Loosely-Coupled Processes

Jayadev Misra*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

1 Introduction

1.1 Message Communicating and Shared-Variable Systems

A system of processes in which the interactions are solely through messages is often called *loosely-coupled*. Such systems are attractive from a programming viewpoint. They are designed by decomposing a specification into its separable concerns, each of which could then be implemented by a process; the operation of the system can be understood by asserting properties of the message sequences transmitted among the component processes. A key attribute of loosely-coupled systems is a guarantee that a message that has been sent cannot be unsent. As a consequence, a process can commence its computation upon receiving a message, with the guarantee that no future message it receives will require it to undo its previous computations.

Processes that communicate through shared variables, where a shared variable may be read from/written to by an arbitrary number of processes, are often called *tightly-coupled*. In contrast to loosely-coupled systems, designs of tightly-coupled systems typically require deeper analysis. Since speeds of the component processes are assumed to be nonzero and finite, but otherwise arbitrary, it is necessary to analyze all possible execution sequences, however unlikely some of them may be, to guarantee the absence of "race conditions." Special protocols for mutual exclusion are often required for a process to access shared-variables in an exclusive manner. Yet, shared-variables often provide succinct, and even elegant, solutions; for instance, broadcasting a message can often be implemented by storing the message in a variable that can be read by every process.

1.2 Loosely-Coupled Processes

To motivate the discussion, we consider two examples of shared-variable systems. In the first case, we have two processes sharing an integer variable x; one process doubles x and the other process increments x by 1, from time to time; both processes read x and assign it to their local variables. We contend that these two processes are tightly-coupled by x; each process needs to know the exact value of x before it can complete its access—read or write—of x; no process can proceed with its computation with only a partial knowledge of the value of x. Contrast this situation with a system in which two processes share an integer variable y; the first process periodically increments y by 1; the second process periodically decrements y by 1 provided y is positive, and then it proceeds with its computation. (The variable y implements a semaphore.) We argue that these two processes are loosely-coupled; the second process can continue with its computation knowing only that y is positive, without knowing the exact value of y. Similarly, the first process need not know the exact value of y; it may merely transmit the message that y should be incremented by 1, to the

*This work was partially supported by ONR Contract N00014-90-J-1640 and by Texas Advanced Research Program grant 003658–065.

second process; the latter increments y upon receiving this message. The value of y at the second process is never more than the true value of y and it tends to "catch up" with the true value; hence the system will not be permanently blocked.

1.3 Contributions of This Paper

The notions of tightly-coupled and loosely-coupled processes depend on the way the shared-variables are accessed. We propose a definition of loosely-coupled processes and show that all point-to-point message passing systems are loosely-coupled under our definition. We argue that large-scale shared-variable programming is feasible only if processes are loosely-coupled. First, the action-sequences in different processes are then serializable [5]. Therefore, it may be imagined that accesses to shared variables are exclusive even though explicit mutual exclusion is not implemented. Second, an important class of progress properties—of the form, if p is *true* now then q is or will become *true*—holds in a loosely-coupled system if it is implemented in a "wait-free manner" by any component process of the system, and the post-condition q is falsified by no other process; this result does not hold for arbitrary—i.e., non-loosely-coupled—systems. Therefore, a possible programming methodology is to design a loosely-coupled system in which each property of the above form is implemented by a single process, with the restriction that no other process falsify the post-condition.

A problem in multiprocessor system design, called *cache-coherence* [9], has its roots in shared-variable programming. Suppose that several processes hold copies of a shared-variable in their local caches. If processes write into their local copies autonomously then these copies may become inconsistent (and then the processes may read different values for the shared-variable). The traditional solutions to cache-coherence restrict accesses to the caches at runtime so that the inconsistencies are avoided; one typical solution is to lock all caches prior to a write by any process, and then broadcast the new value to all processes after completion of the write.

The cache-coherence problem vanishes for loosely-coupled processes. Each process that accesses a shared variable, x, initially keeps a copy of x in its local cache. Reads and writes are performed on the local copies. Whenever a local copy of x is changed the new value is transmitted, asynchronously, to all processes that hold a copy of x. Each process, upon receiving notification of a change, updates its local copy. We prove that this implementation scheme is correct.

The suggested implementation allows the processes to compute asynchronously, using only a partial knowledge of the values of the shared variables. Unlike traditional multiple-copy systems the copies of a shared variable will, typically, have different values; furthermore, no permission is ever sought by a process from other processes nor are the caches locked before a local copy is changed. Therefore, the performance of the system will not degrade if the number of processes accessing a shared-variable is increased substantially; see [8] for some of the issues in scaling up shared-memory multiprocessors. Since the implementation by local copies is transparent to the programmer, programming loosely-coupled systems will retain much of the succinctness of expression while admitting efficient implementations.

1.4 Related Work

It is easy to see that read-only shared variables cause no race conditions; however, processes cannot communicate information about the progress of their computations through such variables. Several concurrent logic programming languages [12] support *logic variables* or write-once variables. A logic variable is initially undefined and it may be assigned a value at most once and by a single designated process, in any computation. Logic variables have been used in place of message-communicating primitives ([1], [14], [3]). In particular, the programming notation PCN by Chandy and Taylor [3] makes a distinction between ordinary program variables—called *mutables*—and logic variables—called *permanents*; concurrently executing processes (in one form of process composition) are prevented from changing the shared mutables, thus communicating through the permanents only. It can be shown that these processes are then loosely-coupled. Kleinman et al [7] introduce *directed logic variable* and use these to effect two-port communication of at most one message. In

particular, by embedding port names as parts of messages, they show how such variables can be used to implement streams, remote procedure calls and process migration.

A slightly general form of shared variables has been employed in [10]. A shared variable assumes an additional value, $\perp$, analogous to the undefined value. A value can be assigned to a shared variable, x, only if its current value is $\perp$; a value can be read from x only if its current value differs from $\perp$; the reading causes x to have the value $\perp$. The variable x may be viewed as a buffer of size 1 for which the writer, reader act as producer, consumer. Unlike logic variables, such shared variables may be assigned value more than once. Processes employing such shared variables where each variable is shared between two processes only can be shown to be loosely-coupled.

Gaifman, Maher and Shapiro [6] have observed that by restricting the updates of the common store to be monotonic, nondeterministic computations can be replayed efficiently, for the purposes of debugging and recovery. They also propose an easy solution to the snapshot problem for such programs.

Steele [13] has argued that asynchrony can be made safe by requiring that the "causally-unrelated" actions of various processes commute. He argues that such restricted systems where each process is additionally deterministic and the computation is terminating, have some of the properties of SIMD systems (in which processes operate in lock-step) including determinacy. He shows how the violation of commutativity can be detected at run-time. Chandy and Misra ([2], Sec. 8.4.2) had proposed a set of access conditions for shared variables that allows the shared variables to be implemented by messages; the present work simplifies and generalizes those conditions.

