Active Objects for Coordinating BSP Computations (Short Paper)

Gaetan Hains, Ludovic Henrio, Pierre Leca, Wijnand Suijlen

Active Objects for Coordinating BSP Computations (short paper)

Gaétan Hains[1], Ludovic Henrio[2], Pierre Leca[1,2], and Wijnand Suijlen[1]

[1] Huawei Technologies, Paris Research Center, France
[2] Université Côte d'Azur, CNRS, I3S, France

Abstract. Among the programming models for parallel and distributed computing, one can identify two important families. The programming models adapted to data-parallelism, where a set of coordinated processes perform a computation by splitting the input data; and coordination languages able to express complex coordination patterns and rich interactions between processing entities. This article takes two successful programming models belonging to the two categories and puts them together into an effective programming model. More precisely, we investigate the use of active objects to coordinate BSP processes. We choose two paradigms that both enforce the absence of data-races, one of the major sources of error in parallel programming. This article explains why we believe such a model is interesting and provides a formal semantics integrating the notions of the two programming paradigms in a coherent and effective manner.

Keywords: parallelism, programming models, active objects, BSP

1 Introduction

This article presents our investigations on programming paradigms mixing efficient data parallelism, and rich coordination patterns. We propose a programming methodology that mixes a well-structured data-parallel programming model, BSP [17] (Bulk Synchronous Parallel), and actor-based high-level interactions between asynchronous entities. This way, we are able to express in a single programming model several tightly coupled data-parallel algorithms interacting asynchronously. More precisely we design an active-object language where each active object can run BSP code, the communication between active objects is remote method invocation, while it is delayed memory read/write inside the BSP code. These two programming models were chosen because of their properties: BSP features predictable performance and absence of deadlocks under simple hypotheses. Active objects have only a few sources of non-determinism and provide high-level asynchronous interactions. Both models ensure *absence of data races*, thus our global model features this valuable property. The benefits we expect from our mixed model are the enrichment of efficient data-parallel BSP with both service-like interactions featured by active objects, and elasticity. Indeed, scaling a running BSP program is not often possible or safe, while adding new active objects participating to a computation is easy.

This short paper presents the motivation for this work and an analysis of related programming models in Section 2. Then a motivating example is shown in Section 3. The main contribution of this paper is the definition of a core language for our programming model, presented in Section 4. An implementation of the language as a C++ library is under development.

2 Context and Motivation

2.1 Active Objects and Actors

The actor model [1] is a parallel execution model focused on task parallelism. Actor-based applications are made of independent entities, each equipped with a different process or thread, interacting with each other through asynchronous messages passing. Active objects hide the concept of message from the language: they call each other through typed method invocations. A call is asynchronous and returns directly giving a Future [11] as a placeholder for its result. Since there is only one thread per active object, requests cannot run in parallel. The programmer is thus spared from handling mutual exclusion mechanisms to safely access data. This programming model is adapted to the development of independent components or services, but is not always efficient when it comes to data-parallelism and data transmission. An overview of active object languages is provided by [5], focusing on languages which have a stable implementation and a formal semantics. ASP [7] is an active object language that was implemented as the ProActive Java library [3]. Deterministic properties were proven in ASP when no two requests can arrive in a different order on the same active object.

Several extensions to the active object model enable controlled parallelism inside active objects [12, 8, 2]. Multi-active objects is an extension of ASP [12] where the programmer can declare that several requests can run in parallel on the same object. This solution relies on the correctness of program annotation to prevent data-races but provides efficient data-parallelism inside active objects. Parallel combinators of Encore [8] also enable some form of restricted parallelism inside an active object, mostly dedicated to the coordination of parallel tasks. A set of parallel operators is proposed to the programmer, but different messages still have to be handled by different active objects. This restricted parallelism does not provide local data parallelism.

