BSP-WHY: an intermediate language for deductive verification of BSP programs

Jean Fortin & Frédéric Gava

October 2009
TR–LACL–2009–7

Laboratoire d’Algorithmique, Complexité et Logique (LACL)
University of Paris–East

Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle,
94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01
BSP-WHY: an intermediate language for deductive verification of BSP programs

Jean Fortin  Frédéric Gava

Laboratory of Algorithmics, Complexity and Logic (LACL)  
University of Paris–East (Paris 12),  
61 avenue du Général de Gaulle, P2 du CMC  
94010 Créteil cedex, France

jean.fortin@ens-lyon.org and frederic.gava@univ-paris-est.fr

Abstract

We present BSP-Why, a tool for verifying BSP programs. It is intended to be used as an intermediate core-language for verification tools (mainly condition generator) of BSP extensions of realistic programming languages such as C, JAVA, etc. BSP is a parallel model which allows an estimation of the execution time of its algorithms, avoids deadlocks and indeterminism. BSP-Why is based on a sequential simulation of the BSP programs which allows to generate pure sequential codes for the back-end condition generator Why and thus benefit of its large range of existing provers (proof assistants or decision procedures). In this manner, BSP-Why is able to generate proof obligations for BSP programs.
## Contents

1 Introduction .......................... 1  
1.1 Generalities .......................... 1  
1.2 Which parallel model ? ............... 1  
1.3 Which condition generator ? .......... 2  
1.4 Contribution of this paper ............ 2  
1.5 Outline ................................ 2  

2 BSP computing .......................... 5  
2.1 The Bulk-Synchronous Parallel Model . 5  
2.2 BSP programing in C .................... 5  
2.2.1 Message Passing and Synchronisation 6  
2.2.2 Remote Memory Access ................ 6  
2.3 Example of crashing BSP programs .... 7  

3 The BSP-Why intermediate language ... 9  
3.1 The Why intermediate language [15] ... 9  
3.2 Syntax of BSP-Why ........................ 9  
3.3 Operational Semantics of BSP-Why .... 11  

4 Parallel memory model .................. 15  

5 Translation from Why to BSP-Why .... 19  
5.1 Sequential Block Decomposition .... 19  
5.1.1 Block Tree .......................... 19  
5.1.2 Algorithm ........................... 19  
5.2 Program transformation .............. 19  
5.2.1 Tree transformation ............... 19  
5.2.2 Local bloc transformation .......... 20  
5.3 Proof of equivalence ............... 21  
5.3.1 Notations ........................... 21  
5.3.2 Correct simulation .................. 21  
5.3.3 Correct assertions ................. 22  
5.3.4 Correct transformation ............ 22  

6 Examples .......................... 23  
6.1 Parallel prefix reduction ............. 23  
6.2 Logarithmic parallel prefix reduction 24  
6.3 Parallel sorting algorithm .............. 25  
6.4 Generalities ........................... 25
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The BSP model of execution</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>The BSP programming C primitives</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Syntax of BSP-Why: expressions (left), assertions, types and programs (right)</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Semantics of BSP-Why: Control Flow Rules</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Semantics: Local BSP Operations</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Semantics: Global reductions</td>
<td>13</td>
</tr>
<tr>
<td>5.1</td>
<td>Transformation of the tree (left) and of a local block (right)</td>
<td>20</td>
</tr>
<tr>
<td>6.1</td>
<td>BSP-Why code of the direct prefix computation (left) and its Why transform (right)</td>
<td>24</td>
</tr>
<tr>
<td>6.2</td>
<td>Parallel sorting algorithm</td>
<td>26</td>
</tr>
<tr>
<td>A.1</td>
<td>Semantics of Why</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 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 fault program, unreliable results or crashing simulations). Formal verification tools that display parallel concepts are thus useful for program understanding and incisive debugging. With the multi-cores, GPUs and peta-scale revolutions looming, such tools are long overdue.

Given the strong heterogeneity of these massively parallel architectures (e.g. cluster of multi-CPUs/GPUs) and their complexity, a frontal attack of the problem of verification of parallel programs is a daunting task that is unlikely to materialize. Some work on the standard MPI [45] exist [43, 51] but are limited to a predefined number of processors: by model-checking the MPI/C source code and by using an abstraction of the MPI calls (the schemes of communications), an engineer, by push-button, can mainly verify that the program do not contains any possible deadlocks. But it is impossible to verify this property for any number of processors which is a scaling problem.

Another approach would be to consider well-defined subsets that include interesting structural properties. In fact, many parallel programs are not as unstructured as that they appear and could be considered as an interlocking of parallel programs: it is the skeletons¹[9] and BSP [4] main idea [12]. For example, one can considered a cluster of multi-CPUs as a pipeline (or a farm) of PCs each doing optimised BSP computations (as numeric computations e.g. sparse matrix multiplication etc.).

Also, more safety properties than deadlocks are necessary to avoid the crash of programs: buffer and integer overflows, non terminating loops, etc. Even if most programming engineers want push-button tools to provide safety (such as model-checking), for critical systems and libraries, one can also want better truth to the code: are results as intended ? Verification Condition Generator (VCG) tools is one of the solutions. They take an annotated program as input and produce verification conditions as output to ensure correctness of the properties given in the annotation. They also provide safety if no annotation is present in the code and the “Push-button spirit” by generating proof obligations to automatic provers.

The goal of this article is to provide a tool for the verification of properties of a special class of parallel programs by providing annotations and generation of evidences to be proved manually or by automatic procedures.

1.2 Which parallel model ?

Since the Dijkstra’s paper “Go To Statement Considered Harmful”, structured sequential programming is the norm. That allows efficient tools for program verification. But it is surprising to see that it is absolutely not the case for parallel programming [19]. Besides compiler-driven automatic parallelisation, programmers have kept the habit of

¹Anyone can observe that many parallel algorithms can be characterised and classified by their adherence to a small number of generic patterns of computation (farm, pipe, etc.) [9]. 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.
using low-level parallel routines (as send/receive of e.g. MPI/PVM [45]) or concurrent languages [31]. In this way, they, less or more, manage the communications with the usual problems of (un)buffered or (un)blocking sending, which are source of deadlocks and non-determinism. Furthermore, programmers forbid optimisations that could be done if more parallel high-level structures as collective operators [19] or skeletons [9] were to be used instead.

In this article, we restrict our study to the BSP framework. BSP is a coarse-grained parallel model which offers a high degree of abstraction, deadlock free, scalable programs and allows an estimation of the execution time of its algorithms on a wide variety of architectures as massively parallel machines (Cray etc.), Beowulf’s clusters of PCs or multi-core PCs or any mix of the two aforementioned architectures. That is particularly helpful in the design of efficient parallel algorithms [2, 10] and routines in API [18]. In fact, many parallel programs fit the BSP model even if many authors do not known it. For example, all MPI programs [45] that only used collective routines can be considered as BSP ones.

A BSP program is executed as a sequence of super-steps [44]. This structural property of the parallel computation is well known by the parallel algorithmic community for doing static [4] and runtime optimisations [2]. It also looks like a good candidate for formal verification [27]: its model of execution is simple to understand than concurrent model as MPI send/received etc.

In fact, writing proof assistants and VCGs are tremendous amount of works which should be left to the field experts. The main idea of this work is to simulate the parallelism by using a transformation of the parallel code into a pure sequential one. Therefore, the goal is using the now “well defined” verification tools of sequential programs as back-ends for parallel verification tools.

1.3 Which condition generator ?

For this work, we choose the VCG Why. First, it takes as input a small core-language close to ML avoiding us to handle all the construction of a full language. Instead, realistic programming languages can be compiled into the Why input language. Why currently interprets C programs and JAVA programs with the help of two companion tools [14, 15], two main languages for parallel computing (lack of Fortran and C++). Our BSP-Why would serve as intermediary for C [6, 22] or JAVA [5, 21] BSP extensions like Why is for sequential programming.

Second, Why is currently interfaced with the main proof assistants (Coq, PVS, HOL) and automatic decision procedures (Simplify, Alt-Ergo, etc.) as back-end for the proofs obligations. This allow us to use these provers for the proof obligations for the parallel programs.

1.4 Contribution of this paper

In this paper, we design a core-language, close to the one of Why for BSP programming. A special syntax for BSP annotations is also provided which is simple to use and seems sufficient to express conditions in most of practical programs (e.g. the ones of [4, 10]). Our core-language is also close to standard BSP programming: primitives offer both message passing and remote memory access.

The structural nature of the BSP programs in super-steps allows us to decompose the programs into sequences of blocks of code, each block corresponding to a super-step. We used the Why code language as a back-end of our own BSP-Why\(^2\) core language by transforming BSP-Why programs into Why ones. That is verifying the de Bruijn criterion because we prove that generated Why programs (with annotations) are as intended: the parallel program and its sequential simulation run in the “same manner”.

