Towards refining BSP programs into loosely coupled distributed systems *

A. Stewart[1] and M. Clint[1] and J. Gabarró[2] and M.J. Serna[2]

[1] Department of Computer Science, The Queen's University of Belfast
[2] Dep. LSI, Universitat Politècnica de Catalunya, Barcelona

Abstract. An experiment is conducted to assess the feasibility of transforming BSP computations into loosely coupled distributed systems employing asynchronous point-to-point communications. The BSP model bundles communications together before performing simultaneous message delivery at a synchronisation point, whereas the loosely coupled framework manages each message in isolation. The potential benefits of the proposed transformation are:

1. a safe development framework for loosely coupled systems; the transformational approach preserves semantics and ensures that the resulting system has matching deadlock-free communications; and
2. the BSP synchronisation mechanism is relaxed into a series of point-to-point communications; in certain circumstances (*visibility*) the transformed program may be more efficient than the original as a result of the removal of synchronisation barriers.

Keywords: Parallel programming: models and methods, BSP, message passing, asynchronous communication, point-to-point communications, refinement rules, normal forms.

1 Introduction

The construction of reliable distributed software requires that complex correctness issues be addressed: typically, individual processes are assembled with assumptions being made about their operating environments. An alternative approach to developing parallel software is to use the BSP (bulk synchronous parallel) framework [15, 6]. The basic computational unit in BSP is a *superstep*; a superstep involves initial and terminal synchronisations over all processes. Thus, a superstep can be treated semantically as a global state transformation [11, 12, 14] in a way akin to reasoning about sequential software.

A potential disadvantage of the BSP model is that a gobal synchronisation operator may be relatively expensive to realise on certain kinds of architecture. For example, global synchronisation may be prohibitively expensive to implement on distributed systems communicating over local area networks or on the *grid* [5]. Here it is proposed that the BSP framework be used to develop parallel programs whose correctness can be readily established [12, 14] [1]. Then transformation rules can be applied to permit, in certain circumstances, the replacement of synchronisation barriers by a set of asynchronous point-to-point communications leading, ideally, to an improvement in efficiency. To this end a number of semantic-preserving transformation rules are developed. The point-to-point communications may be MPI-like operations, such as *send* and *receive*, or lower level instructions. Note, however, that the removal of synchronisation barriers means that the BSP cost calculus may no longer be used to predict execution times [15].

In §2 a semantic model of BSP [11–14] is summarised. An alternative semantic model - that of a loosely coupled system employing asynchronous MPI-like communications - is given in §3. A transition system which maps BSP supersteps to semantically equivalent distributed systems is defined in §4.

* Partially supported by (i) The British Council and Spanish Ministry of Education ACI HB1999-0093, (ii) Spanish CICYT under grant TIC1999-0754-C03-02 (Mallba) and (iii) the IST programme of EU under contract number IST-1999-14186 (ALCOM-FT).

[1] The idea of executing synchronous programs on asynchronous hardware (by means of *synchronizers*) is well known [3].

2 A Programming Model for BSP

In this section the basic properties of a BSP programming model are summarised. The notation used is based on the Unifying Theories of Programming [8]. Communications are treated as buffer updates and, as such, they obey assignment laws. Superstep composition can be expressed in terms of parallel independent composition and a *barrier* operation which delivers buffered messages to their destinations by means of a special form of simultaneous assignment. The proposed model is consistent with Valiant's original definition of BSP [15].
Let S, S_1 etc. denote superstep processes; $\alpha(S)$, $\alpha_R(S)$, $\alpha_W(S)$ denote respectively the set of free variables, the set of free read-only variables and the set of free read-write variables in S; x, y etc. denote simple variables; a,b etc. denote array variables; and $\|^{ss}$ denote the superstep composition operator. The superstep $S_1 \|^{ss} S_2$ is valid provided that its constituent processes are disjoint:

$$disjoint(S_1, S_2) \quad =_{df} \quad (\alpha_W(S_1) \cap \alpha(S_2) = \{\}) \wedge (\alpha_W(S_2) \cap \alpha(S_1) = \{\}) \qquad (1)$$

