- [136] S. Gorbach. Send-receive Considered Harmful: Myths and Realities of Message Passing. *ACM TOPLAS*, 26(1):47–56, 2004. Pages 1, 2, 10 and 117.
- [137] S. Gorbach and M. Cole. Parallel Skeletons. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1417–1422. Springer, 2011. Page 8.
- [138] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular Shape Analysis. In *Programming Language Design and Implementation (PLDI)*, 2007. Page 67.
- [139] J. Goubault-Larrecq. Finite Models for Formal Security Proofs. *Journal of Computer Security*, 18(6):1247–1299, 2010. Page 89.
- [140] L. Graebin and R. da Rosa Righi. jMigBSP: Object Migration and Asynchronous One-Sided Communication for BSP Applications. In S.-S. Yeo, B. Vaidya, and G. A. Papadopoulos, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 35–38. IEEE Computer Society, 2011. Pages 17 and 29.
- [141] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A System for the High-level Parallelization and Cooperation of Constraint Solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601. IASTED/ACTA Press, 1998. Page 10.
- [142] C. Grelck and S.-B. Scholz. Classes and Objects as Basis for I/O in SAC. In *Implementation of Functional Languages (IFL)*, pages 30–44, 1995. Page 39.

- [143] I. Grudenic and N. Bogunovic. Modeling and Verification of MPI Based Distributed Software. In B. Mohr, J. L. Träff, J. Worringer, and J. Dongarra, editors, *Recent Advances in PVM and MPI, User's Group Meeting*, volume 4192 of *LNCS*, pages 123–132. Springer, 2006. Page 11.
- [144] Y. Gu, B.-S. Le, and C. Wentong. JBSP: A BSP programming library in JAVA. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001. Pages 15 and 27.
- [145] I. Guérin-Lassous and J. Gustedt. Portable List Ranking: an Experimental Study. *ACM Journal of Experiments Algorithms*, 7(7):1–18, 2002. Page 10.
- [146] J. Guitton, J. Kanig, and Y. Moy. WHY Hi-Lite ADA? In *Boogie Workshop*, 2011. Page 13.
- [147] K. Hamidouche, A. Borghi, P. Esterie, J. Falcou, and S. Peyronnet. Three High Performance Architectures in the Parallel APMC Boat. In *Parallel and Distributed Methods in Verification (PDMC)*, pages 20–27, 2010. Page 8.
- [148] K. Hamidouche, J. Falcou, and D. Etiemble. A Framework for an Automatic Hybrid MPI + OPENMP Code Generation. In L. T. Watson, G. W. Howell, W. I. Thacker, and S. Seidel, editors, *Simulation Multi-conference (SpringSim) on High Performance Computing Symposia (HPC)*, pages 48–55. SCS/ACM, 2011. Pages 17, 32 and 123.
- [149] K. Hammond. Parallel Functional Programming: An Introduction. In *International Symposium on Parallel Symbolic Computation*. World Scientific, 1994. Page 38.
- [150] K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, 2003. Page 8.
- [151] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 2000. Page 38.
- [152] Y. Hayashi and M. Cole. Automated BSP Cost Analysis of a Parallel Maximum Segment Sum Program Derivation. *Parallel Processing Letters*, 12(1):95–112, 2002. Page 39.
- [153] B. Hendrickson. Computational Science: Emerging Opportunities and Challenges. *Journal of Physics: Conference Series*, 180(1), 2009. Pages 1 and 10.
- [154] D. S. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *SuperComputing (SC)*, 2000. Page 8.
- [155] P. Herms. Certification of a Chain for Deductive Program Verification. In Y. Bertot, editor, *COQ Workshop, satellite of ITP*, 2010. Pages 88, 116, 118 and 119.
- [156] P. Herms. *Certification of a Tool Chain for Deductive Program Verification*. PhD thesis, Université Paris-Sud (LRI), 2013. Pages 119 and 122.
- [157] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPLIB: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998. Pages 15, 21 and 22.
- [158] J. M. D. Hill and D. B. Skillicorn. Practical Barrier Synchronisation. In *Parallel and Distributed Processing (PDP)*. IEEE, 1998. Page 8.
- [159] K. Hinsen, H. P. Langtangen, O. Skavhaug, and Å. Ødegård. Using BSP and PYTHON to Simplify Parallel Programming. *Future Generation Comp. Syst.*, 22(1-2):123–157, 2006. Pages 17 and 32.