In response to an earlier draft of this paper, Cohen [4] gives a somewhat more general asymmetric theory of loose-coupling in terms of program properties (i.e., without commutativity). The theory includes rules to derive loose-coupling of systems from loose-coupling of their components and rules to lift arbitrary progress properties of components to progress properties of a system. These rules have been successfully applied to the solution of several concurrent programming problems.

2 A Theory of Loosely-Coupled Processes

2.1 Processes

We consider a system consisting of a finite set of processes. Each process has a local store which only that process can access, and there is a global store which all processes can access. (The control point of a process, i.e., the point in the process-text where control resides at some point in the computation, is a part of its local store). Let A be the set of possible values that can be held in the global store, and L_i, the set of possible values for the local store of some specific process, P_i. The set of system states is the cartesian product $A \times_i L_i$, ranging over all process indices i.

The states of the stores are modified by the actions of the processes. An action in process P_i is given by a function, f,

$$f \ : \ A \times L_i \ \ \rightarrow \ \ A \times L_i$$

denoting that P_i may read only from the global store and its own local store, and may modify only those stores. A function may be partial; i.e., it may be defined for a subset of $A \times L_i$.

The effect of applying an action in a given system state is to evaluate the corresponding function, if it is defined, in that state and overwrite the appropriate part of the system state—the global store and the local store of the process—by the function value; if the function is not defined in a system state the action has no effect.

Each process consists of a finite set of actions. Execution of a system consists of an infinite number of steps. In each step, an action is chosen from some process and applied in the current state; both choices—the process and the action—are nondeterministic. The nondeterministic choice is constrained by the fairness rule that every action from every process is chosen eventually.

This process-model is from UNITY [2]. At first sight, it may seem restrictive because the traditional control structures—if-then-else, do-while, sequencing, etc.—have been eliminated. Yet, this model has all

the generality of the traditional models (except process creation and deletion), and it provides a simpler basis for constructing a theory of loosely-coupled processes, by ignoring the issues of program syntax. {To see how traditional control structures are encoded within this model, consider two actions, f and g, that are to be applied in sequence. Let the program-counter, pc, be 0 when f is to be applied and 1 when g is to be applied. In our model, pc is a part of the local store of the process. We define a function f' at all system states where pc is 0 *and* f is defined; the effect of applying f' is the same as applying f except that the former results in additionally setting pc to 1. Similarly, we define g' (for all states where g is defined and pc is 1). This scheme implements the desired sequencing implicitly.}

Convention: A function f,

$$f \ : \ A \times L_i \ \ \rightarrow \ \ A \times L_i$$

has the same effect as a function f', that is defined on system states:

$$f' \ : \ A \times_i L_i \ \ \rightarrow \ \ A \times_i L_i$$

where f' simply does not "read" or "update" any L_j, $j \neq i$. Therefore, henceforth we view each function as a mapping from system states to system states.

2.2 Definition of Loose-Coupling

A set of processes is *loosely-coupled* if for every system state x and every pair of functions, f, g, whenever $f.x$ and $g.x$ are both defined, then so are $f.g.x$ and $g.f.x$, and they are equal in value.

Observation 1: Any two functions from a single "sequential" process commute because there is no system state where both functions are defined (since the functions are defined at system states where the program counter for this process has different values). □

Observation 2: Let f, g be functions from distinct processes. If variables accessed by *both* f, g are read-only variables, i.e., no variable written by an action is read or written by another action, then the functions commute. Therefore, the test of commutativity need be applied only to functions where one accesses (i.e., reads or writes) a variable that the other writes into. □

2.3 Examples of Loosely-Coupled Processes

We give several examples of processes that are loosely-coupled. We adopt the UNITY syntax for writing the process-codes (though the results are entirely independent of the syntax).

counting

An integer variable, c, is shared among processes $P_0 \ldots P_N$. Process P_i, $1 \leq i \leq N$, accesses c in a statement,

$$\{P_i\} \quad c \ := \ c + d_i \qquad \text{if} \quad b_i$$

where b_i, d_i are local to P_i and $d_i > 0$. Process P_0 references c in

$$\{P_0\} \quad l \ := \ \mathit{true} \qquad \text{if} \quad c > 0$$

where l is a local variable of P_0.

The functions from P_i, P_j, $i \neq j$, $1 \leq i \leq N$, $1 \leq j \leq N$, clearly commute (because b_i, d_i are local to P_i and addition is commutative). The functions from P_0 and P_i, $1 \leq i \leq N$, commute because, since $d_i > 0$,

$$b_i \ \wedge \ c > 0 \quad \Rightarrow \quad \begin{array}{ll} c + d_i > 0 & \{P_0 \text{ can be applied after applying } P_i\} \wedge \\ b_i & \{P_i \text{ can be applied after applying } P_0\} \end{array}$$

Also, the value of (c, l) after applying P_0, P_i in either order is $(c_0 + d_i, \mathit{true})$, where c_0 is the value of c before the operations were applied. □

parallel search

N processes, $N > 0$, carry out a search in parallel. Shared variables are P, r, where P is a set of process indices, 0 through $N-1$ (P holds the indices of all processes that have yet to complete the search) and r is the index of the lowest numbered process that has succeeded in the search; if no process has yet succeeded in the search then $r = N$. Initially, $P = \{i |\ 0 \leq i \leq N-1\}$ and $r = N$. The code of process i, $0 \leq i < N$, is as follows:

$$\begin{array}{lll} \{f_i\} & P, r := P - \{i\}, r \ \min \ i & \text{if} \quad b_i \ \wedge \ i \in P \\ [\!] \ \{g_i\} & P \ := P - \{i\} & \text{if} \quad c_i \ \wedge \ i \in P \end{array}$$

Here b_i, c_i are local predicates of process i denoting successful or unsuccessful completion of the search. Process N accesses the shared variables, using

$$\{h\} \quad d \ := \ r \qquad \text{if} \quad P = \phi$$

where d is a local variable of this process. Thus, d records the index of the lowest numbered successful process, or N if there is no such process.

To see that these processes are loosely-coupled:

- f_i, g_i commute because b_i and c_i cannot hold simultaneously, i.e., the search cannot be both successful and unsuccessful.
- f_i, f_j, $i \neq j$, commute because removing two different items from a given set in either order results in the same set and *min* is commutative and associative.
- f_i, g_j, $i \neq j$, commute for similar reasons.
- g_i, g_j, $i \neq j$, commute for similar reasons.
- h, f_i (or h, g_i) are not both defined in any state because $P = \phi \ \wedge \ i \in P$ never holds. Therefore, they commute trivially.

point-to-point communication

Two processes, a *sender* and a *receiver*, communicate using a FIFO channel, c. The sender sends messages; the messages are eventually delivered to the receiver in the order sent. It is not surprising that this pair of processes is loosely-coupled; we verify this, formally, below.