Thus, distinct superstep processes may have read only variables in common. For example, in the computation

$$\textbf{for } k := 1 \textbf{ to } n \textbf{ do } (S_1 \|^{ss} S_2)$$

we may have $\{k, n\} \subseteq \alpha_R(S_1) \wedge \{k, n\} \subseteq \alpha_R(S_2)$. A superstep process is constructed using sequential programming operators and two asynchronous communication instructions *put* and *get*: these denote, respectively, a future assignment of the value of a local expression to a non-local destination and a request for non-local information.

The current state of ongoing communications is recorded by an auxiliary variable m, a relation from variable names (message destinations) either to values or (unevaluated) variable names (message content).

$$m : VAR \mapsto (VAL \cup VAR)$$

where VAR denotes the space of variable addresses and VAL denotes the space of values. The relation m is used to record all communications that are invoked in a superstep. Inside the composition $S_1 \|^{ss} S_2$, the data spaces of S_1 and S_2 are augmented by local message relations, $1.m$ and $2.m$ respectively. A message relation m may be modified by the execution of *get* and *put* instructions as follows:

$$\begin{aligned} put(x, e) \quad &=_{df} \quad m := m \cup \{``x" \mapsto e\} \text{ with } \alpha(put(x, e)) = \alpha(e) \\ get(x, z) \quad &=_{df} \quad m := m \cup \{``x" \mapsto ``z"\} \text{ with } \alpha(get(x, z)) = \{x\} \end{aligned}$$

where e denotes an expression and "z" denotes the syntactic string z rather than the value denoted by z. The definitions for *put* and *get* can be extended to realise conditional communications. For example:

$$\begin{aligned} put(x, e) \lhd b \rhd \Pi \quad &=_{df} \quad m := (m \cup \{``x" \mapsto e\}) \lhd b \rhd m \\ get(x, z) \lhd b \rhd \Pi \quad &=_{df} \quad m := (m \cup \{``x" \mapsto ``z"\}) \lhd b \rhd m \end{aligned}$$

where Π is the empty statement (SKIP) and the conditional statement $S_1 \lhd b \rhd S_2$ satisfies $S_1 \lhd true \rhd S_2 \;=\; S_1$ and $S_1 \lhd b \rhd S_2 \;=\; S_2 \lhd \neg b \rhd S_1$.

In many practical scientific applications, it is often necessary to communicate parts of arrays. Array based communication statements are more complex because they require the evaluation of index expressions. Consider the instruction $get(x, a[e])$. Here, $a[e]$ is assumed to be a non-local array reference (to be evaluated in a later state) whereas e is assumed to be a local index expression to be evaluated in the current state. Then,

$$\begin{aligned} put(a[e_1], e_2) \quad &=_{df} \quad m := m \cup \{``a"[e_1] \mapsto e_2\} \\ get(z, a[e]) \quad &=_{df} \quad m := m \cup \{``z" \mapsto ``a"[e]\} \\ get(a_1[e_1], a_2[e_2]) \quad &=_{df} \quad m := m \cup \{``a_1"[e_1] \mapsto ``a_2"[e_2]\} \end{aligned}$$

The characteristic feature of a superstep is the merging of individual process computations. Assume that processes S_1 and S_2 are such that all asynchronous communication statements (*puts* and *gets*) have been replaced by their definitions (updates of m). Then,

$$\begin{array}{rl} S_1 \,\|^{ss}\, S_2 & =_{df} \text{ var } \ 1.m, 2.m := \{\mapsto\}, \{\mapsto\} \\ & \qquad\qquad ((S_1)^m_{1.m} \,\|^I\, (S_2)^m_{2.m}) \ ; \ barrier(1.m \cup 2.m) \\ & \qquad \text{end } \ 1.m, 2.m \end{array}$$

Here, $S^m_{1.m}$ denotes S with all free occurrences of m replaced by $1.m$ (i.e. a local copy of m, $1.m$, is made within S_1), $\|^I$ denotes the independent parallel composition operator and $barrier(m)$ denotes a merge operation over the message space m. The merge operation $barrier$ is defined as a multiple conditional assignment:

$$barrier(m) \quad =_{df} \quad \forall \text{``}z\text{''} \in dom(m).\ \text{``}z\text{''} := \sqcap m(z)$$

where $m = 1.m \cup 2.m$. Here, $\sqcap$ denotes non-deterministic selection.