- [160] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969. Pages 4, 5 and 12.
- [161] C. A. R. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The Verified Software Initiative: A Manifesto. *ACM Comput. Surv.*, 41(4), 2009. Page 1.
- [162] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *European Symposium on Programming (ESOP)*, number 4960 in *LNCS*, pages 353–367. Springer, 2008. Page 67.
- [163] A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic: Now With Tool Support! *Logical Methods in Computer Science*, 8(2), 2012. Page 116.
- [164] G. Horvitz and R. H. Bisseling. Designing a BSP version of ScaLAPACK. In B. H. et al., editor, *Conference on Parallel Processing for Scientific Computing*. SIAM, 1999. Page 10.
- [165] Q. Hou, K. Zhou, and B. Guo. BSGP: Bulk-Synchronous GPU Programming. *ACM Trans. Graph.*, 27(3), 2008. Pages 8, 10, 17 and 26.
- [166] C.-H. Huang and S. Rajasekaran. High-performance Parallel Bio-computing. *Parallel Computing*, 30(9-10):999–1000, 2004. Page 10.
- [167] A. Iosup, O. Sonmez, S. Anoep, and D. Epema. The Performance of Bags-of-tasks in Large-scale Distributed Systems. In *Symposium on High performance distributed computing (HPDC)*, pages 97–108. ACM, 2008. Page 85.
- [168] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and JAVA. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011. Page 68.
- [169] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A Statically Verifiable Programming Model for Concurrent Object-oriented Programs. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006. Page 68.
- [170] H. Jifeng, Q. Miller, and L. Chen. Algebraic Laws for BSP Programming. In L. Bouge and Y. Robert, editors, *Euro-Par*, number 1124 in *LNCS*, pages 359–368. Springer, 1996. Page 70.
- [171] C. B. Jones, P. W. O'Hearn, and J. Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, 2006. Page 1.
- [172] S. A. Joseph. Evaluating Threading Building Blocks Pipelines, 2007. Page 7.

- [173] L. Kale. Charm++. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011. Page 38.
- [174] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software (2nd Ed.)*. John Wiley, 1999. Page 3.
- [175] J. Kanig and J.-C. Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, 2009. Page 67.
- [176] C. W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. *The Journal of Supercomputing*, 17(3):245–262, 2000. Pages 17 and 30.
- [177] T. Kielmann and S. Gorlatch. Bandwidth-Latency Models (BSP, LogP). In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 107–112. Springer, 2011. Page 8.
- [178] W. J. Knottenbelt, M. A. Mestern, P. G. Harrison, and P. Kritzing. Probability, Parallelism and the State Space Exploration Problem. In R. Puigjaner, N. N. Savino, and B. Serra, editors, *Computer Performance Evaluation-Modeling, Techniques and Tools (TOOLS)*, number 1469 in LNCS, pages 165–179. Springer-Verlag, 1998. Page 89.
- [179] L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of LNCS, pages 470–485. Springer, 2009. Page 12.
- [180] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI Analysis and Checking Tool. *Parallel Computing*, 2003. Pages 11 and 68.
- [181] P. Krusche and A. Tiskin. New Algorithms for Efficient Parallel String Comparison. In F. Meyer auf der Heide and C. A. Phillips, editors, *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 209–216. ACM, 2010. Page 10.
- [182] H. Kuchen and M. Cole. The Integration of Task and Data Parallel Skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002. Pages 8 and 39.
- [183] H. Kuchen and J. Striegnitz. Features from Functional Programming for a C++ Skeleton Library. *Concurrency - Practice and Experience*, 17(7-8):739–756, 2005. Page 8.
- [184] A. Kukanov. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4):309–322, 2007. Pages 7 and 25.
- [185] L. Lamport and F. B. Schneider. The Hoare Logic of CSP, and All That. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296, 1984. Page 68.
- [186] I. G. Lassous. *Algorithmes Paralleles de Traitement de Graphes: une Approche Basee sur l'Analyse Experimentale*. PhD thesis, University de Paris VII, 1999. Page 10.
- [187] D. S. Lecomber. A Semantics for Parallel Programming with BSP. Page 116.
- [188] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006. Pages 2, 7, 10, 68 and 117.
- [189] K. R. M. Leino. Verifying Concurrent Programs with Chalice. In G. Barthe and M. V. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5944 of LNCS, page 2. Springer, 2010. Page 68.