1.5 Outline

The remainder of this paper is organized as follows. First, we briefly recall in Chapter 2 the BSP model, the BSP programming in C and typical examples of crashing programs. Then, in Chapter 3 we present our kernel language that is the BSP-Why. We continue with the definition of the translation of BSP-Why programs into Why ones (Chapter 4) and results (Chapter 5). We presents some simple examples of BSP algorithms in Chapter 6.

to demonstrates the usefulness of this method and its limits. We end with related works (Chapter 7), conclusion and future work (Chapter 8).
Chapter 2

BSP computing

2.1 The Bulk-Synchronous Parallel Model

A BSP computer is a set of uniform processor-memory pairs and a communication network allowing interprocessor delivery of messages (for sake of conciseness, we refer to [4,44] for more details). A BSP program is executed as a sequence of super-steps, each one divided into three successive disjoint phases (see Fig. 2.1): each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; the network delivers the requested data; a global (collective) synchronisation barrier occurs, making the transferred data available for the next super-step. Supercomputers, clusters of PCs, multi-core [15] and GPUs etc. can be considered as BSP computers.

The performance of the BSP machine is characterised by 4 parameters: the local processing speed $r$; the number of processor $p$; the time $L$ required for a barrier; 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 $g \times h$. 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. The total cost of a BSP program is the sum of its super-steps costs, and to accurately estimate its execution time the BSP parameters could be easily benchmarked [4].

On most of today’s distributed architectures, barrier synchronisations are often expensive when the number of processors dramatically increases (more than 10,000). However, future shared memory architecture developments (such as multi-cores and GPUs) may make them much cheaper. They have also a number of attractions: it is harder to introduce the possibility of deadlock or livelock, since barriers do not create circular data dependencies. Barriers also permit novel forms of fault tolerance [44].

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 also provides better performances since it is faster to send a block of data than individual ones (less network latency).

2.2 BSP programing in C

BSPlib [22] and PUB [6] (a BSPlib with few additional features\(\textsuperscript{1}\)) are C-libraries of communication routines to support the development of parallel algorithms based on the BSP model. They offer functions for both message passing (BSMP) and remote memory access (DRMA\(\textsuperscript{2}\)). Some collective communication operations like broadcast are also provided, but they are not interesting for our purpose because they can easily be simulated by BSMP operations.

As in the standard MPI, we first need to initialise our parallel computation which is done using the function bspilib_init (C types are given in Fig. 2.2). Now, we can query some informations about the machine: bsp_nprocs

\(\textsuperscript{1}\)These extensions are not modelled here because too complex, too architecture dependant and less useful.
\(\textsuperscript{2}\)RMA allows processes to specify shared memories and distant read/write in this memories.
returns the number of processors $p$ and `bsp_pid` returns the processor id which is in the range $0, \ldots, p - 1$. To terminate a BSP computation, we use `bsplib_done` which cleans up all BSP resources.

### 2.2.1 Message Passing and Synchronisation

According to the BSP model all messages are received during the synchronisation barrier and cannot be read before. 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.

Sending a single message is done using `bsp_send(dest, buffer, s)` where `buffer` is a pointer to a memory address to send to processor id `dest` and `s` is the size in bytes of this block. After calling this routine the buffer may be overwritten or freed.

In the next super-step, each processor can access the received messages (type `t_bspmsg`). This can be done using `bsp_findmsg(proc_id, index)` 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` its size. Also `bsp_nmsgs(id)` returns the number of messages received in the last super-step from node id. 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.

These routines are also the ones used in the two BSP libraries for JAVA which only communicate serializable objects.

### 2.2.2 Remote Memory Access

Another way of communication is through remote memory access: after every processor has registered a variable for direct access, all processors can read or write the value on other processors. Registering a variable or deleting it from global access is done using: `void bsp_push_reg(ident, size)` and `bsp_pop_reg(ident)`.

Due to the SPMD (Single Program Multiple Data) 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, BSP libraries provide 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:

```c
void x[5], y[5];
bsp_push_reg(x, 5)  // BSP: Given
bsp_push_reg(y, 5)  // BSP: Given
```

On processor 1:

```c
void x[5], y[5];
bsp_push_reg(x, 5)  // BSP: Given
bsp_push_reg(y, 5)  // BSP: Given
```
### Tools

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bsplib_init(t Bsplib_Params* parameter)</code></td>
<td>initialises the BSP computation</td>
</tr>
<tr>
<td><code>bsplib_done()</code></td>
<td>exits and frees resources</td>
</tr>
<tr>
<td><code>int bsp_nprocs()</code></td>
<td>returns the number of processors</td>
</tr>
<tr>
<td><code>int bsp_pid()</code></td>
<td>returns own processor-id</td>
</tr>
<tr>
<td><code>void bsp_sync()</code></td>
<td>BSP synchronization</td>
</tr>
</tbody>
</table>

### BSMP primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void bsp_send(int dest, void* buffer, int s)</code></td>
<td>Bulk sending of a value</td>
</tr>
<tr>
<td><code>int bsp_nmsgs(int proc_id)</code></td>
<td>number of received messages</td>
</tr>
<tr>
<td><code>t_BspMsg* bspmsg_data(t_BspMsg* msg)</code></td>
<td>returns a pointer to the data of a message</td>
</tr>
<tr>
<td><code>int bspmsg_size(t_BspMsg* msg)</code></td>
<td>returns the size of a message</td>
</tr>
</tbody>
</table>

### DRMA primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void bsp_send_reg(void* ident, int size)</code></td>
<td>register a variable for remote access</td>
</tr>
<tr>
<td><code>void bsp_pop_reg(void* ident)</code></td>
<td>delete the registration of a variable</td>
</tr>
<tr>
<td><code>void bsp_get(int srcPID, void* src, int offset, void* dest, int nbytes)</code></td>
<td>remote writing to another processor</td>
</tr>
<tr>
<td><code>void bsp_put(int destPID, void* src, int offset, int nbytes)</code></td>
<td>remote reading from another processor</td>
</tr>
</tbody>
</table>

---

Figure 2.2: The BSP programming C primitives

---

The simplicity and yet efficiency of the BSP model makes it a good framework for teaching (few hours are needed to teach BSP programming and algorithms), low level model for multi-cores/GPUs system optimisations [18], etc., since it has been conceived as a bridging model for parallel computation. The simplicity and yet expressivity of BSP programming makes it look like a good candidate for the formal proof of parallel computations. Since BSP programs are portable and cost estimate features power consumption, they can enjoy cloud-computing (see Section [8]).

### 2.3 Example of crashing BSP programs

Even if the BSP model is deadlock and livelock free, many errors can appear in BSP programs in addition of classical sequential programming errors (buffer and integer overflows, non terminating loops, etc.). Take for example the following code:

```c
if (bsp_pid()==0) bsp_sync();
else asynchronous_computation();
```

here, a deadlock can occur and the parallel machine would crash on some architectures. Communications can also generate errors:

```c
void x=bsp_pid();
/* All processors except 0 send a message to 0 */
```

---

3Even deadlock which is not a natural case.
if (bsp_pid() != 0) bsp_send(0, x, sizeof(int));
bsp_sync();

(+ processor 0 reads these messages +)
if (bsp_pid() == 0)
    for (int i = 0; i < bsp_nprocs() - 1; i++)
        x += bspmsg_data(bsp_findmsg(i, 0));

processor 0 will read a message from itself too. Another example:

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

the last processor would write over the limits of x of processor 0 and a segfault will occur. Many other errors can
be cited: forgetting to register a variable, and all errors with pointers of the messages that one can imagine.

Proving that BSP programs do not have these incorrect using of the routines would increase confidence in the
programs.
Chapter 3

The BSP-Why intermediate language

3.1 The Why intermediate language [15]

Generalities

Implementing a verification condition generator (VCG) for a realistic programming language needs a lot of work: too many constructs require a specific treatment[1]. Reducing the VCG to a core language thus seems a good approach.

The input syntax of the VCG Why is an intermediate and specific language dedicated to program verification (http://why.lri.fr/). As a programming language, it is an imperative 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 [15]) (4) uses program labels to refer to the values of variables at specific program points.

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 [13], incorporating exceptional post-conditions and computation of effects over mutable variables.

Why also provides a multi-prover output for the main proof assistants and automatic decision procedures. Why currently interprets C programs and JAVA programs with the help of the companion tools Caduceus and Krakatoa [15]. These tools transform annotated programs from the two above real languages to Why programs. Syntax and semantics of Why can be found in [13] and in Annex A

3.2 Syntax of BSP-Why

The syntax of BSP-Why is very close and strongly inspired from the one of Why [13] with an additional syntax for parallel instructions. It is given in Fig. 3.1

BSP-WHY Programs

A program $P$ is a list of declarations (Fig. 3.1 in the right). A declaration $d$ is either a definition introduced with let or a declaration introduced with val, or an exception declaration.