We regard the channel, c, as a variable—of type message sequence—shared between the sender and the receiver. The action at the sender is of the form:

$$\{\mathit{send}\} \ c \ := \ c; m \qquad \text{if} \quad bs$$

Here ";" denotes concatenation and m, bs are local to the sender denoting, respectively, the next message being sent and the condition for sending m. The action at the receiver is of the form:

$$\{\mathit{receive}\} \ h, c \ := \ \text{head}.c, \text{tail}.c \qquad \text{if} \quad br \ \wedge \ c \neq \langle\rangle$$

Here h, br are local to the receiver (the received message is copied into h; the condition for receiving a message is given by br) and "$\langle\rangle$" denotes the empty sequence.

The two corresponding functions commute in that they produce the same values for h, c when applied in either order, i.e.,

$bs \ \wedge \ br \ \wedge \ c \neq \langle\rangle \ \Rightarrow$ {condition for sending and receiving}
$(\text{head}(c; m), \text{tail}(c; m))$
{value of h, c by first sending and then receiving}
$=$ $(\text{head}(c), \text{tail}(c); m))$
{value of h, c by first receiving and then sending}

multi-input channel

This is a variation of the previous example. Two senders use a single channel to send messages to a receiver.

The send actions are

$\{S1\} \quad c \ := \ c; m1 \qquad \text{if} \quad bs1 \quad \text{and}$
$\{S2\} \quad c \ := \ c; m2 \qquad \text{if} \quad bs2$

where$m1, bs1$ are local to Sender 1 and $m2, bs2$ to Sender 2. As before, the receiver action is

$\{receive\} h, c \ := \ \text{head}.c, \ \text{tail}.c \qquad \text{if} \quad bf \ \wedge \ c \neq \langle \ \rangle.$

S1, S2 do not commute because, given $bs1 \ \wedge \ bs2$

$(c; m1); \ m2 \neq (c; m2); \ m1.$

This reflects the possibility that relative speeds of the processes and the communication network could affect the outcome of the computation.

Note: The last example shows that a system containing a "fair-merge" process (in our case, the multi-input channel implements a fair-merge) cannot be loosely-coupled. However, if the multi-input channel implements a bag, rather than a sequence, then the system is loosely-coupled. In this case, the receiver cannot exploit the order of the messages it receives. {The treatment of bags is problematic in our theory because no function (at the receiver) can remove *some* element from a bag, nondeterministically. Special cases, however, can be handled. For instance, if only a bounded set of messages can be sent—say, each message is a natural number below 100—then the receiver could be structured as a set of functions, the i^{th} function removes message i if it is in the bag.} It can also be shown that multiple senders and multiple receivers sharing a bag—as is the case for a client-server system sharing a task-pool—are *not* loosely-coupled; this implies that some degree of central control is required in this case to ensure that no task gets distributed to more than one server. □

broadcast

The value of a shared variable x is to be broadcast to processes $P_1 \ldots P_N$. Process P_0 stores a new value into x provided the previous value has been read by all processes. A process P_i, $1 \leq i \leq N$, records the value of x, in a local variable x_i, provided it is a new value. In order to determine if a value is new and also if the processes have read the previous value, we introduce boolean variables $b_0 \ldots b_N$, and the invariant: For any i, $1 \leq i \leq N$, P_i has read the current value of x iff $b_i = b_0$. Thus, if b_0 equals every other b_i then every P_i has read the value of x; a new value may then be stored in x and b_0 changed (so that it differs from every other b_i). Also, P_i reads x only if b_i differs from b_0; reading is accompanied by setting b_i to b_0. Specifically, the write action in P_0 is

$\{P_0\} \quad x, b_0 \ := \ w, \neg b_0 \qquad \text{if} \quad (\forall i \ : \ 1 \leq i \leq N \quad :: \quad b_0 = b_i) \ \wedge \ cw$

where w, cw are local to P_0; variable w holds the next value to be broadcast and cw is the condition for broadcasting. The read action in P_i is

$\{P_i\} \quad x_i, b_i \ := \ x, b_0 \qquad \text{if} \quad b_i \neq b_0 \ \wedge \ cr_i$

where x_i, cr_i are local to P_i; the value read is stored in x_i and cr_i is the condition for reading.

The functions in P_0, P_i, $1 \leq i \leq N$, commute because the corresponding actions cannot both be performed in any state:

$$b_i \neq b_0 \ \wedge \ (\forall i \ : \ 1 \leq i \leq N \quad :: \quad b_0 = b_i)$$

is *false*.

The operations in P_i, P_j, $1 \leq i \leq N$, $1 \leq j \leq N$, commute because common variables accessed by these two actions are x and b_0, and both are only read by these two actions. Hence the processes are loosely-coupled.

As a consequence of loose-coupling, we can show that simultaneous access to all b_i's, as required in P_0's action, may be replaced by asynchronous accesses. □

2.4 Properties of Commuting Functions

In this section, we prove a number of properties of loosely-coupled processes. In particular, we show that if in a given system state finite executions of two different processes are defined then so are their interleavings; further, applications of all such interleavings result in the same system state. These results are essential for proving properties of loosely-coupled systems and correctness of their implementations.

Henceforth, we deal with (partial) functions from D to D, for a fixed set D; in the context of loosely-coupled processes D is the set $A \times_i L_i$—the set of system states—and each function corresponds to an action in some process.

Notation: Function composition is shown by concatenation and function application by a "." written between the function and its argument. Function application is from the right to the left. Thus, for a finite sequence of functions α: if α is the empty sequence then $\alpha.x$ is defined and is equal to x; if $\alpha = \alpha' f$ then $\alpha.x$ is defined only if both $f.x$ and $\alpha'.(f.x)$ are defined (and then, $\alpha.x = \alpha'.f.x$). □

Functions f, g *commute* iff for every x in D,

$$\begin{array}{ll} & f.x \text{ and } g.x \text{ defined} \\ \Rightarrow & f.g.x \text{ and } g.f.x \text{ defined, and } f.g.x = g.f.x \end{array}$$

Notational Convention: We write $e = e'$ to denote that both e, e' are defined and they are equal. Thus, the commutativity condition for f, g can be written:

$$f.x \text{ and } g.x \text{ defined} \quad \Rightarrow \quad f.g.x = g.f.x$$

Note (Leibniz Rule with Partial Functions)**:** Given $e = e'$ we can conclude $f.e = f.e'$ *only if* $f.e$ or $f.e'$ is known to be defined. □

Two sequences of functions, α, β, *commute* if every function from α commutes with every function from β.

Lemma 1: Let function g commute with every function in sequence α. For any x in D,

$$g.x \text{ and } \alpha.x \text{ defined} \quad \Rightarrow \quad g.\alpha.x = \alpha.g.x$$

Proof: Proof is by induction on the length of α.

