BSP-WHY: an Intermediate Language for Deductive Verification of BSP Programs

Jean Fortin
LACL, University of Paris-East,
61 avenue du Général de Gaulle, 94010 Créteil, France
jean.fortin@ens-lyon.org

Frédéric Gava
LACL, University of Paris-East,
61 avenue du Général de Gaulle, 94010 Créteil, France
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 generators) of BSP extensions of realistic programming languages such as C, JAVA, *etc.* 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 automatic decision procedures. In this manner, BSP-Why is able to generate proof obligations for BSP programs.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.1.2 [*Computation by Abstract Devices*]: Modes of Computation, Parallelism and concurrency

General Terms Languages, Verification

Keywords Hoare Logic, Parallel Programs, BSP

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 and their complexity, a frontal attack of the problem of verification of parallel programs is a daunting task that is unlikely to materialize. Some works on the standard MPI exist [Siegel 2007, Vo et al. 2009] 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 does not contain any possible deadlocks. But it is impossible to verify this property for any number of processors, which is a scaling problem.

HLPP'10, September 25, 2010, Baltimore, Maryland, USA.

Another approach would be to consider well-defined subsets that include interesting structural properties. In fact, many parallel programs are not as unstructured as they appear: it is the skeletons [Cole 2004] and BSP [Bisseling 2004] main idea.

Also, avoiding deadlocks is not sufficient to ensure that the programs will not crash. Mainly, we need to check buffer and integer overflows (safety) and liveness. For critical systems and libraries, one can also want a better trust in 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 (proof obligations to provers) as output to ensure correctness of the properties given in the annotation[1]. An advantage of this approach is to allow the mixing of the manual proof of properties using proof assistants and automatised checks of simple properties using automatic decision procedures.

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 proof obligations using a VCG.

1.2 Which parallel model?

In this article, we restrict our study to the BSP framework[2]. In fact, many parallel programs fit the BSP model even if many authors do not know it. For example, all MPI sub-programs that only use collective routines (or using the API of [Ghuloum et al. 2007]) can be considered as BSP programs.

A BSP program is executed as a sequence of *super-steps* [Skillicorn et al. 1997]. This structural property of the parallel computation is well known by the parallel algorithmic community for doing static [Bisseling 2004] and runtime [Bamha and Exbrayat 2003] optimisations. It also looks like a good candidate for formal verification [Jifeng et al. 1996]: its model of execution is simpler to understand than any concurrent model.

The structural nature of the BSP programs will allow us to decompose the programs into sequences of blocks of code, each block corresponding to a super-step, and help in the generation of the proof obligations.

1.3 Which condition generator?

Writing proof assistants and VCGs requires a tremendous amount of work which should be left to the field experts. The main idea of

[1] Even if many 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 exists some tools [Kovács and Voronkov 2009] that automatically provide annotations for some properties. But defining the exact meaning of a computation is clearly a human work. Adapting these kind of tools is not the subject of this paper.

[2] We refer to [Bisseling 2004, Skillicorn et al. 1997] for a gentle introduction to BSP and its C libraries.

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.

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 constructs of a full language. Instead, realistic programming languages can be compiled into the Why input language. Why currently interprets C and JAVA programs with the help of two companion tools [Filliâtre and Marché 2004, 2007]. BSP-Why would serve as intermediary for C [Bisseling 2004] or JAVA [Bonorden et al. 2006] BSP extensions similarly to the way 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 proof obligations. This allows us to use these provers for the proof obligations obtained from the parallel programs.

1.4 Contribution of this paper and Outline

In this paper, we design a core-language (Section 3), as an extension 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 practical programs — *e.g.* the ones of [Bisseling 2004, Dehne 1999]. Our core-language is also close to standard BSP programming: primitives offer both message passing and remote memory access. We give the operational semantics of BSP-Why in Section 3. We used the Why core language as a back-end of our own BSP-Why[3] core language.

