Axiomatic frameworks for developing BSP-style programs

A. Stewart and M. Clint
Department of Computer Science
The Queen's University of Belfast
Belfast BT7 1NN
N. Ireland

J. Gabarró*
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo 5
08028-Barcelona
Spain

Abstract

In BSP a superstep comprises a collection of concurrently executed processes with initial and terminal synchronisations. Data transfer between processes is realised through asynchronous communications. BSP programs can be organised either as explicit compositions of supersteps or as parallel compositions of threads (processes) which include synchronisation alignment operations. In this paper axiomatic semantics for the two approaches are proposed: in both cases the semantics are based on a new form of multiple substitution - *predicate substitution* which generalises previous definitions of substitution. *Predicate substitution* together with global synchronisation provide a means of linking state based and process semantics of BSP.

Features of both models are illustrated through correctness proofs of matrix multiplication programs.

1 Introduction

A BSP computation [30] may be organised as a sequence of supersteps. Each superstep comprises a collection of parallel processes. After a superstep is executed its constituent processes are synchronised. (There is an implicit initial synchronisation of the processes before the

*This work has been partially supported by DGICYT under grant PB95-0787 (project KOALA), also by ESPRIT LTR Project no. 20244 —ALCOM-IT, also by CICYT TIC97-1475-CE, also by PR98-11 of the Universitat Politècnica de Catalunya.

first superstep.) Inter-process communications within a superstep are asynchronous and non-interfering [12, 14, 22, 30]. Thus, information transferred during a particular superstep cannot be utilised until that superstep has been completed.

One approach to constructing a semantics for BSP programs is to concentrate on the definition of supersteps. For each superstep global pre-and post-conditions can be established because of the initial and terminal synchronisations. Superstep termination has the effect of resolving all ongoing communications - thus, a superstep may be considered to be an example of a communication closed layer [8] (i.e. the communications of a superstep are resolved internally, so no communications cross superstep boundaries). Semantically, it is convenient (and sufficient because of the non-interference requirement) to collect together all of the asynchronous communications of a superstep in a "bundle" and treat the communications which they denote as a form of multiple substitution of variable names by expressions [6, 13]. An illustration of the semantic model based on this approach is furnished in this paper by a correctness proof of a BSP program for matrix multiplication.

Alternatively, BSP programs may be constructed using processes as basic building blocks. Synchronisation points internal to a process can be specified by means of explicit commands [22, 29]. One way to reason about such a process model is to: (i) make environmental assumptions [3, 19] at synchronisation points; and (ii) introduce a mechanism for matching the synchronisation points of participating processes. This approach resembles axiomatic definitions of CSP [15, 1, 9, 21]. The matrix multiplication example is recast in the process framework in order to illustrate this alternative approach to constructing BSP programs.

The work reported here extends research reported in [14, 27]. In particular, two semantic models are proposed to provide a bridge linking traditional state based transformations with process models. In both cases a precise definition of BSP results. The semantic definitions throw light on the nature of BSP computation and provide frameworks wherein correctness proofs of BSP programs may be conducted.

2 BSP Computations as Sequences of SuperSteps

A bulk-synchronous parallel (BSP) computer consists of a set of processors, an asynchronous communication mechanism and a barrier synchronisation facility. A BSP computation can be viewed as a collection of supersteps. A superstep comprises distributed computation/communication pairs followed by a barrier synchronisation. A barrier cannot be passed until all outstanding asynchronous communications have been resolved. For example, the fragment

$$(\|^{ss}_{i\in\{1,\cdots,100\}} S_i); \|^{ss}_{i\in\{1,\cdots,100\}} T_i)$$

comprises two supersteps each of which has the same internal process structure. There is an implicit global synchronisation at the start and end of the fragment and also between the two supersteps (indicated by the composition operator ";").

In the derived semantics of a superstep (discussed below) global state properties are established as pre- and post-conditions. Internally, global memory is partitioned over a superstep's processes. Thus, each superstep process may be considered to have external access to a slice of global memory (specified using notation borrowed from VDM [20]). For example, a process S_k which has access only to the (global) array segment A(10..20) may be defined as:

S_k ::
ext $A(10 \ldots 20)$
$BodyS_k$

where $BodyS_k$ represents the computation which S_k performs. Data may persist after completion of the execution of S_k. In order to prevent possible interference on the global memory it is necessary to ensure that no two processes of a superstep have access to common external data. Let

$$Disjoint(S_i, i \in J)$$

denote that for every pair $(S_i, S_j), i, j \in J, i \neq j$, S_i and S_j operate on disjoint external data spaces. The details of such a data partition are straightforward and are omitted. A superstep computation

$$\mathcal{S} = \|_{i \in J}^{ss} S_i$$

proceeds as follows. Each component process S_i performs a sequence of "local" computations and asynchronous communications. These computations and communications must be non-interfering. Thus, data transmitted by a process during a particular superstep cannot be used by the designated recipient process during that superstep. A superstep terminates with a barrier which cannot be passed until all of its communications have been completed.

Two kinds of asynchronous communications are conventionally employed (see, for example, BSPlib [22]); a send communication, realised by an instruction *put* and a receive communication, realised by an instruction *get*. The instruction *put* may be considered to comprise the evaluation of a local expression, e, the asynchronous communication of the resulting value and its assignment, at the synchronisation point, to a pre-specified non-local variable (address). Operationally, the value may be delivered to the receiving process at any point in the superstep: assignment to the "variable name" may take place before the global synchronisation point if it can be guaranteed that the variable name is not referenced in the remainder of the superstep. The statement

$$put(x, e)$$

specifies that the value of the local expression, e, in the current process state, is to be communicated asynchronously to the appropriate process and assigned, on completion of the current superstep (or before, if legal), to the remote location denoted by x. Note: it is assumed that a sending process cannot have external access to the destination location - see [28] for a discussion on self addressed messages.

The asynchronous transfer of information may be modelled by means of a message relation m which maps variable names (addresses) and associated indices (if any) to values. An individual "in transit" relation is defined for each process. Let m_i denote the "local" communication relation of component process S_i. The effect of an asynchronous $put(x, e)$ instruction within process i is to modify the local message relation. Thus,

$$put(x, e) \equiv m_i := m_i \cup \{x \rightarrow e\} \tag{1}$$

Alternatively, *put* can be defined axiomatically as:

$$\{q^{m_i}_{m_i \cup \{x \to e\}}\} put(x,e) \{q\} \tag{2}$$

where q is a program assertion.

