Reactive Bulk-Synchronous Programming - A Semantic Approach

Chen, Yifeng

Oxford University Computing Laboratory
Parks Road, Oxford OX1 3QD, UK
Yifeng.Chen@comlab.ox.ac.uk

Abstract. This paper presents a semantic model for Bulk-Synchronous Programming (BSP) which extends previous models by incorporating reactive (non-terminating) programs. It is shown that the traditional relational calculus fails to describe BSP's non-termination precisely. A set of laws is given and the normal form of BSP programs is identified. The notion of public variable is proposed to aid reactive programming. Finally, the paper shows a BSP treatment of the dining-philosopher problem which illustrates the power of BSP in reactive programming.

1 Introduction

1.1 BSP

The BSP model [12] provides an important paradigm bridging sequential and parallel programming, as shown by its recent popularity [8, 11, 5]. A BSP computer consists of several processors, each with its own local memory, connected by a network with an efficient barrier-synchronisation mechanism; and a BSP computation consists of a sequence of parallel supersteps, during each of which each processor executes local operations (steps). Communication between processors is completed only at the end of the superstep in which it was initiated. We suppose that, during a superstep, a variable can be accessed by more than one processor with *put* and *get* commands. Although that extends the original version of BSP [12], it has since turned out to be convenient.

We follow the BSP Would-Wide Standard Library [6] and employ the communication commands *put* and *get* (called bsp-put and bsp-get in [12]). The former copies its current local data into the local memory of its destination; it is buffered at each end and so the data of communication are available only at the beginning of the next superstep. The latter achieves the reverse. *sync* command sets a synchronisation point, i.e. the end of a superstep. Further details appear in section 3.

1.2 Formal Semantics

The first semantic model of BSP was given in [9]; it supposes communication nondeterminism, communication delay and *open processes* (whose variables and communications are free for instantiation by global values from the environment).

However it supported only terminating processes, did not encompass iteration or recursion, and experienced difficulty in dealing with unmatched supersteps in parallel composition. The model was improved in [10] where the latter problem was resolved by introduction of divergence at the end of the shorter process to match supersteps; healthiness conditions for BSP programs were also provided.

In this paper, we shall study different kinds of non-terminations. By clarifying their semantics, we can, at the first time, give the correct semantics to recursion and iteration in BSP, which are necessary for supporting reactive programming.

1.3 BSP Challenges

Non-termination Recursion was not considered in [9, 10]. It is easy to have recursion within a superstep, but recursion over supersteps can not be incorporated in traditional relational calculus.

Before we give the semantics of recursion, we must understand what is non-termination first.

Without considering non-termination, one can describe a sequential specification as a relation $r \subseteq X \times X$, such that, for $x\ r\ x'$, the program starts from initial state x and terminates in the state x'. We need define the non-termination to be the worst specification in the refinement ordering. However the most chaotic relation $X \times X$ already stands for the specification of Arbitrary But Terminate (ABT for short). No relation $r \subseteq X \times X$ can be reasonably used to represent non-termination.

The most traditional way to provide such information is to add a special state $\perp$ in X. The non-termination is then represented by $X_\perp \times X_\perp$ where $X_\perp \mathrel{\hat{=}} X \cup \{\perp\}$. A number of healthiness conditions are needed to make sure that once a specification does not terminate, it can reach any state including $\perp$.

A relation $r : X_\perp \times X_\perp$ in relational semantics can also be represented as a predicate on four free variables $p(x, x', ok, ok')$ with three healthiness conditions [7]. ok/ok' play the same roles as $\perp$. The non-termination is represented as $true$ and ABT is represented as $ok \Rightarrow ok'$. ok records that the program starts properly and ok' records its successful termination.

Neither of the two approaches provides enough information for the correct semantics for recursion in BSP. BSP has different and more complicated kinds of non-terminations:

$$P \mathrel{\hat{=}} do \quad true \rightarrow skip \quad do$$
$$Q \mathrel{\hat{=}} ABT; sync;\ ABT; sync;\ do \quad true \rightarrow skip \quad do$$
$$R \mathrel{\hat{=}} do \quad true \rightarrow\ ABT; sync \quad do$$

We suppose ABT to be the terminate specification that assigns variables with arbitrary values and executes arbitrary communication commands within a superstep. P diverges immediately at the beginning. It should be the worst specification (i.e. chaos) in the refinement ordering of BSP. Its behaviour is totally chaotic. Q diverges after 2 legal supersteps. The rest of its behaviour is chaotic.