Expressions

Program expressions (Fig. 3.1 in the left) mostly follows ML’s ones. They contain pure terms ($t_e$) made of constants (integers, booleans, void, etc.), variables, dereferences (written $!x$) and application of function symbols from the logic to pure terms. A special constant nprocs and a special variable pid are also used: nprocs returns the number of processors $p$ and pid returns the own processor id in the range $0, \ldots, p - 1$. In pure terms, we also

\footnote{This also happens when modelling a communication library such as MPI [32]; it forces to use the smallest possible number of routines.}
have the two special function symbols `nmsg(t)` and `findmsg t1 t2`: the former corresponds to `bsp_nmsgs(id)` and the latter to `bsp_findmsg(proc_id, index)`.

ref e introduces a new reference initialized with e. loop e {invariant p variant t1} is an infinite loop of body e, invariant p and which termination is ensured by the variant t1. The raise construct is annotated with a type τ since there is no polymorphism.

There are two ways to insert proof obligations in programs: 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.

The six parallel operations are: (1) sync, barrier of synchronisation, which holds a pre-condition to define how processes are at the end of the super-step; (2) push x, registers a variable x for global access; (3) pop x, delete x from global access; (4) put e x y, distant writing of x to y of processor e; (5) get e x y, distant reading from x to y; (6) send e1 e2, sending value of e1 to processor e2.

For the sync operation, it is necessary in order to prove the correctness of a program to give a logic assertion just before the sync instruction. This assertion should describe the computation done during the previous superstep. 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.

We consider the following classical syntax sugar:

\[ e_1; e_2 \equiv \text{let } \_ = e_1 \text{ in } e_2. \]
The traditional \texttt{while} is also a syntactic sugar for a combination of an infinite loop with the use of an exception \texttt{Exit} to exit the loop.

\subsection*{Assertions: Types and Specifications}

Annotations are written using a simple minimal first-order logic defined in Fig.~3.1 in the right. A logic term $t_l$ can be a constant $\epsilon$, a variable $x$, the contents of a reference $x$ (written $!x$) or the application of a function symbol $\phi$. Notes that $\phi$ 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 construct $\text{at}(t_l, L)$ denotes the value of the term $t_l$ at the program point $L$. $t_l<i>$ denotes the value of a term $t_l$ at processor id $i$, and $<x>$ denotes the array $x$ (a value on each processor) by opposition to the notation $x$ which means the value of $x$ on the current processor).

We assume the existence of a set of \textit{pure types} ($\beta$) in the logical world, containing at least \texttt{unit}, \texttt{bool}, \texttt{int} and messages \texttt{value}.

As in \cite{13}, predicates necessarily include conjunction, implication and universal quantification. An atomic predicate is the application of a predicate symbol $A$ and is not interpreted.

Messages to be sent that is values and distant reading and writing are stored in environment of communications as simple list of messages. There are thus five additional components in the environment, one per primitives that need communications each of them is a special variable of the assertions: (1) \texttt{Cget} for \texttt{get} primitives; (2) \texttt{Cput} for \texttt{put} ones; (3) \texttt{Csend} (4) \texttt{Cpush} and (5) \texttt{Cpop}. We have also to deal with the list of received messages $R$. For DRMA primitives, there is also the registration $T$ which is describe later.

Take from \cite{13}, a value of type $\tau$ is either an immutable variable of a pure type ($\beta$), a reference containing a value of a pure type ($\beta$ ref) or a function of type $(x : \tau) \rightarrow \{p\} \beta \epsilon \{q\}$ mapping the formal parameter $x$ to the specification of its body, that is a precondition $p$, the type $\tau$ for the returned value, an effect $\epsilon$ and a post-condition $q$. An effect is made of tree lists of variables: the references possibly accessed (\texttt{reads}), the references possibly modified (\texttt{writes}) and the exceptions possibly raised (\texttt{raises}). 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).

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

We also used \texttt{farray} which is the abstract type for purely applicative arrays of size $p$ (with some obvious axioms) and \texttt{list}. They are used for the description of \texttt{p}-values, one per process and for the environment of communications.

\subsection*{3.3 Operational Semantics of BSP-Why}

The semantics of the Why language is a big-step operational one without surprise. Rules were taken from \url{http://why.lri.fr/} and are given in Fig. ~A.1 in appendix A. We also describe the semantics of BSP-Why using a big-step semantics. Even if the semantics contains many rules and many environments (due to the parallel routines), there is no surprise and it has to be read naturally.

In order to simplify the reading of the semantics, parallel operations take simple variables as argument, instead of buffers. In practice, BSP-Why does manipulate buffers, and adds proof obligations to avoid buffer overflows.

The notions of values and states are the same as in the Why semantics with the additional possible value \texttt{SYNC}($\epsilon$), which describes the awaiting of a synchronisation.

\[
f ::= \text{SYNC}($\epsilon$) | v \\
v ::= \epsilon | E | \text{rec} f x = e
\]

A value $v$ is either a constant (integer, boolean, \texttt{etc.}), an exception $E$ carrying a constant $c$ or a closure $\text{rec} f x = e$ representing a possibly recursive function $f$ binding $x$ to $e$. As in the semantics of Why, it is convenient to add the notion of closure to the set of expressions: $e ::= \ldots | \text{rec} f x = e$.

To simplify the notation and reduce the number of rules, we use the notation $f$ to design an execution flow, which is either a state of synchronisation, or a value. It is used when it is possible to factor several rules using this notation.
We note $s$, of a component of an environment without modifying other components, to deal with exceptions: $R$ to access to the component $X$ of the environment $s$, $\phi$ the update of a component of an environment without modifying other components and $\in$ to test the presence of a data in the component.

As the BSPlib, DRMA variables are registered using a registration mechanism that is each processor contains a registration $T$ which is a list of registered variables: the first one in the list of a processor $i$ corresponds to first one
of the processor \(j\).

We first give 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 given in Fig. 3.2. 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 \(\text{SYNC}(c)\).

To not have a 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 \(\text{Why}\) also forbid this case to be aliasing free.

Semantics rules of the BSP operation are given in Fig. 3.3 (executed on a single processor \(i\)). Basically, a communication operation adds the corresponding message in the environment. We note \(n^\text{th}(s', \mathcal{T}, y) = n\) to learn for a variable \(y\) how \(n\)th is it in the registration \(\mathcal{T}\). We also note \(\text{Size}(s, \mathcal{R}, \text{to})\) to know how many message from processor \(\text{to}\) has been received during the last super-step.

BSP programs are SPMD ones so an expression \(e\) is started \(p\) times. We model this as a \(p\)-vector of \(e\) with the environments of execution. A final configuration is a value on all processors. We note \(\Downarrow\) for this evaluation and the two rules are given in Fig. 3.4.

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 \(\text{AllComm}\) 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 $p$ environments.

Note that if at least one processor finishes his execution while others are waiting for a synchronisation, a deadlock will occur. That also happen when a processor still is in an uncaught exception.

Finally we have the following results.

**Lemma 3.3.1.** $\forall i \downarrow^i$ is deterministic.

**Lemma 3.3.2.** $\downarrow$ is deterministic.

The two lemmas are trivially prove by induction on the evaluation. The execution of the program is the execution of its declaration.
Chapter 4

Parallel memory model

The main idea of our approach is to simulate the execution of a BSP program on a parallel machine by an execution in a sequential way which will simulate the entire parallel machine. This way we are able to use the Why tools, which are designed to help the proof of sequential programs via the mean of numerous back-end provers, to help with the proof of our parallel program.

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, and then sequentialized into a Why 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 a prefix file, prefix.why, to define basic operations common to all Why programs, we use the bspwhyprefix.mlw file, which contains the common data types, the basic operations on these data types, the axiomatization of the BSP operations, and of the memory model used.

It includes the definition of data types used in the transformation, such as the fparray type, a functional array of length p. The parray type corresponds to the non-functional array, which is a reference to the fparray.

As in the semantics, three separates message queues, send_queue (Csend), put_queue (Cput), and get_queue (Cget), 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 (Cpush) and pop (Cpop) mechanisms.

To be as close as possible of 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 axiomatisation given by the post-condition, not the effective sequential code used. An actual sequential code would only make the proofs more difficult.

In the EnvR section of the file, we describe 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 two functions of serialization and deserialization are defined for each data type used in the program. One axiom for each data type ensures that the composition of deserialization and serialization is the identity.

The most complex part of the file is the definition of the DRMA mechanism. As Why does not allow pointers, we use a global two-dimensional array, named global, to store all 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 the BSPlib, we define a registration T. The push instruction can associate different variables on different processors. This is modeled 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.

