A Class of Synchronization Systems that Permit the Use of Large Atomic Blocks

Paolo A.G. Sivilotti [*]
Department of Computer and Information Science
The Ohio State University
2015 Neil Ave., Columbus, OH 43210-1277, USA

Abstract

This paper revisits the formal justification of a common practice used in formal and informal reasoning about distributed systems: considering certain sections of code to be implicitly atomic. This practice is extremely useful as it allows distributed and concurrent programs to be developed, tested, and verified with large atomic blocks, yet executed with a much finer granularity of parallelism for efficiency. We expose the elements on which this practice is based and characterize the synchronization systems for which this practice is valid. Unlike previous justifications for this practice, our approach is based on a weakest precondition semantics. Owing to the generality of our model of computation, the result is applicable to both distributed-memory and shared-memory systems.

1 Introduction

An atomic block of code is a sequence of actions with which other actions are not interleaved. Atomic blocks are useful in distributed computations because they allow us to reason about a section of code in isolation, without interference from other concurrent threads of execution. Pragmatically, atomic blocks of code are useful because they reduce the number of interleavings that must be considered. It is no surprise, then, that large atomic blocks are helpful for reasoning.

Atomic blocks of code are limited in size by the synchronization structure of the application. In particular, a synchronization operation is not permitted to suspend inside an atomic block, since this would result in deadlock. In effect, a suspension must correspond to the termination of the enclosing atomic block.[1] A common and intuitive practice when reasoning about distributed systems is to consider a blocking receive operation as the beginning of an atomic block, followed by all the sends and local actions performed by that process. We precisely characterize a class of synchronization systems where this practice is valid. This class includes such common paradigms as distributed memory message-passing over first-in first-out channels with blocking receive, and shared memory unbounded semaphores. Our formal characterization allows us to observe that this practice also applies to some less intuitive paradigms, such as shared-memory monotonic counters.

The rest of the paper is organized as follows: Section 2 presents a motivating example of how the implicit creation of atomic blocks simplifies reasoning about an application; Section 3 points to some related work, contrasting it with our own approach; Section 4 defines our generic model of computation; Section 5 introduces notation and terminology concerning traces; Section 6 introduces notation and terminology concerning computations; Section 7 states and proves the theorem and corollary that are the central results of this paper; Sec-

[*]This work was partially funded by an IBM Cooperative Computer Science Fellowship.

[1] For example, a Java **wait** operation inside a synchronized block releases the associated lock, terminating the atomic block.

tion 8 illustrates the application of the theorem to several real synchronization systems; and Section 9 concludes.

2 Motivating Example: The Gossip Algorithm

The gossip algorithm is a simple implementation of a diffusing computation [4, 12]. It can be used to synchronize a collection of processes. Informally, the gossip algorithm uses an expanding wave to include all processes, followed by a contracting wave to signal that all processes were included.

The algorithm for a single process in this computation is given in Figure 1. Sends are denoted by "!" and receives by "?". Each process has a set N of neighbouring processes with which it can exchange messages. In the first line of the program, the process waits to receive a message from any neighbour. The sender of this message (once it is received) becomes this process's parent. In line 2, all neighbours that are not the parent are sent a message (*i.e.* the "gossip"). In line 3, the process waits to hear a reply back from each of the neighbours from line 2. Once all these replies have been received, the process replies to its parent.

```
1   $?\mathit{parent}$
2   $(\forall c : c \in N \wedge c \neq \mathit{parent} : !c)$
3   $(\forall c : c \in N \wedge c \neq \mathit{parent} : ?c)$
4   $!\mathit{parent}$
```

Figure 1: Gossip Algorithm

The computation begins when a special process, known as the initiator, sends a message to a process in the collection. This message is passed along from parent to children until all processes have been included in the gossip. Only once all process have been included can the second phase begin, where acknowledgements are sent from children back to parents.

One of the difficulties in reasoning about this computation is that many parents can be sending messages to many (even the same) children at the same time. Consider a process c with two neighbours p_1 and p_2, both of which have heard the gossip. There are two messages on the way to c, one from each of p_1 and p_2. Depending on which arrives first, one of these processes will be c's parent. Can the other message be received as an acknowledgement from a child? What if there is only one message on the way, as the other potential parent hasn't yet sent the gossip on? Is this implementation correct despite this nondeterminism? Perhaps surprisingly, this distinction turns out not to matter for the correctness of the algorithm. Seeing that this is so, however, requires some careful thought.