A sequential composition of parallel programs (supersteps) may be rewritten as a parallel composition of threads provided that composite threads have disjoint alphabets [14]. Let $\|^T$ be a thread composition operator and syn be a local (single process) synchronisation statement:

$$(S_1 \,\|^{ss}\, S_2) \quad =_{df} \quad \left(\begin{array}{l} (\text{var } 1.m, 2.m \\ \\ \qquad \begin{pmatrix} 1.m := \{\mapsto\}; \\ (S_1)^m_{1.m}; \\ syn(1.m \cup 2.m) \end{pmatrix} \|^T \begin{pmatrix} 2.m := \{\mapsto\}; \\ (S_2)^m_{2.m}; \\ syn(1.m \cup 2.m) \end{pmatrix} \\ \\ \text{end } 1.m, 2.m \end{array} \right)$$

For non-interfering statements $\|^T$ reduces to independent composition where S_1 and S_2 do not contain syn statements; $barrier$ and syn are related, for this example, as follows:

$$barrier(1.m \cup 2.m) \quad = \quad syn(1.m \cup 2.m) \ \|^T \ syn(1.m \cup 2.m)$$

Example. Given an array $a[0\ ..\ N-1]$ such that $N = 2^l$ a prefix sum calculation computes the partial sums $prefix[r] = \sum_{0 \le i \le r} a[i]$ for $0 \le r < N$. A superstep based program for prefix sum can be derived by modifiying a well known PRAM program [1]:

Prefix_Sum $\equiv$

$\quad$ **var** $to\_add[0 \ldots N-1]$

$\quad \|^I_{0 \le r < N}\ prefix[r] = a[r];$

$\quad$ **for** $k := 0$ **to** $\log_2 N - 1$ **do**

$\qquad \|^{ss}_{0 \le r < N}(\quad put(to\_add[r+2^k], prefix[r]) \lhd r \le N - 2^k - 1 \rhd \mathit{\Pi}\ );$

$\qquad \|^I_{0 \le r < N} \quad (\ prefix[r] := prefix[r] + to\_add[r] \lhd r \ge 2^k \rhd \mathit{\Pi}\ )$

$\quad$ **end** end to_add

where the alphabet of the rth superstep process is $\{prefix[r], to\_add[r], a[r]\}$. The algorithm can also be expressed as a thread composition:

$$Thread\_Sum \equiv \ \|^T_{0 \le r < N}\ Thread\_Prefix\_Sum_r$$

where each thread has a local loop and a local loop counter:

$Thread\_Prefix\_Sum_r \equiv$

$\qquad$ **var** $r.k, to\_add[0 \ldots N-1];$

$\qquad prefix[r] = a[r];$

for $r.k := 0$ **to** $\log_2 N - 1$ **do**
$\quad put((to\_add[r + 2^{r.k}], prefix[r]) \;\lhd\; r \leq N - 2^{r.k} - 1 \;\rhd\; \Pi;$
$\quad syn\;;$
$\quad (prefix[r] := prefix[r] + to\_add[r]) \;\lhd\; r \geq 2^{r.k} \;\rhd \Pi;$
end
end $r.k, to\_add$