As an example, we show the different parts of the prefix 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 (\textit{nil} and \textit{cons}):

\begin{verbatim}
  type send_queue
  logic ifadd_send : int, value, send_queue -> send_queue
  logic ifnil_send : send_queue

  parameter add_send : dest:int -> v:value -> q:send_queue
                  -> {} send_queue {result = ifadd_send(dest,v,q)}
\end{verbatim}

The \texttt{add_send} parameter (⊕ in the semantics) will be used to effectively add a send message in a \texttt{send_queue}.

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 \textit{nsend} function, used to determine the number of messages waiting (\textit{Size}(\textit{s.R}, to) in the semantics):

\begin{verbatim}
  logic nsend : send_queue -> int
  axiom nsend_nil : nsend(ifnil_send) = 0
  axiom nsend_cons : forall q:send_queue. forall n,j:int.
                  forall v:value. nsend(q)=n -> nsend(ifadd_send(j,v,q))= n+1
\end{verbatim}

The \texttt{in_send} functions are used to test the fact that a message is in the list. Lastly, we can define the variable used for the global environment, which is a \texttt{parray} of \texttt{send_queues}, and the method \texttt{bsp_send} defined in the semantics. \texttt{isproc} is a useful predicate defined earlier in the prefix file, stating that an index is a valid processor id (\textit{i.e.} \(0 \leq i < p\)).

\begin{verbatim}
  parameter envCsend : send_queue fparray ref
  parameter bsp_send: dest:int -> v:value -> { isproc(proc_i) }
                  unit reads proc_i writes envCsend
                  {envCsend = pupdate(envCsend@,proc_i,ifadd_send(dest,v,(paccess(envCsend@,proc_i))))}
\end{verbatim}

In the next step, we define the environment used to store the values received during the previous synchronisation.

\begin{verbatim}
  type rvaluet
  logic rvalue_get : rvaluet, int, int -> value
  parameter envR : rvaluet fparray ref
  parameter bsp_findmsg: src:int -> n:int -> {} value
                  reads proc_i,envR
                  {result=rvalue_get(paccess(envR,proc_i),src,n)}
\end{verbatim}

The logic function \texttt{rvalue_get} allows to retrieve the \(n\)-th message sent by a processor \texttt{src}. \texttt{envR}, as previously, is defined as a \texttt{parray}. The \texttt{bsp_findmsg} is the corresponding parameter, and it can be used in the \texttt{BSP-Why} programs.

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

\begin{verbatim}
  predicate comm_send(envCsend:send_queue fparray,
              envCsend’:send_queue fparray, envR’:rvaluet fparray)=
  (forall i: int. isproc(i) ->
  (paccess(envCsend’,i) = ifnil_send)) and
  (forall i: int. isproc(i) ->
  (forall j:int. forall n:int. forall v:value.
  (rvalue_get(paccess(envR’,i),j,n)=v) <-->
\end{verbatim}
(in_send_n_j(paccess(envCsend, j), n, i, v)))

predicate comm(envCsend:send_queue fparray,
envCsend’:send_queue fparray,
envR’:rvaluet fparray, ...) =
  comm_send(envCsend, envCsend’, envR’) and ...

parameter bsp_sync : unit ->
  () unit writes envCsend, envR, ...  
  { comm(envCsend0, envCsend, envR, ...) }
Chapter 5

Translation from Why to BSP-Why

Now that we have the necessary structures to simulate the environments and communication functions of the parallel machine, we can do the actual transformation of a in BSP-Why program into a Why one, that will simulate its parallel execution. We will of course use in the transformation the tools defined in the previous section to manage the parallel environments and the communications.

5.1 Sequential Block Decomposition

The first step of the transformation is a decomposition of the program into blocks of sequential instructions. 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 remain sequential.

5.1.1 Block Tree

The decomposition into blocks is straightforward: it is a simple recursive algorithm on the syntax tree. The sync instruction is the only one that can affect the parallelism of the program, so we build the largest blocks (subtrees) of the program that do not contain the sync instruction.

In addition to this decomposition, in this phase of the process we check if a function is composed of a single block. In that case, it means that the function is purely sequential, and it is tagged accordingly. It allows to know later if a call to a function can be included in a sequential block, or if it is necessary to retain the parallel organisation for that function call, because it will raise a synchronisation. 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.

5.1.2 Algorithm

The algorithm is a recursive pattern matching on the AST:

- A variable or a constant is transformed into a block;
- A control instruction is transformed into a block if all its components are recursively transformed into a block, or the corresponding control instruction in the block data type in the other case;
- A sync is translated into the Sync block instruction.

5.2 Program transformation

5.2.1 Tree transformation

Once that we have 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 bloc tree is a simple traversal of the tree where we apply recursively the transformation (noted $[[e]]$).
The translation of a single block to the code that can be executed within the “for” loop is perhaps the most difficult one. A variable $x$ can be translated in different ways depending on its use.

- If the variable is declared locally, and is only used within the sequential block, it is simply translated in a similar variable $x$;
- 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;
- 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.

Figure 5.1: Transformation of the tree (left) and of a local block (right)

For instance, the rule for a control flow instruction such as the if instruction is as follows:

$$[[ABI f(a_1, a_2, a_3)]] = W S i f([a_1], [a_2], [a_3])$$

The complete transformation is given in Fig. 5.1 (left).

The base case, the transformation of a sequential block, is actually the interesting one: we create a “for” loop, to execute the code sequentially on each processor. $[[Bloc(e)]] = for p ([e], i)$

Since we run the program in a special environment that simulates its natural parallel environment, we need to transform the sequential code accordingly. For instance, the access to variables that have different values on different processors must be replaced by the access to an array, etc. The transformation of a simple sequential code into the corresponding code on the processor $i$ in our model is denoted by $[e]$, and defined in Fig. 5.1

5.2.2 Local bloc transformation

The translation of a single block to the code that can be executed within the “for” loop is perhaps the most difficult one. A variable $x$ can be translated in different ways depending on its use.

- If the variable is declared locally, and is only used within the sequential block, it is simply translated in a similar variable $x$;
- 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;
- 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.
The transformation of control instruction is straightforward, in the same way as previously, by walking the tree recursively. The rules are given in Fig. 5.1 (right).

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, \( x^{<i>} \) is transformed in the access to the \( i \)-th component of \( x \).

The parallel instructions (\textit{put}, \textit{send}, etc.) are not directly translated in an equivalent sequential code. They are replaced by calls to the parameters axiomatized in the prefix file.

5.3 Proof of equivalence

We now prove that our program transformation is correct \( i.e. \):

- If we prove using \textbf{Why} that the generated sequential program is correct, then the original program is correct;
- If the original program is correct, then the Hoare triplet composed of the precondition, code and postcondition of the sequential simulation is correct too.

Full proofs of the lemmas are available in Annex B.

5.3.1 Notations

We use a function of translation \( f_s : E_p \rightarrow E_s \), from parallel environment to sequential environment, and \( f_p \), the inverse function.

Another function of translation \( g_s : P_p \rightarrow P_s \), from parallel predicates to sequential predicates, and \( g_p \), the inverse function, are used on predicates.

We also used the following notation:

1. \( E_s, c \rightarrow_s E'_s \) denotes that the sequential execution of the program \( c \) in the environment \( E_s \) gives the environment \( E'_s \);
2. \( E_p, c \rightarrow_p E'_p \) denotes that the parallel execution of the program \( c \) in the environment \( E_p \) gives the environment \( E'_p \);
3. \( \{p\} c \{q\} \) is the usual Hoare triplet.

5.3.2 Correct simulation

We first need to prove that if a code executed with the parallel semantics give a result, the execution of the sequential translation will give the same result:

\textbf{Lemma 5.3.1.} If \( E_s = f_s(E_p) \), \( E'_s = f_s(E'_p) \), if \( E_s, \{\{c\}\} \rightarrow_s E'_s \) then \( E_p, c \rightarrow_p E'_p \).

\textbf{Lemma 5.3.2.} If \( E_s = f_s(E_p) \), \( E'_s = f_s(E'_p) \), if \( E_p, c \rightarrow_p E'_p \) then \( E_s, \{\{c\}\} \rightarrow_s E'_s \).

Since we chose to stay as close as possible to the semantics in the definition of the BSP operations in the sequential definitions, the proof of these lemmas is rather straightforward. The idea of the proof is to prove first that the execution following the decomposition in blocks corresponds to the global synchronisation rule of the semantics, and then to prove that the parallel synchronisation and the sequential simulation have the same effects. The definition of \texttt{bsp\_sync} in the prefix file is directly inspired from the communication predicate in the semantics rule, so there is no real difficulty.
5.3.3 Correct assertions

The first two lemmas were about the correctness of the transformation in regard to the operational semantics, the next two lemmas concern the correctness of the transformation in the logical point of view.

**Lemma 5.3.3.** If \( E_s = f_s(E_p) \), for all \( P_s \) and \( P_p \) such as \( P_s = g_s(P_p) \), if \( E_s \vdash P_s \) then \( E_p \vdash P_p \).