2.2 Bulk Synchronous Parallel

BSP is another parallel execution model, it defines algorithms as a sequence of supersteps, each made of three phases: computation, communication, and synchronization. BSP is adapted to the programming of data-parallel applications, but is limited in terms of application elasticity or loose coupling of computing components as it relies on the strong synchronization of all computing entities.

Interactions between BSP processes occur through communication primitives sending messages or performing one-sided Direct Remote Memory Access

(DRMA) operations, reading or writing the memory of other processes without their explicit consent. BSP is generally used in an SPMD (Single Program Multiple Data) way, it is suitable for data-parallelism as processes may identically work on different parts of a data. BSPlib is a C programming library for writing BSP programs [13], it features message passing and DRMA. Variants of BSPlib exist such as the Paderborn University BSP (PUB) library [6], or BSPonMPI [14]. The PUB library offers subset synchronization, but this feature is argued against by [10] in the context of a single BSP data-parallel algorithm. Using subset synchronization to coordinate different BSP algorithms in the same system seems possible but even more error prone. Formal semantics were defined for BSPlib DRMA [15] and the PUB library [9].

2.3 Motivation and Objectives

The SPMD programming model in general and BSP are well adapted for the implementation of specific algorithms, but composing different such algorithms in a single application requires coordination capabilities that are not naturally provided by the SPMD approach. Such coordination is especially difficult to implement in BSPlib because a program starts with all processes in a single synchronization group. For any communication to occur, all processes of the application need to participate in the same synchronization barrier, making it difficult to split a program into parallel tasks with different synchronization patterns. The PUB library can split communication groups to synchronize only some of the processes, but still lacks high level libraries for coordinating the different groups. On the other hand, asynchronous message sending of active objects is appropriate for running independent tasks, but inefficient when there are many exchanged messages inside a given group of process or following a particular communication pattern.

In this paper, we use active objects for wrapping BSP tasks, allowing us to run different BSP algorithms in parallel without requiring them to participate in the same synchronization. Active objects provide coordination capabilities for loosely coupled entities and can be used to integrate BSP algorithms into a global application. To our knowledge, this is the first model using active objects to coordinate BSP tasks.

Among related works, programming languages based on stream processing, like the StreamIt [16] language, feature data parallelism. While splitting a program into independent tasks could be considered similar to our approach, stream processing languages do not feature the strong synchronization model of BSP. They are also less convenient for service-like interaction between entities, particularly when those sending queries are not determined statically. In summary, ABSP features an interesting mix between locally constrained parallelism using BSP (with fixed number of processes and predefined synchronization pattern), and flexible service oriented interactions featured by active objects (more flexible but still with some reasonable guarantees). This makes our approach quite different from StreamIt and other similar languages.

```
1  class IPActor : public activebsp::ActorBase {
2  public:
3    double ip(vector<double> v1, vector<double> v2) {
4      // ...
5      for (int i = 0; i < bsp_nprocs(); ++i) { // Split data
6        int beg = (v1.size()/bsp_nprocs()) * i,
7            end = (v1.size()/bsp_nprocs()) * (i+1);
8        bsp_put(i, &v1[beg], _x_part, 0, (end-beg)*sizeof(double));
9        bsp_put(i, &v2[beg], _y_part, 0, (end-beg)*sizeof(double));
10     }
11
12     bsp_run(&IPActor::bspinprod);
13
14     return _alpha;
15   }
16
17   void bspinprod() {
18     // call BSPedupack inner product, returns result on all p
19     _alpha = bspip(bsp_nprocs(),bsp_pid(),_n_part,_x_part,_y_part);
20   }
21 };
22
23 int main() {
24   // ...
25   vector<double> v;
26   // ...
27   Proxy<IPActor>   actorA = createActor<IPActor>  ({1,2});
28   Proxy<MultActor> actorB = createActor<MultActor>({3,4});
29
30   Future<double> f1 = actorA.ip(v,v);
31   double ip = f1.get();
32   Future<vector<double>> f2 = actorB.multiply_all(v, ip);
33   v = f2.get();
34   // ...
35 }
```

Fig. 1. A ABSP example

3 Example

In this section we show an example written using our C++ library under development. This library uses MPI for actor communications and reuses the BSPonMPI library for BSP communications.