We give the axiomatisation of the BSP primitives in Why in Section 4. In Section 5, we transform BSP-Why programs into Why, then we prove that the generated Why programs (with annotations) are as intended: the parallel program and its sequential simulation run in the "same manner". We present some simple examples of BSP algorithms in Section 6 to demonstrate the usefulness of this method and its limits.

2. Why verifying programs: examples of crashes

Even if the BSP model is theoretically deadlock and livelock free and simplifies the writing of parallel codes, many errors, even deadlocks, 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 C code:

```
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:

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

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

```
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();
```

[3] The prototype BSP-Why tool is available at `http://lacl.fr/fortin/BSP-why/`

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

Our last example is not really an error since it does not crash the machine but gives undeterministic 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 example:

```
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 of the processors. We have currently chosen the second case since we assume a deterministic semantics of BSP programs but changing to the first case is trivial to do.

Many other errors can be cited: forgetting to register a variable, 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.

3. The BSP-Why intermediate language

3.1 The Why framework

Implementing a verification condition generator (VCG) for a realistic programming language needs a lot of work: too many constructs require a specific treatment[4]. Reducing the VCG to a core language thus seems a good approach.

The input syntax of the VCG Why is an intermediate and specific alias-free ML language dedicated to program verification (`http://why.lri.fr/`). As a programming language, it is an 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 [Filliâtre and Marché 2007]) (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 completed by a functional interpretation of the imperative features [Filliâtre 2003], incorporating exceptional post-conditions and computation of effects over mutable variables.

The Why language also provide possibility of defined axioms, pure logical assertions and parameters: primitives that have type definitions (with possibles side-effects and logical assertions) but no implementation.

Annotations in Why are Hoare triple $\{P\}c\{R\}$ where P (precondition) and R (post-condition) denote logical expressions and c denotes a source-code fragment. Informally, this means "If P holds before the execution of c, then R will hold after the execution". Classically in Hoare logic, loops require to find an appropriate formula, P, which is true before and after the loop and preserved by each execution of the loop body. P is called an *invariant*. To find it requires some intuition in many cases. It merely assures us that, if the loop has terminated, the post-condition holds. To guarantee that the loop will always terminate, another formula

[4] This also happens when modeling MPI [Li et al. 2008]; it forces to use the smallest possible number of routines.

```
parameterg x: int ref
parameterg z: int ref

logic sigma_prefix : int fparray,int → int
axiom sigma_prefix1 :
  forall t:int fparray. sigma_prefix(t,−1) = 0
axiom sigma_prefix2 : forall i:int. isproc(i) →
  forall t:int fparray.
    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 < nprocs) do
       {
         invariant (y > (bsp_pid)) and
          envCsendIs(j,bsp_pid + 1,y,lcast_int(x))
          and modify(envCsend)
           variant nprocs − y
       }
       bsp_send !y (cast_int !x);
       y := !y + 1
     done);
  { envCsendIs(j,bsp_pid + 1,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) and modify(z)
           variant bsp_pid − y
       }
       z := !z + uncast_int (bsp_findmsg !y 0);
       y := !y + 1
     done )
       { z=sigma_prefix(<x>, bsp_pid)}
```

```
parameter x : int fparray ref
parameter z : int fparray ref

logic sigma_prefix : int fparray,int → int
axiom sigma_prefix1 :
  forall t:int fparray. sigma_prefix(t,−1) = 0
axiom sigma_prefix2 : forall i:int. isproc(i) →
  forall t:int fparray.
    sigma_prefix(t,i) = paccess(t,i) + sigma_prefix(t,i−1)

let prefixes () = { comm_empty(envCsend) }
  proc_i := 0 ;
loop0: while (!proc_i < nprocs) do
       {
           invariant (proc_i>=0) and (...)
           variant nprocs − proc_i
       }
    let y = ref (bsp_pid (void))+1 in
      while (!y < nprocs) do
      {
        invariant (...)
          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 (...)
           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−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)}
```

Figure 1. BSP-Why code of the direct prefix computation (left) and its Why transform (right)

called *variant* is needed: it defines a measure that strictly decrease. All the annotations are used to generated the output: the proof obligations.

