BSP-Why: A Tool for Deductive Verification of BSP Algorithms with Subgroup Synchronisation

Article *in* International Journal of Parallel Programming · March 2015

DOI: 10.1007/s10766-015-0360-y

2 authors, including:

Frédéric Gava
Université Paris-Est Créteil Val de Marne - Université Paris 12
64 PUBLICATIONS **403** CITATIONS

SEE PROFILE

Int J Parallel Prog (2016) 44:574–597
DOI 10.1007/s10766-015-0360-y

BSP-Why: A Tool for Deductive Verification of BSP Algorithms with Subgroup Synchronisation

Jean Fortin[1] · **Frédéric Gava**[1]

Received: 14 August 2014 / Accepted: 10 March 2015 / Published online: 31 March 2015

Abstract We present BSP-WHY, a tool for deductive verification of BSP algorithms with subgroup synchronisation. From BSP programs, BSP-WHY generates sequential codes for the back-end condition generator WHY and thus benefits from its large range of existing provers. By enabling subgroups, the user can prove the correctness of programs that run on hierarchical machines—e.g. clusters of multi-cores. In general, BSP-WHY is able to generate proof obligations of MPI programs that only use collective operations. Our case studies are distributed state-space construction algorithms, the basis of model-checking.

Keywords BSP · Verification · Subgroup synchronisation · State-space

1 Introduction

Context of the work Because parallel code is the norm in many areas, *formal verification* [19] of parallel programs is necessary. Indeed formal verification seems essential when considering the growing number of parallel architectures (e.g. GPUs, multi-cores, etc.), the complexity of distributed architectures and the *cost* of conducting large-scale simulations, the losses due to faulty programs, unreliable results, *unexpected crashing* simulations, etc. [24] This is especially true when parallel programs are executed on architectures which are expensive and consume many resources. Therefore, it seems more appropriate to find errors before the said programs are executed.

✉ Jean Fortin
jean.fortin@ens-lyon.org

Frédéric Gava
frederic.gava@univ-paris-est.fr

[1] LACL, University of Paris-East, Créteil, France

In the last decades, methods have been developed to study the *correctness* of sequential programs [19]. For the verification of parallel ones it should be possible to avoid starting again. A question immediately follows: how easy is it to adapt these methods (developed to ensure the safety of sequential programs) for a parallel environment? Given the strong *heterogeneity* of the 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.

In this paper, we propose a method of "*sequentialisation*" [10] of *annotated* parallel programs. Annotations are added to programs in order to verify some properties such as the absence of failure, overflows, liveness, etc. and total correctness, that is to check if results are as intended. In this way, if we can have a *machine-checked* proof that the (generated) sequentialized annotated parallel program is correct, then the original program is correct too. There is thus no need of new specific complex tools for checking correctness of parallel programs. Therefore, the goal is using the now "well defined" verification tools of sequential programs as back-ends for tools of parallel programs.

Which model for parallelism? An approach is 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 and BSP (**B**ulk **S**ynchronous **P**arallel [36]) main idea. A *bridging model* [36] such as BSP has great advantages in respect to the correctness of parallel programs. Take for example a proof of correctness of a GPU-like program. Although interesting in itself it cannot be used directly for clusters of PCs. A bridging model has the advantage that if a program is correct, then this is the case for "all" physical architectures. Note that the problem does not arise for *portable* libraries such as MPI but algorithm design would be clearly *architecture independent*, which will not be the case using a bridging model. Moreover, it is known and accepted that correctness of programs is more costly in terms of work than just programming and designing algorithms. Hence the choice in this article of BSP to provide both *portability* for proofs of correctness and a cost model for algorithmic design of *efficient* programs [1].

The choice of allowing *subgroup synchronisation* is motivated by the fact that some BSP libraries already allow subgroups, for instance the PUB [2]. Moreover, a large number of MPI programs use only global communications [4,32]. These can be viewed as BSP programs, if we allow BSP programs to synchronise over a subgroup of processes. Subgroup synchronisation is also convenient for clusters of multi-cores and *hierarchical* architectures in general [37]. In order to be able to study these kinds of programs, we extend our previous work on verifying strict BSP programs [10] for those that use subgroups.

Which tool of correctness? **V**erification **C**ondition **G**enerators (VCGs) are tools that work properly for checking annotated programs: they take an annotated program as input and produce *verification conditions* (also called goals or proof obligations) as output to provers to ensure correctness of the properties given in the annotations. The key advantages of using a VCG are: it allows the verification of simple properties of a program (such as "no overflow") without formally proving its total correctness; using *automatic provers* enables the quick detection of simple errors; the manual proof of

properties ("programming with a theorem prover" such as COQ) can be mixed with automatised checks of simple properties using automatic provers.

For this work, we choose the VCG WHY [8]. It takes as input a small *core-language* close to ML, avoiding us the need 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, ADA and JAVA programs with the help of companion tools. Our BSP-WHY would serve as intermediary for C [2,18] and JAVA [28,40] BSP extensions, like WHY is for sequential programming. In addition, WHY is interfaced with the main theorem provers (automatic or not) as back-end for the proofs obligations. This allows us to use these provers for the proof obligations obtained from the BSP programs with subgroups.

Contribution of this paper and outline The objective of this work is to provide a tool for the formal verification of logical properties of annotated BSP (well-structured) programs with subgroup synchronisation. Writing a theorem prover or a VCG is a huge task which should be left to the experts. The main idea of our work is to *simulate* BSP parallelism by transforming the parallel code into a sequential form. 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 of a group. Our BSP-WHY tool (available at http://lacl.fr/fortin/BSP-why/) works by transforming the parallel code (with logical annotations) into a *semantically equivalent* WHY code to generate proof obligations. Another advantage of generating a sequential program with assertions is that we would be able to use any kind of dedicated tools for code analysis that work on annotated programs; thus avoiding the need to recreate these tools for BSP programs. Furthermore, implementing a formal tool for a realistic programming language (such as C or JAVA) needs a lot of work even for a big team: too many constructs require specific treatment. Reducing the work to a core language is a good approach: it is a first step which helps to measure the difficulties caused by the BSP model (with subgroups) while avoiding the complexity of a real-world language. Subsequently, this work can be extended to a real-world language with a sufficient team.

Throughout this paper, we illustrate our work with the example of the *state-space* construction of systems, which is the basis of model-checking (MC for short) [6]. Parallelizing the construction of the state-space on several machines is a standard method more and more used in the industry [12]. The correctness of their answers is thus important.

The remainder of this paper is structured as follows. We first give in Sect. 2 an overview of the WHY tool (Sect. 2.1) following with a short presentation in Sect. 2.2 of how the BSP-WHY tool (without subgroups) works. We also present how to program with the subgroup synchronisation in the PUB [2] and MPI context (Sect. 2.3), following with typical errors. We then present in Sect. 3 how to adapt the BSP-WHY tool to manage subgroups. We then give in Sect. 4 the total correctness of a BSP+subgroups state-space construction algorithm. Section 5 discusses some related work and finally, Sect. 6 concludes the paper and gives a brief outlook on future work.

2 Context and General Definitions

2.1 Deductive Verification of Sequential Algorithms Using WHY

2.1.1 The WHY *Tool*

WHY [8] is a framework for the verification of algorithms. It is composed of two parts: a logical language and an intermediate programming language called WHY-ML with a VCG. WHY takes an algorithm with *logical assertions* and generates goals to theorem provers that ensure the correctness of the algorithm. The logic of WHY is a *polymorphic first-order logic* with definitions and axioms. WHY's library proposes finite sets of data and several operations with their axiomatisation. In the logical formula, x@ is the notation for the value of x in the pre-state, i.e. at the precondition point and x@label for the value of x at a certain point (marked by a label) of the algorithm.