α is the empty sequence: Since $g.x$ is defined, $g.\alpha.x = g.x$ and $\alpha.g.x = g.x$

$\alpha = \alpha' f$:

	$f.x$ defined	, $\alpha.x$ defined and $\alpha = \alpha' f$
	$g.x$ defined	, given
(1)	$f.g.x = g.f.x$	, f, g commute
	$\alpha'.(f.x)$ defined	, $\alpha.x$ defined and $\alpha = \alpha' f$
	$g.(f.x)$ defined	, from (1)
	$\alpha'.g.(f.x) = g.\alpha'.(f.x)$	, using induction hypothesis on the above two
	$\alpha'.f.g.x = g.\alpha'.f.x$	, replacing $g.f.x$ in the lhs of the above by $f.g.x$, using (1)
	$\alpha.g.x = g.\alpha.x$	, using $\alpha = \alpha' f$ in both sides of the above

□

Theorem 1 gives a generalization of Lemma 1, for a pair of sequences.

Theorem 1: Suppose α, β commute. Let γ, δ be any two interleavings of α, β. For any x in D,

$$\alpha.x \text{ and } \beta.x \text{ defined} \quad \Rightarrow \quad \gamma.x = \delta.x$$

Proof: It suffices to show that for any interleaving γ, $\gamma.x$ is defined and it equals a particular interleaving, say $\alpha\beta$, applied to x. Proof is by induction on the length of β.

β is empty: Then $\gamma.x = \alpha.x$ (because $\gamma = \alpha$ and $\alpha.x$ is defined). Trivially, $\gamma.x = \alpha.\beta.x$.

$\beta = \beta' g$: Since γ is an interleaving of α, β
$\gamma = AgB$
where A, B are sequences of functions and B is a suffix—possibly empty—of α.
Furthermore, the sequence AB is an interleaving of α, β'. For any x in D,

	$g.x$ defined	, $\beta.x$ defined and $\beta = \beta' g$
	$\alpha.x$ defined	, given
	$\alpha.g.x$ defined	, using Lemma 1 on the above two
	$\beta'.g.x$ defined	, $\beta.x = \beta'.g.x$
(1)	$(AB).(g.x) = (\alpha\beta').(g.x)$	, using induction hypothesis, any interleaving of α, β'—in particular AB—is defined at $g.x$ and its value equals $(\alpha\beta').(g.x)$
	$B.x$ defined	, B is a suffix of α and $\alpha.x$ is defined
	$g.x$ defined	, $\beta.x$ defined and $\beta = \beta' g$
	$B.g.x = g.B.x$	, using Lemma 1 on the above two
	$A.B.g.x = A.g.B.x$	, from the above; note $A.B.g.x$ is defined (from 1)
	$\alpha.\beta.x = \gamma.x$	, in the above replace $A.B.g.x$ by $\alpha.\beta.x$ (from 1) in the lhs and γ by AgB in the rhs

□

Often, arguments about commuting functions involve exchanging two such functions in a given sequence; thus, a sequence $\alpha f g \beta$ is transformed to $\alpha g f \beta$ provided f, g commute. Such exchanges are permissible provided for *all* x: $f.g.x = g.f.x$. Our notion of commuting is weaker and it does not allow us to perform such exchanges. We do show a result—Theorem 2, below—which allows exchanges or, equivalently, moving one sequence to the right of another under more stringent conditions.

Theorem 2: Suppose α, β commute. Let γ be an interleaving of α, β. For any x,

$$\beta.x \text{ defined}, \ \gamma.x \text{ defined} \quad \Rightarrow \quad \gamma.x = \alpha.\beta.x$$

Proof: Proof is similar to that of Theorem 1; we apply induction on the length of β.

β is empty: Then $\gamma = \alpha$. If $\gamma.x$ is defined then so is α and $\gamma.x = \alpha.\beta.x$
$\beta = \beta' g$: Then γ is of the form AgB where B is a suffix of α and AB is an interleaving of α, β'.

	$B.x$ defined	, B is a suffix of γ and $\gamma.x$ defined
	$g.x$ defined	, $\beta = \beta' g$ and $\beta.x$ defined
	$B.g.x = g.B.x$	, Lemma 1 on the above two
	$(ABg).x = (AgB).x$	, from the above and $(AgB).x$, i.e., $\gamma.x$, defined
(1)	$(ABg).x = \gamma.x$	, replacing AgB by γ in the rhs of the above
	$(AB).(g.x)$ defined, $\beta'.(g.x)$ defined	, from above and $\beta = \beta' g$ and $\beta.x$ defined
	$(AB).(g.x) = \alpha.\beta'.(g.x)$	, induction hypothesis on above: AB is an interleaving of α, β'
	$\gamma.x = \alpha.\beta.x$	, replace lhs using (1) and rhs using $\beta = \beta' g$. □

The reader should note that in Theorem 2, $\alpha.x$ may not be defined. In fact, it is possible for neither of $\alpha.x, \beta.x$ to be defined and yet $\gamma.x$ to be defined. {To see this, consider three functions, f, g, h, each operating on a pair of natural numbers to yield a pair of natural numbers as the result. The functions are given by

$f.(x, y) = (x, y - 1)$ if $y > 0$
$g.(x, y) = (x + 1, y)$
$h.(x, y) = (x - 1, y + 1)$ if $x > 0$

It is easy to verify that every pair of these functions commutes. Now, suppose $\alpha = fg$, $\beta = h$, $\gamma = fhg$. Then $\alpha.(0, 0)$, $\beta.(0, 0)$ are undefined whereas $\gamma.(0, 0)$ is defined.}

The next theorem—motivated by the above observation—asserts that if any two interleavings of α, β are defined at x (and neither $\alpha.x$ nor $\beta.x$ may be defined) then the two interleavings have the same value when applied to x. First, we note a generalization of Theorem 2 which says that any suffix of β can be moved to the "right" of γ, provided the suffix and γ are both defined at x. Its proof is along the same lines as Theorem 2.

Corollary 1: Suppose α, β commute. Let $\beta = \beta' B$ and γ be an interleaving of α, β.

$$B.x \text{ defined}, \ \gamma.x \text{ defined} \quad \Rightarrow \quad \gamma.x = \gamma'.B.x$$

where γ' is some interleaving of α, β'.

Corollary 2: Suppose α, $g\beta$ commute. Let γ be an interleaving of α, β.

$$g.\beta.x \text{ and } \gamma.x \text{ defined} \quad \Rightarrow \quad g.\gamma.x \text{ defined}$$

Proof: Since $\beta.x$ is defined (from $g.\beta.x$ defined) and $\gamma.x$ is defined, we have $\gamma.x = \alpha.\beta.x$, from Theorem 2. Now,

$g.(\beta.x)$	defined	, from the antecedent
$\alpha.(\beta.x)$	defined	, from the above argument
$g.\alpha.\beta.x$	defined	, Lemma 1 applied to the above two
$g.\gamma.x$	defined	, $\gamma.x = \alpha.\beta.x$ □

A consequence of this corollary is that if $\gamma.x$ is defined and $g.\gamma.x$ is undefined then $g.\beta.x$ is undefined; this is the form in which we use this corollary in the proof of Theorem 6.

Theorem 3: Suppose α, β commute. Let γ, δ be two interleavings of α, β. Then,

$$\gamma.x \text{ defined}, \delta.x \text{ defined} \quad \Rightarrow \quad \gamma.x = \delta.x$$

Proof: Proof is by induction on the length of γ (which is same as the length of δ).