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 [Filliâtre and Marché 2007]. These tools transform annotated programs from these real languages to Why programs. Syntax and semantics of Why can be found in [Filliâtre 2003].

In the case of alias-free programs, simpler proof obligations is the main goal of Why: for communications, we will see that BSP-Why generates obligations that can be "hard" to read; the simpler the obligations for the sequential parts of a program, the less complex they are for the parallel parts.

3.2 Syntax of BSP-Why

The syntax of BSP-Why is the one of Why [Filliâtre and Marché 2007, Filliâtre 2003] with an additional syntax for parallel instructions. For more details see [Fortin and Gava 2010] or the Why syntax available at `http://why.lri.fr/`.

A program is a list of declarations. A declaration is either a definition (logic terms, axioms, parameters, *etc.*) or a program expression or an exception declaration.

Program expressions mostly follow ML's ones. Fig 1 gives an example of a BSP-Why program (left) and a Why program (right). One can remark that they are ML programs with logical annotations inside the brackets. In order to simplify the reading of the semantics and of the paper, parallel operations (notably DRMA — Direct Remote Memory Access primitives) take simple variables as argument, instead of buffers. In practice, BSP-Why does manipulate buffers, and adds proof obligations to avoid buffer overflows.

A special constant $nprocs$ (equal to $\mathbf{p}$) and a special variable `bsp_pid` (with range $0, \ldots, \mathbf{p}-1$) were added to Why expressions. In pure terms (terms without possible side effects), we also have introduced the two special function symbols `bsp_nmsg`(t) and `bsp_findmsg` t_1 t_2: the former corresponds the number of messages received from a processor id t (C function `bsp_nmsgs(t)`) and the latter to get the t_2-th message from processor t_1 (C function `bsp_findmsg(t1,t2)`).

The six parallel operations added in the syntax are: (1) `bsp_sync`, barrier of synchronisation, which holds a pre-condition to describe the environments of communications of the processes at the end of the super-step; (2) `bsp_push` x, registers a variable x for global access; (3) `bsp_pop` x, delete x from global access; (4) `bsp_put` e x y, remote writing of x to y of processor e; (5) `bsp_get` e x y, remote reading from x to y; (6) `bsp_send` e_1 e_2, sending value of e_1 to processor e_2.

For the `bsp_sync` operation, it is necessary, in order to prove the correctness of a program, to give a logic assertion just before the `bsp_sync` instruction. This assertion should describe the computations 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 exe-

cution of the program, *i.e.* the fact that the program will terminate without failing by an array overflow, an illegal message read, *etc.*

From the Why's simple minimal first-order logic, we add the construct $t{<}i{>}$ which denotes the value of a term t at processor id i, and $<x>$ denotes the **p**array x (a value on each processor), in contrast with the notation x which means the value of x on the current processor.

We also add $fparray$ to logical term which is the abstract type for purely applicative arrays of size **p** (with some obvious axioms) and $list$. They are used for the description of **p**-values, one per process and for the environment of communications.

3.3 Operational Semantics of BSP-Why

The semantics of the Why language is a big-step operational one without surprise available at `http://why.lri.fr/`. Thus 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.

Values to be sent and remote reading/writing are stored in the environment 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, and each of them is a special variable of the assertions. Each is a simple list of messages. 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).

The notions of values v and states are the same as in the Why semantics with the additional possible value $\texttt{SYNC}(e)$, which describes the awaiting of a synchronisation with e as program to be executed after the global synchronisation: $v ::= c \mid (Ec) \mid \texttt{rec}\ g\ x = e \mid \texttt{SYNC}(e)$. A value v can be a constant (integer, boolean, *etc.*), an exception E carrying a constant c, a closure $\texttt{rec}\ g\ x = e$ representing a possibly recursive function g binding x to e, or the synchronisation state. It is convenient to add the notion of closure to the set of expressions: $e ::= \ \ldots \mid \texttt{rec}\ g\ x = e$. We also use a natural set of contexts R to deal with exceptions.