R never diverges within finite supersteps. It is a reactive process with normal behaviour during each superstep.

Recall the philosophy that "non-termination should be worse than ABT". We actually should expect more: Q is worse than R and P is worse than Q. The later a process diverges, the less chaotic its behaviour is. Unfortunately, P, Q and R can only be represented as the same kind of non-termination if we use ok/ok' or $\perp$. Traditional approaches are neither precise nor fully abstract for BSP.

We solve this problem by providing the information of divergent point in BSP semantics. The idea is to represent termination as $\top$, represent the non-termination of P as a non-deterministic choice in set $\{0, 1, 2, \ldots, \infty, \top\}$, represent the non-termination of Q as $\{2, 3, \ldots, \infty, \top\}$ and represent the non-termination of R as $\{\infty, T\}$. The non-deterministic choice should always be upwards-closed. We shall use notation $@/@'$ instead of ok/ok' in our semantics. Details can be found in section 3.

Parallel Composition One of the difficulties this paper overcomes concerns the parallel composition of processes each having different numbers of supersteps. Previous attempts broached that problem by producing divergence at the start of the processes or the end [10] of the shorter process's execution. We solve that problem by introducing empty supersteps and automatically appending those to the shorter process. The parallel composition no longer generates divergence not existing in program components.

Resource Sharing The previous challenges were semantic. A further challenge confronting BSP is its effectiveness in expressing distributed programs. With only asynchronous communication, how can the latest value of a variable be read immediately by another processor?

We introduce the notion of *public variables*. A public variable is a variable with copies on all processes. Modification to a public variable must be done by communication, which broadcasts to all of its copies. Therefore, each public variable always has a unique value on all processes. Reading the local copy of a public variable is already accessing its latest value. This notion substantially simplifies reactive programming.

2 BSP language

2.1 BSP Syntax

BSP is usually embedded in a normal imperative language such as Fortran or C. The Guarded-Command language (*code*) abstracts such by containing only the basic imperative language operators. BSP commands can be embedded in code.

The following table lists the basic BSP commands.

$abort$	divergence	$skip$	no operation
$\mathbf{x} := \mathbf{e}$	local assignment	$P; Q$	sequential composition
$P \lhd b \rhd Q$	binary conditional	$P \sqcap Q$	non-deterministic choice
$var\ \mathbf{x}\ P$	variable block	$put\ \mathbf{x} := \mathbf{e}$	put $\mathbf{e}$ to variable $\mathbf{x}$
$get\ \mathbf{x} := \mathbf{y}$	get value of $\mathbf{y}$ into $\mathbf{x}$	$sync$	synchronisation
$\lvert P \rvert$	closure of process P	$P_{xs} \parallel Q_{ys}$	parallel composition
$b * P$	iteration		

We use bold lowercase characters to denote vectors of variables and italic lowercase single characters to denote single variables. P and Q denote BSP processes.

We shall use some some conventions in the following sections.

$var\ \mathbf{x} = \mathbf{e}\ P \mathrel{\hat{=}}\ var\ \mathbf{x}\ (\mathbf{x} := \mathbf{e};\ P)$	New variable with initialisation
$var\ \mathbf{x}/\mathbf{y}\ P \mathrel{\hat{=}}\ var\ \mathbf{y} = \mathbf{x}\ (P;\ \mathbf{x} := \mathbf{y})$	The same as $P[\mathbf{x}/\mathbf{y}, \mathbf{x}'/\mathbf{y}']$
$0.var\ \mathbf{x}\ P \mathrel{\hat{=}}\ var\ 0.\mathbf{x}/\mathbf{x}\ P$	$0.\mathbf{x}$ denotes new symbol(s)
$x : \bullet\ e \mathrel{\hat{=}}\ (x := x \bullet e)$	$\bullet$ can be any binary operator[2]
$P \times Q \mathrel{\hat{=}}\ P; Q$	iff P and Q commute: $P; Q = Q; P$

More generally, $\prod_{0 \le i < n} P_i \ \hat{=}\ P_0 \times P_1 \times \ldots P_{n-1}$ denotes $P_0; P_1; \ldots; P_{n-1}$, provided the P_i commute.