We chose C++ because we put higher priority on the efficiency of BSP data-parallel code, C++ allows us to re-use BSP implementations written in C while allowing objects and more transparent serialization. An implementation mixing incompatible languages would, at this point, yield unnecessary complexity in our opinion. We chose a motivating example based on this implementation instead of the formal language of the next section to show the re-use of existing code and because we think it is more convincing.

This example shows how an active object can encapsulate process data and how its function interface can act as a parameterized sequential entry point to a parallel computation. We also show the result of a call being used to call another active object to do another computation, which is not shown.

The IPActor class interfaces the inner product implementation included in the BSPedupack software package [4]. We only show parts of the code we deem

interesting to present our model. Object variables begin by '_', their declarations are not shown. We assume bsp_nprocs() divides $v1.size()$.

In the *main* function, the MPI process of pid 0 creates two active objects with two processes each, see the parameter of the *createActor* primitive. Then the *ip* function of the first one is called with vector v as the two parameters. This asynchronous call returns with a future $f1$. The *ip* function of this active object is then called sequentially. Using BSP primitives, the input vectors are split among the processes of the active object. Then a *bsp_run* primitive is used to run *bspinprod* in parallel. It calls the *bspip* function of BSPedupack. Immediately after the call on the first object, the main method requests the result with a *get* primitive on $f1$, blocking until the result is ready. This result is sent to another active object as request parameter.

4 A Core Language for Coordinating BSP Computations

4.1 Syntax

ABSP is our core language for expressing the semantics of BSP processes encapsulated inside active objects. Its syntax is shown in figure 2, x ranges over variable names, m over method names, α, β over actor names, f over future names, and i, j, k, N over integers that are used as process identifiers or number of processes. A program P is made of a main method and a set of object classes with name `Act`, each having a set of fields and a set of methods. The main method identifies the starting point of the program. Each method M has a return type, a name, a set of parameters x, and a body. The body is made of a set of local variables and a statement. Types T and terms are standard for object languages, except that `new` creates an active object, `get` accesses a future, and $v.m(\overline{v})$ performs an asynchronous method invocation on an active object and creates a future. The operators for dealing with BSP computations are: `BSPrun`(m) that triggers the parallel execution of several instances of the method m; `sync` delimits BSP supersteps; and `bsp_put` writes data on a parallel instance, to prevent data-races the effect of `bsp_put` is delayed to the next `sync`. `bsp_get` is the reverse of `bsp_put`, reading remote data instead of writing it. Sequence is denoted ; and is associative with a neutral element `skip`. Each statement can be written as $s; s'$ with s neither `skip` nor a sequence.

Design choices. We chose to specify a FIFO request service policy like in ASP because it exists in several implementations and makes programming easier. In ABSP, all objects are active, a richer model using passive objects or concurrent object groups [5] would be more complex. We choose a simple semantics for futures: futures are explicit and typed by a parametric type with a simple `get` operator. We chose to model DRMA-style communications although message passing also exists in BSPlib; modelling messages between processes hosted on the same active object would raise no additional difficulty.

$$
\begin{array}{rcll}
P & ::= & \overline{\texttt{Act}\{\overline{T\ x}\ \ \overline{M}\}}\ \{\overline{T\ x}\ s\} & \text{program} \\
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{Act} \mid \texttt{Fut} < T > & \text{type} \\
M & ::= & T\ \texttt{m}(\overline{T\ x})\ \{\overline{T\ x}\ s\} & \text{method} \\
s & ::= & \texttt{skip} \mid x = z \mid \texttt{if}\ v\ \{\,s\,\}\ \texttt{else}\ \{\,s\,\} \mid s\ \texttt{;}\ s & \\
 & \mid & \texttt{return}\ v \mid \texttt{BSPrun(m)} \mid \texttt{sync} \mid \texttt{bsp\_put}(v, v, x) & \text{statements} \\
z & ::= & e \mid v.\texttt{m}(\overline{v}) \mid \texttt{new Act}(N, \overline{v}) \mid \texttt{get}\ v & \text{rhs of assign} \\
e & ::= & v \mid v \oplus v & \text{expressions} \\
v & ::= & x \mid \texttt{null} \mid \textit{integer-values} & \text{atoms}
\end{array}
$$