A *get* operation within a process requests that a non-local expression e be evaluated and assigned to a "local" memory location at the end of the superstep. Since e refers to "non-local" data an assumption about its point of evaluation needs to be made. The assumption is that non-local data are evaluated at process termination (i.e. at the synchronisation point). Consider, for example, the operation $get(x,y)$ where y is a variable name. Then $get(x,y)$ may be defined as:

$$get(x,y) \equiv m_i := m_i \cup \{x \to \overrightarrow{y}\} \tag{3}$$

where $\overrightarrow{y}$ denotes the value of the variable y at process termination.

The relationship between a *get* instruction and a *put* instruction is now made explicit.

1. **Rule get-to-put**:
   Let

$$\begin{array}{llll} E :: & \textbf{ext } x & F :: & \textbf{ext } y \\ & s_1; get(x,y); s_2 & & s_3 \\ \\ E' :: & \textbf{ext } x & F' :: & \textbf{ext } y \\ & s_1; s_2 & & s_3; put(x,y) \end{array}$$

   Then, $E \parallel^{ss} F = E' \parallel^{ss} F'$.

2. **Rule put-to-get**:
   Let

$$\begin{array}{llll} G :: & \textbf{ext } x & H :: & \textbf{ext } y \\ & s_1; put(y,x); s_2 & & s_3 \\ \\ G' :: & \textbf{ext } x, t & H' :: & \textbf{ext } y \\ & s_1; t := x; s_2 & & s_3; get(y,t) \end{array}$$

   where s_1, s_2, s_3 do not refer to t. Then, $G \parallel^{ss} H = G' \parallel^{ss} H'$.

Rule "rule get-to-put" can be used to convert processes which use *get* instructions into processes which use *put* instructions; rule "rule put-to-get" permits translation in the opposite direction. For convenience of expression it is useful to retain both forms of communication. Note that the laws are not symmetric - replacement of *put* operations by *get* operations involves the introduction of a fresh variable which is used to retain a copy of a value.

Rule (1) may be extended, in an obvious way, to cover the case of array element arguments. However, complications arise when the second argument of a *get* expression is an array element. Consider, for example, the operation $get(x, a(i))$ where $a(i)$ is an array element. It is assumed that the array index i is a local variable and that the *get* operation utilises the value of i in the current state. However, $a(i)$ refers to a non-local location. Consequently, a semantics of $get(x, a(i))$ involves a mixed state evaluation of the term $a(i)$ - i being evaluated in the current state and the expression $a(\text{"}i\text{"})$ being evaluated in the state obtaining immediately before synchronisation. Rule (3) may be extended by introducing a mixed state expression evaluation - see the Appendix for details. In special circumstances, when a *get* instruction occurs after all local computation, the mixed state evaluation requirement degenerates to solely final state expression evaluation.

A reasoning system for BSP programs can be constructed by adding the new proof rules to those used for reasoning about sequential programs. The example below illustrates correctness proofs of individual superstep processes (excluding composition interference).

S_1 ::
ext x
$\{Pre - S_1 : m_1 = \{\mapsto\}\ \}$
if $x = 1$ then $put(y, 1)$ else $get(x, 1)$ fi
$\{Post - S_1 : x = 1 \Rightarrow m_1 = \{y \rightarrow 1\} \wedge$
$x \neq 1 \Rightarrow m_1 = \{x \rightarrow 1\}\ \}$

S_2 ::
ext y
$\{Pre - S_2 : m_2 = \{\mapsto\}\ \}$
if $y = 1$ then $put(x, 1)$ else $get(y, 1)$ fi
$\{Post - S_2 : y = 1 \Rightarrow m_2 = \{x \rightarrow 1\} \wedge$
$y \neq 1 \Rightarrow m_2 = \{y \rightarrow 1\}\ \}$

The precondition/postcondition relationships can be established using (2), (3), the axiom of assignment, the rules of composition and the rule of selection [6, 13]. It remains to add a superstep composition proof rule in order to reason about the combined system $\mathcal{SMALL} = S_1 \parallel^{ss} S_2$. In the remainder of this section the nature of the interference between processes at a synchronisation point is specified. In order to simplify matters it is assumed that the "in transit" data to each variable is consistent - i.e. processes do not initiate conflicting data transfers by sending two or more distinct values to the same destination (see [28, 14] for a treatment of nondeterminism). Let the *superstep message bundle* be defined, for a set of processes $\{S_i | i \in J\}$, by:

$$m = \cup_{i \in J} m_i \tag{4}$$

Messages are non-interfering if m is a function. Superstep initialisation consists in identifying m with the empty relation

$$\overleftarrow{m} = \{\mapsto\} \tag{5}$$

The semantics of a superstep is constructed in two phases;

1. non-interfering computation and message generation; and

2. message delivery.

Let $\parallel^I$ denote the parallel composition of independent processes, $\parallel^{ss}$ denote superstep construction and *barrier* denote a message delivery operator. Then,

$$\parallel^{ss}_{i \in J} S_i = \parallel^I_{i \in J} S_i; barrier \tag{6}$$

The semantics of $\parallel^I$ is well known [23, 9]:

$$\frac{\{P_i\}S_i\{Q_i\}, Disjoint(S_i, i \in J)}{\{\bigwedge_{i \in J} P_i\}\ \parallel^I_{i \in J}\ S_i\{\bigwedge_{i \in J} Q_i\}} \tag{7}$$

Let *ext* be a function which returns the (external) data space (set of variables) of a process. The predicate $Disjoint(S_i, i \in J)$ is defined by:

$$\forall i, j \in J. i \neq j \Rightarrow ext(S_i) \cap ext(S_j) = \{\} \tag{8}$$