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 $s.\mathcal{X}$ to access to the component $\mathcal{X}$ of the environment s, $\oplus$ the update of a component of an environment without modifying other components and $\in$ to test the presence of a data 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.

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 [Fortin and Gava 2010]. 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 $\texttt{SYNC}(e)$ — rules for the put primitive in Fig. 2 gives a good examples of this fact.

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.

Rules of the BSP operations are given in Fig. 2 (executed on a single processor i). Basically, a primitives adds the corresponding message to the environment. We note $n^{\text{th}}(s'.\mathcal{T}, y) = n$ to learn for a variable y how nth is it in the registration $\mathcal{T}$. We also note $Size(s.\mathcal{R}, to)$ to know how many messages from processor 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 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. 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 $\texttt{SYNC}(c)$ state: the communication are effectively done during the synchronisation phase and the current super-step is finished. The $\mathbf{AllComm}$ 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 its execution while others are waiting for a synchronisation, a deadlock will occur. Finally we have the following results.

LEMMA 1. $\forall i \ \ \Downarrow^i$ *is deterministic.*

LEMMA 2. $\Downarrow$ *is deterministic.*

The two lemmas are trivially proved by induction on the evaluation.

4. Parallel memory model

The main idea of our approach is 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 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, 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 prelude files to define basic operations common to all Why programs, we use a bspwhyprelude.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.

4.1 Data types

Several data types are used in the transformation, and are defined in the prelude file. The `fparray` 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 `fparray`. 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` datatype used to transmit any kind of data, and the `rvaluet` type used to represent the values received with `send` messages.

4.2 Communication Environments

As in the semantics, three separates message queues, `send_queue` ($\mathcal{C}^{\text{send}}$), `put_queue` ($\mathcal{C}^{\text{put}}$), and `get_queue` ($\mathcal{C}^{\text{get}}$), are used to store

$$\frac{}{s, \{\mathrm{p}\}\texttt{bsp\_sync} \Downarrow^i s, \texttt{SYNC(void)}} \qquad \frac{s' = s.\mathcal{C}^{\text{push}} \oplus x}{s, \texttt{bsp\_push}\ x \Downarrow^i s', \texttt{void}} \qquad \frac{s' = s.\mathcal{C}^{\text{pop}} \oplus x}{s, \texttt{bsp\_pop}\ x \Downarrow^i s', \texttt{void}} \qquad \frac{}{s, nprocs \Downarrow^i s, \mathbf{p}} \qquad \frac{}{s, pid \Downarrow^i s, i}$$

$$\frac{s, e \Downarrow^i s', to \qquad 0 \leq to < \mathbf{p} \qquad \{x \mapsto c\} \in s'.\mathcal{E} \qquad n^{\text{th}}(s'.\mathcal{T}, y) = n \qquad s'' = s'.\mathcal{C}^{\text{put}} \oplus \{to, c, n\}}{s, \texttt{bsp\_put}\ e\ x\ y \Downarrow^i s'', \texttt{void}}$$

$$\frac{s, e \Downarrow^i s', \texttt{SYNC}(e')}{s, \texttt{bsp\_put}\ e\ x\ y \Downarrow^i s', \texttt{SYNC}(\texttt{bsp\_put}\ e'\ x\ y)} \qquad \frac{s, e \Downarrow^i s', \texttt{SYNC}(e')}{s, \texttt{bsp\_get}\ e\ x\ y \Downarrow^i s', \texttt{SYNC}(\texttt{bsp\_get}\ e'\ x\ y)} \qquad \frac{s, e \Downarrow^i s', \texttt{SYNC}(e')}{s, \texttt{bsp\_send}\ x\ e \Downarrow^i s', \texttt{SYNC}(\texttt{bsp\_send}\ x\ e')}$$