**Lemma 5.3.4.** If \( E_s = f_s(E_p) \), for all \( P_s \) and \( P_p \) such as \( P_s = g_s(P_p) \), if \( E_p \vdash P_p \) then \( E_s \vdash P_s \).

Once again, the transformation of the logical expressions is designed so that the predicate on the sequential environment has the same meaning as the predicates in the parallel environment. So the proof is a direct induction over the transformation rules.

5.3.4 Correct transformation

With the help of the lemmas given above, we can now prove the correctness and completeness of the transformation:

**Theorem 5.3.5 (Correctness).** If \( P_s = g_s(P_p) \), \( P'_s = g_s(P'_p) \), if \( \{P_s\} [[(c)] \} \{P'_s\} \) then \( \{P_p\} \ c \ \{P'_p\} \).

**Proof:** Let \( E_p \) such as \( E_p \vdash P_p \). Let \( E_s = f_s(E_p) \). Let \( E'_s \) be the result of the execution \( E_s \), \( [[[c]]] \rightarrow_s E'_s \), and \( E'_p = f_p(E'_s) \). Then by the Lemma 3, we have \( E_p \ c \rightarrow_p E'_p \). By Lemma 6, we have \( E_s \vdash P_s \). Then, since \( \{P_s\} [[(c)] \} \{P'_s\} \), we can deduce \( E'_s \vdash P'_s \). We can then apply the Lemma 5, which gives \( E'_p \vdash P'_p \). Hence \( \{P_p\} \ c \ \{P'_p\} \) is a valid Hoare triplet.

**Theorem 5.3.6 (Completeness).** If \( P_s = g_s(P_p) \), \( P'_s = g_s(P'_p) \), if \( \{P_p\} \ c \ \{P'_p\} \), then \( \{P_s\} [[(c)] \} \{P'_s\} \).

**Proof:** Let \( E_s \) such as \( E_s \vdash P_s \). Let \( E_p = f_p(E_s) \). Let \( E'_p \) be the result of the execution \( E_p \), \( c \rightarrow_p E'_p \), and \( E'_s = f_s(E'_p) \). Then by the Lemma 4, we have \( E_s \rightarrow_p E'_s \). By Lemma 5, we have \( E_p \vdash P_p \). Then, since \( \{P_p\} \ c \ \{P'_p\} \), we can deduce \( E'_p \vdash P'_p \). We can then apply the Lemma 6, which gives \( E'_s \vdash P'_s \). Hence \( \{P_s\} [[(c)] \} \{P'_s\} \) is a valid Hoare triplet.
Chapter 6

Examples

6.1 Parallel prefix reduction

Our first example is a simple one-step parallel prefix reduction that is having $\sum_{k=0}^{i} v_i$ on each processor where each processor $i$ initially hold $v_i$ (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. 6.1.

The program starts with a distributed parameter $x$, which contains the initial values, with one value on each processor. The prefixes are computed by the program in the $z$ parameter.

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 $\text{pid}$ than itself. The instruction $\text{bsp\_sync}$ then executes the synchronisation barrier. In the second loop, each processor computes the sum of all the received values.

In BSP-Why, as in the generated Why code, the while instructions should be followed by the variant statement, which ensures that the loop terminates. An invariant is also generally used to ensure the correctness of the program. The final post-condition is used to prove that the program gives the expected result.

Note the use of our notations in the program: $x$ designs the value on the current processor, $<x>$ refers to the global array and $x[i]$ refers to the value of $x$ at processor $i$. A user defined logic function, $\text{sigma\_prefix}$, is used to describe the partial sums. $\text{envCsendIs}$, on the other hand, is a predefined macro to describe the communication environment, without having to use the intern list description and its associated functions.

The send and $\text{find\_msg}$ functions can be used to transfer any type of data. For this reason, we use the $\text{cast\_int}$ and $\text{uncast\_int}$ functions, that encapsulates the date in a generic value data type.

The generated Why code is in Fig. 6.1. 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 $\text{proc\_i}$.

We can note that the distributed variables, such as $x$ and $z$, are translated into arrays of size $p$, using the type $p$-array. Reading or writing such a variable is done with the $\text{parray\_get}$ and $\text{parray\_set}$ functions, or in the logic world their counterparts $\text{paccess}$ and $\text{pupdate}$.

Local variable, 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. Lastly, we can note that the macro in the first loop invariant has been expanded, and that a special predicate has been introduced in the precondion to ensure that the environments are clean at the start of the program.

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. Here is an example of a proof obligation, generated in the Coq format:

(*Why goal*) Lemma prefixes_po_1 :
forall (envCsend: (fparray send_queue)),
forall (HW_1: (init_envCsend envCsend)),
forall (proc_i: Z),
forall (HW_2: proc_i = 0),
Let prefixes () = []

let y = ref (bsp_pid + 1) in
while (!y < nprocs) do
  invariant envCsendSize[j, bsp_pid + 1, y, x]
  variant nprocs — y
  bsp_send !y (cast_int (parray_get x !proc_i));
  y := !y + 1
done;

let y = ref 0 in
while (!y < bsp_pid) do
  invariant z = x + sigma_prefix(<x>, y)
  variant bsp_pid — y
  z := !z + uncast_int (bsp_findmsg !y 0);
  y := !y + 1
done;

Figure 6.1: BSP-Why code of the direct prefix computation (left) and its Why transform (right)

proc_i >= 0.
Proof.
intuition.
Save.

This simple proof obligation corresponds to the base case of a loop invariant, and is solved automatically by the intuition tactic of Coq. In this example, a dozen of proof obligations are generated. A lot of these obligations are relatively heavy, with a lot of unnecessary hypothesis, and for now, it is not possible to automatically prove all of them. The future developments in automatic provers, or the use of specialized tactics could change that.

6.2 Logarithmic parallel prefix reduction

If we suppose that data are of constant length, the bsp cost of the communications of the simple parallel prefix reduction would be $size(data) \times p \times g + L$. Thus, the above reduction does not make use of parallelism. If the combination operator has some cost, we may prefer to reduce in a multi-step manner (a classical logarithmic way), doing the combinations locally. This algorithm combines the values of processors $i$ and $i+2^n$ at processor $i+2^n$ for every step $n$ from 0 to $\lceil \log_2 p \rceil$. The bsp cost of the communications would be $\lceil \log_2 p \rceil (size(data) \times g + L)$.

One can write the main loop as:

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

This is a case where our block decomposition fails: not all the processors run the same sync and it would be impossible to prove the contrary (the condition generated by Why). But the program can be rewritten by factoring the two syncs. Here is the entire code with assertions: (we use the external logic term \( \Sigma_{i=n_1}^{n_2} X[i] \))

```
let scan () =
  let Xin = ref undefined 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) < nprocs ) do
      \{
        invariant
          (0<=pow_int(2,i)<=nprocs)
          and X'=sigma_prefix(<X'@init>,pid−pow_int(2,i−1),pid)
          variant nprocs−i
      \}
    if (pid >= (pow_int 2 !i)) then
      get (pid−(pow_int 2 !i)) X' 0 Xin sizeof_int;
    bsp_sync;
    if (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,pid) \}
end
```

6.3 Parallel sorting algorithm

Our last example is the sampling sort algorithm (PSRS) of Schaeffer in its BSP version [48]. The PSRS algorithm proceeds as follows. First, the lists of the parallel vectors (we assume that their lengths are \( \geq p^i \)) are sorted independently with a sequential sort algorithm. The problem now consists of merging the \( p \) sorted lists. Each process selects from its list \( p+1 \) elements for the primary sample and there is a total exchange of these values. In the second super-step, each process reads the \( p \times (p+1) \) primary samples, sorts them and selects \( p \) secondary samples. 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 \( j \). The complete code is given in Fig [6.2]

The BSP cost of the first super-step is \( \frac{n}{p} \times \log(\frac{n}{p}) \times c_c + \frac{n}{p} + (p \times (p+1) \times s_e) \times g + L \), where \( c_c \) is the time used to compare two elements, and with the assumption that all elements are of size \( s_e \). For the second super step and the merging, the cost is \( \frac{n}{p^2} \times \log(\frac{n}{p}) \times c_c \times \frac{n}{p^2} \times s_e \times g + L + \text{time}_\text{merge} \), with the merging time for each processor about \( \frac{n}{p} \).

6.4 Generalities

It is easy to see that the number of super-steps is always bounded in the above examples. This is also the case in most BSP programs. Terminations of them is thus generally simple to show.

In this table, we can show how many assertions 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.).
The complete proof of the proof obligations generated for the three examples is still a work in progress.