- [190] K. R. M. Leino and P. Muller. A Basis for Verifying Multi-threaded Programs. In *ESOP*, LNCS. Springer, 2009. Page 68.
- [191] F. Lerda and R. Sista. Distributed-Memory Model Checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of SPIN*, number 1680 in LNCS, pages 22–39. Springer-Verlag, 1999. Page 89.
- [192] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009. Pages 116, 118, 122 and 123.
- [193] X. Leroy and H. Grall. Coinductive Big-step Operational Semantics. *Inf. Comput.*, 207(2):284–304, 2009. Pages 93 and 101.
- [194] M. Leyton and J. M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *PDP*, pages 289–296. IEEE, 2010. Pages 8 and 39.
- [195] C. Li and G. Hains. SGL: Towards a Bridging Model for Heterogeneous Hierarchical Platforms. *IJHPCN*, 7(2):139–151, 2012. Page 123.
- [196] G. LI. *On-the-fly Model Checking of Security Protocols*. PhD thesis, Japan Advanced Institute of Science and Technology, 2008. Page 88.
- [197] G. Li, M. Delisi, G. Gopalakrishnan, and R. M. Kirby. Formal Specification of the MPI-2.0 Standard in TLA+. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 283–284. ACM SIGPLAN, 2008. Pages 11, 13 and 68.
- [198] X. Liu, W. Tong, and Y. Hou. BSPCloud: A Programming Model for Cloud Computing. In *Computer and Information Technology (CIT)*, pages 1109–1113. IEEE Computer Society, 2012. Pages 33 and 34.
- [199] A. Lluch-Lafuente. Simplified Distributed Model-checking by Localizing Cycles. Technical Report 176, Institute of Computer Science at Freiburg University, 2002. Page 89.
- [200] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, A. J. Rebón, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher Order and Symb. Comp.*, 16(3):203–251, 2003. Page 38.
- [201] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of LNCS, pages 147–166. Springer, 1996. Page 88.
- [202] G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998. Page 88.

- [203] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003. Pages 11 and 68.
- [204] B. Maggs, L. Matheson, and R. Tarjan. Models of Parallel Computation: A Survey and Synthesis. In *Hawaii International Conference on Systems Sciences*, 1995. Page 8.
- [205] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, and N. Leiser. Pregel: A System for Large-scale Graph Processing. In *Management of data*, pages 135–146. ACM, 2010. Pages 17 and 33.
- [206] A. Malkis and A. Banerjee. Verification of Software Barriers. In J. Ramanujam and P. Sadayappan, editors, *Principles and Practice of Parallel Programming (PPOPP)*, pages 313–314. ACM, 2012. Page 116.
- [207] A. Malkis and L. Mauborgne. On the Strength of Owicki-Gries for Resources. In H. Yang, editor, *Programming Languages and Systems (APLAS)*, volume 7078 of *LNCS*, pages 172–187. Springer, 2011. Page 67.
- [208] W. R. Marrero. *Brutus: a Model-checker for Security Protocols*. PhD thesis, Carnegie Mellon University, 2001. Page 88.
- [209] J. M. R. Martin and Y. Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. In P. H. Welch and A. W. P. Bakkers, editors, *Communicating Process Architectures*, pages 1–14, 2000. Page 89.
- [210] D. R. Martinez, J. C. Cabaleiro, T. F. Pena, F. F. Rivera, and V. Blanco. Accurate Analytical Performance Model of Communications in MPI applications. In *Parallel and Distributed Processing Symposium, International (IPDPS)*, pages 1–8. IEEE, 2009. Page 10.
- [211] B. D. Martino, A. Mazzeo, M. Mazzocca, and U. Villano. Parallel Program Analysis and Restructuring by Detection of Point-to-point Interaction Patterns and their Transformation into Collective Communication Constructs. *Science of Computer Programming*, 40(2–3):235–263, 2001. Pages 17, 69 and 122.
- [212] K. Matsuzaki. Efficient Implementation of Tree Accumulations on Distributed-Memory Parallel Computers. In *Practical Aspects of High-Level Parallel Programming (PAPP)*, part of *The International Conference on Computational Science (ICCS)*, 2007. Page 10.
- [213] S. Meier, C. J. F. Cremers, and D. A. Basin. Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs. In *Computer Security Foundations (CSF)*, pages 231–245. IEEE Computer Society, 2010. Page 89.
- [214] Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, 2002. Page 17.