3 BSP Semantics

3.1 Sequences

A sequence s of type T is a function from natural numbers to values $s : \{0, 1, 2, \ldots\} \to T \cup \{\dashv\}$ satisfying $s[i] = \dashv \Rightarrow s[i+1] = \dashv$. There $\dashv \notin T$ and is used to mark the end of useful information in a finite sequence. Of course a sequence may be infinite; in that case for each i, $s[i] \neq \dashv$. The empty sequence $\langle\rangle$ satisfies, for any i, $\langle\rangle[i] = \dashv$. The set of all sequences of T is defined as *sequence of T*. The length $|s| \in \mathbb{N} \cup \{\infty\}$ of sequence s is defined by $|s| \mathrel{\hat{=}} \bigsqcup\{i \in \mathbb{N} \mid s[i] \neq \dashv\}$. The concatenation is written as s_t. Truncation of a sequence is defined by $(s \downarrow n)[i] = (s[i] \lhd 0 \le i < n \rhd \dashv)$, and $s \uparrow n$ denotes the rest of the sequence.

3.2 Observables

In this section we give the observables on which our model is based. The following table lists conventions for the use of symbols in this paper.

$x, y \ldots$	single variable	$\partial x, \partial y, \ldots$	name of a variable
$xs, ys, \ldots$	set of variables	$xq, yq, \ldots$	sequence variable
$xqs, yqs, \ldots$	set of sequence variables	$\mathbf{x}, \mathbf{y}, \ldots$	vector of variables

If δ denotes a set of variables, we let $\overline{\delta} \mathrel{\hat{=}}\ \{\overline{d} \mid d \in \delta\}$; if δ denotes a vector of variables, we let $\overline{\delta} \mathrel{\hat{=}}\ (\overline{d} \mid d\ in\ \delta)$. For any variable x, x' denotes its output value; symbol x denotes the input value of itself.

The following table lists the meanings of the symbols that we shall use.

us	set of variables owned by P	vs	set of variables known by P
c	bag of P's communications	$\#$	superstep counter
$@$	divergence counter	uqs	traces of variables in us
vqs	traces of variables in vs	cq	trace of P's communications
$\overline{vqs}$	traces of variables in $\overline{vs}$	$\overline{cq}$	trace of $\overline{c}$

A variable of type T is said to be *known* to process P if and only if it is accessible by *put* and *get* commands. A known variable is said to be *owned* by P if and only if its modified value can be read in P without communications. A known variable which is not owned is said to be *remote*.

The record of a communication request is a 3-tuple ($name$, $dest$, src) in which $name$ can be either *put* or *get*, $dest$ is the name of the destination variable, src is either the name of the copied variable for *get* or the value of an expression for *put*. For instance, $(get, \partial x, \partial y)$ and $(put, \partial x, 2)$ are valid requests. c is a variable whose value can be any bag of valid communication requests from the set $\{(get, \partial x, \partial y) \mid x, y \in vs\} \cup \{(put, \partial x, e) \mid x \in vs, e \in T\}$.

cq denotes the *sequence of bags of communications* as the program evolves.

Process P evolves as supersteps are executed. The record of the values of each owned variable x during the new supersteps is appended to xq to form xq' in the output alphabet; the record of communication requests issued by P during the new supersteps is appended to cq to form cq'.

$\#$ ($\#'$) stands for the serial number of the superstep in which P starts (terminates). $@'$ denotes the earliest superstep in which the process can diverge. $@' = \top$ means that the process has not diverged. It has the same meaning as $ok' = true$ in [7]. We call extra symbols representing non-termination *indicators*. Since $@'$ already provides information of termination, ok and ok' are no longer needed in our semantics. The type of $\#$ and $\#'$ is $\mathbb{N}_\infty \mathrel{\hat{=}} \mathbb{N} \cup \{\infty\}$, and the type of $@$ and $@'$ is $\mathbb{N}_\infty \cup \{\top\}$.