Now consider a modification to the original gossip algorithm, given in Figure 2. The modification is to create an atomic block, denoted by $\langle ... \rangle$. This block allows us to treat the reception of the message from the parent and the sending of messages out to other neighbours as a single atomic action.

```
1   $\langle$
2       $?\mathit{parent}$
3       $(\forall c : c \in N \wedge c \neq \mathit{parent} : !c)$
4   $\rangle$
5   $(\forall c : c \in N \wedge c \neq \mathit{parent} : ?c)$
6   $!\mathit{parent}$
```

Figure 2: Gossip Algorithm With Atomic Block

In general, larger atomic blocks allow us to state stronger invariants. This is because the states inside an atomic block are not observable, and so can violate the invariant, so long as its validity is restored prior to termination of the block. For example, an invariant of the modified gossip algorithm (but not of the original version) is that all processes that have a parent have sent messages out to all their other neighbours.

A general heuristic is that stronger invariants simplify the task of establishing the correctness of a computation. Indeed, it is relatively easier to visualize the progress of the modified gossip algorithm as processes are added as leaves to a growing tree of parents, where the act of becoming a leaf also corresponds to the distribution of messages to all other neighbours.

Interestingly, this modified program is equivalent to the original. We can therefore reason about the correctness of the original by reasoning about the correctness of the modified version. This process of increasing the size of atomic blocks can be extremely useful, therefore, if it preserves the semantics of the program. This paper addresses the following question: Under what conditions is such a modification valid?

3 Related Work

The problem of reasoning about a concurrent program by considering a program with larger atomic blocks was first formally addressed by Lipton [10]. His approach was to classify actions as "right movers" and "left movers", which is the core of the proof of our theorem as well. Lipton's reduction method considers partial correctness only, however, while we use total correctness. Also, Lipton's main result is for "PV parallel programs" (*i.e.*, semaphore-based programs), whereas our result applies more generally to any synchronization system with a pair of synchronization primitives. We use our result to analyze a collection of synchronization paradigms, including semaphores.

Lipton's results were extended by Doeppner [6] to a larger class of safety properties, then further generalized by Lamport and Schneider [8] to the general class of safety properties, namely invariants. They generalize the notion of atomicity by introducing a predicate $\varepsilon(A)$ (for "external") that holds when control is outside of an atomic action A. This predicate characterizes when an invariant is required to hold. Lamport [7] further generalizes the result to include progress properties by introducing fairness requirements.

Our approach differs from these results in several ways. First, we use a weakest precondition [3, 5] semantics, while the work cited above uses a next-state transition system semantics. Our approach is therefore more general as we consider actions that nondeterministically may or may not terminate, a behaviour which was not examined by these authors.[2] The distinction between our main theorem and its corollary, for example, is lost if such nondeterministic actions are not considered (and hence this distinction did not appear in the work cited above). Second, we base our approach on total correctness as a semantic definition of a program, rather than safety properties. Although this restricts our results to programs with a precondition and postcondition specification (as opposed to reactive programs [11] such as operating systems), it allows us to use the traditional refinement calculus [1, 14, 13] as the foundation for our proof.

Our work is perhaps closest in spirit to Back's extension of the refinement calculus to include concurrent action systems [2]. He presents sufficient conditions for permitting the refinement of a concurrent algorithm. Our work is similar to this extension of the refinement calculus in that both use weakest precondition semantics, both consider nondeterministic actions that may or may not terminate, and both preserve total correctness. The focus of the refinement calculus, however, is on algorithms and on the development of programs through a series of correctness-preserving transformations. Our emphasis here, on the other hand, is on synchronization systems and on the development of conditions on synchronization primitives that guarantee the correctness of a specific refinement step. We use these conditions to examine a collection of common (and some less common) synchronization paradigms.

4 State, Shared State, and Communication

A distributed system consists of a collection of processes. Each process has an independent thread of control and local data that is not visible outside the process. Communication and synchronization is achieved through some element of state that is shared by two or more processes. We do not further qualify this shared state. This is a very general model of process communication, capturing shared memory primitives (*e.g.* locks, semaphores, and single-assignment variables) as well as distributed

[2] It is possible to model such behaviour in next-state transition sytems by the introduction of a special state, often denoted ∞ or $\perp$. The introduction of such a state, however, complicates the rule for composition and was not done in the cited work.

memory primitives (*e.g.* ordered and unordered channels with blocking and nonblocking receives). Nor do we subdivide this shared state; a collection of processes has a single pool of shared state. For example, the sending of a message on a channel between two particular processes is viewed to modify the state of the message-passing layer of the entire system. This approach does not strengthen the result of our theory, but it does permit a unified presentation.

The state of the system is the union of the states of the individual processes and the shared state. For example, the state of a message-passing system is the local state of each process and the state of the channels between processes.

An action that reads values from or writes values to the shared state is called a *synchronization action.* Such an action has an associated enabling predicate that the shared state must satisfy in order for the action to complete. The enabling predicate of a synchronization action S can be defined using weakest precondition semantics [3, 5] as $wp.S.true$. In the trace of the computation, the state immediately before the execution of a synchronization action must meet that action's enabling condition. For example, for a blocking receive on a channel, the enabling condition is that there be a delivered message in the channel.