- [215] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. In *Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society, 1997. Page 88.
- [216] K. S. Namjoshi. Certifying Model Checkers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 2–13. Springer, 2001. Pages 75, 76, 77 and 88.
- [217] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the Awkward Squad. In *International Conference on Functional Programming (ICFP)*, 2008. Page 67.
- [218] G. C. Necula. Proof-Carrying Code. In *Principles of Programming Languages (POPL)*, pages 106–119. ACM, 1997. Page 75.
- [219] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008. Page 7.
- [220] L. P. Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Institut für Informatik, Technische Universität München, 2001. Pages 67 and 116.
- [221] L. P. Nieto and J. Esparza. Verifying Single and Multi-mutator Garbage Collectors with Owicki-Gries in Isabelle/HOL. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of *LNCS*, pages 619–628. Springer, 2000. Page 67.
- [222] T. Nipkow and L. P. Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE)*, part of *the European Joint Conferences on the Theory and Practice of Software (ETAPS)*, volume 1577 of *LNCS*, pages 188–203. Springer, 1999. Page 67.
- [223] P. W. O’Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007. Page 67.
- [224] P. Ohly and W. Krotz-Vogel. Automated MPI Correctness Checking: What if There Were a Magic Option? In *High-Performance Clustered Computing*, 2007. Page 11.
- [225] S. Orzan, J. van de Pol, and M. Espada. A State Space Distributed Policy Based on Abstract Interpretation. In *ENTCS*, volume 128, pages 35–45. Elsevier, 2005. Page 89.
- [226] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, 1976. Page 67.
- [227] M. F. Pace. BSP vs MapReduce. *Procedia CS*, 9:246–255, 2012. Page 17.
- [228] C. Pajault. *Model Checking Parallèle et Réparti de Réseaux de Petri Colorés de Haut-niveau*. PhD thesis, Conservatoire National des Arts et Métiers, 2008. Page 89.
- [229] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998. Page 88.
- [230] D. Peled, A. Pnueli, and L. D. Zuck. From Falsification to Verification. In R. Hariharan, M. Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2245 of *LNCS*, pages 292–304. Springer, 2001. Pages 87 and 88.

- [231] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Formal Verification of Programs that use MPI One-sided Communication. In *Recent Advances in PVM and MPI (Euro PVM/MPI)*, volume 4192 of *LNCS*, pages 30–39. Springer, 2006. Page 68.
- [232] D. Petcu. Parallel Explicit State Reachability Analysis and State-space Construction. In *International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 207–214. IEEE Computer Society, 2003. Page 89.
- [233] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software Verification with VeriFast: Industrial Case Studies. *Science of Computer Programming*, 2013. to appear. Page 68.
- [234] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. *Cluster Computing*, 10:127–143, 2007. Page 10.
- [235] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003. Page 8.
- [236] M. Redekopp, Y. Simmhan, and V. K. Prasanna. Optimizations and Analysis of BSP Graph Processing Models on Public Clouds. In *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2013. to appear. Pages 33 and 34.
- [237] Y. Régis-Gianas and F. Pottier. A Hoare Logic for Call-by-Value Functional Programs. In *Mathematics of Program Construction (MPC)*, pages 305–335, 2008. Page 67.
- [238] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. Page 39.
- [239] X. Rival and J. Goubault-Larrecq. Experiments with Finite Tree Automata in COQ. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 2152 of *LNCS*, pages 362–377. Springer, 2001. Page 87.
- [240] R. O. Rogers and D. B. Skillicorn. Using the BSP Cost Model to Optimise Parallel Neural Network Training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998. Page 10.
- [241] M. Rusinowith and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *Computer Security Foundations Workshop (CSFW)*, pages 174–190. IEEE, 2001. Page 77.
- [242] F. K. A. Salem. *Factorisation Algorithms for Univariate and Bivariate Polynomials over Finite Fields*. PhD thesis, University of Oxford (Merton College), 2004. Page 10.
- [243] F. K. A. Salem and L. T. Yang. Parallel Methods for Absolute Irreducibility Testing. *The Journal of Supercomputing*, 46(3):181–212, 2008. Page 10.
- [244] S. Sanjabi and F. Pommereau. Modelling, Verification, and Formal Analysis of Security Properties in a P2P System. In *Collaboration and Security (COLSEC'10)*, pages 543–548. IEEE, 2010. Page 77.