WHY-ML is a *first-order language* that provides the usual constructs of imperative programming. All symbols from the logic can be used in the algorithms. Mutable data types can be introduced, by means of polymorphic references: a reference r to a value is created with the function **ref**, is accessed with !r, and assigned with r ← e. Algorithms are annotated using pre- and post-conditions, *loop invariants* (a property that holds before and after each repetition), and *variants* (a strictly decreasing measure) to ensure termination. VCG are computed using a weakest precondition calculus and then passed to the back-end of WHY to be sent to provers. Notice that in WHY, sets are immutable (manipulated only with purely functional routines) and thus only a reference on a set can be modified and assigned to another set.

2.1.2 Example: Verification of a State-Space Construction Algorithm

Model-**C**heckers (MC for short) are often used to verify safety-critical systems. The correctness of their answers is thus vital: many MC produce the answer "yes" or generate a counterexample computation (if a property of the model fails), which forces, in the two cases, to assume that the algorithm and its implementation are both correct. Having trusty state-space is fundamental in order to get MC that you can trust [7].

Let us recall that the finite *state-space construction* problem (also known as the Orbit skeleton [22]) is exploring all the *states reachable* through a *successor* function succ (which returns a set of states) from an *initial state* s_0. This is done without loss of generality and it is a trivial extension to compute the full Kripke structure—usually preferred for checking temporal logic formulas. Generally, during this operation, all the explored states must be kept in memory in order to avoid multiple explorations of a same state. To represent this set StSpace in the logic of WHY, we used the following axiom (for consistency, it has been proved in COQ using an inductive definition of the state-space, also available in the source code):

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

Now ss is the state-space (ss=StSpace) if and only if, the two following *properties hold*: (A) ss ⊆ StSpace and (B) StSpace ⊆ ss. Note that using this first-order

```
1  let random_walk () =
2   let known = ref ∅ and todo = ref {s0} in
3    while todo ≠ ∅ do
4      { invariant (1) (known ∪ todo) ⊆ StSpace
5              and (2) (known ∩ todo)=∅
6              and (3) s0 ∈(known ∪ todo)
7              and (4) (∀ e:state. e ∈known →
8                              succ(e) ⊆ (known ∪ todo))
9      variant |StSpace \ known| }
10     let s = pick todo in
11       known←!known ⊕ s;
12       todo←!todo ∪ (succ(s) \ !known)
13   done;
14   !known {result=StSpace}
```

Fig. 1 A sequential state-space algorithm

definition makes the automatic solvers prove more proof obligations than using an inductive definition for the state-space.

For example, we give in Fig. 1 the usual sequential algorithm in WHY-ML where a set called known contains all the states that have been processed and would finally contain StSpace. It involves a set todo that is used to hold all the states whose successors have not been constructed yet; each state s from todo is processed in turn (lines 3 and 8) and added to known (line 10) while its successors are added to todo unless they are known already—line 11.

Note that this algorithm can be made strictly depth-first by choosing the most-recently discovered state (i.e. implementing todo as a stack), or breadth-first by choosing the least-recently discovered state—i.e. todo as a heap. This has not been studied here and pick randomly chooses a state in the set: we cannot have information about any order in the annotations.

For correctness, the previously presented code needs three *properties*: (a) it does not fail; (b) it indeed computes the state-space; (c) and it terminates. (a) is immediate since the only operation that could fail is pick (where the precondition of this parameter[1] is "not take any element from an empty set") and this is ensured by the guard of the **while** statement. (b) is given by the final post-condition which is only valid after the loop using the four following invariants: (1) known and todo are subsets of StSpace ((A) property); at the end, (3) and (4) known is a subset of StSpace and has the "same" inductive property; and when todo will be empty, then known contains StSpace — (B) property. Finally, termination is ensured by the variant: the algorithm only adds a new state s since (known ∩ todo)=∅; thus known is growing and thus the measure is strictly decreasing.

2.2 Deductive Verification of BSP Algorithms Using BSP-WHY

2.2.1 BSP Model and Programming Without Subgroup Synchronisation

A BSP program is executed as a sequence of *super-steps* —Fig. 2. This *structured* model enforces a strict *separation* of communication and computation: during a super-step,

[1] A parameter is a routine of the program for which we do not know the code; we only have its type and the effect on its arguments.

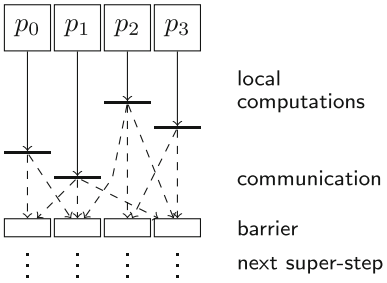

Fig. 2 A BSP super-step

no communication between the processors is allowed but only transfer requests; only at the synchronisation *barrier* information is actually exchanged. Note that a BSP library can send messages during the computation phase of a super-step, but this is hidden to programmers. There exist different BSP programming libraries. The most known are BSPLIB [18,40], PUB [2] and HAMA [28]. An MPI program only using *collective operations* can also be viewed as a BSP program.

The only important routines are those that perform barriers because they terminate a super-step. How BSP communication routines work (with few differences due to the host language) is not of important in this work: mainly, the semantics is growing a data-structure with the messages to be sent and the pending DRMA operations[2], copying buffers of data and modifying the data-structures of received messages only when the barrier is performed—the synchronous effect. For example:

```
1 parameter bsp_send: dest0:int → v:value →
2  {valid_proc(dest0)} unit writes envCsend {envCsend=pupdateSend(envCsend@,pid,dest0,v)}
3 parameter bsp_sync: unit→ {}unit Sync,writes envCsend,
4  ... {comm(envCsend@,envCsend,...)}
```

so that the primitive **bsp_send** updates the set of sending messages using the destination dest0, the value v to send and pid of the processor; **bsp_sync** creates new environments of communication using a relation comm between the past environments and the new ones. Note that as WHY does not allow pointers, we also use a global two-dimensional array to store all variables that need DRMA access. More details could be found in [9,10]. We do not consider in this work high-performance routines of some BSP libraries: even if they are more efficient, they break the abstract/theoretical BSP model of execution. Consequently, they are unsafe and introduce too much non-determinism. For example, if we modify the buffer of an unbuffered sending, it is unknown which value will be sent: for a deductive verification tool, all the possibilities must be kept into account, which is not reasonable.

Throughout this paper, we abstract all communications by using a parameter of *exchange* over arrays of values—it is mainly the MPI's alltoall primitive. Source codes of the examples can be found at http://lacl.fr/gava/cert-mc.tar.gz.

[2] For DRMA operations, in case of a distributed architecture, the buffers have to be sent; in case of a shared-memory architecture, the library simulates this sending.

2.2.2 *The* BSP-WHY *Tool (Without Subgroups) [9,10]*

BSP-WHY extends the syntax of WHY-ML with BSP primitives (message passing of values and synchronisation) and definitions of collective operations. BSP-WHY-ML codes are written in an **S**ingle **P**rogram **M**ultiple **D**ata (SPMD) fashion. We used the WHY-ML language as a *back-end* of our own BSP-WHY-ML language. In this way, BSP-WHY-ML programs are transformed into WHY-ML ones and then the VCG of WHY is used to generate the appropriate goals for the verification of the BSP algorithms. It is thus a kind of "*sequentialisation*": a simulation of a distributed program by a sequential one. The objective is to have a sequential program where we can test to prove the correctness of the final post-condition [9,10]: if it is valid (thus all generated invariants and pre/post-conditions of functions are also valid too) then the post-condition of the parallel program is valid too. In this way, the logical annotations of both parallel and sequential versions of the the program are valid.

A special constant **nprocs** (equal to **p** the number of processors) and a special variable **bsp_pid** (with range $0, \ldots, \mathbf{p}-1$) were also added to WHY-ML expressions. A special syntax for BSP annotations is also provided which is simple to use and seems sufficient to express conditions in most practical programs: we add the construct $t{<}i{>}$ which denotes the value of a term t at processor id i, and $<x>$ denotes a **p** value x (represented by *fparray*, purely applicative arrays of constant size **p**) that is a value on each processor as opposed to the simple notation x which means the value of x on the current processor.