Fig. 2. Static syntax of ABSP

$$
\begin{array}{rcll}
cn & ::= & \overline{\alpha(N, A, p, \overline{q}, \mathit{Upd})}\ \overline{f(\bot)}\ \overline{f(w)} & \text{configuration} \\
p & ::= & \varnothing \mid q : \left([\overline{i} \mapsto \overline{\mathit{Task}}]\ ;\ j \mapsto \mathit{Task}\right) & \text{current request service} \\
q & ::= & (f, m, \overline{w}) & \text{request id} \\
\mathit{Task} & ::= & \{\ell | s\} & \text{task} \\
w & ::= & x \mid \alpha \mid f \mid \texttt{null} \mid \textit{integer-values} & \text{runtime values} \\
\ell, a & ::= & [\overline{x} \mapsto \overline{w}] & \text{local store} \\
A & ::= & [\overline{i} \mapsto \overline{a}] & \text{object fields} \\
e & ::= & w \mid v \oplus v & \text{runtime expressions} \\
\mathit{Upd} & ::= & \overline{(i_{src}, v, i_{dst}, x)} & \text{DRMA operations}
\end{array}
$$

Fig. 3. Runtime Syntax of ABSP(terms identical to the static syntax omitted).

4.2 Semantics

The semantics of ABSP is expressed as a small-step operational semantics; it relies on the definition of *runtime configurations* which represent states reached during the intermediate steps of the execution. The syntax of configurations and runtime terms is defined in Figure 3. Statements and expressions are the same as in the static syntax except that they can contain runtime values.

A runtime configuration is an unordered set of active objects and futures where futures can either be unresolved or have a future value associated. An active object has a name α, a number N of processes involved in α, these processes are numbered $[0..N-1]$, A associates each pid i to a set of field-value pairs a. It has the form $(0 \mapsto [x \mapsto \texttt{true}, y \mapsto 1], 1 \mapsto [x \mapsto \texttt{true}, y \mapsto 3])$ for example meaning that the object at pid 0 has two fields x and y with value `true` and 1, and the object at pid 1 has the same fields with different values. Note that the object has the same fields in every pid. The function $A(i)$ allows us to select the element a at position i. $\overline{q}$ is the request queue of the active object. The active object might be running at most one request at a time. If it is not running a request, then $p = \varnothing$. Otherwise $p = q : \left([\overline{i} \mapsto \overline{\mathit{Task}}]\ ;\ j \mapsto \mathit{Task}\right)$ where q is the identity of the request being served, and $\left([\overline{i} \mapsto \overline{\mathit{Task}}]\ ;\ j \mapsto \mathit{Task}\right)$ is a two level mapping of processes to tasks that have to be performed to serve the request. The first level represents parallel execution, it maps process identifiers to tasks, the second represents sequential execution and contains a single process identifier and task. Tasks in each process consist of a local environment ℓ and a current

$$\frac{w \text{ is not a variable}}{[\![w]\!]_\ell \;=\; w} \qquad \frac{x \in \mathrm{dom}(\ell)}{[\![x]\!]_\ell \;=\; \ell(x)} \qquad \frac{[\![v]\!]_\ell = k \qquad [\![v']\!]_\ell = k'}{[\![v \oplus v']\!]_\ell \;=\; k \oplus k'}$$