- [245] A. Schimpf, S. Merz, and J.-G. Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*. Springer, 2009. Page 88.
- [246] S. Seo, E. J. Yoon, J.-H. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *Cloud Computing (CloudCom)*, pages 721–726. IEEE, 2010. Pages 17, 27 and 121.
- [247] N. Shankar. Trust and Automation in Verification Tools. In S. D. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 5311 of *LNCS*, pages 4–17. Springer, 2008. Pages 75 and 76.
- [248] N. Shankar and M. Vaucher. The Mechanical Verification of a DPLL-Based Satisfiability Solver. *Electr. Notes Theor. Comput. Sci.*, 269:3–17, 2011. Page 87.
- [249] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. A Formal Approach to Detect Functionally Irrelevant Barriers in MPI Programs. In *Recent Advances in PVM and MPI (Euro PVM/MPI)*, volume 5205 of *LNCS*, pages 265–273. Springer, 2008. Page 68.
- [250] S. V. Sharma, G. Gopalakrishnan, and R. M. Kirby. A Survey of MPI Related Debuggers and Tools. Technical Report UUCS-07-015, University of Utah, School of Computing, 2007. Page 68.
- [251] S. F. Siegel. Verifying Parallel Programs with MPI-Spin. In F. Cappello, T. Hérault, and J. Dongarra, editors, *Euro PVM/MPI*, volume 4757 of *LNCS*, pages 13–14. Springer, 2007. Pages 11, 68 and 122.
- [252] S. F. Siegel and G. S. Avrunin. Verification of Halting Properties for MPI Programs Using Nonblocking Operations. In F. Cappello, T. Hérault, and J. Dongarra, editors, *Recent Advances in PVM and MPI (Euro PVM/MPI)*, volume 4757 of *LNCS*, pages 326–334. Springer, 2007. Page 11.
- [253] S. F. Siegel, A. M., G. S. Avrunin, and L. A. Clarke. Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–34, 2008. Page 68.
- [254] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008. Pages 69 and 119.
- [255] S. F. Siegel and T. K. Zirkel. FEVS: A Functional Equivalence Verification Suite for High-Performance Scientific Computing. *Mathematics in Computer Science*, 5(4):427–435, 2011. Page 69.
- [256] S. F. Siegel and T. K. Zirkel. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science*, 5(4):395–426, 2011. Page 69.
- [257] S. F. Siegel and T. K. Zirkel. Loop Invariant Symbolic Execution for Parallel Programs. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7148 of *LNCS*, pages 412–427. Springer, 2012. Page 69.

- [258] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. Pages 8 and 9.
- [259] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998. Pages 7, 9 and 20.
- [260] D. X. Song, S. Berezin, and A. Perrig. Athena: A Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001. Page 88.
- [261] C. Sprenger. A Verified Model Checker for the Modal μ -calculus in COQ. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 167–183. Springer-Verlag, 1998. Pages 76, 87 and 88.
- [262] J. M. Squyres. Zen and the Art of MPI Collectives. *ClusterWorld Magazine*, MPI Mechanic Column, 2(3):32–34, 2004. Page 69.
- [263] U. Stern and D. L. Dill. Parallelizing the μ rv Verifier. *Formal Methods in System Design*, 18(2):117–129, 2001. Page 88.
- [264] A. Stewart. A Programming Model for BSP with Partitioned Synchronisation. *Formal Asp. Comput.*, 23(4):421–432, 2011. Pages 70 and 123.
- [265] A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14:271–292, 2000. Page 70.
- [266] A. Stewart, M. Clint, and J. Gabarró. Barrier Synchronisation: Axiomatisation and Relaxation. *Formal Asp. Comput.*, 16(1):36–50, 2004. Page 70.
- [267] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT Proof Checking using a Logical Framework. *Formal Methods in System Design*, 42(1):91–118, 2013. Page 88.
- [268] J. Sun, Y. Liu, and B. Cheng. Model Checking a Model Checker: A Code Contract Combined Approach. In J. S. Dong and H. Zhu, editors, *Formal Engineering Methods (ICFEM)*, volume 6447 of *LNCS*, pages 518–533. Springer, 2010. Pages 76 and 88.
- [269] L. Tan and R. Cleaveland. Evidence-Based Model Checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 455–470. Springer, 2002. Pages 87 and 88.