$@'$ is upwards-closed. If $@'$ can be chosen only from $\{\top\}$, the process terminates; if the range of $@'$ is $\{n, n+1, \dots\} \cup \{\top\}$, the process diverges during superstep n; if the range is $\{\infty, \top\}$, the process is reactive (challenge, section 1.3).

$\overline{vqs}$ denotes the traces of all known variables. $\overline{cq}$ denotes the trace of the collections of all communication requests during each superstep (not only those issued by P). $\overline{vqs}$ and $\overline{cq}$ contain all environmental information that should be given before a process's own behaviour can be described (challenge, section 1.3).

Pair vs, us is called the *signature* of the alphabet. The input alphabet with signature vs, us consists of the set of input variables; the output alphabet consists of its set of output variables. $\alpha_{vs,us}$ is the core alphabet shared by both input and output alphabets. $\alpha'_{vs,us}$ denotes the set $\{x' | x \in \alpha_{vs,us}\}$.

$$\begin{aligned} \alpha_{vs,us} &\mathrel{\hat{=}} \{\#, @, cq\} \cup us \cup uqs \\ in\alpha_{vs,us} &\mathrel{\hat{=}} \alpha_{vs,us} \cup \overline{vqs} \cup \{\overline{cq}\} \\ out\alpha_{vs,us} &\mathrel{\hat{=}} \alpha'_{vs,us} \end{aligned}$$

3.3 Basic Semantics

Given alphabet signature vs, us, the semantic denotation of BSP process P is a predicate $[P]_{vs,us} : \alpha P \to \mathbf{Boolean}$. The basic semantics is defined in terms of the guarded-command language, whose relational semantics is standard (see [4]). In the following definitions, we suppose $\mathbf{x} = (x_1, x_2, \ldots, x_t)$ and $\mathbf{e} = (e_1, e_2, \ldots, e_t)$. We also use the notation of $A \vdash B$ (*design* [7]). A is the precondition under which the computation terminates in which case it does so in a state satisfying the postcondition B. If execution begins in an environment for which A is false or does not begin in finite time, termination is not guaranteed.

BSP 1 $[abort]_{vs,us} \mathrel{\hat{=}} \perp$

$\perp$ is the specification for total chaos.

Laws 1 $\perp \sqcap P = P \sqcap \perp = \perp$
$\perp \| P = P \| \perp = \perp$
$\perp; A = A; \perp = \perp$
if A is a local computation within one superstep.

BSP 2 $[skip]_{vs,us} \mathrel{\hat{=}} I\!I$

Refer to section 3.4 for the definitions of $\perp$ and $I\!I$

BSP 3 $[\mathbf{x} := \mathbf{e}]_{vs,us} \mathrel{\hat{=}} true \vdash (\forall x_i \ in \ \mathbf{x}.\ x_i' = e_i) \wedge (\forall y \in \alpha_{vs,us} \backslash \mathbf{x}.\ y = y')$
where every variable appearing in $\mathbf{e}$ *lies in us.*

In other words, only the variables in $\mathbf{x}$ are changed.

Laws 2 $(\mathbf{x} := \mathbf{e}_1;\ \mathbf{x} := \mathbf{e}_2) = (\mathbf{x} := \mathbf{e}_2[\mathbf{e}_1/\mathbf{x}])$

BSP 4 $[P \lhd b \rhd Q]_{vs,us} \mathrel{\hat{=}} [P]_{vs,us} \lhd b \rhd [Q]_{vs,us}$
where every variable appearing in b lies in us.