NEW

$$\frac{[\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \overline{y} = fields(\texttt{Act}) \qquad A = [j \mapsto (\overline{y} \mapsto \overline{w}, pid \mapsto j, nprocs \mapsto N) \mid j \in [0..N-1]] \qquad \beta \text{ fresh}}{\mathcal{R}_k[a, \ell, x = \texttt{new Act}(N, \overline{v}) \;;\; s] \to \mathcal{R}_k[a, \ell, x = \beta \;;\; s]\; \beta(N, A, \varnothing, \varnothing, \varnothing)}$$

IF-TRUE

$$\frac{[\![v]\!]_{a+\ell} = \texttt{true}}{\mathcal{R}_k[a, \ell, \texttt{if } v \;\{\, s_1 \,\}\; \texttt{else} \;\{\, s_2 \,\} \;;\; s] \to \mathcal{R}_k[a, \ell, s_1 \;;\; s]}$$

IF-FALSE

$$\frac{[\![v]\!]_{a+\ell} \neq \texttt{true}}{\mathcal{R}_k[a, \ell, \texttt{if } v \;\{\, s_1 \,\}\; \texttt{else} \;\{\, s_2 \,\} \;;\; s] \to \mathcal{R}_k[a, \ell, s_2 \;;\; s]}$$

GET

$$\frac{[\![v]\!]_{a+\ell} = f}{\mathcal{R}_k[a, \ell, y = \texttt{get } v \;;\; s]\; f(w) \to \mathcal{R}_k[a, \ell, y = w \;;\; s]\; f(w)}$$

ASSIGN

$$\frac{[\![e]\!]_{a+\ell} = w \qquad (a + \ell)[x \mapsto w] = a' + \ell'}{\mathcal{R}_k[a, \ell, x = e \;;\; s] \to \mathcal{R}_k[a', \ell', s]}$$

INVK

$$\frac{[\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad f \;\text{ fresh}}{\mathcal{R}_k[a, \ell, x = v.\texttt{m}(\overline{v}) \;;\; s]\; \beta(N, A, p, \overline{q}, Upd) \to \mathcal{R}_k[a, \ell, x = f \;;\; s]\; \beta(N, A, p, \overline{q} :: (f, m, \overline{w}), Upd)\; f(\bot)}$$

SERVE

$$\frac{\mathrm{bind}(\alpha, m, \overline{v}) = \{\ell | s\} \qquad i = head(N)}{\alpha(N, A, \varnothing, (f, m, \overline{v}) :: \overline{q'}, Upd)\; cn \to \alpha(N, A, (f, m, \overline{v}) : (\varnothing \;;\; i \mapsto \{\ell | s\}), \overline{q'}, Upd)\; cn}$$

BSPRUN

$$\frac{\mathrm{bind}(\alpha, m, \varnothing) = \{\ell' | s'\}}{\alpha(N, A, q : (\varnothing \;;\; i \mapsto \{\ell \mid \texttt{BSPrun}(m) \;;\; s\}), \overline{q'}, Upd)\; cn \to \alpha(N, A, q : ([k \mapsto \{\ell' | s'\} | k \in [0..N-1]] \;;\; i \mapsto \{\ell \mid s\}), \overline{q'}, Upd)\; cn}$$

RETURN-VALUE

$$\frac{[\![v]\!]_{A(i)+\ell} = w}{\alpha(N, A, (f, m, \overline{w}) : (\varnothing; i \mapsto \{\ell \mid \texttt{return } v \;;\; s\}), \overline{q}, Upd)\; f(\bot)\; cn \to \alpha(N, A, \varnothing, \overline{q}, Upd)\; f(w)\; cn}$$

RETURN-SUB-TASK

$$\alpha(N, A, q : \left([\overline{i \mapsto Task}] \uplus [k \mapsto \{\ell \mid \texttt{return} v \;;\; s\}] \;;\; j \mapsto Task'\right), \overline{q}, Upd)\; cn \to \alpha(N, A, q : \left([\overline{i \mapsto Task}] \;;\; j \mapsto Task'\right), \overline{q}, Upd)\; cn$$