The *transformation* of BSP-WHY-ML codes into WHY-ML ones is based on the fact that, for each super-step, if we execute *sequentially* the code for each processor and then perform the simulation of the communication by copying the data, we have the same validity of the logical properties of the environments as if it is truly done in parallel. We thus have the same logical properties, and it is the charge of the programmer to have sufficiently annotated the program to prove the correctness; for example, proving that the result does not depend of the order of reception of the messages in case of a set of received values.

The first step of the transformation is a decomposition of the program into *blocks* of sequential instructions —Fig. 3. For this, we need to be able to know if an instruction has a *synchronize effect*—doing a barrier or not. Two instructions potentially influence the parallelism of the program: (1) a routine defined with the synchronize effect, such as **bsp_sync**; (2) a function call, if the function body is determined to have a parallel code. The first step of our transformation is to tag each part of the code with a boolean

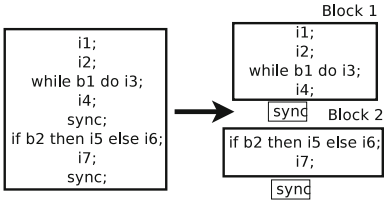

Fig. 3 BSP-WHY's block decomposition

that says whether the "subcode" includes parallel code, or not. This static analysis is fully described in [9].

After having *regrouped* the sequential parts of the code into blocks, we create a "for loop" to execute sequentially the block **p** times. That is the block is executed **p** times, once for each processor. Local variables remain unchanged and any variable that was created outside the block, had been transformed into an array of size **p** (one value per processor) and we thus modify the code of the block accordingly to this fact. Moreover, when making explicit the "for loop", one thing is immediately visible: it is necessary to give an invariant to the loop, if we hope to prove anything about the program. We thus generate invariants to keep track of which variables are modified. Since we are using arrays to represent the variables local to every processor, and programs are run in a SPMD fashion, it is necessary to say that we only modify a variable on the current processor and that the rest of the array stays unchanged. Fortunately, the invariant can in general be inferred automatically. The loop consists of the independent execution of the sequential code e, simulated for the processors 0 to **p**. This means that one iteration of the loop will have executed the code e for one processor. Hence, if we know the post condition $post$ that we would like to ensure after the block e, the invariant should include $\forall j{:}int.\ 0 \leq j < i \rightarrow post[j]$ (for all the processors $j < i$, the code e has already been executed, so the post condition $post$ at the processor j holds) and $\forall j{:}int.\ i \leq j < \mathbf{p} \Rightarrow v[j] = v[j]@loopstart$: the value of v for the processor j must still be the same as the value at the beginning of the loop, denoted by the label @loopstart; ensuring that at the ith iteration of the loop, we have not modified the array for the processors $i + 1$ to $\mathbf{p} - 1$ yet.

Outside the blocks of sequential codes, the code is not altered. However, several cases need more attention: when transforming **if** or **while** statements, there is a risk that a synchronous code (a **bsp_sync** or any code containing a barrier) might be executed on a processor and not on the others—resulting in a program failure; more details in Sect. 2.3.2. We thus enforce that the condition associated with the instruction will always be true on every processor at the same time using a call to a parameter, called `valid`. This parameter is called on the result of the boolean expression and a proof obligation will then be generated by WHY to ensure that every processor enter the same branch of the **if**, or that no processor exits the loop prematurely. Figure 4 gives a naive example to illustrate the method.

Finally, when translating the logic expressions in annotations, it is necessary to translate the variable in the same way as previously. When it is necessary to refer to

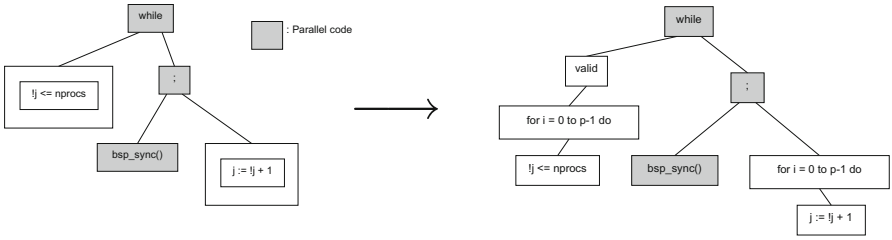

Fig. 4 Example of tagging the code, adding the "valid" condition and the "for loop"

```
1  let bsp_state_space () =
2    let known = ref ∅       and let todo = ref ∅ and
3    let pastsend = ref ∅    and let total = ref 1 in
4      if cpu(s0) = bsp_pid then
5        todo ←s0 ⊕ !todo;
6      while total>0 do
7          { invariant (1) ⋃(<known>) ∪ ⋃(<todo>) ⊆ StSpace
8                  and (2) (⋃(<known>) ∩ ⋃(<todo>))=∅
9                  and (3) (∀ i,j:int. isproc(i) → isproc(j) → total<i> = total<j>)
10                 and (4) total<0> ≥ |⋃(<todo>)|
11                 and (5) s0 ∈(⋃(<known>) ∪ ⋃(<todo>))
12                 and (6) (∀ e:state. e ∈⋃(<known>) → succ(e) ⊆ (⋃(<known>) ∪ ⋃(<todo>)))
13                 and (7) (∀ e:state. ∀i:int. isproc(i)
14                                → e∈ known<i>→ succ(e) ⊆ (known<i> ∪ pastsend<i>))
15                 and (8) ⋃(<pastsend>) ⊆ StSpace
16                 and (9) (∀ i:int. isproc(i) → ∀e:state. e ∈pastsend<i> → cpu(e)≠ i)
17                 and (10) ⋃(<pastsend>) ⊆ (⋃(<known>) ∪ ⋃(<todo>))
18         variant pair(total<0>,| StSpace \ ⋃(known) |) for lexico_order }
19         let tosend=(local_successors known todo pastsend) in
20         exchange todo total !known !tosend
21     done;
22     !known {StSpace=⋃(<result>)}
```

Fig. 5 Parallel (distributed) BSP-WHY-ML algorithm for state-space construction

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 details and some examples are available in [9]. Note the the special treatment for exceptions handling is also available in [9].

2.2.3 Example: Verification of a BSP State-Space Algorithm [9,15]

Algorithm "random_walk" can be parallelised [12] using a *partition* function cpu that returns for each state a processor id, i.e., the processor numbered cpu(s) is the owner of s: **logic** cpu: state → int **axiom** cpu_range: ∀s:state. 0≤cpu(s)<**nprocs**

The idea is that each process computes the successors only for the states it owns. This is rendered as the BSP algorithm of Fig. 5. Sets known and todo are still used but become local to each processor and thus provide only a partial view on the ongoing computation. Initially, only state s_0 is known and only its owner puts it in its todo set. To ensure termination of the algorithm, we use the additional variable *total* in which we count the total number of sent states. It can be noted that the value of *total* may be greater than the intended count of states in *todo* sets. Indeed, it may happen that two processors compute a same state owned by a third processor, in which case two states are exchanged but then only one is kept upon reception—depending on the partition function. In the worst case, the termination requires one more super-step during which all the processors will process an empty *todo*, resulting in an empty exchange and thus $total = 0$ on every processor, yielding the termination. We have thus not used any complicated methods such as the ones presented in [12].

The function local_successors computes the successors of the states in todo where each computed state that is not owned by the local processor is recorded in a set tosend together with its owner number. The set pastsend contains all the states

that have been sent during the past super-steps—the past exchanges. This prevents returning a state already sent by the processor.

The synchronous primitive **exchange** is responsible for performing the actual communications: it computes the set of received states that are not yet known locally (and update todo with this set) together with the new value of *total*—it is essentially the MPI's alltoall primitive. Pastsend is also updated with the union of all states from tosend. The primitive performs a global (collective) synchronisation *barrier* which makes data available for the next super-step so that all the processors are now synchronised.