Laws 3 $(x := e); (P \lhd b \rhd Q) = (x = e; P) \lhd b[e/x] \rhd (x := e; R)$
$(P \lhd b \rhd Q); R = (P; R) \lhd b \rhd (Q; R)$
$(\mathbf{x} := \mathbf{e};\ P \lhd b \rhd Q) = ((\mathbf{x} := \mathbf{e}; P) \lhd b[\mathbf{e}/\mathbf{x}] \rhd (\mathbf{x} := \mathbf{e}; Q))$

BSP 5 $[P \sqcap Q]_{vs,us} \mathrel{\hat{=}} [P]_{vs,us} \vee [Q]_{vs,us}$

Laws 4 $(P \sqcap Q) \| R = (P \| R) \sqcap (Q \| R)$

$P \lhd b \rhd Q$ and $P \sqcap Q$ satisfy a few other laws available in sequential semantics.

BSP 6 $[var\ x\ P]_{vs,us} \mathrel{\hat{=}} \exists x, x', xq, xq', \overline{xq}.\ [P]_{vs_1,us_1}$
$[var\ \mathbf{x}\ P]_{vs,us} \mathrel{\hat{=}} [var\ \ x_1, x_2, \ldots, x_t\ \ P]_{vs,us}$

var introduces new program variable x; P is then interpreted under the expanded alphabet with signature vs_1, us_1 where $vs_1 \mathrel{\hat{=}} vs \cup \{x\}$ and $us_1 \mathrel{\hat{=}} us \cup \{x\}$.

BSP 7 $[put\ \mathbf{x} := \mathbf{e}]_{vs,us} \hat{=}\ cq[\#] :\cup\{(put, \partial x_i, e_i)|\ x_i\ in\ \mathbf{x}\}$
where every variable appearing in $\mathbf{e}$ *lies in us.*

$:\cup$ is a command, implying that the only variables which change are those appearring on the left-hand side of the definition. Command *put* collects its communication requests in $cq[\#]$. If x_i is a variable owned by the process itself, the new value will be available at the beginning of the next superstep — a decision convenient to the refinement calculus but different from BSPLib [4] where it is considered to be a local assignment taking effect immediately. Our decision makes the semantics more unified and straightforward.

BSP 8 $[get\ \mathbf{x} := \mathbf{y}]_{vs,us} \hat{=}\ cq[\#] :\cup\{(get, \partial x_i, \partial y_i)|x_i\ in\ \mathbf{x}\}$

Similarly if x_i and y_i are located on the same processor, semantically the copying operation is postponed until the end of the current superstep.

Laws 5
$$\begin{array}{l}(get\ \mathbf{gx} := \mathbf{gy}; \mathbf{x} := \mathbf{e}) = (\mathbf{x} := \mathbf{e}; get\ \mathbf{gx} := \mathbf{gy})\\ (\mathbf{x} := \mathbf{e};\ put\ \mathbf{px} := \mathbf{pe}) =\ (put\ \mathbf{px} := \mathbf{pe}[\mathbf{e}/\mathbf{x}];\ \mathbf{x} := \mathbf{e})\\ (get\ \mathbf{x}_1 := \mathbf{y}_1;\ get\ \mathbf{x}_2 := \mathbf{y}_2) =\ (get\ \mathbf{x}_1, \mathbf{x}_2 := \mathbf{y}_1, \mathbf{y}_2)\\ (put\ \mathbf{x}_1 := \mathbf{e}_1;\ put\ \mathbf{x}_2 := \mathbf{e}_2) =\ (put\ \mathbf{x}_1, \mathbf{x}_2 := \mathbf{e}_1, \mathbf{e}_2)\\ (get\ \mathbf{gx} := \mathbf{y};\ put\ \mathbf{px} := \mathbf{e}) =\ (put\ \mathbf{px} := \mathbf{e};\ get\ \mathbf{gx} := \mathbf{y})\end{array}$$

BSP 9 $[P; Q]_{vs,us} \hat{=}\ \exists \alpha^0_{vs,us}.\ [P]_{vs,us}[\alpha^0_{vs,us}/\alpha'_{vs,us}] \wedge [Q]_{vs,us}[\alpha^0_{vs,us}/\alpha_{vs,us}]$

The definition of sequential composition is essentially the same as that in [7]. The only difference is that we have extra symbols $\overline{vqs}$ and $\overline{cq}$ in the input alphabet (or we lack $\overline{vqs}$ and $\overline{cq}$ in the output alphabet). These symbols refer to the global information shared by all program components and need not be substituted in sequential composition.