$$\frac{s, e \Downarrow^i s', to \qquad 0 \leq to < \mathbf{p} \qquad \{x \mapsto c\} \in s'.\mathcal{E} \qquad n^{\text{th}}(s'.\mathcal{T}, y) = n \qquad s'' = s'.\mathcal{C}^{\text{get}} \oplus \{to, n, x\}}{s, \texttt{bsp\_get}\ e\ x\ y \Downarrow^i s'', \texttt{void}}$$

$$\frac{s, e \Downarrow^i s', to \qquad 0 \leq to < \mathbf{p} \qquad \{x \mapsto c\} \in s'.\mathcal{E} \qquad s'' = s.\mathcal{C}^{\text{send}} \oplus \{to, c\}}{s, \texttt{bsp\_send}\ x\ e \Downarrow^i s'', \texttt{void}}$$

$$\frac{s, t_1 \Downarrow^i s', to \qquad 0 \leq to < \mathbf{p} \qquad s', t_2 \Downarrow^i s'', n \qquad \{to, n, c\} \in s''.\mathcal{R}}{s, \texttt{bsp\_findmsg}\ t_1\ t_2 \Downarrow^i s'', c} \qquad \frac{s, t \Downarrow^i s, to \qquad 0 \leq to < \mathbf{p} \qquad n = Size(s.\mathcal{R}, to)}{s, \texttt{bsp\_nmsg}(t) \Downarrow^i s, n}$$

Figure 2. Semantics: Local BSP Operations

$$\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, v_i}{(s_0, c_0), \ldots (s_{\mathbf{p}-1}, c_{\mathbf{p}-1}) \Downarrow (s'_0, v_0), \ldots (s'_{\mathbf{p}-1}, v_{\mathbf{p}-1})}$$

$$\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, \texttt{SYNC}(e'_i) \qquad \mathbf{AllComm}\{(s'_0, e'_0), \ldots (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1})\} \Downarrow (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}{(s_0, e_0), \ldots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}$$

Figure 3. Semantics: Global reductions

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` ($\mathcal{C}^{\text{pop}}$) 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 axiomatisation 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.

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 $\mathcal{T}$. 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 $\mathbf{p} = 6$, 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 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 -> 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 -> int
axiom nsend_nil : nsend(lfnil_send) = 0
axiom nsend_cons : forall q:send_queue.
      forall n:int. forall v:value.
        nsend(q) = n -> 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

logic in_send_n : send_queue,int,value -> prop
```

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.* $0 \leq i < \mathbf{p}$).

```
parameter envCsend : send_queue fparray fparray 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)))))}
```

In the next step, we 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 fparray 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.

4.3 Synchronisation

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 fparray fparray,
            envCsend':send_queue fparray fparray,
            envR':rvaluet fparray)
 = (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 fparray fparray,
          envCsend':send_queue fparray fparray,
          envR':rvaluet fparray, ...) =
       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.

5. Translation from BSP-Why to Why

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 do the actual transformation of a BSP-Why program into Why that will simulate its parallel execution.

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 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 of the BSP-Why AST, and returns a block tree:

- 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

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 a simple traversal of the tree where we apply recursively the transformation (noted $[[e]]$).

The complete transformation is given in [Fortin and Gava 2010]. 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. $[[Bloc(e)]] = forp([[e]]_i)$ For each inductive case, the transformation simply gives the appropriate structure in the Why language.

Special care must be taken to generate correct loop invariants and variant in the "for" loop executing the sequential code. If the invariant is not strong enough, it will not be possible to prove the resulting program using Why. For this reason, we ask the programmer to write explicitly an assertion at the end of each superstep. It can be the assertion before the $sync$ instruction, or a postcondition if there is no synchronisation afterwards. This assertion is then used to generate the invariant of the sequentialisation loop: since the assertion must be true at the end of the superstep of one processor, the invariant is that after i iterations, the invariant is true for the i first processors.

The invariant also keeps track of which variables are modified, and which are not. Since we are using arrays to represent the variables on each processors, it is necessary to say that we only modify a variable on the current processor, and that the remaining of the array stays unchanged after the iteration of the loop.

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
```