<table>
<thead>
<tr>
<th>Prog</th>
<th>Nb of assertions</th>
<th>Safety assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Prefix</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>Log Prefix</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>BSP Sort</td>
<td>41</td>
<td>19</td>
</tr>
</tbody>
</table>

Many of them are currently not automatically proved by automatic provers.

let psrs (a:int array) =
{
    (size of a > p+1 on each processor
array_length(a)>p+1)
    (local sort
local_sort(a);
    (first sampling
let samp1 = sampling(a) in
    (= create the buffer of reception)
let total_samp1 = ref (make_array_int (nprocs*(nprocs+1)) 0)
in
let i = ref 0 in
    push(total_samp1,nprocs*(nprocs+1));
    { sorted_array(a, 0, array_length(a)-1)
and envCpush=cons(total_samp1,nil)
    } sync void;
    (= total exchange of the sampling
while (i<nprocs) do
    |
    invariant envCsendk(pid,k,0,i,put(i,(cast_int_farray samp1,total_samp1,(nprocs+1)*i,(nprocs+1)))))
    variant nprocs
    |
    put (i,(cast_int_farray samp1) total_samp1 ((nprocs+1)*i,(nprocs+1)));
i:=i+1
    done;
    (= local sort of the sampling
    local_sort(uncast_int_farray !total_samp1);
    (= second sampling
    let samp2 = sampling(uncast_int_farray !total_samp1) in
    Second_step:i:=0;
    (= same samp2 on each processor
    assert \forall pid1,pid2:int. (0<=pid1<nprocs) → (0<=pid2<nprocs)
assert \forall pid1,pid2:int. (0<=pid1<nprocs) → (0<=pid2<nprocs) → samp2<pid1>=samp2<pid2>
while (i<nprocs) do
    |
    invariant envCsendk(pid,k,0,i,send(cast_int_farray (selection samp2 !a !i),!i))
    variant nprocs
    |
    send (cast_int_farray (selection samp2 !a !i)) !i;
i:=i+1
    done;
    (= sending data within the samplings
    envCsendk(pid,k,0,i,send(cast_int_farray (selection samp2 !a !i),!i))
and sorted_array(total_samp1, 0, array_length(total_samp1)-1)
and (local pid1 pid2 int. (0<pid1<nprocs) → (0<pid2<nprocs) → samp2<pid1>=samp2<pid2>)
)} sync void;
assert \forall pid1 pid2 int. (0<pid1<nprocs) → \exists pid2 (0<pid2<nprocs) → bsp_findmsg pid2 1=(selection samp2 !a !pid)
Third_step:a:= (make_array_int 0 0);
\forall pid:int. (0<=pid<nprocs-1) → (bsp_findmsg pid 1)=(selection samp2 !a !pid)
while (i<nprocs) do
    |
    invariant a=\sigma_merge(\epsilon R,0,0)
    variant nprocs
    |
a:=\sigma_merge_arrays la (uncast_int_farray (bsp_findmsg l1 1));
i:=i+1
    done
    (= merge the data
    envCsendk(pid,k,0,i,put(i,(cast_int_farray samp2,total_samp2,(nprocs+1)*i,(nprocs+1)))))
    invariant envCsendk(pid,k,0,i,put(i,(cast_int_farray samp2,total_samp2,(nprocs+1)*i,(nprocs+1)))))
    variant nprocs
    |
    put (i,(cast_int_farray samp2) total_samp2 ((nprocs+1)*i,(nprocs+1)));
i:=i+1
while (i<nprocs) do
    |
    invariant envCsendk(pid,k,0,i,put(i,(cast_int_farray samp2,total_samp2,(nprocs+1)*i,(nprocs+1)))))
    variant nprocs
    |
    put (i,(cast_int_farray samp2) total_samp2 ((nprocs+1)*i,(nprocs+1)));
i:=i+1
    done
    (= sorting array locally and different processors and set global length and same elements as in the beginning
sorted_array(a, 0, array_length(a)-1)
and (local pid1 pid2 int. (0<pid1<nprocs-1) → (bsp_findmsg pid1 1)=(selection samp2 !a !pid) )
and (global permutation(<a>,<a@>)
and \sigma_size <a> = \sigma_size <a@>)
)}

Figure 6.2: Parallel sorting algorithm
Chapter 7

Related Works

7.1 Other Verification Condition Generators

Close to Why, we can cite Spec# [3]. The Ynot System [34] is an extension to the Coq proof assistant, capable of reasoning 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. In the case of alias-free programs, simpler proof obligations is the main goal of Why: for communications, BSP-Why generates 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.

We can also cite Who [28] which is a derivation of Why for ML programming (polymorphic functions), and in the same spirit the Pangolin system [38]. Both could be interesting for proofs of ML extensions of BSP [16].

7.2 Concurrent programs

They are now many studies of proof obligations for concurrent programs, for example, the use of a shape analysis [20]. [35] presented a Concurrent Separation Logic as an extension of Separation Logic for reasoning about shared-memory concurrent programs with Dijkstra semaphores. It provides substantial flexibility in the dynamic assignment of resources to be locked, and at the same time permits modular reasoning.

[24] presents an operational semantics for Concurrent C minor which preserves as much sequentiality 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 [20]. However, currently no tools for generated proof obligations are provided and it is not clear how much harder the obligations would be.

[7] presents a type system that prevents data races and deadlocks (as in [24]) 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 permits concurrent reading only for immutable objects. [26] extend Spec#’s verification methodology to concurrent programs.

[39] 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 SMT solvers [37]). 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.

It is not clear whether locking strategies are very suitable for high-performance applications [31]. BSP is by nature a coarse-grained and deadlock-free model which is used for high-performance problems [4] and now in multi-core applications [18].

Even if proof of concurrent programs is clearly useful (servers, critic problems on multi-core architectures, etc.), parallel programming (high-performance computing) is not concurrent programming. Parallel programs are much simpler [31] (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 by copies of the data (this is the main idea of [47]). 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.

7.3 MPI programs

MPI is the most used C/Fortran 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 lack of formal specifications (even in works such as [32,49], some cases are not taken into account because of the lack of specification and too much dependence on the architecture) and the concurrent/deadlock nature of the programs. This enormous number of routines in the API makes it difficult to trust any formal specification of a complete MPI e.g., is there a non-trivial case that has been forgotten?

They are many works and tools dedicated to MPI. Surveys could be found in [25,41]. We can cite: debugger [11], tests methods [33,50], and checker [29,30]. These tools help to find some classical errors but not all. Note that this kind of tools works well for many situations in development phases but is not sufficient.

[51] presents a tool that directly model-check the MPI/C 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/C code against deadlocks on a p-processors machine before running it on this machine. A large number of MPI routines (including DRMA ones [36]) are considered. This technique was also used to find Irrelevant Barriers [40]. The main idea (which is also the one of [42,43]) is to analyse by abstract interpretation the code 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 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) to 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 [51]. 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.

Currently, we are not aware of verification condition generators tools for MPI programs. We think that a sequential simulation of any kind of MPI programs is not reasonable. Continuations would be needed to simulate the interleaving of processes: that would generate unintelligible assertions. But collective MPI routines can be seen as BSP programs, and many MPI programs could be transformed into BSP ones. Automatically translating this class of programs is a possible way to analyse MPI programs. We leave the work of substantiating this claim for future work.

7.4 Proof of BSP programs

The simplicity of the structural nature of BSP programs allows to prove properties and correctness of BSP programs. Different approaches for proofs of BSP programs have thus been studied, such as BSP functional programming using Coq [16] or the derivation of BSP imperative programs using Hoare’s axiom semantics [8,27,46]. The main drawback of these approaches is that they use their own languages that are in general not subsets of real programming languages. Also they neither used any proof assistant (except [16]) nor implemented dedicated tools which is a lack of safety: users make hand proofs so they are just theoretical works. It is also not clear how such tools can be built and how these BSP extensions of the Hoare logic can be used.

[17] presents a formal deterministic operational semantics in Coq (a natural semantics) for BSP programs and uses it to prove the correctness of a classical numerical computation (the N-body problem) and the divergence of some programs. But using the semantics in Coq, proofs were too hard even for just proving that the program run without faults.
Chapter 8

Conclusion

In this paper, we have presented the tool BSP-Why, an intermediate language aimed at the verification of BSP programs. It is an extension of the verification tool Why which is designed for sequential programs. We give and prove a block transformation of the BSP-Why language into the Why one, featuring obligation generation for BSP programs. Some examples are given.

The current prototype implementation is still limited. We plan to extend it in several ways.

First, we intend to add a companion tool for C programs as in [15]. Java programs are interesting but C++ and Python [23] languages are currently more used for high-performance and scientific computations. They are both “untyped” object languages but some type system seems possible for selected parts of the languages. That could allow Why interpretation of these languages.

Second, 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 [1], we can imagine a scheduler server that distributes the parallel programs depending on the cost certificates to optimise power consumption.

Third, conditions generated by Why from BSP-Why programs are not friendly, even for theorem provers (especially for them ?). 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 counter for loops, local sorting of data, etc. 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) are needed to facilitate the writing of logical assertions.