Interference between processes can occur only at *barrier*s where (non-interfering) messages in transit must be delivered. Semantically, *barrier* ensures that all communications are delivered to their destinations and then deletes all message routing information. Operationally, *barrier* corresponds to the execution of a collection of assignments followed by the emptying of the message relation:

$$barrier \equiv \forall v \in dom(m).v := m(v); m := \{\mapsto\} \tag{9}$$

Note that the set of names whose values are reassigned is not explicitly specified - rather, it is extracted from the relation m. The asynchronous communication mechanism is *confluent* in that the order in which messages are added to m is unimportant. Thus, given a consistent set of messages, the *barrier* operation is a unique state transformer. The meaning of the simultaneous assignment in (9) is a generalisation of data parallel assignment [10, 26]. A semantic treatment of such simultaneous substitutions based on the idea of *predicate substitution* is given in [28]. In this approach ${}^{\lambda z.\wp(z)}Q_{\lambda z.e(z)}$ denotes the set of simultaneous substitutions $\{x \to e(x) | \wp(x) \wedge x \in freevar(Q)\}$. The message delivery operation (9) can then be modelled by the predicate-substitution:

$$\{{}^{\lambda z.z \in dom(m)}q_{\lambda z.m(z)}\}\forall v \in dom(m).v := m(v)\{q\} \tag{10}$$

Informally, all stores with name $v \in dom(m)$ are updated with the values $m(v)$. The predicate $\lambda z.z \in dom(m)$ determines whether or not a variable is to be modified by a message while $\lambda z.m(z)$ determines the content of a message. Formally, predicate-substitution is defined inductively over assertions. Let y be an arbitary name, P a predicate, $\Box$ a monadic operation, $\circ$ a dyadic operation, R a relation and U a universal quantifier (either $\forall$ or $\exists$).

$$\begin{array}{rcl}
{}^{\lambda z.\wp(z)}(\Box P)_{\lambda z.e(z)} & = & \Box({}^{\lambda z.\wp(z)}P_{\lambda z.e(z)}) \\
{}^{\lambda z.\wp(z)}(P_1 \circ P_2)_{\lambda z.e(z)} & = & {}^{\lambda z.\wp(z)}(P_1)_{\lambda z.e(z)} \circ {}^{\lambda z.\wp(z)}(P_2)_{\lambda z.e(z)} \\
{}^{\lambda z.\wp(z)}(Ux.P)_{\lambda z.e(z)} & = & Ux.({}^{\lambda z.\wp(z) \wedge z \neq x}P_{\lambda z.e(z)}) \\
{}^{\lambda z.\wp(z)}(y_1 R y_2)_{\lambda z.e(z)} & = & {}^{\lambda z.\wp(z)}(y_1)_{\lambda z.e(z)} R^{\lambda z.\wp(z)}(y_2)_{\lambda z.e(z)} \\
{}^{\lambda z.\wp(z)}y_{\lambda z.e(z)} & = & \left\{ \begin{array}{ll} e(y) & \text{if } \wp(y) \\ y & \text{otherwise} \end{array} \right.
\end{array}$$

Thus,

$${}^{\lambda z.false}P_{\lambda z.e(z)} = P \tag{11}$$

and so

$$(m = \{\mapsto\}) \Rightarrow {}^{\lambda z.z \in dom(m)}P_{\lambda z.e(z)} = P \tag{12}$$

Finally, provided that the set of delayed messages is consistent, the barrier operation can be identified with message delivery and the reassignment of m:

$$\frac{function(m)}{\{{}^{\lambda z.z \in dom(m)}(q^m_{\{\mapsto\}})_{\lambda z.m(z)}\}barrier\{q\}} \tag{13}$$

Consequently (see (12)),

$$barrier; barrier = barrier \tag{14}$$

The use of the *barrier* proof rule is illustrated by a correctness argument for the system $\mathcal{SMALL} = S_1 \parallel^{ss} S_2$ where S_1 and S_2 are as defined earlier:

Lemma 1 *correctness of* $\mathcal{SMALL}$
$\{true\}S_1 \parallel^{ss} S_2\{x = 1 \vee y = 1\}$

Proof:

1. $S_1 \parallel^{ss} S_2$
   $= <$ by the definition of $\parallel^{ss}$ $>$
   $S_1 \parallel^I S_2; barrier$
2. $\{true\}S_1 \parallel^I S_2\{Post - S_1 \wedge Post - S_2\}$
   $<$ by the definition of $\parallel^I$ and the correctness of S_1 and S_2 $>$
   It remains to show that $\{Post - S_1 \wedge Post - S_2\}barrier\{x = 1 \vee y = 1\}$. This requirement reduces to:
   $(Post - S_1 \wedge Post - S_2) \Rightarrow {}^{\lambda z.z \in dom(m)}(x = 1 \vee y = 1)_{\lambda z.m(z)}$
3. ¿From $\{Post - S_1 \wedge Post - S_2\}$ the following cases may be distinguished:
   (a) $(x = 1 \wedge y = 1) \vee (x \neq 1 \wedge y \neq 1) \Rightarrow m = \{y \rightarrow 1, x \rightarrow 1\}$
      i. $m = \{y \rightarrow 1, x \rightarrow 1\} \Rightarrow$
         $function(m)$
      ii. $m = \{y \rightarrow 1, x \rightarrow 1\} \Rightarrow$
         ${}^{\lambda z.z \in dom(m)}(x = 1 \vee y = 1)_{\lambda z.m(z)} = (1 = 1 \vee 1 = 1) = true$
         $<$ by the definition of predicate substitution $>$
   (b) $(x = 1 \wedge y \neq 1) \Rightarrow m = \{y \rightarrow 1\}$
      i. $m = \{y \rightarrow 1\} \Rightarrow$
         $function(m)$
      ii. $m = \{y \rightarrow 1\} \Rightarrow$
         ${}^{\lambda z.z \in dom(m)}(x = 1 \vee y = 1)_{\lambda z.m(z)} = (x = 1 \vee 1 = 1) = true$
         $<$ by the definition of predicate substitution $>$
   (c) $(x \neq 1 \wedge y = 1) \Rightarrow m = \{x \rightarrow 1\}$
      i. $m = \{x \rightarrow 1\} \Rightarrow$
         $function(m)$
      ii. $m = \{x \rightarrow 1\} \Rightarrow$
         ${}^{\lambda z.z \in dom(m)}(x = 1 \vee y = 1)_{\lambda z.m(z)} = (1 = 1 \vee y = 1) = true$
         $<$ by the definition of predicate substitution $>$

This concludes the proof. □

$\mathcal{SMALL}$ illustrates a situation in which a single location may be sent multiple messages. In this case the messages are consistent (with value 1). The framework above may be extended [14, 28] to provide non-deterministic selection [7] of conflicting messages.