BSP 10
$$\begin{array}{rl}[sync]_{vs,us} \hat{=} & \prod_{x\in us}(xq[\#] := x);\\ & \prod_{x\in us}(\mathit{II} \lhd chcs_{\#}(x) = \emptyset \rhd x :\in chcs_{\#}(x));\\ & \#,\ @ := \# + 1,\ @ + 1;\\ & cq[\#] := \emptyset\end{array}$$
where $chcs_{\#}(x) \hat{=}\ \{e \mid (put, \partial x, e) \in \overline{cq}[\#]\} \cup \{\overline{yq}[\#] \mid (get, \partial x, \partial y) \in \overline{cq}[\#]\}$

At a synchronisation point, the final value of each variable x at the end of each superstep is recorded in sequence variable xq. Communication intervention takes place afterwards and only affects the following superstep. x will be changed to the value non-deterministically chosen from $chcs_{\#}(x)$. If the collection is empty, x is not changed. Some preparation is done before the beginning of the next superstep: # and @ are increased by 1 respectively and the local communication collection is initialised to be empty.

Laws 6
$$\begin{array}{l}(A; sync; P)\|(B; sync; Q) =\ (A\|B); sync; (P\|Q)\\ (A; sync; P)\|B =\ (A\|B); sync; (P\|\mathit{II})\end{array}$$
A and B are components of local computation.

BSP 11 $[\ |P|\ ]_{vs,vs} \hat{=}\ [P]_{vs,vs}\ \wedge\ (\forall xq \in vqs.\ xq' = \overline{xq} \downarrow \#') \wedge (cq' = \overline{cq} \downarrow \#')$

The behaviour of an open process depends on its environment. Using command *put*, a process can modify any remote variable by communication. $|P|$ states that P is no longer involved in any communication with its environment, i.e. the local state and the local collection of communication requests have been the global state and the global collection respectively. This definition is different from those in [9] and [10] in which intermediate states are hidden in closures. This is again consistent with our objective to support reactive processes; otherwise a reactive process may equal to sequential program *skip*.

It is easy to obtain the pre-post information from a closed process by hiding intermediate states: $hide(P) \hat{=} \exists \#, \#', cq, cq', vqs, vqs', \overline{vqs}, \overline{cq}.P$ in which the non-termination indicator ok can be defined as $@ = T$.

BSP 12 $[\, P_{\otimes xs} \parallel Q_{\otimes ys}]_{vs,us} \hat{=}$

$$var\ 0.\mathbf{s} = 1.\mathbf{s} = \mathbf{s}$$

$$\begin{pmatrix} 0.var\ \mathbf{s}\ [P]_{vs,xs}\ \wedge\ 1.var\ \mathbf{s}\ [Q]_{vs,ys}; \\ \quad 0.var\ \mathbf{s}\ [(\# < 1.\#) * sync]_{vs,xs} \\ \wedge\ 1.var\ \mathbf{s}\ [(\# < 0.\#) * sync]_{vs,ys} \\ \wedge\ \#' \in \{0.\#', 1.\#'\}\ \wedge\ @' \in \{0.@', 1.@'\} \\ \wedge\ (\forall xq \in uqs.\ xq' = xq \downarrow \#')\ \wedge\ (cq' = cq \downarrow \#'); \\ I\!I \end{pmatrix}$$

where $\mathbf{s} \hat{=} (\#, @, cq)$ *and* $xs, ys \subseteq us$

Different semantic decisions on parallel composition have been made in the past. Ours i the first time supports reactive processes (challenge, section 1.3).

In the definition, $\mathbf{s}$ consists of all shared symbols with different meanings on the two sides. It is duplicated so that P and Q can be calculated in disjoint alphabets. P and Q may have different numbers of supersteps. Unbalanced parallel composition is undefined in BSPLib [6], defined inaccurately in [9], defined as immediate divergence in an early model and defined to be divergent at the end of shorter process in [10]. In our approach, empty supersteps are appended to the shorter one. Traces uqs and cq are curtailed to meet healthiness conditions before $I\!I$ forces it to be divergent if either of them diverges. Parallel composition itself does not generate divergence. In operating systems, a service process will not fail because one of its customer processes terminates after being served.