For brevity, we only present the verification of the parallel part of this algorithm and not the sequential local_successors —similar to "random_walk" but with additional invariants on states to send. They are still available in the source code and fully explained in [9]. We use the following predicates:

- isproc(i) defines what is a valid processor id, that is 0≤i<**nprocs**;
- ⋃(<p_set>) is the union of the sets of the **p**-value p_set, i.e. $\bigcup_{i=0}^{\mathbf{p}-1}$ p_set<i>;

As before, we need to prove that (1) the code does not fail; (2) indeed computes the entire state-space and (3) terminates. The first property follows immediately since only the routine pick is used as before in the local_successors function (we never pick a state in a empty set); and to also prove that the code is well-structured enough to never have a processor doing an extra super-step (the loop contains **exchange** which implies a global synchronisation), we can easily maintain that total (which gives the condition for termination) has the same value on all the processors during the entire execution of the algorithm. Let us now focus on the two other properties.

The invariants (lines 7–17) of the main parallel loop work as follows: (1) as in "random_walk", we need to maintain that known (even distributed) is a subset of StSpace which finally ensures (A) property when todo is empty; (2) as before, the states to be treated are not already known; (3) total is a global variable, we thus ensure that it has the same value on each processor; (4) ensures that no state remains in todo (to be treated) when leaving the loop since total is at least as big as the *cardinality* of todo, total is an over-approximation of the number of sent states; (5–7), as before, ensure (B) property; (8) ensure states sent in the past are in the state-space; (9) pastsend only contains states that are not owned by the processor and (10) all these states, that were sent, are finally received and stored by a processor. In the post-condition (line 22), we can also ensure that the result is well distributed: the state-space is complete and each processor only contains the states it owns according to the function "cpu".

The termination of the main loop has two cases: (a) in general the set known globally (that is for of all processors) grows and we have thus the cardinality of StSpace minus known which is strictly decreasing; (b) if there is no state in any todo of a processor, no new states would be computed and thus total would be equal to 0 in the last stage of the main loop. We thus used a lexicographic order on the two values: sum of known across all processors and total.

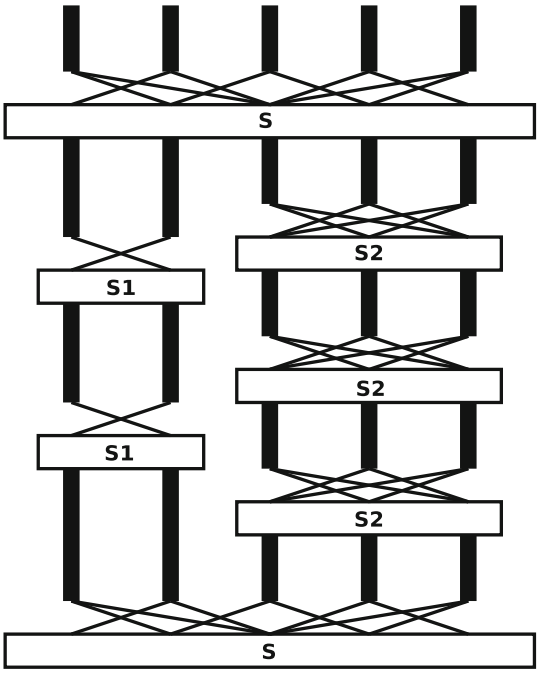

Fig. 6 Example of execution of subgroup synchronisation

2.3 Subgroup Synchronisation

2.3.1 Model

The BSP model is based on global barriers. However, an algorithm may include problems that can be solved using only a subset of the processors. Some libraries thus extend the basic BSP model, and allow the definition of *subgroups*, which are pairwise disjoint *subsets* of the set of processors. It is then possible to write a part of the program with the subgroup acting as an independent BSP computer. A call to **bsp_sync** will then synchronise over the subgroup, instead of the whole machine. In Fig. 6, we show an example of execution of subgroup synchronisation. In this example, the overall group of processors "S" is split into two subgroups ("S1" and "S2") which run independent BSP computations. Finally the two subgroups are merged and the whole machine continues its work. Handling subgroups is an important feature considering cluster of multi-cores: a program can synchronise and communicate only over the cores making it faster.

2.3.2 Programming with Subgroup Synchronisation and Possible Errors

Allowing the synchronisation on a subset of processors means that the communication procedures need to be able to tell in which group they are working. An additional argument, called a *communicator*, is thus added to all the parallel procedures, representing a group of processors linked together.

MPI allows to create sub-communicators and collective operations can then be performed on a subset of the processors. The main routine is MPI_Comm_split(...) which creates a new communicator by partitioning the group into disjoint subgroups using a set of colors. Unlike in the PUB, the MPI routines require a collective call between the processors of the group and there is almost no restriction to the way groups are formed.

For the PUB, the main function is **void** bsp_partition(...). The subgroups are described as a partition of the current BSP computer in contiguous subsets. It is impossible to synchronize or create other subgroups from the parent subgroup until the current subgroup is released with the routine bsp_done. It is however possible to create different new subgroups of the current subgroup. The partition also has to be made of subgroups of consecutive processors. The organisation of subgroups is similar to a stack, with only the lowest subgroup being allowed to create new subgroups. As shown in the next section, our work is independent of these two kinds of routines. We can add a synchronise creation of subgroups *à la* MPI (if the routine has a synchronise effect) or aysnchronous one *à la* PUB or both depending on what the user wants to use. Despite the limitation of the PUB, using subgroups can introduce different errors. In addition to the usual sequential programming errors (out-of-bound or bad results due to non-deterministic communications [9]), a failure can happen when one processor is performing an extra super-step. Considering the following code:

```
1 group1←subgroup(group2,...);
2 if (bsp_pid=0)
3  then bsp_sync(group1);
4  else asynchronous_computation();
5 bsp_sync(group2);
```

where group1 would be a subgroup of group2 (no matter how subgroup works); because processor 0 performs an additional synchronisation, it is a programming error. In some BSP libraries, this translates by a program failure. Generally, it is not a good way of programming. To forbid this case, we force programs (with specific annotations) to be well-structured enough: any synchronous operation must be performed by the entire group. We thus do not allow programs that have the same barrier of the same super-step at different points of the program. A reader can notice that this strict decomposition does not accept all valid BSP programs. Take for example the following code:

```
1 if (bsp_pid=0)
2  then comp1;bsp_sync();comp2;
3  else comp3;bsp_sync();comp4;
```

This is a case where our decomposition fails: not all the processors run the same **bsp_sync** and our tool will generate unprovable assertions. But the program can be rewritten by factoring the two **bsp_sync**[3]:

```
1 if (bsp_pid=0) then comp1;else comp3;
2 bsp_sync();
3 if (bsp_pid=0) then comp2;else comp4;
```

[3] Doing this transformation automatically is perhaps possible in some specific cases; however, this is not the subject of this work.

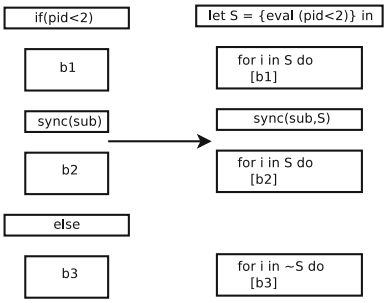

Fig. 7 Scheme of the transformation

In practice and by reading many BSP algorithms (those cited in [1]), we only find this problem in reduction-like (logarithmic) loops where the code can clearly be refactored. This does not seem too restrictive.

3 Managing Subgroups

We have presented so far the transformation of programs in the strict BSP model. In this section, we will show how to extend the above transformation to include the possibility of synchronising over a subgroup of processors.