Last, there are many more MPI programs than BSP ones. Our tool is not intended to manage all MPI programs. It can not be used to model send/receive ones (with possible deadlocks depending on the MPI scheduler), only programs which are BSP-like. Analysing MPI/C programs to find which are BSP-like and to interpret them in BSP-Why is a great challenge.
Bibliography


Appendix A

Operational rules of Why

This description of the Why semantics is taken from http://why.lri.fr/. In Why, the notions of values and states are the following:

\[
\begin{align*}
  v &::= \ c \mid E \ c \mid \text{rec } f x = e \\
  s &::= \ \{(x, c), \ldots, (x, c)\}
\end{align*}
\]

A value \(v\) is either a constant value (integer, boolean, etc.), an exception \(E\) carrying a value \(c\) or a closure \(\text{rec } f x = e\) representing a possibly recursive function \(f\) binding \(x\) to \(e\). For the purpose of the semantic rules, it is convenient to add the notion of closure to the set of expressions:

\[
\begin{align*}
e &::= \ldots \mid \text{rec } f x = e
\end{align*}
\]

In order to factor out all semantic rules dealing with uncaught exceptions, we have the following set of contexts \(R\):

\[
\begin{align*}
  R &::= \[] \mid x := R \mid \text{let } x = R \text{ in } e \mid \text{let } x = \text{ref } R \text{ in } e \\
  &\quad\mid \text{if } R \text{ then } e \text{ else } e \mid \text{loop } R \{\text{invariant } p \text{ variant } t\} \\
  &\quad\mid \text{raise } (E \ R) : \tau \mid R \ e
\end{align*}
\]
Figure A.1: Semantics of Why
Appendix B

Proofs of Technical lemmas

B.1 Notations and first results

We use a function of translation, \( f_s : E_p \rightarrow E_s \), from parallel environment to sequential environment, and \( f_p \), the inverse function.

Another function of translation, \( g_s : P_p \rightarrow P_s \), from parallel predicates to sequential predicates, and \( g_p \), the inverse function, are used on predicates.

\( E_s, c \rightarrow_s E_s' \) denotes that the sequential execution of the program \( c \) in the environment \( E_s \) gives the environment \( E_s' \).

\( E_p, c \rightarrow_p E_p' \) denotes that the parallel execution of the program \( c \) in the environment \( E_p \) gives the environment \( E_p' \).

\( \{p\} c \{q\} \) is the usual Hoare triplet.

**Definition** \( Es\}_i = (EnvE[i], EnvCSend[i], EnvCPut[i], EnvCGet[i], EnvCPush[i], EnvCPop[i], EnvR[i]) \)

**Definition** A program \( c_s \) is called \( i \)-local if it does only read and modify the \( i \)-th component of its environment, \( E\}_i\).

**Definition** We note \( E_{si}, [[(c_s)]]) \rightarrow_{si} E_{si}' \) if \( c_s \) is \( i \)-local and if there exist two environments \( E_s \) and \( E_s' \) such as \( E_{si} = E_s\}_i, E_{si}' = E_s'\}_i, \) and \( E_s, [[(c_s)]]) \rightarrow_s E_s' \).

**Lemma B.1.1.** Let \( c \) be an expression. Then \( [[c]]_i \) is \( i \)-local.

**Proof.** By induction on \( c \).

- If \( c \) is \( e_1; e_2 \), then \( [[c]]_i = [[e_1]]_i; [[e_2]]_i \). By induction, \( [[e_1]]_i \) and \( [[e_2]]_i \) are \( i \)-local, so \( [[e_1]]_i; [[e_2]]_i \) is \( i \)-local. So \( [[c]]_i \) is \( i \)-local.

- If \( c \) is \( \text{loop } e' \), then \( [[c]]_i = \text{loop } [[e']]_i \). By induction, \( [[e']]_i \) is \( i \)-local, so \( \text{loop } [[e']]_i \) is \( i \)-local. So \( [[c]]_i \) is \( i \)-local.

- If \( c \) is \( x := e' \), then we have several cases.
  - If \( [[c]]_i = x := [[e']]_i \), by induction \( [[e']]_i \) is \( i \)-local, so \( [[c]]_i \) is \( i \)-local.
  - If \( [[c]]_i = \text{arrayset}(x, i, [[e']]_i) \), by induction \( [[e']]_i \) is \( i \)-local, and we modify the \( i - th \) component, so \( [[c]]_i \) is also \( i \)-local.

- If \( c = \text{send } e_1 e_2 \), \( [[c]]_i = \text{send } [[e_1]]_i, [[e_2]]_i \). By induction, \( [[e_1]]_i \) and \( [[e_2]]_i \) are \( i \)-local. By construction, the \( \text{send} \) instruction only modifies \( EnvCSend[i] \), so it is \( i \)-local. So \( [[c]]_i \) is \( i \)-local.

- For the other communication functions, a similar proof as for the \( \text{send} \) instruction applies.

- For the other inductive cases, a similar proof as for the \( \text{loop} \) instruction applies.
B.2 Correct logic assertions

Lemma B.2.1. If \( E_s = f_s(E_p) \), forall \( P_s \) and \( P_p \) such as \( P_s = g_s(P_p) \), if \( E_p \vdash P_p \) then \( E_p \vdash P_p \).

Proof. By induction on the predicate \( P_p \).

- If \( P_p = P_1 \land P_2, P_s = g_s(P_1) \land g_s(P_2) \). \( E_s \vdash P_s \), so \( E_s \vdash g_s(P_1) \) and \( E_s \vdash g_s(P_2) \). By the induction hypothesis, we have \( E_p \vdash P_1 \) and \( E_p \vdash P_2 \). Hence \( E_p \vdash P_p \).
- The other induction cases follow the same structure.

Lemma B.2.2. If \( E_s = f_s(E_p) \), forall \( P_s \) and \( P_p \) such as \( P_s = g_s(P_p) \), if \( E_p \vdash P_p \) then \( E_s \vdash P_s \).

Proof. By induction on the predicate \( P_p \).

- If \( P_p = P_1 \land P_2, P_s = g_s(P_1) \land g_s(P_2) \). \( E_p \vdash P_p \), so \( E_p \vdash P_1 \) and \( E_p \vdash P_2 \). By the induction hypothesis, we have \( E_s \vdash g_s(P_1) \) and \( E_s \vdash g_s(P_2) \). Hence \( E_s \vdash P_s \).
- The other induction cases follow the same structure.

B.3 Correctness of the transformation

In this section, we prove that if we give using Why a proof that the generated sequential program is correct, then the original program is correct.

B.3.1 Correctness of the execution

First, we need to prove that if the sequential execution gives a result, then the parallel execution gives the same result.

Lemma B.3.1. Let \( b \) be a block.

\[ E_s, [[b]] \rightarrow s E'_s \Rightarrow \forall i, E_s[i], [[b]]_i \rightarrow_{si} E'_s[i] \]

Proof. By the lemma[B.1.1] for all \( i, [[c]]_i \) is \( i \)-local. \([ [b]] \) is a for loop that executes all the \([ [c]]_i \) in turn, hence the result.

Lemma B.3.2. Let \( b \) be a \( i \)-local expression. If \( E_s, [[c]]_i \rightarrow_{si} E'_s \), then for \( E_{pi} = f_{pi}(E_{si}) \) and \( E'_{pi} = f_{pi}(E'_s) \), \( E_{pi}, e \rightarrow_{pi} E'_{pi}, END \).

Proof. By induction on \( e \):

- If \( e \) is \( e_1; e_2 \), then \([ [e]]_i = [[e_1]]_i; [[e_2]]_i \). \( e \) is \( i \)-local, so \( e_1 \) and \( e_2 \) are \( i \)-local too. Let \( E_s \) and \( E'_s \) such as \( E_s, [[e_1]]_i; [[e_2]]_i \rightarrow_{si} E'_s \), with \( E_s = E_s[i], E'_s = E'_s[i] \). By the semantics rule, we have a \( E''_s \) such as \( E_s, [[e_1]]_i \rightarrow_{si} E''_s \) and \( E''_s, [[e_2]]_i \rightarrow_{si} E'_s \). Then \( E_{si}, [[e_1]]_i \rightarrow_{si} E''_s \) and \( E''_s, [[e_2]]_i \rightarrow_{si} E'_s \), with \( E''_s = E''_s[i] \). Using the induction hypothesis, we have \( E_{pi}, e_1 \rightarrow_{pi} E''_{pi}, END \) and \( E''_{pi}, e_2 \rightarrow_{pi} E'_{pi}, END \), with \( E''_{pi} = f_{pi}(E''_s) \). By the semantics rule of the sequence, we obtain \( E_{pi}, e_1; e_2 \rightarrow_{pi} E'_pi, END \), hence the result.
- The other cases of control flow instructions follow the same structure of proof.
- For the communication functions, we have to assume that the axiomatisation given in the BSP-Why prefix file corresponds to the BSP-Why semantics rule, as expected.
Lemma B.3.3. Let $b$ be a block.
For all sequential environment $E_s$ and $E_s'$, if $E_s, [(b)] : s E_s'$, then with $E_p$ and $E_p'$ defined by $E_s = f_s(E_p)$ and $E_s' = f_s(E_p')$, we have $E_p', b : s E_p'$.