SYNC

$$\frac{A' = \left[j \mapsto A(j)\left[(y \mapsto [\![v]\!]_{A(i)}) | (i, v, j, y) \in Upd\right] \mid j \in I\right]}{\alpha(N, A, q : \left([\overline{k} \mapsto \overline{\{\ell_k | \texttt{sync } ; s_k\}}] \;;\; i \mapsto Task\right), \overline{q'}, Upd)\; cn \to \alpha(N, A', q : \left([\overline{k} \mapsto \overline{\{\ell_k | s_k\}}] \;;\; i \mapsto Task\right), \overline{q'}, \varnothing)\; cn}$$

BSP-GET

$$\frac{[\![v]\!]_{a+\ell} = i}{\mathcal{D}_k[a, \ell, \texttt{bsp\_get}(v, x_{src}, x_{dst}) \;;\; s, Upd] \to \mathcal{D}_k[a, \ell, s, Upd \cup (i, x_{src}, k, x_{dst})]}$$

BSP-PUT

$$\frac{[\![v]\!]_{a+\ell} = i \qquad [\![v_{src}]\!]_{a+\ell} = v'}{\mathcal{D}_k[a, \ell, \texttt{bsp\_put}(v, v_{src}, x_{dst}) \;;\; s, Upd] \to \mathcal{D}_k[a, \ell, s, Upd \cup (k, v', i, x_{dst})]}$$

Fig. 4. Semantics of ABSP.

statement s. For example, $p = q : ([k \mapsto \{\ell_k | s_k\} | k \in [0..N-1]] \ ; \ i \mapsto \{\ell \mid s\})$ means that the current request q first requires the parallel execution on all processes of $[0..N-1]$ of their statements s_k in environments ℓ_k; *then* the process i will recover the execution and run the statement s in environment ℓ. Concerning future elements, these have two possible forms: $f(\bot)$ for a future being computed, or $f(w)$ for a resolved future with the computed result w.

We adopt a notation inspired from reduction contexts to express concisely a point of reduction in an ABSP configuration. A global reduction context $\mathcal{R}_k[a, \ell, s]$ is a configuration with four holes: a process number k, a set a of fields, a local store ℓ, and a statement s. It represents a valid configuration where the statement s is at a reducible place, and the other elements can be used to evaluate the statement. This reduction context uses another reduction context focusing on a single request service and picking the reducible statement inside the current tasks. This second reduction context $\mathcal{C}_k[\ell, s]$ will allow us to conveniently define rules evaluating the current statement in any of the two execution levels, it provides a single entry for two possible options: the sequential level and the parallel one. It also defines that the parallel level is picked first instead of the sequential one if it is not empty. The two reduction contexts are defined as follows:

$$\begin{aligned} \mathcal{R}_k[a, \ell, s] ::= & \alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, s], \overline{q}, \mathit{Upd}) \ \mathit{cn} \\ \mathcal{C}_k[\ell, s] ::= & (\varnothing \ ; \ k \mapsto \{\ell | s\}) \quad | \quad \left([\overline{i} \mapsto \overline{\mathit{Task}}] \uplus [k \mapsto \{\ell | s\}]; \ j \mapsto \mathit{Task}\right) \end{aligned}$$

Taking the assignment as example, it applies in two kinds of configurations: $\alpha(N, A \uplus [k \mapsto a], q : \left([\overline{i} \mapsto \overline{\mathit{Task}}] \uplus [k \mapsto \{\ell | x = e \ ; \ s\}]; j \mapsto \mathit{Task}\right), \overline{q}, \mathit{Upd}) \ \mathit{cn}$ and $\alpha(N, A \uplus [k \mapsto a], q : (\varnothing \ ; \ k \mapsto \{\ell | x = e \ ; \ s\}), \overline{q}, \mathit{Upd}) \ \mathit{cn}$. Using contexts both greatly simplifies the notation and spares us from having to duplicate rules.