the assertion generated will be

```
assert {forall proc_i, 0<=proc_i<bsp_nprocs →
        (((pow_int(2,paccess(i,proc_i))) < bsp_nprocs)
    ↔ (forall proc_j, 0<=proc_j<bsp_nprocs →
        (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.

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 $[[e]]_i$ — rules in [Fortin and Gava 2010].

5.2.2 Local block transformation

The translation of a single block, denoted by $[[e]]_i$, 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 [Fortin and Gava 2010].

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 $<x>$, 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 (put, $send$, *etc.*) are not directly translated in an equivalent sequential code. They are replaced by calls to the parameters axiomatized in the prelude file.

5.3 Proof of equivalence

We now prove that our program transformation is correct *i.e.*:

- If we prove using 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.

By lack of place, we don't give the full proofs here. They are available in [Fortin and Gava 2010].

5.3.1 Notations

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

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

$E_s,\ c \to E'_s$ denotes that the sequential execution of the program c in the environment E_s gives the environment E'_s.

$E_p,\ c \to E'_p$ 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.

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:

LEMMA 3. *If* $E_s = f_s(E_p)$, $E'_s = f_s(E'_p)$, *if* $E_s,\ [[c]] \to E'_s$ *then* $E_p,\ c \to E'_p$, .

LEMMA 4. *If* $E_s = f_s(E_p)$, $E'_s = f_s(E'_p)$, *if* $E_p,\ c \to E'_p$ *then* $E_s,\ [[c]] \to E'_s$, .

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 `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 with regard to the operational semantics, the next two lemmas concern the correctness of the transformation in the logical point of view.

LEMMA 5. *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 6. *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 1 (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 $E_s,\ [[c]] \to E'_s$, and $E'_p = f_p(E'_s)$. Then by the Lemma 3, we have $E_p,\ c \to E'_p$. 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 2 (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 $E_p,\ c \to E'_p$, and $E'_s = f_s(E'_p)$. Then by the Lemma 4, we have $E_s,\ [[c]] \to E'_s$. 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.

6. Examples

The code not given in the paper is available at the BSP-Why web page: http://lacl.fr/fortin/BSP-why/.

6.1 Parallel prefix reduction

Our first example is a simple one-step parallel prefix reduction that is having $\oplus_{k=0}^{\mathbf{i}} v_i$ 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. 1 (left).

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. We use the user-defined logic term sigma_prefix(X,n_1,n_2) to describe the partial sums, that is $\Sigma_{i=n_1}^{n_2} X[i]$.

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` 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, <x> refers to the global array and x <i> refers to the value of x at processor i. `envCsendIs` is a predefined macro to describe the communication environment, without having to use the intern 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 date in a generic $value$ data type.

The generated Why code is in Fig. 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 5, 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 `parray`. 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.

6.2 Logarithmic parallel prefix 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 $\lceil \log_2 \mathbf{p} \rceil$. 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 a case where our block decomposition fails: not all the processors run the same `bsp_sync` and our tool will genereted unprovable assertions. But the program can be rewritten by factoring the two `bsp_sync`[5]:

```
if (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'));
```

6.3 Horner Method

Suppose we are given a polynomial of degree n, $p(x) = \sum_{i=0}^{n} a_i \times x^i$, where $\forall i\ a_i \neq 0$ and we would like to evaluate $p(x_0)$. Our third example is a BSP version of the Horner method [Gerbessiotis 1993]. We suppose that for each $pid = 0, \cdots, \mathbf{p}-1$, we have $t = \lfloor \frac{n-pid}{\mathbf{p}} \rfloor$ and polynomial $q_{pid}(x) = \sum_{j=0}^{t} a_{pid+pj} \times x^{\mathbf{p}j}$. Now we have: $p(x) = \sum_{pid=0}^{\mathbf{p}-1} q_{pid}(x) \times x^{pid}$. Each of the $\mathbf{p}$ processors computes separately one $q_i(x_0)$ by precomputing $x_0^{\mathbf{P}}$ and using a classical sequential Horner method. Subsequently, all the partial results are sent to processor 0. Then processor 0 evaluates $p(x_0)$ by applying Horner's rule again.

6.4 Parallel sorting algorithm

Our last example is the *sampling sort algorithm* (PSRS) of Schaeffer in its BSP version [Tiskin 1998]. The goal is that the elements of a distributed array (we assume that each processor's array length is $\geq \mathbf{p}^3$) are sorted on each processor and elements of processor i are smaller than the ones of processor $i+1$.

The PSRS algorithm proceeds as follows. First, each processor sorts its own array independently with a sequential sort algorithm. The problem now consists of merging the $\mathbf{p}$ sorted arrays. Each process selects from its array $\mathbf{p}+1$ 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 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 [Fortin and Gava 2010].

6.5 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 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

[5] Note that doing this transformation automatically is perhaps possible in some specific cases but this is not the subject of this article.

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), Gappa (0.13.0) and CVC3 (2.2).

Table 1. Number of proof obligations generated / discharged

Program	Correctness/AP	Safety/AP
Direct Prefix	37/37	19/19
Log Prefix	41/37	21/19
Horner	31/30	17/17
BSP Sort	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 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 $\mathbf{p}$ for local computations. That also generated harder proof obligations for the provers. Furthermore, automatic provers are also work in progress. For example, logarithm's (and power-of-two) properties are not currently well interpreted by any automatic provers 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. This is a work in progress.

7. Related Works

7.1 Concurrent programs

There are now some studies of proof obligations for concurrent programs, for example [O'Hearn 2007] presented a Concurrent Separation Logic as an extension of Separation Logic for reasoning about shared-memory concurrent programs with Dijkstra semaphores. [Hobor et al. 2008] 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 [Gotsman et al. 2007]. However, currently no tools for generated proof obligations are provided and it is not clear how much harder the obligations would be.

In the same way, [Leino and Muller 2009] 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. [Jacobs et al. 2006] extends the Boogie framework for concurrent programs with a kind of locking strategy.

It is not clear whether locking strategies are very suitable for high-performance applications [Lee 2006]. BSP is by nature a coarse-grained and deadlock-free model which is used for high-performance problems [Bisseling 2004] and now in multi-core applications [Ghuloum et al. 2007].

Even if proof of concurrent programs is clearly useful (servers, *etc.*), parallel programming is not concurrent programming. High-performance programs are much simpler [Lee 2006] (many time more coarse-grained) and BSP programs are even simpler. They can clearly be simulated by shared-memory fork/lock concurrent programs by explicitly separating the local memories and allowing communications by copies of the data [Tiskin 1998]. 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.

7.2 MPI programs

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 lack of formal specifications — even in works such as [Li et al. 2008, Vakkalanka et al. 2009]. There are many works and tools dedicated to MPI. Surveys could be found in [HPCBugBase, Sharma et al. 2007]. 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.

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 certainly 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.

7.3 Proof of BSP programs

Different approaches for proofs of BSP programs have thus been studied as formal proof of BSP functional programming using Coq [Gava 2003].

There is also the derivation of BSP imperative programs using Hoare's axiom semantics ([Chen and Sanders 2004] is the last we have found) following by the generatation of correct C code [Zhou and Chen 2005]. The two main drawbacks of this approach is that it is not real programs that are analysed and they used their own language of specifications with their own axiomatic: there is no implementation of a dedicated tool for the logical derivation which is a lack of safety; users make hand proofs, not machine checked.