γ is empty: The consequenct holds, trivially, as $x = x$.

$\gamma = \gamma' g$:

$g.x$ defined	, $\gamma = \gamma' g$ and $\gamma.x$ defined
$\delta.x$ defined	, given
$\delta.x = \delta'.g.x$ for some δ'	, applying Corollary 1 (with $B = g$)
$\delta'(g.x), \gamma'.(g.x)$ defined	, above and $\gamma = \gamma' g$
$\delta'.(g.x) = \gamma'.(g.x)$	, induction hypothesis

□

Note: Our theorems readily generalize to interleavings of several finite sequences where each pair of sequences commute. In fact, we will normally be dealing with this more general case. □

2.5 Compositional Design of Loosely-Coupled Systems

We show how properties of a system of loosely-coupled processes may be deduced from the properties of its component processes. These deduction rules can be used as a basis for system design.

We compose processes using the *union* operator of UNITY [2]; for processes F, G their union, written as $F \parallel G$, is a process in which actions from F, G are executed concurrently and asynchronously. In each step of the execution of $F \parallel G$ an action from either F or G is chosen and executed; the choice is to be made *fairly*, in the sense that every action is chosen eventually for execution; see ([2], Sec. 7.2) for details.

The two classes of program properties—*safety* and *progress*—are expressed using the operators, *unless* and *leads-to*. (UNITY employs another operator, *ensures*, as the basis for the inductive definition of *leads-to*; we won't be needing that operator for the theory developed in this paper.)

The basic safety property is of the form, p *unless* q, where p, q are predicates (defined on the state space of the program). The operational interpretation of p *unless* q is that once p is *true* it remains *true* as long as q is *false*. An important special case is p *unless false* which means that p remains *true* once it becomes *true*; we write p *stable* for p *unless false*. A predicate p is invariant in a program if p is initially *true* and p is stable.

The basic progress property is of the form $p \mapsto q$ (read p *leads-to* q); its operational interpretation is: Once p is *true*, q is or will become *true*. See ([2], Chapter 3) for formal definitions of these operators.

union theorem

Safety properties of composite programs can be deduced from the safety properties of their component processes, as given below. This result applies to all processes, not just loosely-coupled processes.

Theorem 5 (union theorem for safety): ([2], Sec. 7.2.1)

$$p \text{ unless } q \text{ in } F \parallel G = p \text{ unless } q \text{ in } F \ \wedge \ p \text{ unless } q \text{ in } G$$

□

This theorem provides a basis for designing a composite system: A system required to satisfy p *unless* q can be designed as a union of two components, each of which satisfies p *unless* q.

There is no corresponding result for progress properties. Indeed, a progress property established by one component process may be affected by the operations of another process, as shown in the following example.

Example: Processes F, G share a variable x that can take three possible values—0, 1 or 2. Each process has a single action.

$$\begin{array}{lcll} F & :: & x & := & (x+1) \bmod 3 \\ G & :: & x & := & (x+2) \bmod 3 \end{array}$$

The initial value of x is irrelevant. It is easy to see that

$$\begin{array}{l} x = 0 \;\mapsto\; x = 2 \quad \text{in } F \\ x = 0 \;\mapsto\; x = 2 \quad \text{in } G \end{array}$$

However,

$$x = 0 \;\mapsto\; x = 2 \quad \text{in } F \;[\!]\; G$$

does not hold: To see this consider the case where F, G alternate forever, starting in a state where $x = 0$. Note, further, that F, G are loosely-coupled according to our definition.

□

(The union theorem [2] can be used to deduce a special class of progress properties, of the form p *ensures* q; the theorem says that p *ensures* q holds in a composite program iff p *ensures* q holds in at least one component and p *unless* q holds in all other components. No such result holds for $p \;\mapsto\; q$.)

The lack of a theorem for progress, analogous to the union theorem for safety, is a serious drawback for concurrent program design. In order to design a system in which $p \;\mapsto\; q$ holds, we cannot easily partition the design into several processes each satisfying some safety and progress properties. Conversely, to assert a progress property for a given system, all its components have to be considered together. A major simplification, proposed in [11], is to ascertain that the proof constructed for a single process is not affected by the executions of the statements in the other processes; construction of such a proof is still an expensive procedure.

We give a theorem that allows us to deduce progress properties of loosely-coupled systems. The essence of the theorem is: If process F establishes $p \mapsto q$ in a "wait-free" manner (defined below), and F is a component in a system of loosely-coupled processes where every other process preserves q, then $p \;\mapsto\; q$ holds in the entire system. Thus, a progress property, $p \;\mapsto\; q$, of a loosely-coupled system can be implemented by designing a single process to implement this property in a wait-free manner, and requiring the other processes to preserve q (i.e., by having "q *stable*").

Process F is *wait-free* for a pair of predicates (p, q) if once $p \;\wedge\; \neg q$ holds, every action of F is *effective* (i.e., execution of the action changes state) at least until q holds. This property can be established as follows. Let c_i be a condition under which the i^{th} action in F is guaranteed to execute effectively. Then,

$$\begin{array}{ll} p \;\wedge\; \neg q \;\;\Rightarrow\;\; (\wedge\, i \;\; :: \;\; c_i) & \{\text{every action can be executed effectively given } p \;\wedge\; \neg q\} \\ (\wedge\, i \;\; :: \;\; c_i) \textit{ unless } q & \{\text{every action can be executed effectively at least until } q \text{ holds}\} \end{array}$$

The notion of wait-freedom captures our intuitive understanding that once p holds in F, process F can execute autonomously without waiting for any external signal.

Theorem 6: Let F, G be loosely-coupled.

$$\frac{\begin{array}{c} p \;\mapsto\; q \quad \text{in } F\ , \\ F \text{ is wait-free for } (p, q)\ , \\ q \textit{ stable in } G \end{array}}{p \;\mapsto\; q \quad \text{in } F \;[\!]\; G}$$

Proof (sketch): Consider an infinite execution, σ, of $F \parallel G$ starting in a state where p holds; call this starting state x. We show that σ has a finite prefix, τ, at the end of which q holds.

Let σ_F be the sequence obtained by retaining only and all the F-actions of σ. Since σ is a (fair) execution of $F \parallel G$, σ_F is a (fair) execution of F. From $p \mapsto q$ in F, we know that every (fair) execution of F starting in a state where p holds has a prefix at the end of which q holds. Therefore, there is a prefix α of σ_F after which q holds. Since F is wait-free for (p, q), $\alpha.x$ defined.

{**Note:** Execution sequences are usually written in the order in which the actions are applied, from left to right, e.g., the execution sequence fgh denotes first applying f, then g and then h. In the notation of Section 2.4.1, executing such a sequence starting in state x is equivalent to computing $(hgf).x$.}

Let τ be a prefix of σ such that $\tau_F = \alpha$. We deduce the system state by applying τ to state x, as follows. First, remove all actions from τ that are undefined in the corresponding state (those that behave as a *skip*); let the new sequence be τ'. Clearly, the system states by applying τ and τ' to x are identical; furthermore, $\tau'.x$ is defined. We show that q holds in the state $\tau'.x$.