3 SuperStep based BSP Matrix Multiplication

The $N \times N$ matrix product C of two $N \times N$ matrices A and B is defined by:

$$C = A \cdot B \Leftrightarrow \forall i, j \in \{1 \ldots N\} : c_{i,j} = \sum_{k=1 \ldots N} a_{i,k} * b_{k,j}$$

One method for computing a matrix product is to *partition* matrices into (square) sub-blocks. The following partitioning functions *lower*, *upper*, *range* and *block* are introduced:

- $lower(r, s)$ defines the lower bound of the rth partition (one dimensional) where each component has size s:
$$lower(r, s) \equiv (r - 1) * s + 1$$
- $upper(r, s)$ defines the upper bound of the rth partition (one dimensional) where each component has size s:
$$upper(r, s) \equiv r * s$$
- $range(r, s)$ defines the index range of the rth partition (one dimensional) where each component has size s:
$$range(r, s) \equiv [lower(r, s)..upper(r, s)]$$
- $block(i, j, s)$ defines the index range of the (i, j)th partition (two dimensional) where each component has size s in each dimension:
$$block(i, j, s) \equiv [range(i, s), range(j, s)]$$

For example, let $A = (a_{ij})$, $1 \leq i, j \leq 6$ be a square matrix (a_{ij} is used as a synonym for $a(i, j)$). Then the components of the matrix partitioned into square blocks of order 3 are shown below:

$$A = \left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ \hline a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{array}\right] = \begin{array}{|c|c|} \hline a(block(1, 1, 3)) & a(block(1, 2, 3)) \\ \hline a(block(2, 1, 3)) & a(block(2, 2, 3)) \\ \hline \end{array}$$

More generally, let p be the number of processes available. Partitions can be formed by splitting each dimension of a matrix evenly (as above). For example, let p and N be such that the following expressions denote natural numbers.

$$\sqrt[2]{p} = \pi \qquad \sqrt[3]{p} = \psi \qquad \frac{N}{\sqrt[2]{p}} = \mu \qquad \frac{N}{\sqrt[3]{p}} = \nu$$

Then an $N \times N$ matrix may be partitioned as:

1. $block(k, l, \nu)$, where $k, l \in \{1 \ldots \psi\}$

2. $block(i,j,\mu)$, where $i,j \in \{1 \ldots \pi\}$

The matrix multiplication algorithm is taken from [22]. The product C may be generated by the composition of three supersteps $\mathcal{SLICE}$, $\mathcal{PRODUCT}$ and $\mathcal{SUM}$:

$$\mathit{SuperStepsMatrixProduct} = \mathcal{SLICE}; \mathcal{PRODUCT}; \mathcal{SUM}$$

Superstep $\mathcal{SLICE}$: Processes in this superstep are structured using a cubic grid - thus, an individual process is identified using a triple (i,j,k) with $1 \leq i,j,k \leq \psi$. The global data space is also organized as a tridimensional set of points and is partitioned into blocks of size ν. Program variables x and y are subscripted in order to distinguish data in separate processes. It is assumed that the definitions of §2 are extended (in an obvious way) to deal with block asynchronous communications.

$$\mathcal{SLICE} = \|^{ss}_{i,j,k \in \{1,\cdots,\psi\}} \; Slice_{i,j,k}$$

$Slice_{i,j,k}$ distributes sections of the matrices A and B so that data blocks $a(block(i,k,\nu))$ and $b(block(k,j,\nu))$ are assigned to the external blocks $x_{i,j,k}(block(i,k,\nu))$ and $y_{i,j,k}(block(k,j,\nu))$ respectively. The body of $Slice_{i,j,k}$ is:

$BodySlice_{i,j,k}$ ::
    $get(x_{i,j,k}(block(i,k,\nu)), a(block(i,k,\nu)))$;
    $get(y_{i,j,k}(block(k,j,\nu)), b(block(k,j,\nu)))$

Thus, we have:

$Slice_{i,j,k}$ ::
ext $x_{i,j,k}(block(i,k,\nu)), y_{i,j,k}(block(k,j,\nu))$
$\{m_{i,j,k} = \{\mapsto\}\}$
    $BodySlice_{i,j,k}$
$\{Post - Slice_{i,j,k}\}$

where $Post - Slice_{i,j,k}$ is

$$m_{i,j,k} = \{x_{i,j,k}(block(i,k,\nu)) \mapsto a(block(i,k,\nu))), y_{i,j,k}(block(k,j,\nu)) \mapsto b(block(k,j,\nu))\}$$

The correctness of process $Slice_{i,j,k}$ follows immediately from the definitions of §2. Note: the (subscripted) variables a and b remain unchanged throughout and thus denote their initial values $\overleftarrow{a}$ and $\overleftarrow{b}$, respectively. The correctness of the entire superstep is established below:

Lemma 2 *correctness of* $\mathcal{SLICE}$
$\{m = \{\mapsto\}\}\mathcal{SLICE}\{Post - \mathcal{SLICE}\}$
where $Post - \mathcal{SLICE} =$
$\forall i,j,k \in \{1 \ldots \psi\} : x_{i,j,k}(block(i,k,\nu)) = a(block(i,k,\nu)) \wedge y_{i,j,k}(block(k,j,\nu)) = b(block(k,j,\nu))$

Proof:

1. $\mathcal{SLICE} =\|^{ss}_{i,j,k\in\{1,\cdots,\psi\}} Slice_{i,j,k}$
   $= <$ by the definition of $\|^{ss} >$
   $\|^{I}_{i,j,k\in\{1,\cdots,\psi\}} Slice_{i,j,k};\ barrier$
2. $\{m = \{\mapsto\}\}$
   $\|^{I}_{i,j,k\in\{1,\cdots,\psi\}} Slice_{i,j,k}$
   $\{\forall i,j,k \in \{1\ldots\psi\} : Post - Slice_{i,j,k}$
   $<$ by the definition of $\|^{I}$ and the correctness of $Slice_{i,j,k} >$
   It remains to show that $\{\forall i,j,k \in \{1\ldots\psi\} : Post - Slice_{i,j,k}\} barrier \{Post - \mathcal{SLICE}\}$.
   This requirement reduces to:
   $(\forall i,j,k \in \{1\ldots\psi\} : Post - Slice_{i,j,k}) \Rightarrow {}^{\lambda z.z\in dom(m)}(Post - \mathcal{SLICE})_{\lambda z.m(z)}$.
3. $(\forall i,j,k \in \{1\ldots\psi\} : Post - Slice_{i,j,k}) \Rightarrow$
   $function(m)$
   $<$ by the definition of $Post - Slice_{i,j,k}$ above and the uniqueness of block decomposition $>$
4. Given $(\forall i,j,k \in \{1\ldots\psi\} : Post - Slice_{i,j,k})$ then
   ${}^{\lambda z.z\in dom(m)}(Post - \mathcal{SLICE})_{\lambda z.m(z)}$
   $= <$ by the definition of predicate substitution $>$
   $\forall i,j,k \in \{1\ldots\psi\} : a(block(i,k,\nu)) = a(block(i,k,\nu)) \wedge b(block(k,j,\nu)) = b(block(k,j,\nu))$
   $=$ true