3 Point-to-Point communications

An alternative to grouping together all messages in a superstep is explicitly to manage individual communications through matching pairs of *send* and *receive* instructions. Message passing may again be realised through buffered communications; each process, i, has a local buffer M_i which is used to receive incoming messages. In order to simplify the semantics of point-to-point communications M_i is treated as a sequence of local buffers:

$$M_i = \cup_{0 \leq j < P} M_{i,j}$$

where P is the number of processes in the system. The semantics of *send* is closely related to the definitions of *get* and *put* - relevant communication information is written to an appropriate buffer. The instruction $send(e, tag, i)$ denotes a communication identified by a unique tag, tag, which results in the value e being delivered to process i [2]. Let j denote the index of the sender process. Then:

$$send(e, tag, i) \quad =_{df} \quad M_{i,j} := M_{i,j} \cup \{tag \mapsto e\} \tag{2}$$

where $tag \notin dom(M_{i,j})$. The instruction $receive(x, tag, j)$ retrieves a message identified by label tag from process j and stores the contents in variable x:

$$\begin{aligned} receive(x, tag, j) \quad =_{df} \quad & \textbf{await } tag \in dom(M_{i,j}) \rightarrow \\ & \quad x := M_{i,j}(tag); \\ & \quad M_{i,j} := \{tag\} \ominus M_{i,j} \\ & \textbf{end} \end{aligned} \tag{3}$$

where the **await** instruction is as given in Owicki & Gries [9]. The symbol $\ominus$ denotes domain deletion defined by:

$$\begin{aligned} &\{a\} \ominus \{\mapsto\} = \{\mapsto\} \\ &\{a\} \ominus (\{a \mapsto b\} \cup M) = \{a\} \ominus M \\ &\{a\} \ominus (\{b \mapsto c\} \cup M) = \{b \mapsto c\} \cup (\{a\} \ominus M),\; a \neq b \end{aligned}$$

The **await** clause excludes the possibility of interference since no two processes can have buffer write conflicts; the uniqueness of *tag* ensures that non-matching *send* and *receive* instructions cannot update the same buffer contents. A *send* instruction involves a remote assignment whereas a *receive* involves only local information. This execution model closely resembles the standard point-to-point communication of MPI [10].

Example. An informal transformation of the thread version of prefix sum, $Thread\_Sum$, into a semantically equivalent system employing point-to-point communications is performed below. The essence of the transformation is the replacement of each *put* instruction (and the implicit delivery operation *barrier*) by a matching pair of *send* and *receive* communications. Thus the instruction:

[2] Note: This definition requires that every process i has a set of local buffers $M_{i,j}$, one for each possible sender process. A single buffer implementation may be realised by prefixing a message identification tag with the sender's process number. The partitioned definition above is used in order to clarify interference issues.

$put(to\_add[r + 2^{r.k}], prefix[r]) \;\lhd\; r \le N - 2^{r.k} - 1 \;\rhd\; \Pi$

is replaced by:

$send(prefix[r], r.tag, r + 2^{r.k}) \;\lhd\; r \le N - 2^{r.k} - 1 \;\rhd\; \Pi$

where *r.tag* is assumed to be a unique identification label. In addition, a matching *receive* instruction is introduced to take the place of *barrier*:

$receive(to\_add[r], (r - 2^{k.r}).tag, r - 2^{k.r}) \;\lhd\; r \ge 2^{r.k} \;\rhd\; \Pi$

The following MPI-like process results:

$Prefix\_Sum_r \equiv$
  var $r.k, M_{r,1}, \ldots, M_{r,N-1} \;:=\; 0, \{\mapsto\}, \ldots, \{\mapsto\}$;
  for $r.k := 0$ **to** $\log_2 N - 1$ **do**
    $send(prefix[r], r.tag, r + 2^{r.k}) \;\lhd\; r \le N - 2^{r.k} - 1 \;\rhd\; \Pi;$
    $receive(to\_add[r], (r - 2^{k.r}).tag, r - 2^{k.r}) \;\lhd\; r \ge 2^{r.k} \;\rhd\; \Pi;$
    $prefix[r] := prefix[r] + r.to\_add \;\lhd\; r \ge 2^{r.k} \;\rhd\; \Pi;$
  end
  end $r.k, M_{r,1}, \ldots, M_{r,N-1}$

and $||^T_{0 \le r < N}\ Prefix\_Sum_r$ denotes the transformed system. □

In the following sections semantics preserving transformation rules for rewriting BSP programs as point-to-point communicating systems are developed and conditions under which the rewrite rules may usefully be employed are determined.