Now, τ is an interleaving of α and some sequence of G-actions, and $\alpha.x$ is defined. Therefore, according to Corollary 2 of Theorem 2, the removed actions from τ are G-actions only. Hence, τ' is an interleaving of α and a sequence, say β, of G-actions. Since F, G are loosely-coupled, α, β commute. Given that $\tau'.x$ and $\alpha.x$ are defined, from Theorem 2,

$$\tau'.x = \beta.\alpha.x$$

We know that q holds for $\alpha.x$ and q is not falsified by any action from β (because q is stable in G and β consists of G-actions only). Hence, q holds for $\tau'.x$. □

2.6 A Programming Methodology for Loosely-Coupled Processes

Theorem 5 provides a basis for programming loosely-coupled processes. We implement a progress property $p \mapsto q$ by a single process in a wait-free manner; then we require that the other processes not falsify q. The methodology for constructing a program from its specification, consisting of safety and progress properties, is as follows:

Require that

- the component processes be loosely-coupled,
- each safety property hold for each component process, and
- each progress property, of the form $p \mapsto q$, hold in a specific process (in a wait-free manner) and q be stable in the other processes.

This design methodology is driven by consideration of the progress properties; the safety properties merely serve as the restrictions in designing the individual processes.

3 Implementing Loosely-Coupled Processes: The Cache Coherence Problem

We show in this section that the cache-coherence problem vanishes for loosely-coupled processes. An implementation can employ an *asynchronous* communication mechanism to bring the caches into coherence, and it is not required for a process to lock other process caches or ask for permission in updating its own cache. The implication of this observation is that loosely-coupled processes can be implemented very efficiently in a distributed, message-passing type architecture. Consequently, such systems are highly scalable, a property that is not enjoyed by arbitrary shared-variable systems. Even when a system is not loosely-coupled, it

pays to identify the processes that are loosely-coupled because their interactions can be implemented asynchronously, as described above. (In fact, with slight assistance from the programmer, compilers can generate code that minimizes cache-locking.)

The implementation of loosely-coupled processes employing a point-to-point communication network is as follows. Initially, each process holds the contents of the global store and the local stores of all processes in a local cache; hence, all caches are initially coherent. A process computes by reading values from its local cache and writing new values into its local cache. Whenever a process changes a value in its local cache it sends a message to every other process informing them of the change. Upon receiving a message a process updates its local cache appropriately (the exact mechanism of update is explained below).

We show that this implementation is correct. Observe that the caches may never be coherent beyond the initial state. Therefore, we cannot show that for every finite execution of the original system there is a finite execution in the implementation that has the same final state. Instead, we show that whatever properties hold in the original system also hold in the implementation; this is made more precise below.

3.1 Asynchronous Implementation of Caches

Let A denote the original system consisting of a global store and local stores for each process. Let s denote the entire system state (i.e., contents of the global store and all local stores) during any computation of A. An action in a process of A is of the form,

$$s \ := \ f.s \quad \text{if } f.s \text{ is defined}$$

(Note that the action can only modify the contents of the global store and the local store of the appropriate process.)

Let B denote the system that implements A using asynchronous communication. In B, each process i has a local store—a cache—which will be denoted by s_i. Initially, all s_i's are equal to s (of A), i.e., all caches are initially coherent and each of them has the entire system state. Processes communicate using directed channels. Let $ch(i, j)$ denote the contents of the channel directed from process i to process j; $ch(i, j)$ is a sequence of function names. Initially all channels are empty. Channels are FIFO: messages sent along a channel are appended to its tail and the message that is removed from a channel is its head.

There are three kinds of actions for a process i in B.

- Actions to change s_i: an action of process i in A of the form

  $$s \ := \ f.s \quad \text{if } f.s \text{ is defined}$$

  is simulated by process i in B through

  $$s_i \ := \ f.s_i \quad \text{if } f.s_i \text{ is defined}$$

  Further, a message is sent by process i (see below).

- Sending messages: Any action in process i of the above form, that changes s_i, is accompanied by sending a message consisting of the function symbol "f" to all other processes; i.e., for all j, $ch(i, j)$ is updated by appending "f" to its end.

- Receiving messages: A process j receives a message "f" from a channel $ch(i, j)$ provided $f.s_j$ is defined (i.e., f can be applied to the current state of process j). In this case, s_j is changed to $f.s_j$. If $f.s_j$ is undefined, the message is not removed from $ch(i, j)$.

  If messages can be received from several channels (when the corresponding functions are defined in the present state) the choice of the message to be received next is arbitrary. However, we make the following fairness assumption about receiving messages: If f is at the head of an incoming channel of process j and $f.s_j$ remains defined forever, message f will be removed, eventually.

3.2 Proof of Correctness of the Implementation

We show that the implementation is correct by proving that every "property" of the original system, A, holds in the implementation, B. We consider the two prototypical classes of properties: p *unless* q, for safety, and $p \mapsto q$, for progress. Note, however, that predicates p, q are over the system state, s, whereas the implementation does not have a global system state. Therefore, we consider yet another system, C, which is same as the system B augmented with an auxiliary variable, s. The auxiliary variable, s, is modified as follows. The action in process i of system B

$$s_i \ := \ f.s_i \quad \text{if } f.s_i \text{ is defined}$$

is replaced by

$$s, s_i \ := \ f.s, f.s_i \quad \text{if } f.s_i \text{ is defined}$$

in System C. (We will show that $f.s$ is defined whenever $f.s_i$ is defined.) We will prove that, for predicates p, q over system state, s:

$$\begin{array}{l} p \textit{ unless } q \text{ in } A \ \Rightarrow \ p \textit{ unless } q \text{ in } C \\ p \mapsto q \quad \text{in } A \ \Rightarrow \ p \mapsto q \quad \text{in } C \end{array}$$

Note: The close correspondence between A, C, as given by the above relationships, does not directly imply that B is a faithful implementation of A. A more direct proof should show that any property of system A also holds in *any process* of system B, i.e., for any i (where $p.s_i$ denotes the predicate p in which s is replaced by s_i)

$$\begin{array}{l} p \textit{ unless } q \text{ in } A \ \Rightarrow \ p.s_i \textit{ unless } q.s_i \text{ in } B \\ p \mapsto q \quad \text{in } A \ \Rightarrow \ p.s_i \mapsto q.s_i \quad \text{in } B \end{array}$$ □

A Safety Property Relating s, s_i

We show that at every point in the computation of C, s may be obtained by applying the functions in the incoming channels of i to s_i in a certain order. Specifically, the following is an invariant for C.

invariant For process i there is an interleaving, γ, of $ch(j, i)$, for all j, $j \neq i$, such that

$$s = \gamma.s_i$$

Proof: Initially, the invariant holds with γ as the empty sequence. To prove that every change to s or s_i preserves the invariant, we observe that these variables may be changed by: applying an action in process i (which may change both s, s_i), applying an action in process j, $j \neq i$ (which may change s and $ch(j, i)$, but does not change s_i) or, process i receiving a message, g (thus changing s_i but leaving s unchanged).