Many synchronization systems consist of a pair of such actions.[3] For example, a message-passing system has send and receive operations, a lock-based system has acquire and a release operations, and a semaphore-based system has increment and decrement operations. In order to be considered a synchronization system, at least one of these actions must have an enabling predicate that is not identically $true$. We restrict our attention to such systems where one action in the pair is always enabled, while the other is not. The action that is always enabled we call *send* (and denote by "!"), while the action that is not necessarily enabled we call *receive* (and denote by "?"). An action that does not access the shared state we call *local* (and denote by "#"). For the systems mentioned above, "!" corresponds to sending a message, releasing a lock, or incrementing a semaphore, respectively.

[3]Although this is not strictly necessary. Consider a token-swap operation that is enabled precisely when either the process or the shared state contains the token. It has the effect of flipping ownership of the token. This system is equivalent in expressive ability to a message-passing synchronization system.

5 Traces and Refinement

A trace is a sequence of actions. Each action is performed by a process in the computation. An action can either access the shared state or it can be entirely local. Because we have restricted attention to systems with only two synchronization commands (send and receive), an action that accesses the shared state is either a send or a receive. Thus, a trace consists only of sends, receives, and local actions.

Sends, receives, and local actions executed by process p are denoted $p!$, $p?$, and $p\#$ respectively. For a collection of processes S, the trace of a computation is projected to a string in the language:

$$\mathcal{L} = (\cup p : p \in S : p? \cup p! \cup p\#)^*$$

As in [9], we use $\cup$ to separate alternative items and * to mean a (possibly empty) sequence of items.

A trace maps the initial system state to a set of possible final system states (the effect of an action, hence a trace, can be nondeterministic). We use weakest preconditions [3, 5] to define trace semantics. Following the notation of [5], $wp.t.Q$ denotes the weakest precondition (*i.e.*, the set of states from which trace t is guaranteed to terminate in a state satisfying Q), while $wlp.t.Q$ denotes the weakest liberal precondition (*i.e.*, the set of states from which trace t is guaranteed to either not terminate or terminate in a state satisfying Q). A trace t is said to be *refined by* a trace t' if and only if any specification satisfied by t is satisfied by t' as well [13]. (Conversely, t' is said to *refine* t.) Refinement is therefore a "correctness preserving" process [14]. Denoting trace refinement by $\sqsubseteq$ (read "refined by"), this can be expressed more formally as:

$$t \sqsubseteq t' \equiv (\forall Q :: [wp.t.Q \Rightarrow wp.t'.Q])$$

We make use of [ ] as "everywhere brackets" [5], denoting universal quantification over the state space.

An immediate consequence of this definition is that trace refinement is transitive:

$$(r \sqsubseteq s) \wedge (s \sqsubseteq t) \Rightarrow (r \sqsubseteq t)$$

One way to prove that two traces are related by refinement is to find a sequence of action swaps that permutes one trace into the other. By transitivity, if each action swap yields a refined trace, the final trace is a refinement of the initial one. We will use this technique to establish our theorem.

6 Computations and Refinement

A computation is defined by a set of traces. Given an initial state, a computation is modeled as the nondeterministic choice of one of the traces for execution from this initial state. Thus, a computation maps an initial state to a set of final states (the union of the possible final states given by the traces of the computation).

A computation C is said to be refined by a computation C' if and only if every trace in C' is a refinement of a trace in C. Like trace refinement, we denote computation refinement by $\sqsubseteq$. This symbol overloading is justified by the following definition:

$$C \sqsubseteq C' \equiv (\forall t' : t' \in C' : (\exists t : t \in C : t \sqsubseteq t'))$$

See Figure 3 for an illustration of computation refinement.

Refinement is a useful property because it represents replaceability. Informally, anywhere C can be used, C' can be used instead. Any specification met by C is also met by C'. This is especially useful if C is easier to reason about than C'. In that case we can prove that C' meets a specification by establishing that C meets that specification.

Operationally, the traces of a computation are formed by the nondeterministic selection of an action for execution from a pool of enabled commands. Any action in the pool can be selected. Furthermore, every action in the pool is selected in some trace. That is, if some subsequence of actions leads to a state satisfying $wp.X.true$, there must be a trace in the computation with that subsequence as a prefix, followed by X as the next action.