4 From *barrier* to point-to-point communications

In this section a formal transformation method is developed that replaces BSP-style message generation instructions and the operation *barrier* with a set of explicit point-to-point communications. Only three types (sub-classes) of parallel programs are considered: *straight line* processes (which serve to introduce the basic transformations), processes with *visible* communication guards (*visibility* defines conditions under which the transformations are likely to produce efficient programs) and *generic* processes.

4.1 Straight line programs

A process is *straight line* if it contains only assignments (no conditional or iterative statements). Consider $S_1 \;||^{ss}\; S_2$ where all S_i are straight line and where all communications are realised by *put* instructions. Note that, for straight line programs, *get* operations can always be rewritten as *put* instructions [14]:

$$((S_1 \;;\; put(y, e)) \;||^{ss}\; S_2) \quad = \quad (S_1 \;||^{ss}\; (S_2 \;;\; get(y, e)))$$

A transformation which replaces each BSP-style message by a corresponding pair of point-to-point communications is defined by means of a transition system. Let

$$[U_1, U_2 \mid \langle V_1 \rangle, \langle V_2 \rangle]$$

denote a partially transformed system; here U_1 and U_2 represent the partially transformed processes S_1 and S_2 where (some of) the *put* instructions have been replaced by *sends*; $\langle V_1 \rangle$ and $\langle V_2 \rangle$ comprise partially constructed collections of *receive* instructions corresponding to the *send* operations generated during the transformations of S_1 and S_2. $\langle V_1 \rangle$ and $\langle V_2 \rangle$ are initially empty. After the transformation process $\langle V_1 \rangle$ (respectively, $\langle V_2 \rangle$) comprises the *receive* instructions corresponding to the *send* instructions resulting from the transformation of the *put* instructions in S_2 (respectively, S_1). Angle brackets $\langle \cdots \rangle$ are used to enclose atomic actions [2]. Note that, since $||^{ss}$ is commutative [14] (this follows from the commutitivity of $||^{I}$), it follows that:

$$[U_1, U_2 \mid \langle V_1 \rangle, \langle V_2 \rangle] = [U_2, U_1 \mid \langle V_2 \rangle, \langle V_1 \rangle] \tag{4}$$

1. *Initial state.* The initial state corresponding to $S_1 \parallel^{ss} S_2$ is:

$$[S_1, S_2 \mid \langle \Pi \rangle, \langle \Pi \rangle]$$

2. *Transformation.* A *put* operation is transformed into a *send-receive* pair as follows:

$$\begin{aligned}[U_{11}; put(x,e); U_{12}\ ,\ U_2 \mid \langle V_1 \rangle\ ,\ \langle V_2 \rangle] \quad &\mapsto \\ &[U_{11}; send(e, tag, 2); U_{12}\ ,\ U_2 \mid \langle V_1 \rangle\ ,\ \langle V_2 \parallel receive(x, tag, 1) \rangle]\end{aligned} \tag{5}$$

   where $x \in \alpha(U_2)$.
3. *Termination.* Since all *receive* operations within $\langle V_1 \rangle$ and $\langle V_2 \rangle$ commute the transition system is confluent and has a unique final state:

$$\begin{aligned}[U_1\ ,\ U_2 \mid \langle V_1 \rangle\ ,\ \langle V_2 \rangle] \quad &\mapsto \\ &(\text{var } M_{1,2} := \{\mapsto\}; U_1\ ;\ V_1 \text{ end } M_{1,2}) \\ &\parallel \\ &(\text{var } M_{2,1} := \{\mapsto\}; U_2\ ;\ V_2 \text{ end } M_{2,1})\end{aligned} \tag{6}$$

   where U_1 and U_2 do not contain any *put* instructions. Here $V_1 \parallel V_2$ takes the place of the original *barrier* operation.