This concludes the proof. □

Superstep $\mathcal{PRODUCT}$: This superstep uses the same data distribution as $\mathcal{SLICE}$:

$$\mathcal{PRODUCT} =\|^{ss}_{i,j,k\in\{1,\cdots,\psi\}} Product_{i,j,k}$$

Process $Product_{i,j,k}$ stores the result of the matrix block product $a(block(i,k,\nu))\cdot b(block(k,j,\nu))$ in the block $c(block(i,j,\nu))(k)$:

$BodyProduct_{i,j,k}$ ::
  for $r := lower(i,\nu)$ to $upper(i,\nu)$ do
    for $s := lower(j,\nu)$ to $upper(j,\nu)$ do
      $c(r,s)(k) := 0.0;$
      for $t := lower(k,\nu)$ to $upper(k,\nu)$ do
        $c(r,s)(k) := c(r,s)(k) + x_{i,j,k}(r,t) * y_{i,j,k}(t,s)$

$Product_{i,j,k}$ satisfies:

$Product_{i,j,k}$ ::
ext $x_{i,j,k}(block(i,k,\nu)), y_{i,j,k}(block(k,j,\nu)), c(block(i,j,\nu))(k)$
var r, s, t: integer;
$\{Pre - Product_{i,j,k} : x_{i,j,k}(block(i,k,\nu)) = a(block(i,k,\nu))$
$\qquad \wedge\ y_{i,j,k}(block(k,j,\nu)) = b(block(k,j,\nu))\ \}$
  $BodyProduct_{i,j,k}$
$\{Post - Product_{i,j,k} : m_{i,j,k} = \{\mapsto\}$
$\qquad \wedge\ c(block(i,j,\nu))(k) = a(block(i,k,\nu)) \cdot b(block(k,j,\nu))\ \}$

In this case no communication is performed since $m_{i,j,k} = \{\mapsto\}$ in $Post - Product_{i,j,k}$ and so $\|^{ss} = \|^{I}$. Thus, the second superstep satisfies:

$$\{Pre - \mathcal{PRODUCT}\}\mathcal{PRODUCT}\{Post - \mathcal{PRODUCT}\}$$

where:
$Pre - \mathcal{PRODUCT} = \forall i, j, k \in \{1, \cdots, \psi\} : Pre - Product_{i,j,k}$ and
$Post - \mathcal{PRODUCT} = \forall i, j, k \in \{1, \cdots, \psi\} : Post - Product_{i,j,k}$
Superstep $\mathcal{SUM}$: This superstep collapses, by aggregation through addition, the third dimension of the tridimensional array c to produce C. The precondition of this operation ensures that:

$$c(block(i, j, \nu))(k) = a(block(i, k, \nu)) \cdot b(block(k, j, \nu))$$

Thus,
$\sum_{k=1..\psi} c(block(i, j, \nu))(k)$
$= <$ from precondition $>$
$\sum_{k=1..\psi} a(block(i, k, \nu)) \cdot b(block(k, j, \nu))$
$= <$ by matrix algebra $>$
$a(range(i, \psi), 1 \ldots N) \cdot b(1 \ldots N, range(j, \psi))$
$= <$ by definition of matrix multiplication $>$
$C(block(i, j, \nu)) <$ as required $>$
For this superstep a different process structure and data partitioning are adopted.

$$\mathcal{SUM} = \|^{ss}_{u,v \in \{1, \cdots, \pi\}} Sum_{u,v}$$

$BodySum_{u,v}$ is defined as:

$BodySum_{u,v}$ ::
  for $r := 2$ to ψ do
    for $s := lower(u, \mu)$ to $upper(u, \mu)$ do
      for $t := lower(v, \mu)$ to $upper(v, \mu)$ do
        $c(s, t)(1) := c(s, t)(1) + c(s, t)(r)$

with $Sum_{u,v}$ satisfying:

$Sum_{u,v}$ ::
ext $c(block(u, v, \mu))(1 \ldots \psi)$
var r, s, t: integer;
$\{Pre - Sum_{u,v} : \forall k \in \{1 \ldots \psi\} : c(block(u, v, \mu))(k) = a(block(i, k, \psi) \cdot b(block(k, j, \psi))\}$
  $BodySum_{u,v}$
$\{Post - Sum_{u,v} : m_{u,v} = \{\mapsto\} \wedge$
  $c(block(u, v, \mu))(1) = a(range(i, \mu), 1 \ldots N) \cdot b(1 \ldots N, range(j, \mu))\}$

Again, the superstep semantics degenerates to independent process composition. Thus,

$$\{Post - \mathcal{PRODUCT}\}\mathcal{SUM}\{Post - \mathcal{SUM}\}$$

where
$Post - \mathcal{SUM} =$
$\forall u, v \in \{1 \ldots \pi, 1 \ldots \pi\} : Post - Sum_{u,v}$
$= <$ by algebra $>$
$\forall l, m \in \{1, \ldots N\} : c(l,m)(1) = \sum_{h=1}^{N} a(l,h) \cdot b(h,m)$
The correctness of the (sequential) composition of the three supersteps follows in the conventional way.

$$\{m = \{\mapsto\}\} SuperStepsMatrixProduct \{Post - \mathcal{SUM}\}$$

The first two supersteps employ the same partition of global memory. Thus, no operational reorganisation is needed when composing them. Between the second and third supersteps, however, an implicit data redistribution is performed. Although such a repartition may be operationally complex it has no semantic implications: (global) state properties are unaffected by such an operation.