Action in process i: Executing

$$s, s_i \ := \ f.s, f.s_i \qquad \text{if} \quad f.s_i \text{ is defined}$$

preserves the invariant, trivially, if $f.s_i$ is undefined. Otherwise, prior to the execution

	$f.s_i$ defined	, assume
	$\gamma.s_i$ defined	, $s = \gamma.s_i$ from the invariant
(1)	$f.\gamma.s_i = \gamma.f.s_i$	, Lemma 1 (f commutes with γ)
	$f.s$ defined	, from the above using $s = \gamma.s_i$

Thus assigning $f.s$ to s is legal whenever $f.s_i$ is defined. Next, observe that

$$\begin{array}{ll} & f.s \\ = & \{s = \gamma.s_i\} \\ & f.\gamma.s_i \\ = & \{\text{from (1)}\} \\ & \gamma.f.s_i \end{array}$$

Hence, the invariant holds with $f.s_i$, $f.s$ in place of s_i, s. No channel $ch(j,i)$, $j \neq i$, is changed by the action in process i; thus, γ remains an interleaving of the incoming channels of i.

Action in process j, $j \neq i$:

$$s, s_j \ := \ g.s, g.s_j \qquad \text{if} \quad g.s_j \text{ is defined}$$

has the effect that s may be changed; also $ch(j,i)$ is then extended by appending "g" to it.

$$\begin{array}{ll} & g.s \\ = & \{s = \gamma.s_i\} \\ & g.\gamma.s_i \end{array}$$

Hence, the invariant is satisfied with the sequence $g\gamma$, which is an interleaving of all $ch(j,i)$, in place of γ.

Receiving a message: Process i executes

$$s_i \ := \ g.s_i \qquad \text{if} \quad g.s_i \text{ is defined}$$

and removes g from a sequence $ch(j,i)$, $j \neq i$. Since for some interleaving γ, $\gamma.s_i$ is defined prior to the execution, $ch(j,i)$ is of the form $\beta' g$ and $g.s_i$ is defined, then using Corollary 1 of Theorem 2 we have $\gamma.s_i = \gamma'.g.s_i$. Therefore, the above action preserves the invariant, where s_i is replaced by $g.s_i$, and γ by γ'.

A Progress Property Relating s, s_i

The invariant, described above, tells us that the local caches—s_i's—are not "incoherent," i.e., by applying suitable interleavings of the messages from their incoming channels each process cache can be made equal to s (and hence, all process caches can be made equal). Our next result concerns progress: We show that system C evolves towards this ultimate coherent state. For instance, if no action in any process changes the state (the state is then known as a fixed point; see [2]) then the messages from the various channels will be consumed in such a way that, eventually, all caches will become identical. In the more general case, when computation never ceases, the caches may never become equal but every progress property of A can be shown to hold in C. The following progress property of C relates s and s_i, for any i.

For any function f in process i:

$$f.s \text{ defined} \ \mapsto \ f.s_i \text{ defined} \quad \text{in } C.$$

Proof(sketch): We will show that

$$(1) \quad s = S \ \mapsto \ (s_i = \alpha.S \ \wedge \ f \notin \alpha) \ \vee \ f.s_i \text{ defined},$$

where S is any (constant) system state, and α is some sequence of functions. Consider a state in C where $(s, s_i) = (S, S_i)$. From the invariant, there is an interleaving, γ, of the incoming channels such that

$$S = \gamma.S_i$$

We show that eventually all functions in γ would be removed from the channels and applied to s_i. If γ is empty, this is trivial. If $\gamma = \gamma' g$, where g is in the channel $ch(j,i)$:

$g.s_i$ is defined , because $\gamma.s_i$ is defined and $\gamma = \gamma' g$.

Also, $g.s_i$ remains defined as along as g is not received, because after applying a sequence of functions $\alpha, s_i = \alpha.S_i$ and g commutes with α (because the functions in α are not from process j). Since $g.S_i$ and $\alpha.S_i$ are defined, using Lemma 1, $g.\alpha.S_i$, i.e., $g.s_i$ is defined. Therefore, according to the fairness rule, g will be removed from $ch(j,i)$, i.e., eventually s_i will become $g.\alpha.S_i$, for some α. By similar reasoning, all functions in γ will be applied, i.e., eventually

$s_i = \delta.S_i$

where γ is a subsequence of δ. Now, $\gamma.S_i$ and $\delta.S_i$ are defined, and by applying (a slight generalization of) Theorem 2, $s_i = \alpha.\gamma.S_i$, for some sequence of functions α (where δ is an interleaving of α, γ). That is, eventually, $s_i = \alpha.S$, replacing $\gamma.S_i$ by S. Now if $f \notin \alpha$ then (1) is established. If $f \in \alpha$ then $f.s_i$ is defined at some point during this computation, and hence (1) is established.

Now, in C:

$f.S$ defined is stable
, S is constant
$s = S \ \wedge \ f.S$ defined $\mapsto \ (s_i = \alpha.S \ \wedge \ f \notin \alpha \ \wedge \ f.S$ defined$) \ \vee \ f.s_i$ defined
,(see [2], Sec. 3.6.3) PSP on the above and (1)
$s = S \ \wedge \ f.s$ defined $\mapsto \ (s_i = \alpha.S \ \wedge \ f \notin \alpha \ \wedge \ f.S$ defined$) \ \vee \ f.s_i$ defined
,rewriting the lhs
$s = S \ \wedge \ f.s$ defined $\mapsto \ (s_i = \alpha.S \ \wedge \ f.\alpha.S$ defined$) \ \vee \ f.s_i$ defined
, Lemma 1 on rhs: f commutes with α because $f \notin \alpha$ and the processes are loosely-coupled; also, $f.S$ and $\alpha.S$ are defined
$s = S \ \wedge \ f.s$ defined $\mapsto \ f.s_i$ defined
, the first disjunct in the rhs implies $f.s_i$ defined
$f.s$ defined $\mapsto \ f.s_i$ defined
, (see [2], Sec. 3.4.5) disjunction over all S

Preservation of Safety Properties by the Implementation

Theorem 7: *p unless q* in $A \ \Rightarrow$ *p unless q* in C
Proof: Consider an action a in A,
$a \ :: \ s := f.s$ if $f.s$ defined
From the *unless*-property in A,
$\{p \ \wedge \ \neg q\} \ a \ \{p \ \vee \ q\}$
The corresponding action c in C is, for some i,
$c \ :: \ s, s_i := f.s, f.s_i$ if $f.s_i$ defined
We have $\{p \ \wedge \ \neg q\} \ c \ \{p \ \vee \ q\}$ because,
$p \ \wedge \ \neg q \ \wedge \ \neg(f.s_i$ defined$) \ \Rightarrow \ p \ \vee \ q$,trivially
and $\{p \ \wedge \ \neg q \ \wedge \ f.s_i$ defined$\}$
$\Rightarrow$ (see the proof of the invariant: $f.s_i$ defined $\Rightarrow \ f.s$ defined)
$\{p \ \wedge \ \neg q \ \wedge \ f.s$ defined$\}$
c
$\{p \ \vee \ q$, because c has the same effect on s as action a when $f.s$ is defined$\}$

The other actions in C—sending and receiving messages—do not affect s, and hence, *p unless q* holds in C.