Figure 7 illustrates the main idea of the simulation of a BSP program with subgroups by a sequential program. The *block decomposition* remains the same as before. However, the **if** statements in the parallel structure need another treatment because they allow the program to branch over different paths of execution depending on the subgroups. A variable S for the set of the processors entering the **if** statement is thus introduced (internally) by BSP-WHY.

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

3.1 Managing the Primitives

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

To do this, we now *dynamically* (automatically) maintain S during the execution of the translated program: S contains the set of the processors that are running the same branch of the code. To avoid failures, for each **bsp_sync**(comm,S) (or every parameter with a synchronize effect), we check that all the processors of a subgroup will synchronise at the same time:
assert{∀i:int i∈S→(comm[i] ⊆ **S and** (∀j:int j∈comm[i]→comm[i]=comm[j]))}.

That is **bsp_sync** is now defined with this *precondition*. It ensures that it is called on a coherent set of processors at any time. For every processor in the set S, which is the set of the processors that will execute the call to **bsp_sync**, the subgroup that includes the processor is included in S. The subgroup of a processor i of S is denoted here by $comm[i]$, since it is an information contained in the communicator argument. The second part of the assertion states that if one processor synchronises over a communicator, then all the other processors of the communicator synchronise on it too.

Well-structured programs If one processor in a subgroup calls the synchronisation on that subgroup, every processors in the subgroup must execute it too. As before, the restriction imposed by BSP-WHY is a bit more restrictive. To ensure that there can not be a failure, BSP-WHY asks that all the processors of a same group that enter the same branch of the **if** synchronise together. However it is still possible for two subgroups to enter two different branches of a **if**. For example, the following code is correct for BSP-WHY:

```
1 let C1 = {0,1} and C2 = {2,3} in
2   if (bsp_pid in C1) then comp1; bsp_sync(C1);
3   else
4     if (bsp_pid in C2) then comp2; bsp_sync(C2);
5     else ...
```

For valid programs that do not meet this restriction, it is often possible to rewrite them using the previous *factorisation*.

3.2 Transformation of Programs with Subgroups

The transformation of programs follows the same general steps as the transformation explained in Sect. 2.2.2. The first step, extracting a *tree of sequential blocks*, is unchanged. The transformation of the block tree (noted $[[e]]_S$) is generally similar to the transformation done by BSP-WHY without subgroup synchronisation. However, as explained earlier, code that do not execute the same for all processors will result in a specific treatment. This includes the **if** and **while** statetments. We give their rules of transformation in Fig. 8. Exceptions also need a specific transformation, but we will not detail it in this article by lack of space. The full rules are available in [9]. They work as follows.

The transformation of a **if** statement is done in three steps. First, the condition is evaluated for all processors in the current subset S. This gives us two new subsets, S_1 and $S \backslash S_1$, respectively the processors of S where the condition is true and the other

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

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

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

Fig. 8 Transformation of **if** and **while** statements

processors. The second step is thus the execution of the first branch for S_1, and the last step the execution of the second branch for $S \setminus S_1$.

The transformation of the **while** loop is more complex, and we give *two variations* of it. The first option closely follows the transformation found in BSP-WHY without subgroup: the valid parameter ensures that the condition remains true on every processor that executes the loop. The only difference is that now, the loop can be executed within a subgroup instead of all the processors. The invariant and variant are thus obtained in the same way as before. In the second option, we enrich the loop by allowing processors to exit it while it progresses. Because of this, we need to update, at each iteration, the set of the processors that are currently executing the loop. This is done by introducing a variable S', which is updated from the computation of the condition on each processor. While the invariant is still much the same, the variant generation is a bit more difficult, since there is no way of knowing beforehand which processor will stay the longest in the loop. Instead, we need to provide some *measure* obtained from the variants on every processors, for instance their sum. Because of the added complexity for the user in the second option, both in the orginal code and with the proof obligations, we chose to provide the first option by default, with the possibility to request the generation of the more complex loop when necessary (if the case when the user wants an algorithm where processors exit the loop early) using the keyword **global** next to **variant**.

Finally, the transformation of a sequential block of code is similar to the one seen in Sect. 2.2.2, but slightly more complex. Instead of executing the block for all the processors successively, we only execute it for the processors that are running that part of the code. This is exactly what is denoted by the variable S. For this, we introduce a shortened notation: `for` i `in` S `do` c. The "for loop" means that we execute sequentially the instructions c for all the processors in S. This generates a code of the following form:

```
1 let i = ref 0 in
2 while !i<nprocs do
3     { invariant inv
4
  variant nprocs − i }
5     if procin i S then c;
6     i← !i+1;
7 done
```

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

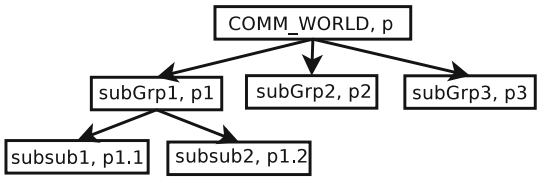

Fig. 9 A tree of BSP subgroups

$[[e]]_S = \texttt{for}\ i\ \texttt{in}\ S\ \texttt{do}\ [[e]]_{i,S}$ where $[[e]]_{i,S}$ is transformation of the sequential code e for processor i.

When transforming local code from BSP-WHY to WHY, there are almost no modification compared to the standard BSP model. The main point here is that for synchronising parameters, we add the S argument containing the set of the processors that run the synchronisation. This is necessary to ensure the *proper* synchronisation. As explained before, the synchronisation parameters are defined with two arguments, the array of the communicators and the set of the processors that execute the synchronisation. Both arguments are then used to define the correct precondition. Similarly, an argument is added for all of the function calls, when the function contains parallel code. This is needed to transmit the information of S inside the code of the functions.

The determination of the invariants of the "for loops" can be made similarly to what we did before—see Sect. 2.2.2. The difference is that for the processors that are not in S , we always need the second form of invariant to hold, that is: ∀j:int. 0≤j<i **and** j ∈ S → post[j]. And, as above, we also need to express that the computation has not yet been processed for some processors. This is done using the two following invariants: (1) ∀j:int. i≤j<**nprocs**→v[j] = v[j]@loopstart, the computation has not been done for these processors; (2) ∀j:int. 0≤j<i **and**¬ (j ∈ S)→v[j]=v[j]@loopstart, processors not in S will never do the computation.

4 Application to Finite State-Space Construction

4.1 Model and Algorithm

Previously, we have *mechanically* verified two state-space algorithms: one sequential and one strictly BSP. The latter uses an extension of the former in a inner loop. We now show how to use the transformation of Sect. 3 to verify an algorithm which is a modification of the BSP algorithm with subgroups. For this algorithm, we assume that the architecture is a "*hierarchical tree*" of disjoint BSP machines, e.g. a cluster of multi-cores. Figure 9 illustrates the architecture, i.e. the tree of disjoint groups. The root is the whole machine (group COMM_WORLD) with the **p** processors. Then, there are three disjoint subgroups ($\text{subGrp}_{\{1,2,3\}}$) of processors where $\mathbf{p} = p_1 + p_2 + p_3$. The first subgroup is itself cut up into two disjoint subsubgroups where $p_1 = p_{1.1} + p_{1.2}$. Also p_1 (resp. $p_{\{2,3\}}$ and $p_{\{1.1,1.2\}}$) is greater than 0.