4 BSP Computations as MultiParty Synchronous Computation

In the earlier discussion of BSP computation single supersteps were isolated and then composed using sequential program operators. BSP programs can also be expressed as the parallel composition of "process" threads - see, for example, the BSP library routines [22, 14]. In this approach processes (threads) are given whose parallel composition defines the entire program. Threads are similar to processes in [9] which interact through multiparty synchronous communication.

The command, *syn*, is used within processes to specify (synchronisation) barriers. For example, a mutiple superstep thread program may be represented as:

$$\|^{T}_{i \in \{1,\ldots,p\}} T_i$$

where $\|^T$ denotes a composition operator for *threads* which may contain multiple interference points (unlike $\|^{ss}$ which includes a single terminal interference point). For example, each $T_i, 1 \leq i \leq p$, may be defined as follows:

$T_i :: S1_i;\ syn;\ S2_i;\ syn;\ S3_i;\ syn$

where $S1_i$, $S2_i$ and $S3_i$ denote computations which may involve asynchronous communications. The relationship between the two approaches to the organisation of BSP computations for the simple case of an explicit sequence of supersteps is given by:

$$(\ \underset{i \in J}{\|^{ss}}\ T1_i); (\ \underset{i \in J}{\|^{ss}}\ T2_i) = \|^{t}_{i \in J} (T1_i; syn; T2_i)$$

The version on the left

$$(\ \underset{i \in J}{\|^{ss}}\ T1_i); (\ \underset{i \in J}{\|^{ss}}\ T2_i)$$

benefits from explicitly aligning the process barriers. However, an implicit data redistribution may be required if corresponding external data declarations in $T1_i$ and $T2_i$ are not identical - as in the matrix multiplication example presented earlier. The thread-based approach necessitates that all redistributions are *explicitly* specified. This may afford the programmer the opportunity to define repartitions in a efficient way.

Notation: let $v^{(i)}$ denote process i's instance of the variable v. Thus, if $i \neq j$ then $v^{(i)}$ and $v^{(j)}$ are distinct variables. Consequently, the data spaces of any two distinct processes are disjoint. It is necessary to extend the formal rules presented in §2 in order to reason about thread programs. In particular, corresponding *syn* statements must be matched. During execution of thread i let a local auxiliary variable $counter^{(i)}$ record the number of synchronisation points encountered so far. A *syn* command ensures that any asynchronous messages in transit are delivered to their pre-specified destinations. For individual processes, correctness arguments can be facilitated by making local assumptions about external communications in a way similar to that employed in the axiomatisation of CSP [1, 9, 21]. These assumptions may be discharged later by means of a composition rule. Thus, the effect of a local *syn* operation in a thread is captured by:

$$\{P\}syn\{Q \wedge m = \{\mapsto\}\} \tag{15}$$

where P is guaranteed by the thread and Q is an assumption about the global state immediately after the synchronisation barrier has been passed. Note again the requirement that the communication buffer (represented by the map m) be empty after synchronisation. It is also necessary to update the auxiliary variable *counter* at each *barrier* operation. This may be achieved by including the auxiliary assignment "$counter^{(i)} := counter^{(i)} + 1$" immediately prior to every *syn* operation in thread i. Let each *syn* operation be identified with the virtual operations:

$$syn = (counter := counter + 1; \overline{syn}) \tag{16}$$

where $\overline{syn}$ is a virtual synchronisation operation, $\overleftarrow{counter} = 0$ and the virtual variable *counter* does not appear in any other statements.

Annotated processes $S_i, i \in I$ $co-operate$ at a designated synchronisation point if they exchange information as in the *barrier* transformation (13):

$co-operate(S_i, i \in I) \equiv$

for every combination of local $\overline{syn}$ transformations

involving exactly one $\overline{syn}$ operation from each of the threads in I

$\{pre_i\}\overline{syn}\{post_i\}, i \in I:$

$\{\exists t \in N^+ : (\forall i \in I : pre_i \wedge counter^{(i)} = t)\} \ \|_{i\in I}^T \ \overline{syn}\{\forall i \in I : post_i\}$

$Co-operation$ ensures that only semantically (operationally) *matching* synchronisation points need to be addressed. For non-matching synchronisations (i.e. $counter^{(i)} \neq counter^{(j)}$) the required co-operation condition degenerates to a vacuously true assertion $\{false\}S\{q\}$. In the case of a matching synchronisation we have:

$$\|_{i\in I}^T \ \overline{syn} = barrier \tag{17}$$

Thus, satisfaction of the co-operation requirement reduces to the correctness of a *barrier* transformation. Under the assumption that local proofs co-operate (as above) the overall system behaviour can be extracted:

$$\frac{(\forall i \in I : \{P_i\} S_i \{Q_i\}), co-operate(S_i, i \in I)}{\{(\forall i \in I : P_i)\} \parallel_{i \in I}^{T} S_i \{(\forall i \in I : Q_i)\}} \tag{18}$$

5 MultiParty Matrix Multiplication

The matrix multiplication example is re-expressed below within a process-style BSP framework. Here, it is necessary to re-structure the supersteps in such a way that each process is allocated a static portion of memory (unlike the state transformation framework in which consecutive supersteps may have a different internal structure and data partition). The evaluation of the matrix product is specified by the parallel composition:

$$\mathit{MultiPartyMatrixProduct} = \parallel_{l \in \{1,\ldots,p\}}^{T} ThreadMatrixProduct_l$$

In order to clarify the presentation the following two space transformation functions are introduced. The function

$$1D3D : (1 \ldots p) \rightarrow (1 \ldots \sqrt[3]{p}) \times (1 \ldots \sqrt[3]{p}) \times (1 \ldots \sqrt[3]{p})$$

converts an index in the range $1 \ldots p$ to a point in 3 dimensional Cartesian space. The x, y and z components of a triple denoting a point in 3-space can be extracted using the component functions $._i$, $._j$ and $._k$, respectively.