To help defining DRMA operations, we will also use $\mathcal{D}_k[a, \ell, s, \mathit{Upd}]$, which is an extension of $\mathcal{R}_k[a, \ell, s]$ exposing the *Upd* field. It is defined as:

$$\mathcal{D}_k[a, \ell, s, \mathit{Upd}] ::= \alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, s], \overline{q}, \mathit{Upd}) \ \mathit{cn}$$

We use the notation $[\overline{i} \mapsto \overline{\mathit{Task}}] \uplus [k \mapsto \{\ell | s\}]$ to access and modify the local store and current statement of a process k. Just as a statement can be decomposed into a sequence $s; s'$ with the associative property, the task mapping can be decomposed into $[\overline{i} \mapsto \overline{\mathit{Task}}] \uplus [k \mapsto \mathit{Task}]$, we use the disjoint union $\uplus$ to work on a single process disjoint from the rest.

The first three rules of the semantics define an evaluation operator $[\![e]\!]_\ell$ that evaluates an expression e using a variable environment l. We rely on $\mathrm{dom}(l)$ to retrieve the set of variables declared in l. While these rules involve a single variable environment, we often use the notation $a + l$ to involve multiple variable environments, e.g. $[\![e]\!]_{a+\ell}$. It is important to note that $[\![e]\!]_{a+\ell} = w$ implies that w is not a variable, it can only be an object or future name, `null`, or an integer value.

New creates a new active objects on N processes with parameters v, used to initialize object fields. We use $\mathit{fields}(\mathit{Act})$ to retrieve names and rely on the declaration ordering to assign values to the right variables. We also add a unique

process identifier and N, respectively as *pid* and *nprocs*. The new active object β is then initialized with N processes and the resulting object environment A.

Assign is used to change the value of a variable. The expression e is evaluated using the evaluation operator, producing value w. Since variable x can either be updated in the object or local variable environments, we use the notation $(a + \ell)[x \mapsto w] = a' + \ell'$ to update either of these and retrieve both updated environments a' and l'. They replace old ones in the object configuration.

If-True and If-False reduces an `if` statement to $s1$ or $s2$ according to the evaluation of the boolean expression v.

Get retrieves the value associated with a future f. The future must be resolved. If the future has been resolved with value w, the get statement is replaced by w.

Invk invokes an existing active object and creates a future associated to the result. This rule requires v to be evaluated into an active object, then enqueues a new request in this object. A new unresolved future f is added to the configuration. Allowing self-invocation would require a simple adaptation of this rule. Parameters $\overline{v}$ that are passed to the method are evaluated locally, into $\overline{w}$. The request queue of the active object β is then appended with a triplet containing a new future identifier f associated to the request, the method m to call and the parameters $\overline{w}$.

Serve processes a queued request. To prevent concurrent execution of different requests, the active object is required to be idle (with the current request field empty). A request $(f, m, \overline{v})$ is dequeued to build and execute a new sequential environment $i \mapsto \{l|s\}$; it relies on bind to build this environment. The process i responsible for the sequential environment is called the *head*, it is the master process responsible for serving requests.

BSPRun starts a new parallel environment from the current active object α and the method m. Every process of the active object is going to be responsible for executing one instance of the same task $\{l'|s'\}$. All parallel processes start with the same local variable environment and the same statement to execute.

Return-Value resolves a future. The expression v is first evaluated into a value w that is associated with the future f. The current request field is emptied, allowing a new request to be processed.

Return-Sub-Task terminates one parallel task. The process that returned is removed from the set of tasks running in parallel. When the last process is removed, the sequential context can be evaluated.

Sync ends the current superstep, the `sync` statement must be reached on every pid k of the parallel execution context before this rule can be reduced. DRMA operations that were requested since the last superstep and stored in the Upd field as (i, v, j, y) quadruplets are taken into account. They are used to update the object variable environment A into A' such that variable y of pid j is going to take the value v as evaluated in process i, for every such quadruplet. As Upd is an unordered set, these updates are performed in any order.