In a transformed program the receiving part of each process acts as a nondeterministic parallel assignment. For example, if the final state of $\langle V_1 \rangle$ is:

$$\langle receive(x, tag, 2) \parallel receive(x, tag', 2) \parallel receive(y, tag'', 2) \rangle$$

then, on execution, the following updates will take place:

$$x, y := \sqcap\{M_{1,2}(tag), M_{1,2}(tag')\}, M_{1,2}(tag'')$$

The correctness of the transition system is guaranteed iff the transformation from *puts* to *send-receive* pairs preserves the orginal semantics. Ignoring local variables and buffers we have:

Lemma 1. $(put(x,e) \parallel^{ss} \Pi) = (\ send(e, tag, 2)) \parallel (\ receive(x, tag, 1))$

4.2 Visible Conditional Communications

Consider a superstep process, S_i, with a conditional communication:

$$\gamma \lhd b \rhd \Pi$$

where γ is a simple BSP-type communication statement. Nested conditional communication statements can always be rewriten with a single guard [8]. The communication is *visible* iff it has a read only guard - that is, if $\alpha(b) \notin \alpha_W(S_i)$. Clearly $\alpha(b) \subseteq \alpha(S_i)$. Then, by superstep disjointness, $\alpha(b) \notin \alpha_W(S_j)$, for any superstep process, S_j, $i \neq j$. Thus, the boolean communication guard b is read-only over all processes and is *visible* to the entire system. *Visible* guards have the desirable property that they can be replicated in any other superstep process. Thus, the conditions under which the original BSP-type communication takes place can be expressed within the receiving processes. There are many practical algorithms which can be expressed as supersteps with *visible* guards. Some examples are:

1. matrix multiplication (and many other BLAS operations) and Gauss-Seidel relaxation which employ unconditional communication schemes.
2. Gram-Schmidt orthogonalisation and Floyd's shortest path algorithm which employ *visible* communication guards in a generic process framework to define which process should broadcast data over the system.

The definition of *visibility* can be extended to cover non-local array references in BSP-type communication statements. An array communication $put(a[i], e)$ in process S_i has *visible* non-local indices iff $i \notin \alpha_W(S_i)$. Similarly, the instruction $get(a[e], b[j])$ in process S_j is *visible* iff $j \notin \alpha_W(S_i)$. A superstep with *visible* guards and non-local array indices may be transformed into a potentially more efficient loosely coupled system with MPI-like communications.
Transformation. An occurrence of a *visible* conditional BSP-type communication can be transformed into a *send-receive* pair as follows:

$$\begin{array}{l} [U_{11};\ (put(a[i], e)\ \lhd\ b\ \rhd\ \Pi)\ ; U_{12}\ ,\ U_2 \mid \langle V_1 \rangle\ ,\ \langle V_2 \rangle] \quad \mapsto \\ \quad [U_{11};\ (send(e, tag, 2)\ \lhd\ b\ \rhd\ \Pi)\ ; U_{12}\ ,\ U_2 \mid \\ \quad\quad \langle V_1 \rangle\ ,\ \langle V_2 \parallel (receive(a[i], tag, 1)\ \lhd\ b\ \rhd\ \Pi)\ \rangle] \end{array} \tag{7}$$

Note that *visible get* instructions can always be rewritten as *put* operations:

$$\frac{visible(b)}{((S_1\ ;\ (put(y,e)\ \lhd\ b \rhd\ \Pi)) \parallel^{ss} S_2) \quad = \quad (S_1 \parallel^{ss} (S_2\ ;\ (get(y,e)\ \lhd b\ \rhd \Pi)))} \tag{8}$$

where $visible(b) \quad =_{df} \quad b \in \alpha_R(S_1) \wedge b \in \alpha_R(S_2)$.