For a 2 dimensional space the function:

$$1D2D : (1 \ldots p) \rightarrow (1 \ldots \sqrt[2]{p}) \times (1 \ldots \sqrt[2]{p})$$

is used. The x and y components of a pair denoting a point in 2-space can be extracted using the component functions $._u$, and $._v$, respectively.
An annotated version of the thread $ThreadMatrixProduct_l$ is:

$ThreadMatrixProduct_l$ ::
let $i = 1D3D(l)._i, j = 1D3D(l)._j, k = 1D3D(l)._k,$
$\quad u = 1D2D(l)._u, v = 1D2D(l)._v$ in
ext $x_{i,j,k}(block(i,k,\nu)), y_{i,j,k}(block(k,j,\nu)), c(block(i,j,\nu))(k), d(block(u,v,\mu))(1 \ldots \psi)$
var r, s, t: integer;
$\quad \{\overleftarrow{m_l} = \{\mapsto\} \wedge counter^{(l)} = 0\}$
$\quad BodySlice_{i,j,k};$
$\quad \{\mathbf{A1} : Post-Slice_{i,j,k} \wedge counter^{(l)} = 0\}$
$\quad (counter^{(l)} := counter^{(l)} + 1;\ \ \overline{syn})$
$\quad \{\mathbf{A2} : Pre-Product_{i,j,k} \wedge counter^{(l)} = 1 \wedge m_l = \{\mapsto\}\}$
$\quad BlockProduct_{i,j,k};$
$\quad put(d(block(i,j,\nu)(k), c(block(i,j,\nu)(k));$
$\quad \{\mathbf{A3} : Post-Product_{i,j,k} \wedge \{m_l = d(block(i,j,\nu)(k) \mapsto c(block(i,j,\nu)(k)\}$
$\quad\quad \wedge\ counter^{(l)} = 1\}$
$\quad (counter^{(l)} := counter^{(l)} + 1;\ \ \overline{syn}\ )$

$\{\mathbf{A4}: d(block(i,j,\mu))(k) = a(block(i,k,\mu) \cdot b(block(k,j,\mu))$
$\quad \wedge\ m^{(l)} = \{\mapsto\} \wedge counter^{(l)} = 2\}$
$BodySum_{u,v}[c \leftarrow d];$
$\{m = \{\mapsto\} \wedge d(block(i,j,\mu))(1) = a(range(i,\mu), 1 \ldots N) \cdot b(1 \ldots N, range(k,\mu))\}$

In the implementation of the algorithm in §3 the array c is repartitioned between the second and third supersteps. In this version two distinctly partitioned arrays (c and d) are used to store the same information at the start of the third superstep. An asynchronous communication is performed at the second *barrier* to send information from c to d. In this way a repartition of the information stored in c is achieved. Note: it may be the case that an asynchronous transfer of information from one block structure to another results in "self-addressed" messages - that is, the same thread may be allocated portions of both the sending array and the receiving array. In such circumstances the technical details need to be slightly modified - see [28] a treatment of self-addressed messages. The co-operation of the threads is established as a first step in the correctness proof of the matrix multiplication algorithm.

Lemma 3 *matrix multiplication co-operation*
$co-operate(ThreadMatrixProduct_l, 1 \leq l \leq p)$.

Proof:
Each annotated thread includes two textual occurrences of $\overline{syn}$:

- $\{A1\}counter^{(l)} := counter^{(l)} + 1;\ \overline{syn}\{A2\}$
- $\{A3\}counter^{(l)} := counter^{(l)} + 1;\ \overline{syn}\{A4\}$

These may be reduced to $\overline{syn}$ transformations, using the axiom of assignment, as follows:

- $\{A1' : Post-Slice_{i,j,k} \wedge counter^{(l)} = 1\}\overline{syn}\{A2\}$
- $\{A3' : Post-Product_{i,j,k} \wedge \{m_l = d(block(i,j,\nu)(k) \mapsto c(block(i,j,\nu)(k)\} \wedge counter^{(l)} = 1\}$
  $\overline{syn}$
  $\{A4\}$

Let
$i = 1D3D(l)._i$, $j = 1D3D(l)._j$, $k = 1D3D(l)._k$, $u = 1D2D(l)._u$ and $v = 1D2D(l)._v$.
Compositions of the form $\|_{l\in 1..p}^{T}\ \overline{syn}$ involving a single $\overline{syn}$ operation from each of the threads may be realised in three different ways:

1. All of the individual thread synchronisations have the form: $\{A1'\}\overline{syn}\{A2\}$
   In this case the co-operation test becomes:

   $\{\exists t \in N^+ : (\forall l : 1 \leq l \leq p : A1'^{(l)} \wedge counter^{(l)} = t)\}\ \|_{l\in\{1\ldots p\}}^{T}\ \overline{syn}\{\forall l : 1 \leq l \leq p : A2^{(l)}\}$
   $= <$ by expansion and simplification $>$
   $\{\exists t \in N^+ : \forall l : 1 \leq l \leq p : counter^{(l)} = 1 \wedge counter^{(l)} = t \wedge Post-Slice_{i,j,k}\}$
   barrier
   $\{\forall l : 1 \leq l \leq p : A2^{(l)}\}$
   $= <$ let $t = 1 >$
   $\{\forall l : 1 \leq l \leq p : counter^{(i)} = 1 \wedge Post-Slice_{i,j,k}\}$

$barrier$
$\{\forall l : 1 \leq l \leq p : A2^{(l)}\}$
The correctness of the last triple follows from Lemma 2.

2. All of the thread synchronisations have the form: $\{A3'\}\overline{syn}\{A4\}$.
   In this case the co-operation test becomes:
   $\{\exists t \in N^+ : (\forall l : 1 \leq l \leq p : A3'^{(l)} \wedge counter^{(l)} = t)\}barrier\{\forall l : 1 \leq l \leq p : A4^{(l)}\}$
   $= <$ by expansion, simplification and letting $t = 2 >$
   $\{(\forall l : 1 \leq l \leq p :$
   $Post - Product_{i,j,k} \wedge \{m_l = d(block(i,j,\nu)(k) \mapsto c(block(i,j,\nu)(k)\} \wedge counter^{(l)} = 2\}$
   $barrier$
   $\{\forall l : 1 \leq l \leq p : A4^{(l)}\}$
   Proof of the correctness of the last triple is straightforward and is left as an exercise.

3. The thread synchronisations have neither of the forms above. In such cases the co-operation triple holds vacuously since at least two threads will have counters set to different values.

This concludes the proof. □

6 Related Work and Summary

The models developed in this paper have similarities to that of He et al. [14] - for example, the construction of communication bundles (m). Some distinguishing features of the models stem from the starting points adopted. He's system is a modification of a trace model - usually employed in developing a semantics for reactive systems. The approach taken here is to develop a model of BSP grounded on sequential and data-parallel computation. Thus, global assertions can be established at synchronisation points and *barrier* is considered to be a generalisation of data parallel assignment. This necessitates the introduction of *predicate substitution*.