Proof. By Lemma B.3.1 we have $\forall i, E_s, [(i)] : s E_s'$, Using Lemma B.3.2 we get $\forall i, E_{pi}, b : s E_{pi}'$, END. Then the first global reduction rule of the semantics gives $E_p, b : s E_p'$.

Definition There is a natural bijection between a code $c$ and its block decomposition $\langle c \rangle$. In order to simplify the writing of the proofs, we note $E_p, t : s E_p'$ with $ab$ a block tree, if we have $E_p, c : s E_p'$ with $c$ the associated code.

Lemma B.3.4. If $E_p, t_1 : s E_p'$ and $E_p', t_2 : s E_p''$, then $E_p, t_1; t_2 : s E_p''$.
Proof. By induction.

Lemma B.3.5. Let $t$ be a parallel program in its block decomposition form.
For all sequential environment $E_s$ and $E_s'$, if $E_s, [[t]] : s E_s'$, then with $E_p$ and $E_p'$ defined by $E_s = f_s(E_p)$ and $E_s' = f_s(E_p')$, we have $E_p, t : s E_p'$.

Proof. The proof is done by induction on the block tree $t$.

- If $t$ is a block $b$, we can apply the lemma B.3.3 which gives the result $E_p, b : s E_p'$.
- If $t$ is $t_1; t_2$. Then $[[t]] = [[t_1]]; [[t_2]]$, so $E_s, [[t_1]]; [[t_2]] : s E_s'$. By the semantics rule of the sequence, we have $E_s, [[t_1]] : s E_s'$ and $E_s, [[t_2]] : s E_s'$. By induction, $E_p, t_1 : s E_p''$ and $E_p', t_2 : s E_p''$, with $E_p'' = f_s(E_p'')$. Then, with the lemma B.3.4 we have $E_p, t_1; t_2 : s E_p''$.
- If $t$ is loop $t$. Then $[[t]] = \text{loop} [[t]]$, so $E_s, \text{loop} [[t]] : s E_s'$. By the semantics rule of the loop, we have $E_s, [[t]] : s E_s''$ and $E_s', \text{loop} [[t]] : s E_s'$. By induction, $E_p, t : s E_p''$ and $E_p', \text{loop} t : s E_p''$, with $E_p'' = f_s(E_p'')$. Then, with the lemma B.3.4 we have $E_p, \text{loop} t : s E_p''$.
- The other inductive cases follow the same kind of proof.

Lemma B.3.6. Let $c$ be a parallel program.
For all sequential environment $E_s$ and $E_s'$, if $E_s, [[c]] : s E_s'$, then with $E_p$ and $E_p'$ defined by $E_s = f_s(E_p)$ and $E_s' = f_s(E_p')$, we have $E_p, c : s E_p'$.

Proof. This is a direct consequence of the lemma B.3.5.

B.3.2 Correct transformation

With the help of the lemmas given above, we can now prove the correctness and completeness of the transformation.

Theorem B.3.7 (Correctness). If $P_s = g_s(P_p)$, $P_s' = g_s(P_p')$, if $\{P_s\} [[\langle c \rangle]] \{P_s'\}$ then $\{P_p\} c \{P_p'\}$.
Proof: Let $E_p$ such as $E_p \vdash P_p$. Let $E_s = f_s(E_p)$. Let $E_s'$ be the result of the execution $E_s$, $|(c)| \rightarrow_s E_s'$, and $E_p' = f_p(E_s')$. Then by the Lemma B.3.6 we have $E_p, c \rightarrow_p E_p'$. By Lemma B.2.2 we have $E_s \vdash P_s$. Then, since $\{P_s\} \{|(c)|\} \{P_s'\}$, we can deduce $E_s' \vdash P_s'$. We can then apply the Lemma B.2.1 which gives $E_p' \vdash P_p'$. Hence $\{P_p\} c \{P_p'\}$ is a valid Hoare triplet.

### B.4 Completeness of the transformation

In this section, we prove that if the original program is correct, then the Hoare triplet composed of the precondition, code and postcondition of the sequential simulation is correct too.

#### B.4.1 Correctness of the execution

First, we need to prove that if the parallel execution gives a result, then the sequential execution gives the same result.

**Lemma B.4.1.** Let $b$ be a block.

$$\forall i, E_s|_i, [b]|_i \rightarrow_s E_s'|_i \Rightarrow E_s, [b] \rightarrow_s E_s'$$

**Lemma B.4.2.** Let $e$ be a i-local expression. If $E_{pi}, e \rightarrow_{pi} E_{pi}', \text{END}$, then for $E_{pi} = f_{pi}(E_s|_i)$ and $E_{pi}' = f_{pi}(E_s'|_i)$,

$$E_s|_i, [e]|_i \rightarrow_s E_s'|_i.$$  

**Proof.** By induction on $e$. \qed

**Lemma B.4.3.** Let $b$ be a block.

For all parallel environment $E_p$ and $E_p'$

1. if $E_p, b \rightarrow_p E_p'$

then with $E_s$ and $E_s'$ defined by $E_s = f_s(E_p)$ and $E_s' = f_s(E_p')$

we have $E_s, [b] \rightarrow_s E_s'$.  

**Proof.** By the first global reduction rule of the semantics gives, we have $\forall i, E_p, b \rightarrow_p E_p', \text{END}$. Using Lemma B.4.2 we get $\forall i, E_s|_i, [b]|_i \rightarrow_s E_s'|_i$. Then with the lemma B.4.1 we have $E_s, [b] \rightarrow_s E_s'$. \qed

**Lemma B.4.4.** If $E_p, t_1; t_2 \rightarrow_p E_p''$, there exists $E_p'$ such as $E_p, t_1 \rightarrow_p E_p'$ and $E_p', t_2 \rightarrow_p E_p''$.

**Proof.** By induction. \qed

**Lemma B.4.5.** Let $t$ be a parallel program in its block decomposition form.

For all parallel environment $E_p$ and $E_p'$

1. if $E_p, t \rightarrow_p E_p'$

then with $E_s$ and $E_s'$ defined by $E_s = f_s(E_p)$ and $E_s' = f_s(E_p')$

we have $E_s, [t] \rightarrow_s E_s'$. \qed

**Proof.** The proof is done by induction on the block tree $t$.

- If $t$ is a block $b$, we can apply the lemma B.4.3 which gives the result $E_s, [b] \rightarrow_p E_s'$.

- If $t$ is $t_1; t_2$. Then $[t] = [t_1];[t_2]$. $E_p, t_1; t_2 \rightarrow_p E_p'$. By the lemma B.4.4 we have $E_p, t_1 \rightarrow_p E_p'$ and $E_p'', t_2 \rightarrow_p E_p$. By induction, $E_s, [t_1] \rightarrow_s E_s'$ and $E_s', [t_2] \rightarrow_s E_s'$, with $E_s'' = f_s(E_p'')$. Then, with the semantics rule of the sequence, we have $E_s, [t_1];[t_2] \rightarrow_s E_s'$.

- The other inductive cases follow the same kind of proof.
Lemma B.4.6. Let $c$ be a parallel program.
For all parallel environment $E_p$ and $E'_p$,
if $E_p, c \rightarrow_p E'_p$,
then with $E_s$ and $E'_s$ defined by $E_s = f_s(E_p)$ and $E'_s = f_s(E'_p)$,
we have $E_s, [[(c)]] \rightarrow_s E'_s$.

Proof. This is a direct consequence of the lemma B.4.5.

Theorem B.4.7 (Completeness). If $P_s = g_s(P_p), P'_s = g_s(P'_p)$, if $\{P_p\} c \{P'_p\}$, then $\{P_s\} [[(c)]] \{P'_s\}$.

Proof. Let $E_s$ such as $E_s \vdash P_s$. Let $E_p = f_p(E_s)$. Let $E'_p$ be the result of the execution $E_p, c \rightarrow_p E'_p$, and $E'_s = f_s(E'_p)$. Then by the Lemma B.4.6 we have $E_s, [[(c)]] \rightarrow_p E'_s$. By Lemma B.2.1 we have $E_p \vdash P_p$. Then, since $\{P_p\} c \{P'_p\}$, we can deduce $E'_p \vdash P'_p$. We can then apply the Lemma B.2.2 which gives $E'_s \vdash P'_s$. Hence $\{P_s\} [[(c)]] \{P'_s\}$ is a valid Hoare triplet.