- [270] J. Tesson and F. Loulergue. Formal Semantics for the DRMA Programming Style Subset of the BSPLIB Library. In J. Weglarz, R. Wyrzykowski, and B. Szymanski, editors, *Parallel Processing and Applied Mathematics (PPAM)*, number 4967 in *LNCS*, pages 1122–1129. Springer, 2007. Page 97.
- [271] J. Tesson and F. Loulergue. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In *International Conference on Computational Science (ICCS)*, *Procedia Computer Science*, pages 36–45. Elsevier, 2011. Pages 12 and 69.
- [272] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998. Pages 10, 68 and 73.
- [273] J. L. Traeff and J. Worringer. Verifying Collective MPI Calls. In *EuroPVM/MPI conference*, *LNCS*. Springer, 2004. Page 68.
- [274] M.-H. Tsai and B.-Y. Wang. Formalization of CTL* in the Calculus of Inductive Constructions. In M. Okada and I. Satoh, editors, *Asian computing science conference on Advances in computer science: secure software and related issues (ASIAN)*, volume 4435 of *LNCS*, pages 316–330. Springer-Verlag, 2007. Pages 88 and 123.
- [275] E. Turner, M. Butler, and M. Leuschel. A Refinement-based Correctness Proof of Symmetry Reduced Model-checking. In *Abstract State Machines, Alloy, B and Z*, *LNCS*, pages 231–244. Springer, 2010. Pages 87 and 123.
- [276] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced Execution Semantics of MPI: From Theory to Practice. In *Formal Methods (FM)*, 2009. Page 68.
- [277] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: a Tool for Model Checking MPI Programs. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 285–286. ACM, 2008. Page 11.
- [278] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. of the ACM*, 33(8):103–111, 1990. Pages 8 and 11.
- [279] L. G. Valiant. A bridging model for multi-core computing. In *European Symposium on Algorithms (ESA)*, pages 13–28. Springer-Verlag, 2008. Page 123.
- [280] K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in COQ. In J. He and M. Sato, editors, *Advances in Computing Science - 6th Asian Computing Science Conference (ASIAN)*, volume 1961 of *LNCS*, pages 162–181. Springer, 2000. Page 87.
- [281] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *SuperComputing (SC)*. IEEE, 2000. Pages 11 and 68.
- [282] J. Villard. *Heaps and Hops*. PhD thesis, École Normale Supérieure de Cachan, LSV, France, 2011. Page 69.
- [283] J. Villard, É. Lozes, and C. Calcagno. Proving Copyless Message Passing. In Z. Hu, editor, *Programming Languages and Systems (APLAS)*, volume 5904 of *LNCS*, pages 194–209. Springer, 2009. Pages 69 and 116.
- [284] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal Verification of Practical MPI Programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 261–269, 2009. Pages 11, 68, 69 and 122.
- [285] F. Vogels, B. Jacobs, and F. Piessens. A Machine-checked Soundness Proof for an Efficient Verification Condition Generator. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *Symposium on Applied Computing (SAC)*, pages 2517–2522, 2010. Page 119.

- [286] T. L. Williams and R. J. Parsons. The Heterogeneous Bulk Synchronous Parallel Model. In *Parallel and Distributed Processing (IPDPS)*, LNCS, pages 102–108. Springer-Verlag, 2000. Page 123.
- [287] A. N. Yzelman and R. H. Bisseling. An Object-oriented Bulk Synchronous Parallel Library for Multicore Programming. *Concurrency and Computation: Practice and Experience*, 24(5):533–553, 2012. Pages 8, 15 and 28.
- [288] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen. MulticoreBSP for C: A High-performance Library for Shared-memory Parallel Programming. *Journal of Parallel Programming*, 2013. to appear. Pages 15 and 25.
- [289] A. Zavanella. Skeletons, BSP and Performance Portability. *Parallel Processing Letters*, 11(4):393–407, 2001. Page 39.
- [290] J. Zhou and Y. Chen. Generating C Code from LOGS Specifications. In D. V. Hung and M. Wirsin, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 3722 of *LNCS*, pages 195–210. Springer, 2005. Page 69.