Skillicorn [24] uses a different approach to the axiomatic definition of BSP-style computation. In particular, the scope of assertions is extended to include variable/process distribution information (*location predicates*). Thus, a statement which alters a location predicate, over, say, x, corresponds, operationally, to the "communication" of x between two processes (albeit a communication in which the sender no longer has direct access to x). This approach avoids the need for *predicate substitution* but requires the use of more complex assertions.

The development of the thread style semantics involves the matching of synchronisation points through the use of local counters. Only a partial semantics has been considered - see [29] for a treatment of deadlock-free conditions.

Predicate substitution provides a conceptual basis for reasoning about systems which resolve sets of ongoing asynchronous communications at synchronising barriers. Thus, it furnishes a semantic basis for BSP-type type programs and generalises previous definitions of substitution. Work on the soundness and completeness of *predicate* substitution is in progress [11]. The barrier operation can be embedded within either a state-transformation based or thread-system based frameworks and, together with global synchronisation, provides a bridge between the two approaches.

References

[1] Apt K. R., Francez N., de Roever W. P.: A proof system for communicating sequential processes, ACM Trans. Programming Languages Systems,. **2** (3) (1980) 359–385.

[2] Bouge L., Cachera D., Le Guyadec Y., Utard G., Virot B.: Formal validation of data-parallel programs: a two-component assertional proof system for a simple language, Theoretical Computer Science **189** (1997) 71–107.

[3] Clint M.: Program proving: coroutines, Acta Informatica, **2** (1) (1973) 50–63.

[4] Clint M., Narayana K. T.: Programming structures for synchronous parallelism, Parallel Computing 83, eds: F. Feilmeier, J. Joubert, U. Schendel, North-Holland, pp 405–412, 1984.

[5] Dahl O.-J.: Verifiable Programming, Prentice-Hall International, 1992.

[6] Dijkstra E. W.: A Discipline of Programming, Prentice-Hall, 1976.

[7] Dijkstra E. W.: Guarded commands, nondeterminacy and formal derivation of programs, Communications ACM, **18** (1975) 453–457.

[8] Elrad T., Francez N.: Decomposition of distributed programs into communication-closed layers, Science of Computer Programming, **2** (1982) 155–173.

[9] Francez N.: Program Verification, Addison-Wesley, 1992.

[10] Gabarro J., Gavalda R.: An approach to correctness of data parallel algorithms, Journal of Parallel and Distributed Computing, **22** (1994) 185–201.

[11] Gabarro J., Clint M., Stewart A.: A sound and complete axiomatic system for reasoning about BSP-type programs (draft manuscript).

[12] Gerbessiotis A. V., Valiant L. G.: Direct bulk-synchronous parallel algorithms, Journal of Parallel and Distributed Computing, **22** (1994) 251–267.

[13] Gries D.: The Science of Programming, Springer-Verlag, 1981.

[14] He, J., Miller, Q., Chen, L.: Algebraic laws for BSP programming, in Euro-Par '96, LNCS 1124, Springer-Verlag, edts: L. Bouge, P. Fraigniaud, A. Mignotte and Y. Robert (1996) 359–368.

[15] Hoare C. A. R.: An axiomatic basis for computer programming, Comm. ACM, **12** (10) (1969) 576–580.

[16] Hoare C. A. R.: Communicating Sequential Processes, Prentice Hall, 1985.

[17] Hoare C. A. R., Jefeng H.: Unifying theories for parallel programming, in Euro-Par '97, LNCS 1300, Springer-Verlag, edts: C. Lengauer, M. Griebl and S. Gorlatch (1997) 3–14.

[18] Hoare C. A. R., Jefeng H.: Unifying theories of programming, Prentice Hall, 1998.

[19] Jones C. B.: Tentative steps towards a development method for interfering programs, ACM Trans. Programming Languages Systems,. **5** (4) (1983) 596–619.

[20] Jones C. B.: Systematic Software Development Using VDM (2nd edn.), Prentice Hall, 1990.

[21] Levin G., Gries D.: Proof techniques for communicating sequential processes, Acta Informatica, **15** (1981) 281–302.

[22] McColl W.F.: Scalable computing, in Computer Science Today: Recent Trends and Developments LNCS Vol. 1000, Springer-Verlag, edt: J. van Leeuwen (1995) 46–61.

[23] Owicki S., Gries D.: An axiomatic proof technique for parallel pograms, Acta Informatica, **6** (1976) 319–340.

[24] Skillicorn, D.B.: Building BSP programs using the refinement calculus, Technical Report 96-400, Department of Computing and Information Science, Queen's University Kingston, 1996.

[25] Stewart A.: An axiomatic treatment of SIMD assignment, BIT, **30** (1990) 70–82.

[26] Stewart A.: Reasoning about data-parallel array assignment, Journal of Parallel and Distributed Computing, **27** (1) (1995) 79–85.

[27] Stewart A., M. Clint: Synchronising asynchronous communications, in Euro-Par '97, LNCS 1300, Springer-Verlag, edts: C. Lengauer, M. Griebl and S. Gorlatch (1997) 511–520.

[28] Stewart A., M. Clint: BSP: a semantic investigation, submitted to The Computer Journal, 1999.

[29] Utard G., Hains G.: Deadlock-free absorption of barrier synchronisations, Information Processing Letters, **56** (4) (1995)221–228.

[30] Valiant L. G.: A bridging model for parallel computation, Comm. ACM, **33** (8) (1990) 103–111.

A Appendix: Mixed State Evaluation

Consider the operation $get(x, e)$ where e denotes an expression which may refer to array elements. It may be defined as follows:

$$get(x, e) \equiv m := m \cup \{x \to \hat{e}\} \tag{19}$$

Here, $\widehat{e}$ denotes a mixed state evaluation of the expression and can be defined inductively over expression structures. Two of the more interesting parts of such a definition are:

$$\widehat{y} = \overrightarrow{y} \tag{20}$$

$$\widehat{a(e)} = \overrightarrow{a}(e) \tag{21}$$

where y is a scalar variable and a is an array. Note: the reference $a(\widehat{b(i)})$ is only meaningful if the process context has external access to $b(i)$.