We also assume a new function of partition cpu(s) which now returns a pair (g, i) where g is a "group leaf" of the tree and i the ith processor in this group. Note that our algorithm provides load balancing of computations only if the partition (hash) allows

```
1  let multi_bsp_state_space () =
2    let known = ref ∅          and todo = ref ∅
3    and pastsend = ref ∅       and total = ref 1
4    and toOther = ref [p1 ⇒ ∅,p2 ⇒ ∅, . . ., p1.2⇒ ∅]
5    and mygroup = ref (leafGroup (bsp_pid COMM_WORLD)) in
6    let (g,i) = cpu(s0) in
7     if (g=mygroup) and (i=(bsp_pid mygroup)) then todo ←s0 ⊕ !todo;
8    while (total>0) or (!mygroup≠ COMM_WORLD) do
9        { invariant (1) and (2) and (3) and (6) and (9) and (11)
10       (2') and (∀i:int.isproc(i)→  (known<i> ∩ AllSum(toOther<i>))=∅)
11       (2') and (∀i:int.isproc(i)→  (todo<i> ∩ AllSum(toOther<i>))=∅)
12       (4') and (∀ i,j:int. isproc(mygroup,i)→ isproc(mygroup,j)→ total<mygroup,i> =total<mygroup,j>)
13       (5') and total<mygroup,0> ≥ |⋃(<todo,mygroup>)|
14       (7') and (∀e:state.e∈⋃(<known>)
15                       → succ(e)⊆ (⋃(<known>)∪⋃(<todo>)∪⋃(<AllSum(toOther)>)))
16       (8') and (∀e:state.∀i:int.isproc(i)→ e∈known<i>
17                       → succ(e)⊆ (known<i>∪pastsend<i>∪AllSum(toOther<i>)))
18       (9') and ⋃(<AllSum(toOther)>) ⊆ StSpace
19       (10') and (∀i:int.isproc(i)→ ∀e:state.∈pastsend<i>
20                       → let(g,j)=cpu(e) in g≠ GrpLeaf(i) and toProper(j)≠ i)
21       (10') and (∀i:int.isproc(i)→ ∀e:state. let(g,j)=cpu(e) in e∈(Sum(toOther<i>[g]))
22                       → g≠ GrpLeaf(i) and toProper(j)≠ i)
23       (10') and (∀i:int.isproc(i)→ ∀e:state. let(g,j)=cpu(e)in ∈(Sum(toOther<i>[g]))→ g≠ mygroup)
24       (12) and mygroup≤COMM_WORLD
25       (13) and (∀i:int.isproc(i)→  ∀grp:group. Valid(toOther<i>[g]) ↔ grp≤COMM_WORLD)
26       (14) and (∀i:int.isproc(i)→  ∀grp:group. ∈(gr,Tree)→ grp≤mygroup→ Sum(toOther<i>[g])=∅)
27       variant pair(total,| StSpace \ ⋃(known) |) for lexico_order }
28       local_successor known todo pastsend toOther;
29       exchange todo total !known (select !mygroup toOther);
30       while (total=0) and (!mygroup≠ COMM_WORLD) do
31          { invariant (1) and (2) and (all 2') and (4') and (5') and (6) and (7') and (8') and (9) and (9')
32          and (all 10') and (11) and (12) and (13) and (14)
33          variant | nprocs(COMM_WORLD)−nprocs(mygroup)| }
34          mygroup←(top !mygroup);
35          exchange todo total !known (select !mygroup toOther);
36       done
37       if total≠ 0 then mygroup←(leafGroup (bsp_pid COMM_WORLD));
38   done
39   !known {StSpace=⋃(<result>)}
```

Fig. 10 BSP-WHY-ML (with subgroups) algorithm for state-space construction

a good distribution of the states. Adding specific procedures of redistribution of the states is out of the scope of this article.

Figure 10 shows the annotated algorithm. It works as follows. First, references are properly initialised —lines 2–7. Second, each processor computes its local successors and, depending of cpu, puts into toOther the states that belong to another processor of a group. toOther is a map where keys are groups and values are $fparray$ of sets of states, one set for each processor within the group. For this, we assume a procedure **leafGroup** indicating to a processor the "leaf group" to which it belongs. Third, we need to send states of toOther. For this, we have a inner loop (lines 30–36) that moves back up in the tree until an exchange is not empty (total>0) or the root is reached; these exchanges are initialised with the procedure **select** which takes from toOther only the states that belong to the selected group; to moves back up in the tree, we use the procedure **top**. If the exchange is not empty, then some states have been received by some processors and we loop to perform again the local computations. Otherwise, we have reached COMM_WORLD without sending any states which terminates the main loop since no new state have been received. local_successors works as before (mainly as "random_walk") but by putting into toOther states that have not to be owned by the

```
1  let local_successors (known: state set ref) (todo:state set ref)
2                       (pastsend:state set ref) (mygroup:group)
3                       (toOther: state set fparray map ref) =
4  let tosend = ref (init_send mygroup ∅) in
5   while todo ≠ ∅ do
6     let s = pick todo in
7     known←!known ⊕ s;
8     let new_states = ref ((succ s)\ !known\ !pastsend) in
9     init:while new_states ≠ ∅ do
10      let new_s = pick new_states in
11      let (grp,tgt)=cpu(new_s) in
12      if (grp=mygroup) and (tgt=bsp_pid) then
13        todo←!todo⊕new_s
14      else
15        toOther[grp]<tgt>←toOther[grp]<tgt>⊕new_s
16    done
17  done
```

Fig. 11 Local successors function

local processor. Figure 11 gives the code without annotations of the local computations. **exchange** works also as before but only synchronises the processors of the subgroup.

This algorithm results in less synchronisations of the whole machine since the processors of a group only synchronize when there were actually states to exchange and all local calculations (of the subgroups) were completed. It also better supports the hardware locality: a group could be the multi-cores of a machine and thus compute a part of the state-space without communications with other processors of other machines. Note that, assuming that the tree is only the root, we recover the original BSP algorithm.

4.2 Mechanised Deductive Verification of the Algorithm

For lack of space, we only present the verification of the parallel part of this algorithm and not the sequential local_successors nor **exchange**, which is more technical and with straightforward properties: it only permutes states in arrays. It is still available in the source code. We use the following predicates:

- Sum(m[gr]) is the union of all states of a fparray of sets, given by the the map m from the key (group) gr;
- AllSum(m) is the union of all sets of states in the map (for each key);
- isproc(grp,i) defines what is a valid processor id of a group grp, that is 0≤i<**nprocs**(grp);
- GrpLeaf(i) gives the "group leaf" of the proper processor i (0≤i<**nprocs**)
- toProper(gr,j) gives the proper processor id of the jth processor of group gr
- grp1≤grp2 is valid if both groups are in the tree of groups, in the same branch and grp1 is lower. Note that all groups are lower than COMM_WORLD.

As previously, we need to prove that (1) the code does not fail; (2) computes the entire state-space (e.g. no state is lost during the exchanges of the subgroups) and (3) terminates. The first property is easy since pick is used as before; the function **top** does not fail since the condition of the loop is that we only trace the tree back until the root (COMM_WORLD) is accessed; and to prove that the code is well-structured enough (a loop contains **exchange** which implies a synchronisation of a subgroup),

we can easily *maintain* that total has the same value on all the processors of a same group during the entire execution of the mentioned group. Let us now focus on the two other properties.

Correctness of the parallel main loop For the main loop, the invariants (lines 9–26) work as follows: (1), (2), (3), (6), (9), (11) as in the strict BSP algorithm; (2') completes (2) so that any state could not be owned by a processor and be sent to another group (depending on the partition function); (4') total is the same value for each processor of the group mygroup, thus ensuring no processor makes another super-step in a group; (5') is as for the BSP algorithm but here total is an over-approximation for each group; (7') as before, ensures the (B) property since todo and toOther will be empty after the loop; (8') successors of a state are owned by the current processor or was (or will be) sent; (9) all states of toOther are in the state-space; (10') pastsend and toOther only contain states that are not owned by the processor; (12) ensuring that mygroup is in the tree and lower than COMM_WORLD ; (13) if a group is a valid key of the map toOther then it is lower than COMM_WORLD ; (14) ensures that all communication of a group has been done which ensures that toOther is empty after the loop since mygroup will be equal to COMM_WORLD .

Invariants of the inner loop are as before—lines 31–32. The most important one is (14) which ensures that we trace the tree back to the root without forgetting to send states. A reader might think that *too many invariants* are added to the code. This is actually fairly *standard* when performing *mechanical proofs* and such constraints would appear if the verification had been made in COQ.