Softwares

- 1 **Open-MP**: <http://openmp.org/wp/>
- 2 **TBB**: <http://threadingbuildingblocks.org>
- 3 **CUDA**: <http://developer.nvidia.com/category/zone/cuda-zone>
- 4 **MPI**: <http://www.mcs.anl.gov/research/projects/mpi/>
- 5 **OCamlP3L**: <http://ocamlp3l.inria.fr/>
- 6 **TLA+**: <http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html>
- 7 **Marmot**: <http://www.hlrs.de/organization/av/amt/projects/marmot/>
- 8 **MPI-SPIN**: <http://vsl.cis.udel.edu/mpi-spin/index.html>
- 9 **ISP**: http://www.cs.utah.edu/formal_verification/ISP-Release/
- 10 **DAMPI**: http://www.cs.utah.edu/formal_verification/DAMPI/
- 11 **Why**: <http://why.lri.fr/index.html>
- 12 **Coq**: <http://coq.inria.fr/>
- 13 **PVS**: <http://pvs.csl.sri.com/>
- 14 **Isabelle**: <http://isabelle.in.tum.de/>
- 15 **HOL**: <http://hol.sourceforge.net/>
- 16 **Simplify**: <http://www.hpl.hp.com/downloads/crl/jtk/index.html>
- 17 **Alt-Ergo**: <http://ergo.lri.fr/>
- 18 **Z3**: <http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html>
- 19 **CVC3**: <http://cs.nyu.edu/acsys/cvc3/>
- 20 **Yices**: <http://yices.csl.sri.com/>
- 21 **Vampire**: <http://www.vprover.org/>
- 22 **Frama-C**: <http://frama-c.com/>
- 23 **BSPlib**: <http://www.bsp-worldwide.org/implmnts/oxtool/>
- 24 **CGMlib**: <http://www.scs.carleton.ca/cgm/>
- 25 **BSPonMPI**: <http://bsponmpi.sourceforge.net/>
- 26 **PUB**: <http://publibrary.sourceforge.net/>
- 27 **BSPMultiCore**: <http://www.multicorebsp.com/>
- 28 **Hama**: <http://hama.apache.org/>
- 29 **Pregel**: <http://googleresearch.blogspot.fr/2009/06/large-scale-graph-computing-at-google.html>
- 30 **NestStep**: <http://www.ida.liu.se/~chrke/neststep/index.html>
- 31 **BSPonGPU**: <http://kunzhou.net/BSGP/BSGP-package.zip>
- 32 **BSP++**: <https://github.com/khamidouche/BSPPP>
- 33 **BSP-Python**: <http://dirac.cnrs-orleans.fr/plone/software/scientificpython/>
- 34 **Hadoop**: <http://hadoop.apache.org/>
- 35 **PUB-WEB**: <http://pubweb.sourceforge.net/>
- 36 **ProActive**: <http://proactive.inria.fr>
- 37 **BSMLlib**: <http://bsmlib.free.fr>
- 38 **OCaml**: <http://caml.org>
- 39 **Giraph**: <http://incubator.apache.org/giraph/>
- 40 **GoldenOrb**: <http://goldenorbos.org/>
- 41 **Split-C**: <http://www.eecs.berkeley.edu/Research/Projects/CS/parallel/castle/split-c/>
- 42 **UPC**: <http://upc.gwu.edu/>
- 43 **Cilk**: <http://supertech.csail.mit.edu/cilk/>
- 44 **Fortress**: <http://projectfortress.java.net/>
- 45 **ZPL**: <http://www.cs.washington.edu/research/zpl/home/index.html>
- 46 **Orca**: <http://www.cs.vu.nl/orca/>
- 47 **Co-Array Fortran**: <http://www.co-array.org/>
- 48 **Occam**: pop-users.org/wiki/occam-pi
- 49 **Charm++**: <http://charm.cs.illinois.edu/software>
- 50 **NESL**: <http://www.cs.cmu.edu/~scandal/nesl.html>
- 51 **Nepal**: <http://homepages.inf.ed.ac.uk/wadler/realworld/nepal.html>
- 52 **Manticore**: <http://manticore.cs.uchicago.edu/>
- 53 **Concurrent ML**: <http://cml.cs.uchicago.edu/>
- 54 **SAC**: www.sac-home.org/
- 55 **Data-Parallel Haskell**: http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

- 56 **Who**: <https://github.com/kanigsson/who>
- 57 **Boogie**: <http://research.microsoft.com/en-us/projects/boogie/>
- 58 **VCC**: <http://research.microsoft.com/en-us/projects/vcc/default.aspx>
- 59 **Verifast**: <http://people.cs.kuleuven.be/~bart.jacobs/verifast/>
- 60 **HPCBugDatabase**: <http://www.hpcbugbase.org/>
- 61 **TASS**: <http://vsl.cis.udel.edu/tass/index.html>
- 62 **FEVS**: <http://vsl.cis.udel.edu/fevs/index.html>
- 63 **Heap-hop**: <http://www.lsv.ens-cachan.fr/Software/heap-hop/>
- 64 **ProVerif**: <http://www.proverif.ens.fr/>
- 65 **Scyther**: <http://people.inf.ethz.ch/cremersc/scyther/>
- 66 **AVISPA**: <http://www.avispa-project.org/>