BSP-Get requests to update a local variable with the value of a remote one. We write a DRMA quadruplet such that the variable x_{src} of the remote process i is going to be read into the variable x_{dst} of the current pid k during the next sychronisation step.

BSP-Put requests to write a local value into a remote variable. The value to be written is evaluated into v, and a new update quadruplet is created in Upd. It will be taken in acocunt upon the next `sync`.

While race conditions exist in ABSP, like in active object languages and in BSP with DRMA, the language has no data race. Indeed, the only race conditions are message sending between active objects, and parallel emission of update requests. The first one results in a non-deterministic ordering in a request queue, and the second in parallel accumulation of update orders in an unordered set. Updates are performed in any order upon synchronisation but additional ordering could be enforced, e.g. based on the time-stamp of the update.

5 Current Status and Objectives

We presented a new programming model for the coordination of BSP processes. It consists of an actor-like interaction pattern between SPMD processes. Each actor is able to run an SPMD algorithm expressed in BSP. The active-object programming model allowed us to integrate these notions together by using object and methods as entry points for asynchronous requests and for data-parallel algorithms. We have shown an example of this model that features two different BSP tasks coordinated through dedicated active objects. This example also shows the usage of an experimental C++ library implementing this model that relies on MPI for flexible actor communications and a BSPlib-like implementation for intra-actor data-parallel computations.

The semantics proposed in this paper will allow us to prove properties of the programming model. Already, by nature both active objects and BSP ensure the absence of data-races and thus our programming model inherits this crucial property. To further investigate race-conditions, we should formally identify the sources of non-determinism in ABSP and show that only concurrent request sending to the same AO and DRMA create non-determinism. Another direction of research could focus on the verification of `sync` statements, checking they can only be invoked in a parallel context.

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
2. Azadbakht, K., de Boer, F.S., Serbanescu, V.: Multi-threaded actors. In: Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016., pp. 51–66 (2016)
3. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Composing, Deploying for the Grid, pp. 205–229. Springer London (2006)
4. Bisseling, R.: Parallel Scientific Computation: A Structured Approach Using BSP and MPI. OUP Oxford (2004)
5. Boer, F.D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**, 76:1–76:39 (2017)
6. Bonorden, O., Juurlink, B., von Otte, I., Rieping, I.: The paderborn university BSP (PUB) library. Parallel Comput. **29**(2), 187–207 (2003)
7. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04, pp. 123–134. ACM, New York, NY, USA (2004)
8. Fernandez-Reyes, K., Clarke, D., McCain, D.S.: Part: An asynchronous parallel abstraction for speculative pipeline computations. In: A. Lluch Lafuente, J. Proença (eds.) Coordination Models and Languages, pp. 101–120. Springer International Publishing, Cham (2016)
9. Gava, F., Fortin, J.: Formal semantics of a subset of the paderborn's BSPlib. In: 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 269–276 (2008)
10. Hains, G.: Subset synchronization in BSP computing. In: PDPTA, vol. 98, pp. 242–246 (1998)
11. Halstead Jr., R.H.: Multilisp: A language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst. **7**(4), 501–538 (1985)
12. Henrio, L., Huet, F., István, Z.: Multi-threaded active objects. In: R.D. Nicola, C. Julien (eds.) COORDINATION, *Lecture Notes in Computer Science*, vol. 7890, pp. 90–104. Springer (2013)
13. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.H.: BSPlib: The BSP programming library. Parallel Comput. **24**, 1947–1980 (1998)
14. Suijlen, W.J., BISSELING, R.: BSPonMPI. URL: http://bsponmpi.sourceforge.net (2013)
15. Tesson, J., Loulergue, F.: Formal semantics of DRMA-style programming in BSPlib. In: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (eds.) Parallel Processing and Applied Mathematics, pp. 1122–1129. Springer Berlin Heidelberg (2008)
16. Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. In: R.N. Horspool (ed.) Compiler Construction, pp. 179–196. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
17. Valiant, L.G.: A bridging model for parallel computation. CACM **33**(8), 103 (1990)