8. Conclusion and Future Work

The paper presents a methodology and its associated tool, called BSP-Why for deductive verification of BSP programs. An extension of the Why intermediate language is proposed, adding some constructs specific to BSP parallelism. An implemented tool translates a subset of the BSP-Why programs (those that are "well-structured" enough) into plain sequential programs. The output programs rely on a generic library of logical axioms and definitions. Since BSP uses barrier synchronisation, parallelism can be removed by replacing a portion of code between barriers with a

loop to repeat that portion for every process. The correctness proof of this translation is sketched — details are available in [Fortin and Gava 2010]. 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 many works — in theory and in practise. Some examples are given and how many generated proof obligations were automatically discharged. In view of this first ratio "number to prove / proved automatically", we believe this method is far from perfect (notably for correctness) but nonetheless can rapidly increase the confidence that can be placed in the code since at least the safety properties are massively proved automatically.

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 [Filliâtre and Marché 2007]. The tool for sequential programming is now a plugin called Jessie for the Frama-C tools (`http://frama-c.com/`) which generates Why programs from C ones. Jessie generates a memory model for the pointers of the C programs which can be clearly identify in the extracted Why codes. We are currently working to adapt the parser to identify the parameters of the memory model for managing more easily the DRMA primitives. We also need to test our method on realistic BSP computations even if the results of our examples are encouraging. Programs of [Bisseling 2004, Dehne 1999] will be used.

Second, BSP is an interesting model because it features a 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.

Third, conditions generated by Why from BSP-Why programs are not friendly. In automatic theorem provers, you don't see them at all but in case of manual proof that could be problematic. This is mainly due to the massive use of the generated **p**-loops: special tactics are needed to simplify the analysis. In the same manner, syntactic sugar to manipulate the environment of communications (list of messages) are needed as an "user library" 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 (*e.g.* MPI's collective primitives). Analysing MPI/C programs to find which are BSP-like and to interpret them in BSP-Why is a great challenge.

References

M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003.

R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

O. Bonorden, J. Gehweiler, and F. Meyer auf der Heide. A Web Computing Environment for Parallel Algorithms in Java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006.

Y. Chen and J. W. Sanders. Logic of global synchrony. *ACM Transactions on Programming Languages and Systems*, 26(2):221–262, 2004.

M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.

J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004. `http://why.lri.fr/caduceus/`.

J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, LNCS. Springer-Verlag, 2007.

J. Fortin and F. Gava. Bsp-why: an intermediate language for deductive verification of bsp programs. Technical Report TR-LACL-2010-07, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (UPEC), 2010.

F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.

A. V. Gerbessiotis. *Topics in Parallel and Distributed Computation*. PhD thesis, Harvard University, 1993.

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.

A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *7th European Symposium on Programming (ESOP)*, LNCS. Springer, 2008.

HPCBugBase. HPCBugBase. `http://www.hpcbugbase.org/`, 2008.

B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *ICFEM*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006.

H. Jifeng, Q. Miller, and L. Chen. Algebraic Laws for BSP Programming. In L. Bougé and Y. Robert, editors, *Euro-Par'96*, volume 1124 of *LNCS*, pages 359–368. Springer, 1996.

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.

E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

K. Rustan M. Leino and Peter Muller. A basis for verifying multi-threaded programs. In *ESOP*, LNCS. Springer, 2009.

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, 2008.

P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.

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. `http://www.cs.utah.edu/research/techreports.shtml`.

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.

D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998.

S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced execution semantics of MPI: From theory to practice. In *FM 2009*, 2009.

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.

J. Zhou and Y. Chen. Generating c code from logs specifications. In D. Van Hung and M. Wirsin, editors, *ICTAC*, volume 3722 of *LNCS*, pages 195–210. Springer, 2005.