Termination For the local computations, the termination is ensured as above since known grows when entering the loop. But to ensure that known actually grows in the main loop or there is no need to send states (toOther remains unchanged), the two following specific invariants are added in the loop of the local computations: (1) $(\text{todo@init} \cup \text{known@init}) \subseteq (\text{todo} \cup \text{known})$ that is known contains the states of the initial known (the label "init" marks the beginning of the loop) union todo, but todo will be empty after the loop and thus known will contain (at most) all the states of the initial todo; (2) $\text{todo@init}=\emptyset \rightarrow (\text{todo}=\emptyset \textbf{ and } \text{toOther}=\text{toOther@init})$ that is if todo is empty then there is no need of sending states and toOther is as at label "init". The termination of the inner loop (line 33) is easy since every group has obviously less processors than its "father". Thus the variant is the difference between the number of processor of mygroup and COMM_WORLD.

Mechanical proof 11 obligations are generated for the sequential "random_walk" whereas 152 are generated for the strict BSP algorithm. With some obvious axioms on the above predicates (such as $\bigcup <\emptyset,\ldots,\emptyset>=\emptyset$) so that solvers can handle the predicates, all the obligations (logical goals) produced by the VCG of WHY are *automatically discharged* by a combination of automatic provers: CVC3, Z3, SIMPLIFY, ALT- ERGO, YCES and VAMPIRE. For each prover, we give a timeout of 10 mins—otherwise some obligations are not proved. The automatic provers SIMPLIFY and Z3 give the best results. In practice, we mostly used them. SIMPLIFY is the fastest and Z3 sometime verified

some obligations that had not be discharged by SIMPLIFY. We have no explanation for this fact.

Currently, 245 goals are generated when subgroups are used and some of them still not proved by the solvers. We are currently working to research what in our *axiomatisation* of subgroups hampers the solvers. We hope to be able to discharge all goals in a near future. Even if proof obligations are as usual when working with a VCG such as WHY, some of them are hard to follow due to the parallel computations. But reading them carefully, we can find the good annotations. Based on this fact, it seems conceivable that a more seasoned team in formal methods can tackle more substantial algorithms.

5 Related Work

Correctness of BSP *programs* Different approaches have been studied. In [13], *functional* BSP programs have been proved correct in COQ. In [14], we presented the correctness of a N-body computation using a *mechanised semantics*. However, proofs of correctness were too hard by only using semantics inside COQ.

The *derivation* of imperative programs using the Hoare's semantics followed by the generation of correct C code [41] also exists. The two main drawbacks of this approach is a lack of an implementation of a dedicated tool for the logical derivation, which implies a lack of safety; users make hand proofs which are not machine checked; moreover, it is impossible to verify users existing codes. Using the Hoare's semantics has also been studied in [5,21,34]. More recently, these works were extended for subgroups in [33]. All of these approaches *lack mechanised proofs*. Moreover, they are close to refinement *à la* B since they give logical rules for deriving algorithms from specifications. On the contrary, using deductive verification, we begin with a program and by adding logical assertions, we prove the correctness of the said program.

A work on proving *determinism*, using assertions in the code, of *multi-threaded* JAVA programs with barriers can be found in [3]. The authors note that *there seemed to be no obvious simpler, traditional assertions that would aid in catching non-deterministic parallelism*. In our case of BSP programs, this work is simple —but still limited to BSP programs.

Another work on concurrent threading with barriers is [20]. The authors have developed and proved sound a *concurrent separation logic* for barriers of threads. An interesting point is that the proofs are machine-checked in COQ. The authors also showcase a program verification toolset that automatically applies some logic rules (Hoare logic) and discharges the associated proof obligations. It is thus a work for derivation of formal specifications into correct parallel programs. The drawback (as in [38] and partly in [26]) is that only programs with a *predefined constant number of threads* (e.g. two for a producer-consumer problem) can be considered. For HPC, we prefer to have correct programs for an unknown number of processors in a data-parallel fashion.

Debugging MPI programs There are many tools dedicated to MPI. A survey could be found in [16]. These tools help to find some classical errors, but not all of them. In practice, defects, which usually appear in large configurations, can often be detected

in much smaller configurations. [39] presents a tool that directly *model-check* the MPI source code, executing its interleaving with the help of a verification scheduler (producing error traces of deadlocks and assertion violations). It allows an engineer to check its MPI code against deadlocks on a **p**-processors machine before running it on this machine. The main advantage of this method is to be fully automatic. The two main drawbacks are (1) they only consider *deadlocks and assertion violations* (2) programs are model-checked only for a *predefined* number of processors which is less than 64 in the original article; their model-checker induces too much computations and communication if the program is checked with a larger number of processors.

In [30], the author proposes a solution using a *symbolic execution* technique (with special "collective loop invariants") that he uses to verify assertions in MPI programs which are then checked, with a model-checker, for unknown sizes of data—but still for a fixed number of processors only. However, the author note that discovering these collective invariants is currently to the charge of the programmer (as in our work) and these invariants are limited: for example, there is no way to express that the number of messages is invariant. The method still has the advantage of greatly reducing the number of necessary invariants. The tool can also be use (with symbolic executions) to verify that two programs are functionally equivalent—i.e. "input-output equivalent". This technique is particularly useful in computational science, to *compare* a complex parallel program, the "implementation", to its simple and *trusted* sequential version, the "specification".

Currently, we are not aware of any verification condition generator tool for MPI programs. We think that performing a sequential simulation of any kind of MPI programs is not reasonable. Continuations would be necessary to simulate the interleaving of processes: that would generate unintelligible assertions. But certainly many MPI programs could be automatically transformed into BSP ones [23]. We leave the aim of substantiating this claim for future work.

The approaches of symbolic verification as well as VCG tools, suffer to the main limitation: as it now stands, models of the programs must be built by hand. This requires significant effort and a degree of skill from the user. The ideal situation would be to have tools that automatically extract the models from source code, at least for specific domains [11].

Correctness of state-space algorithms and model-checkers For verifying model-checkers, different solutions have been proposed. The first one is to prove MC inside theorem provers and use the *extraction* facilities to get pure functional machine-checked programs [7,31]. Extracted functional programs are known to be less efficient than imperative ones and currently only sequential MC algorithms have been studied. Second, derive a model-checker from its specification *à la* B [35], but also only sequential codes are currently generated.

The third and more common approach is to generate a "*certificate*" during the execution of the MC that can be checked later or on-the-fly by a dedicated tool or a theorem prover. This is the so-called "certifying model-checking" [25]. In this way, users can re-execute the certificate/trace and have some *safety guarantees* because even if the MC is buggy, its results can be checked by a trustworthy tool. But, any explicit MC may

enumerate a very large state-spaces (the famous state-space *explosion problem*), and mimicking this enumeration inside any theorem prover would be unreasonable [29].

We are also not aware of another state-space algorithm using subgroups. An exception is the work of [27] but a concurrent data-structure is used for the cores and the naive algorithm of [12] is used for the whole parallel machines.

6 Conclusion and Future Work

Summary of the contribution The paper presents a *methodology* and its associated tool, called BSP-WHY, for the deductive *verification* of BSP algorithms with *subgroups*. BSP-WHY-ML extends the WHY-ML intermediate language by adding some constructs specific to BSP parallelism and subgroups. The tool *translates* a subset of BSP-WHY-ML (programs that are "well-structured" enough) into sequential WHY-ML: since BSP programs are made of super-steps, even with subgroups, parallelism can be removed by replacing a portion of code between barriers of a subgroup with a loop to repeat that portion for every process of the subgroup. In view of the ratio "number to prove/proved automatically" of the generated proof obligations, we believe this method is far from perfect but nonetheless can rapidly increase the confidence that can be placed in the code.