4.3 Iterative Processes and Communication Extraction

Consider an iterative superstep process S_1 defined as $b * (S_1\ ;\ \gamma\ ;\ S_2)$ where γ is a *visible* BSP-type communication and $b * S$ denotes a computation which repeatedly performs S until $\neg b$. The transformation rules developed earlier cannot be applied to such a system; clearly, $b \in \alpha_W(S_1)$ - if not the iteration must degenerate into either Π or $\perp$ (undefined). Consequently, b is $non-visible$ to the recipent process. Thus, the recipent process cannot determine the number of *receive* instructions that are needed to match the *send*s generated from S_1.

In order to transform iterative superstep processes into a loosely coupled system it is helpful to separate communications from computations. Some simple examples of communication extraction are:

$$b * (get(x, z); S) \quad = \quad (get(x, z)\ \lhd b\ \rhd \Pi); (b * S) \tag{9}$$

$$b * (put(x, e); S) \quad \sqsubseteq \quad (put(x, e)\ \lhd b\ \rhd \Pi); (b * S) \tag{10}$$

In the last case preference is given to the first communication by refinement. It is possible to devise a set of *general* rewrite rules that separate communication from computation:

Theorem 2 (Normal form). *Each superstep process can be rewritten as $S\ ; C$ where S does not contain communication statements and C is a sequence containing (i) unconditional communications, or (ii) conditional communication or (iii) block communications.*

A *block* communication results from the extraction of an array communication from a loop. For example, $b_1 * (S\ ;\ (get(a_1[e_1], a_2[e_2])\ )$ may be rewriten as:

$$\begin{array}{l} \textbf{var}\ \ t_1,\ t_2, := [], []; \\ \quad b_1 * (S\ ;\ t_1 := t_1\hat{}[e_1]; t_2 := t_2\hat{}[e_2]; \\ \quad \textbf{get}^*(a_1[t_1], a_2[t_2]) \\ \textbf{end}\ t_1, t_2, \end{array}$$

Here ˆ denotes sequence concatenation and t_1 and t_2 are temporary vectors which record (i) the indices of a_1 that are to be updated and (ii) the corresponding replacement values; $\textbf{get}^*$ denotes a generalised version of *get* which defines a *block* communication [3].

[3] Work on block communication transition systems and a normal form for iterative processes is currently in progress

4.4 Generic Processes

In the examples in the previous sections a single BSP communication is transformed into a point-to-point communication. The situation is more complex for supersteps composed of generic processes; in these a single communication statement may be replicated across all processes giving rise to a set of communications.

Consider a superstep composed of generic processes, S_i, $0 \leq i < P$, in which S_i performs a communication:

$$put(a[g(i)], e)$$

where g is some *visible* index relation. The effect of the command on the system is to route a set of messages over a communication transfer map h where

$$h : [0 \ldots P-1] \mapsto [0 \ldots P-1]$$

and $\forall i$: $0 \leq i < P \Rightarrow a[g(i)] \in \alpha(S_{h(i)})$. In order to transform $\|_{0 \leq i < P}\ S_i$ into a loosely coupled system it is necessary to generate transformed processes S_i', $0 \leq i < P$, which both send and, possibly, receive messages. Clearly, $put(a[g(i)], e)$ can be transformed into a *send* instruction:

$$send(e, i.tag, h(i))$$

Here i is prefixed to tag in order to distinguish distinct communications in the system. The corresponding set of receiving processes is $rng(h)$ where rng is a function which returns the range of its relation argument. Note that h may not be a function. A process S_i receives a message iff $i \in rng(h)$. However, a receiving process may have more than one message sent to it.

A transition system for a system composed of generic processes has the form:

$$[U_i, \mid \langle V_i \rangle]$$

where U_i represents the partially transformed process S_i and $\langle V_i \rangle$ represents the partially constructed receiving part which will eventually be appended to the first component. The *transformation* phase of the transition system for *straight line* generic processes is:

$$\begin{aligned} [U1_i; put(a[g(i)], e); U2_i \mid \langle V_i \rangle] \quad \mapsto & \\ [U1_i; send(e, i.tag, h(i)); U2_i \mid & \\ \langle V_i \parallel ((\|_{r \in h^{-1}(i)}\ receive(a[g(i)], i.tag, r)) \lhd i \in rng(h) \rhd \Pi) \rangle] & \end{aligned} \tag{11}$$

where $a[g(i)] \in \alpha(S_{h(i)})$.