Laws 7 $P \| Q = Q \| P \qquad (P \| Q) \| R = P \| (Q \| R)$

BSP 13 $[b * P]_{vs,us} \hat{=} \bigvee \{X | X \in \mathbf{BSP}_{vs,us},\ X \Rightarrow (P; X) \lhd b \rhd I\!I\}$

The semantics of iteration is the weakest fixed-point of the recursive equation. Recursion can be defined in the same way, although here we consider only iteration. Note that $X \in \mathbf{BSP}_{vs,us}$ is necessary (c.f. [7]) for BSP; otherwise unhealthy *true* could be a fixed-point.

Using those laws, a finite BSP program can be written in normal form [7].
$ST =\ put\ \mathbf{y} := \mathbf{e};\ \mathbf{x} := \mathbf{e};\ get\ \mathbf{x} := \mathbf{y}$
$NF =\ abort \mid ST \mid ST; sync; NF \mid NF \lhd b \rhd NF \mid NF \sqcap NF$

3.4 Healthiness Conditions

Healthy specifications with signature vs, us form the semantic space $\mathbf{BSP}_{vs,us}$. A healthy specification P in $\mathbf{BSP}_{vs,us}$ satisfies a number of healthiness conditions:

$$
\begin{aligned}
H_c &: cq \downarrow \# = cq' \downarrow \# \quad cq[\#] \subseteq cq'[\#] \quad cq'[n] \subseteq \overline{cq}[n] \quad (\forall n \leq \#') \\
H_{us} &: xq \leq xq' \leq \overline{xq} \quad (\forall xq \in uqs) \\
H_{c,us} &: |cq| - 1 = |xq| = \# \quad (\forall xq \in uqs) \quad |cq'| - 1 = |xq'| = \#' \\
H_{@} &: \# \leq @' \quad @' \neq \top \Rightarrow \forall m, n \geq @'.\ P[m/\#', n/@'] \\
H_{\#} &: \# \leq \#' \\
H_{pre} &: P \Rightarrow P[a_x^n / xq[n] \mid n < \#,\ xq \in uqs \cup \overline{vqs} \cup \{cq, \overline{cq}\}] \\
&\quad (a_x^n \text{ can be any value of the same type as } xq[n]) \\
H_{stt} &: P = P \lhd @ = \top \rhd \bot \\
H_{II} &: P = P; II
\end{aligned}
$$

where $\bot$ and II are defined as follows: $\bot \mathrel{\hat{=}} H_c \wedge H_{us} \wedge H_{c,us} \wedge H_{\#} \wedge H_{@}$, $A \vdash B \mathrel{\hat{=}} ((A \wedge @ = \top) \Rightarrow (B \wedge @' = \top)) \wedge \bot$ and $II \mathrel{\hat{=}} \mathit{true} \vdash \forall x \in \alpha_{vs,us}.\ x' = x$. H_{stt} combined with H_{II} states that P has no further output if it does not start properly; $H_{@}$ combined with H_{II} means that $@'$ records whether the process diverges; if yes, it behaves chaotically after the divergent superstep. It can be proved that $\mathbf{BSP}_{vs,us}$ is a complete lattice and all BSP commands satisfy these healthiness conditions.

4 Example: Five Dining Philosophers

The problem of the dining philosophers is described in [3]. Five philosophers are seated in a circle. Each philosopher cycles through states *thinking*, *hungry*, and *eating*. Neighbours can not eat at the same time. A hungry philosopher will eat eventually if thinking and eating are always done in finite time.

Our solution is similar to that in [1]. Let $vs = \mathbf{s}\_\mathbf{f}$, $\mathbf{s} = s_0\_ \ldots\_ s_4$, $\mathbf{f} = f_0\_ \ldots\_ f_4$, $\mathbf{P} = \{0, 1, 2, 3, 4\}$, and ℓ be, for any $k \in \mathbf{P}$, $s_k^\ell = \{k^+, k, k^-\}$ and $f_k^\ell = \mathbf{P}$. The types of s_k and f_k are $\{t, h, e\}$ (the three states) and $\{l, r\}$ (priority between neighbours) respectively. $f_k = l$ means that philosopher k^- has priority to philosopher k; $f_k = r$ *vice versa*. Note that k^+ is increment modulo 5. $\mathbf{s}$ and $\mathbf{f}$ are public variables.