Résumé. Cette thèse s'inscrit dans le domaine de la vérification formelle de programmes parallèles. L'enjeu de la vérification formelle est de s'assurer qu'un programme va bien fonctionner comme il le devrait, sans commettre d'erreur, se bloquer, ou se terminer anormalement. Cela est d'autant plus important dans le domaine du calcul parallèle, où le coût des calculs est parfois très élevé. Le modèle BSP (Bulk Synchronous Parallelism) est un modèle de parallélisme bien adapté à l'utilisation des méthodes formelles. Il garantit une forme de structure dans le programme parallèle, en l'organisant en super-étapes où chacune d'entre elle est composée d'une phase de calculs, puis d'une phase de communications entre les unités de calculs. Dans cette thèse, nous avons choisi d'étendre un outil actuel pour l'adapter à la preuve de programmes BSP. Nous nous sommes basés sur WHY, un VCG (générateur de condition de vérification) qui a l'avantage de pouvoir s'interfacer avec plusieurs prouveurs automatiques et assistants de preuve pour décharger les obligations de preuves. Les contributions de cette thèse sont multiples. Dans un premier temps, nous présentons une comparaison des différentes bibliothèques BSP disponibles, afin de mettre en évidence les primitives de programmation BSP les plus utilisées, donc les plus intéressantes à formaliser. Nous présentons ensuite BSP-WHY, notre outil de preuve des programmes BSP. Cet outil repose sur la génération d'un programme séquentiel qui simule le programme parallèle, permettant ainsi d'utiliser WHY et les nombreux prouveurs automatiques associés pour prouver les obligations de preuves. Nous montrons ensuite comment BSP-WHY peut-être utilisé pour prouver la correction de quelques algorithmes BSP simples, mais aussi pour un exemple plus complexe qu'est la construction distribuée de l'espace d'états (model-checking) de systèmes et plus particulièrement dans les protocoles de sécurité. Enfin, pour garantir la plus grande confiance possible dans l'outil BSP-WHY, nous formalisons les sémantiques du langage dans l'assistant de preuve COQ. Nous démontrons également la correction de la transformation utilisée pour passer d'un programme parallèle à un programme séquentiel.

Mots clefs. Parallélisme, BSP, Preuves Dédicatives, WHY, VCG, State-space, Model-checking, Sémantiques formelles, COQ

Abstract. This thesis falls within the formal verification of parallel programs. The aim of formal verification is to ensure that a program will run as it should, without making mistakes, blocking, or terminating abnormally. This is even more important in the parallel computation field, where the cost of calculations can be very high. The BSP model (Bulk Synchronous Parallelism) is a model of parallelism well suited for the use of formal methods. It guarantees a structure in the parallel program, by organising it into super-steps, each of them consisting in a phase of computations, followed by communications between the processes. In this thesis, we chose to extend an existing tool to adapt it for the proof of BSP programs. We based our work on WHY, a VCG (verification condition generator) that has the advantage of being able to interface with several automatic provers and proof assistants to discharge the proof obligations. There are multiple contributions in this thesis. In a first part, we present a comparison of the existing BSP libraries, in order to show the most used BSP primitives, which are the most interesting to formalise. We then present BSP-WHY, our tool for the proof of BSP programs. This tool generates a sequential program to simulate the parallel program in input, thus allowing the use of WHY and the numerous associated provers to solve the proof obligations. We then show how BSP-WHY can be used to prove the correctness of some basic BSP algorithms. We also present a more complex example, the generation of the state-space (model-checking) of systems, especially for security protocols. Finally, in order to ensure the greatest confidence in the BSP-WHY tool, we give a formalisation of the language semantics, in the COQ proof assistant. We also prove the correctness of the transformation used to go from a parallel program to a sequential program.

Keywords. Parallelism, BSP, Deductive Verification, WHY, VCG, State-space, Model-checking, Formal Semantics, COQ