Preservation of Progress Properties by the Implementation

Theorem 8: $p \mapsto q$ in $A \Rightarrow p \mapsto q$ in C

Proof (sketch): It is sufficient to show that
p *ensures* q in $A \Rightarrow p \mapsto q$ in C

Then, by structural induction on the proof of $p \mapsto q$ in A, the result follows. The proof of p *ensures* q in A consists of (1) p *unless* q in A, and hence p *unless* q holds in C, from the last theorem, and (2) there is a function, f, in process i, say, in A such that

$\{p \wedge \neg q\} \;\; s := f.s \qquad$ if $f.s$ defined $\{q\}$

It follows from the above that

(3) $p \wedge \neg q \Rightarrow f.s$ defined

We will show that,

(4) $p \mapsto (p \wedge f.s_i$ defined$) \vee q \quad$ in C

(5) $p \wedge f.s_i$ defined $\mapsto q \quad$ in C

Then, using cancellation (see [2]; Sec. 3.6.3)

$p \mapsto q \quad$ in C.

Proof of (4): In C

p *unless* q	, see (1) above
$p \wedge \neg q$ *unless* q	, from the above: a fact about *unless*
$f.s$ defined $\mapsto f.s_i$ defined	, proven earlier
$p \wedge \neg q \wedge f.s$ defined $\mapsto (p \wedge \neg q \wedge f.s_i$ defined$) \vee q$	, PSP (see [2], Sec. 3.6.3) on the above two
$p \wedge \neg q \mapsto (p \wedge f.s_i$ defined$) \vee q$	, simplifying the lhs using (3) and the rhs using predicate calculus
$p \wedge q \mapsto q$	, implication (see [2], Sec. 3.6.3)
$p \mapsto (p \wedge f.s_i$ defined$) \vee q$	, disjunction on the above two (see [2], Sec. 3.6.3)

Proof of (5) (sketch):

Show that this is an *ensures* property in C.

3.3 Notes on the Implementation

Implementing Tightly-Coupled Processes

The implementation scheme, proposed in Section 3.1, eliminates cache-locking completely for loosely-coupled processes. Another way to interpret this result is that cache-locking is required only for accessing those variables on which the processes are tightly-coupled. As an example, consider a task-pool shared between a set of clients and a set of servers; a client may add a task to the task-pool and a server removes

an arbitrary task from this pool whenever it is idle. As we have shown for multi-input channels (Section 2.3), these processes are not loosely-coupled. However, locking is required only when reading, i.e., removing a task; no locking is required when adding tasks to the pool.

Our theory suggests exactly where locking of caches is required. In fact, we can design a compiler to tag those accesses that require cache-locking (see below).

Optimizing the Implementation

The implementation proposed in Section 3.1 can be optimized in several ways. A process that does not read a variable need not retain its value in its local cache; thus a local cache of a process need only contain the values of the shared variables and its local variables. Thus, a process that updates a shared variable has to send messages only to those processes that may read this variable. In many cases, variables are shared between two processes, and often one of the processes is the writer and the other is the reader. Then, the total storage and the communications can be reduced drastically.

The contents of messages are slightly more elaborate when processes include only the shared variable values and their own local variable values in the local caches. When a process sends a function name, f, to another process it has to include the relevant portion of its local store that will be required by the latter process in order to apply f. For instance, when process i adds the value of a local variable, d_i, to a shared variable c (see the example of counting, Section 2.3) then it sends a message (to any process that may read c) that includes both the addition function *and* the value of d_i.

There are many ways to encode the functions that are sent as parts of messages. Clearly, if all functions in a channel are identical they may be omitted and only the relevant parameters may be transmitted. Further, two successive functions f, g to be sent along a channel may be replaced by sending a single function, h, where $h = gf$. This is particularly useful in the case when multiple updates can be replaced by a single update, as in incrementing an integer variable k times, that can be replaced by adding k to it.

Compiler-Check for Loose-Coupling

We have not yet specified how the check for loose-coupling should be carried out. It is clearly impractical to design a system and then check its processes for loose-coupling. Instead, variables to be shared should be drawn only from certain specified types and structures (with a set of operations that are known to commute). A compiler can then check for loose coupling and generate code for asynchronous message transmission following shared variable updates. For instance, for a shared variable of type integer, addition and subtraction commute whereas addition and multiplication do not; a compiler check of the shared variable type and its access functions can in many cases establish that processes are loosely-coupled.

4 Summary

Message communication has played a central role in the development of asynchronous computing. The traditional model of message communication involves a channel directed from a sender to a receiver (that are engaged in point-to-point communication). We have proposed that certain types of shared variables may be viewed as "generalized channels." If the different operations on these variables commute, then this form of interaction possesses most of the desirable properties of message communication. In particular, systems can then be designed without explicit checks for interference, and they can be implemented using point-to-point asynchronous message communication. These are prerequisites, we believe, for developments of large-scale multiprocessor systems.

Acknowledgements I am grateful to Ernie Cohen and J. R. Rao for several discussions; their efforts at extending this theory are appreciated. The Austin Tuesday Afternoon Club, under the direction of E. W. Dijkstra, read parts of this draft and helped me reformulate some of the results of Section 2.4.

References

[1] Arvind, R. S. Nikil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[2] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[3] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett, Boston, 1992.

[4] E. Cohen. *Modular Progress Proofs of Concurrent Programs*. PhD thesis, The University of Texas at Austin, August 1992.

[5] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[6] H. Gaifman, M. J. Maher, and E. Shapiro. Replay, recovery, replication and snapshot of nondeterministic concurrent programs. Technical report, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, 76100, Israel, July 1990.

[7] A. Kleinman, Y. Moscowitz, A. Pnueli, and E. Shapiro. Communication with directed logic variables. In *Proc. POPL 91*, (to appear).

[8] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of scalable shared-memory multiprocessors: The dash approach. In *Proc. ACM, Compcon*, February 1990.

[9] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proc. IEEE, 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[10] J. Misra. Specifying concurrent objects as communicating processes. *Science of Computer Programming*, 14(10):159–184, October 1990.

[11] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(1):319–340, 1976.

[12] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[13] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA*, pages 218–231, January 1990.

[14] Prasad Vishnubhotla. Concurrency and synchronization in the alps programming language. Technical Report 56, Ohio State University, 1989.