The transition system can be extended to include a *transformation* phase for generic processes with *visible* communication guards. Consider a superstep process S_i with a *visible* $b(i)$ and let h be a routing relation (as above). If h is bijective then the generated *receive* guard is $b(h^{-1}(i))$. More generally, it may be necessary to generate a *set* of receive guards for process S_i (as above). For each $r \in h^{-1}(i)$ a guard $b(r)$ is generated.

A *visible* generic transformation system may be used to transform the BSP version of prefix sum into the loosely coupled MPI-like system; however, after applying the transition system it is necessary to rewrite the resulting SPMD computation as threads. Details of rewrite systems which achieve this purpose are provided in [14].

5 Related and Future Work

The difficulties associated with the design of *correct* loosely coupled systems are well known [8, 9]. In this paper it is proposed that a simple, secure design approach for developing distributed systems can be devised by imposing a $non-reactive$ environment on the development process; to this end BSP supersteps are used as intermediate designs in the development of loosely coupled systems. In a similar vein [4, 7] propose design methods which simplify the development of correct distributed software. In [4] a design scheme which arranges communications into layers (in a similar way to BSP supersteps) is used. In [7] specialised communication patterns are represented by functions which act across processes. A functional (SPMD) development method is then employed to calculate efficient code.

The transformation system defined in this paper provides a mechanism for developing loosely coupled systems that have matching deadlock-free communication schemes. It remains to:

1. refine a parallel composition of *receive* commands (such instructions may occur in a transformed program) into a sequential compositon (with a possible loss of non-determinism);
2. investigate the efficiency of transformed programs relative to their BSP parents by conducting timing experiments.
3. determine if the increased overhead associated with a transformed $non-visible$ communication is prohibitively expensive.

References

1. Akl S.G.: The Design and Analysis of Parallel Algorithms, Prentice-Hall, 1989.
2. Apt K.R, Olderog E-R.: Verification of Sequential and Concurrent Programs, Second Edition, Springer-Verlag, 1999.
3. Awerbuch B.: Complexity of network synchronization, JACM, **32** (4) (1985), 804–823.
4. Elrad T., Francez N.: Decomposition of distributed programs into communication-closed layers, Science of Computer Programming, **2** (1982) 155–173.
5. The Grid : blueprint for a new computing infrastructure, editor I. Foster, Morgan Kaufmann, 1999.
6. Gerbessiotis A. V., Valiant L. G.: Direct bulk-synchronous parallel algorithms, Journal of Parallel and Distributed Computing, **22** (1994) 251–267.
7. Gorlatch S.: Toward Formally-Based Design of Message Passing Programs, IEEE Transactions on Software Engineering, **26** (2000) 276–288.
8. Hoare C. A. R., Jefeng H.: Unifying Theories of Programming, Prentice Hall, 1998.
9. Owicki S., Gries D.: An axiomatic proof technique for parallel pograms, Acta Informatica, **6** (1976) 319–340.
10. MPI: A Message-Passing Interface Standard. University of Tennessee, 1995.
11. Stewart A., Clint M.: Synchronising asynchronous communications *in* Euro-Par '97, LNCS 1300, Springer-Verlag, edts: C. Lengauer, M. Griebl and S. Gorlatch (1997) 511–520.
12. Stewart A., Clint M., Gabarró J.: Axiomatic frameworks for developing BSP-style programs, Parallel Algorithms and Applications, **14** (2000) 271–292.
13. Stewart A., Clint M.: BSP-style computation: a semantic investigation, Computer Journal (to appear).
14. Stewart A., Clint M., Gabarró, J.: Algebraic rules for reasoning about BSP programs, Proceedings of Constructive Mathematics for Parallel Programming, 2000 (to appear).
15. Valiant L. G.: A bridging model for parallel computation, Comm. ACM, **33** (1990) 103–111.