The strategy is as follows: the priority-dependency graph ($\mathbf{f}$) is maintained acyclic; a thinking philosopher never obtains priority; a hungry philosopher loses priority and he will obtain priorities from the adjacent thinking philosophers; with all priorities, a philosopher begins eating, then thinking. Thinking and eating should be completed in finite supersteps. It can be proved that the strategy is deadlock free and starvation free.

$$
\begin{aligned}
DiningPhils \mathrel{\hat{=}}\ & put\ \mathbf{s} := t,t,t,t,t;\ \ put\ \mathbf{f} := l,r,r,r,r;\ \ sync;\\
& Phil_0 \parallel Phil_1 \parallel \ldots \parallel Phil_4\\
Phil_k \mathrel{\hat{=}}\ & do\ \ true \rightarrow\\
& \quad \left(\begin{array}{l}
Thinking_k;\ \ put\ s_k := h;\ sync;\\
do\ (s_k = h) \rightarrow\\
\ \ if\ (s_{k^-} = t \wedge f_k = l = f_{k^+}) \rightarrow\ \ (put\ f_k := r;\ sync)\\
\ \ \ \ []\ (s_{k^+} = t \wedge f_{k^+} = r = f_k) \rightarrow\ \ (put\ f_{k^+} := l;\ sync)\\
\ \ \ \ []\ (s_k = s_{k^+} = t \wedge f_k = l \wedge f_{k^+} = r) \rightarrow\\
\qquad\qquad\qquad (put\ f_k, f_{k^+} := r, l; sync)\\
\ \ \ \ []\ (f_k = r \wedge f_{k^+} = l) \rightarrow (put\ s_k := e;\ sync)\\
\ \ \ \ []\ otherwise \rightarrow sync\quad fi\\
do;\\
Eating_k;\ \ put\ s_k := t;\ sync;
\end{array}\right)\\
& od\\
Thinking_k \mathrel{\hat{=}}\ & Eating_k \mathrel{\hat{=}}\ \ var\ (n \in \mathbb{N})\ (do\ n > 0 \rightarrow (n := n-1; sync)\ do);
\end{aligned}
$$

References

1. K.M. Chandy and J. Misra. Parallel Program Desgin. Addison-Wesley, Reading, Massachusetts, 1998.
2. O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming, Academic Press, 1972.
3. E.W. Dijkstra, Two Starvation Free Solutions to General Exclusion Problem, EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands 1978.
4. E.W. Dijkstra, Guarded commands, non-determinacy and the formal derivation of programs, CACM, 18, 453-457, 1975.
5. A. V. Gerbessiotis, C. J. Siniolakis and A. Tiskin, Parallel priority queue and list contraction: The BSP approach, In C. Lengauer, M. Griebl and S. Gorlatch, editors, Proceedings of Euro-Par'97, volume 1300 of Lecture Notes in Computer Science, pages 409–416. Springer-Verlag, 1997.
6. M.W. Goudreau, J.M.D. Hill, K. Lang, M.F. McColl, S.B. Rao, D.C. Stefanescu, T. Suel, T. Tsantilas, A proposal for the BSP world-wide standard library (preliminary version), 1996. with program examples.
7. C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice Hall Europe 1998.
8. W.F. McColl, Scalability, portability and predictability: The BSP approach to parallel programming, Future Generation Computer Systems 12(1996) 265-272.
9. J. He, Q. Miller, L. Chen, Algebraic Laws for BSP Programming, Euro-Par'96, LNCS 1124, 359-368 1996.
10. David Lecomber, Methods of BSP Programming, Oxford University Computing Laboratory DPhil thesis, 1998.
11. K. R. Sujikhan, J. M. D. Hill, Collection Types for Database Programming in the BSP Model, Tech Report, OXCL.
12. L.G. Valiant, A bridging model for parallel computation, Comm. ACM, 33(8) (1990) 103-111.