In this work, we also focus on examples for *finite state-space* construction of systems (basis of model-checking) and notably on a BSP algorithm (with subgroups) designed by the authors. We *annotated* the algorithms and used the VCG WHY (certified in COQ [17]) as back-end of our tool BSP-WHY to obtain goals. These goals ensure the *termination* of the algorithms as well as their *correctness* for any successor function—assumed correct and generating a finite state-space. We thus gained more confidence in the code. We also hope to have convinced the reader that this approach is humanly feasible and applicable to other kinds of BSP algorithms: graphs, bioinformatics, etc..

Future work The current prototype is still limited. We plan to extend it in several ways. First, we are currently proving algorithms and not real codes. Regarding the code structure, this is not really an issue and translating the resulting proof into a verification tool for true programs should be straightforward, especially if high level data-structures are used, e.g., the WHY framework allows a user to generate WHY-ML code from JAVA using a tool called KRAKATOA.

Second, adapt this work for true MC algorithms—for instance able to check temporal logics such as LTL/CTL*; it is mostly Tarjan/NDFS like algorithms. This is challenging in general but using appropriate algorithms, we believe that a team can "quickly" do it.

And to finish, without the insurance of a machine-checked proof, the transformation of BSP-WHY-ML into WHY-ML could potentially contain bugs. The first author is working on this using *machine-checked semantics* in COQ [9].

References

1. Bisseling, R.H.: Parallel Scientific Computation. A Structured Approach Using BSP and MPI. Oxford University Press, Oxford (2004)

2. Bonorden, O., Judoiink, B., von Otte, I., Rieping, O.: The Paderborn University BSP (PUB) library. Parallel Comput. **29**(2), 187–207 (2003)
3. Burnim, J., Sen, K.: Asserting and checking determinism for multithreaded programs. Commun. ACM **53**(6), 97–105 (2010)
4. Cappello, F., Guermouche, A., Snir, M.: On communication determinism in HPC applications. In: Computer Communications and Networks (ICCCN), pp. 1–8. IEEE (2010)
5. Chen, Y., Sanders, W.: Top-down design of bulk-synchronous parallel programs. Parallel Process. Lett. **13**(3), 389–400 (2003)
6. Clarke, E., et al. (eds.): Handbook of Model Checking. Springer, Berlin (2012)
7. Esparza, J., et al.: A fully verified executable LTL model checker. In: Computer Aided Verification (CAV), LNCS, vol. 8044, pp. 463–478. Springer (2013)
8. Filliâtre, J.C.: Verifying two lines of C with why3: an exercise in program verification. In: Verified Software: Theories, Tools and Experiments (VSTTE) (2012)
9. Fortin, J.: BSP- WHY: a tool for deductive verification of BSP programs; machine-checked semantics and application to distributed state-space algorithms. Ph.D. thesis, University of Paris-East (2013). http://lacl.fr/gava/papers/fortin_thesis
10. Fortin, J., Gava, F.: BSP- WHY: an intermediate language for deductive verification of BSP programs. In: HLPP, pp. 35–44. ACM (2010)
11. Furia, C.A., Meyer, B.: Inferring loop invariants using postconditions. In: Fields of Logic and Computation, LNCS, vol. 6300, pp. 277–300. Springer (2010)
12. Garavel, H., Mateescu, R., Smarandache, I.M.: Parallel state space construction for model-checking. In: SPIN Conference, LNCS, vol. 2057, pp. 217–234. Springer (2001)
13. Gava, F.: Formal proofs of functional BSP programs. Parallel Process. Lett. **13**(3), 365–376 (2003)
14. Gava, F., Fortin, J.: Formal semantics of a subset of the PUB. In: Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 269–276. IEEE (2008)
15. Gava, F., Fortin, J., Guedj, M.: Deductive verification of state-space algorithms. In: Integrated Formal Methods (IFM), LNCS, vol. 7940, pp. 124–138. Springer (2013)
16. Gopalakrishnan, G., Kirby, R.M., Siegel, S.F., Thakur, R., Gropp, W., Lusk, E.L., de Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs: present and future. Commun. ACM **54**(12), 82–91 (2011)
17. Herms, P.: Certification of a chain for deductive program verification. In: Bertot, Y. (ed.) COQ Workshop, Satellite of ITP (2010)
18. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.: BSPLIB: the BSP programming library. Parallel Comput. **24**, 1947–1980 (1998)
19. Hoare, C.A.R., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: a manifesto. ACM Comput. Surv. **41**(4), 1–8 (2009)
20. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic: now with tool support. Log. Methods Comput. Sci. **8**(2), 1–32 (2012)
21. Jifeng, H., Miller, Q., Chen, L.: Algebraic laws for BSP programming. In: Bouge, L., Robert, Y. (eds.) Euro-Par, no. 1124 in LNCS, pp. 359–368. Springer (1996)
22. Lübeck, F., Neunhöffer, M.: Enumerating large orbits and direct condensation. Exp. Math. **10**(2), 197–205 (2001)
23. Martino, B.D., Mazzeo, A., Mazzocca, M., Villano, U.: Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs. Sci. Comput. Program. **40**(2–3), 235–263 (2001)
24. Merali, Z.: Computational science: error, why scientific programming does not compute. Nature **467**(7317), 775–777 (2010)
25. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification (CAV), LNCS, vol. 2102, pp. 2–13. Springer, Berlin (2001)
26. Nieto, L.P.: Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL. Ph.D. thesis, Technische Universitat Munchen (2001)
27. Saad, R.T., Dal-Zilio, S., Berthomieu, B.: Mixed shared-distributed hash tables approaches for parallel state space construction. In: Parallel and Distributed Computing (ISPDC), pp. 9–16. IEEE (2011)
28. Seo, S., Yoon, E.J., Kim, J.H., Jin, S., Kim, J.S., Maeng, S.: HAMA: an efficient matrix computation with the mapreduce framework. In: Cloud Computing (CloudCom), pp. 721–726. IEEE (2010)

29. Shankar, N.: Trust and automation in verification tools. In: Cha, S.D., Choi, J.Y., Kim, M., Lee, I., Viswanathan, M. (eds.) Automated Technology for Verification and Analysis (ATVA), LNCS, vol. 5311, pp. 4–17. Springer, Berlin (2008)
30. Siegel, S.F., Zirkel, T.K.: Loop invariant symbolic execution for parallel programs. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 7148, pp. 412–427. Springer, Berlin (2012)
31. Sprenger, C.: A verified model checker for the modal μ-calculus in COQ. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS, vol. 1384, pp. 167–183. Springer (1998)
32. Springel, V.: The cosmological simulation code gadget-2. Mon. Not. R. Astron. Soc. **364**, 1105–1134 (2005)
33. Stewart, A.: A programming model for BSP with partitioned synchronisation. Form. Asp. Comput. **23**(4), 421–432 (2011)
34. Stewart, A., Clint, M., Gabarró, J.: Axiomatic frameworks for developing BSP-style programs. Parallel Algorithms Appl. **14**, 271–292 (2000)
35. Turner, E., Butler, M., Leuschel, M.: A refinement-based correctness proof of symmetry reduced model-checking. In: Abstract State Machines, Alloy, B and Z, LNCS, pp. 231–244. Springer (2010)
36. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
37. Valiant, L.G.: A bridging model for multi-core computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
38. Villard, J., Lozes, É., Calcagno, C.: Proving copyless message passing. In: Programming Languages and Systems (APLAS), LNCS, vol. 5904, pp. 194–209. Springer (2009)
39. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: Principles and Practices of Parallel Programming (PPoPP), pp. 261–269 (2009)
40. Yzelman, A.N., Bisseling, R.H.: An object-oriented BSP library for multicore programming. Concurr. Comput. Pract. Exp. **24**(5), 533–553 (2012)
41. Zhou, J., Chen, Y.: Generating C code from LOGS specifications. In: Theoretical Aspects of Computing (ICTAC), LNCS, vol. 3722, pp. 195–210. Springer (2005)