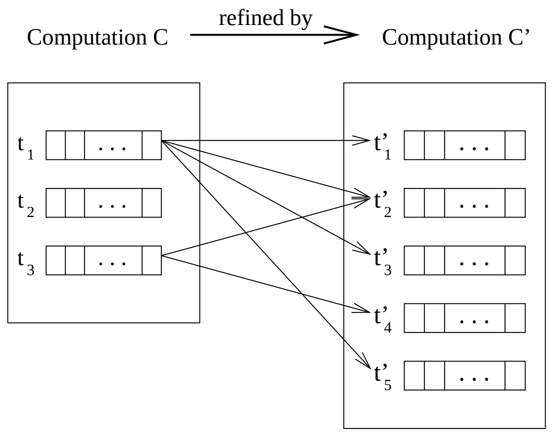

Figure 3: Graphical Representation of a Computation Refinement

7 Theorem

7.1 Definitions of Send and Local Actions

We begin with some properties that follow immediately from our model of computation. In fact, these properties can be taken as definitions of the associated actions.

Sends are always enabled.

$$[wp.p!.true \equiv true] \qquad (1)$$

Here we introduce the convention that any unbound dummy is implicitly universally quantified over the natural domain. The equation above holds for all processes p in the computation.

Local actions are not externally visible. Let X denote any of the three possible actions (send, receive, or local action, denoted $!$, $?$, and $\#$ respectively).

$$(\forall q : q \neq p : [wp.(pX; q\#).Q \equiv wp.(q\#; pX).Q]) \qquad (2)$$

Here we introduce the symbol ; to represent sequential composition of actions.

7.2 Nomenclature and Properties

Before stating the theorem, the following nomenclature is introduced to facilitate the presentation.

Atomic Computations. We wish to establish the conditions under which a collection of actions by a process can be treated as an atomic block. In particular, we are interested in the sequence of actions by a process that either:

1. precede the first receive of that process, or
2. begin with a receive and are followed by some number of sends and local actions by that same process.

A trace where these sequences are atomic (*i.e.* not interleaved with other actions) we will call an *atomic trace*. See Figures 4 and 5 for examples of atomic and nonatomic traces. A computation consisting only of atomic traces is called an *atomic computation.*

Receive Properties. A receive is *enabled-stable* if, once enabled, the execution of a send on a different processor does not make it unenabled. This condition says nothing about what happens when a *receive* on a different processor is executed. More formally, this property is defined by:

$$\begin{array}{l}(\forall\, q \,:\, q \neq p \\ \quad :\; [\mathit{wp}.p?.\mathit{true} \;\Rightarrow\; \mathit{wp}.(q!;p?).\mathit{true}]\;)\end{array} \tag{3}$$

Another useful (and common) property of receive actions is that when they are swapped with a prior send action, they return the same or stronger result if they return anything at all. Such a receive is called *send-monotonic.* More formally, this property is defined by:

$$\begin{array}{l}(\forall\, q \,:\, q \neq p \\ \quad :\; [\mathit{wp}.(q!;p?).Q \;\Rightarrow\; \mathit{wlp}.(p?;q!).Q]\;)\end{array} \tag{4}$$

This property can be weakened by requiring that it hold only when the anticipated receive terminates. Such a receive is called *weakly send-monotonic.* More formally, this property is defined by:

$$\begin{array}{l}(\forall\, q \,:\, q \neq p \\ \quad :\; [\mathit{wp}.p?.\mathit{true} \wedge \mathit{wp}.(q!;p?).Q \\ \qquad\qquad \Rightarrow\; \mathit{wlp}.(p?;q!).Q]\;)\end{array} \tag{5}$$

Send Property. The send operation is said to *commute* exactly when two sends on different processes can be exchanged in any trace with no effect. More formally, this property is defined by:

$$\begin{array}{l}(\forall\, q \,:\, q \neq p \\ \quad :\; [\mathit{wlp}.(p!;q!).Q \;\Rightarrow\; \mathit{wlp}.(q!;p!).Q]\;)\end{array}$$

The symmetry in this equation allows us to use the equivalent expression:

$$\begin{array}{l}(\forall\, q \,:\, q \neq p \\ \quad :\; [\mathit{wlp}.(p!;q!).Q \;\equiv\; \mathit{wlp}.(q!;p!).Q]\;)\end{array}$$

7.3 Theorem, Corollary, and Proofs

Theorem. Any computation in a synchronization system is a refinement of an atomic computation if that synchronization system has enabled-stable and weakly send-monotonic receives, as well as commuting sends.

Proof. Given an arbitrary computation, G, we must establish the existence of an atomic computation, A, such that every trace in G is a refinement of some trace in A.

Consider a trace $g \in G$. If g is not an atomic trace, then there must exist a pair of actions of the form $(iX;j!)$ or $(iX;j\#)$, where $i \neq j$ and X is one of "!", "?", or "#". For each of these cases, we must show that swapping these two actions yields a new trace, a, such that:

1. g refines a, and
2. a is also an element of the computation G.

For any finite trace, there exists a sequence of such action swappings that produces an atomic trace. In particular, consider the sequence of swappings that builds an atomic trace from left

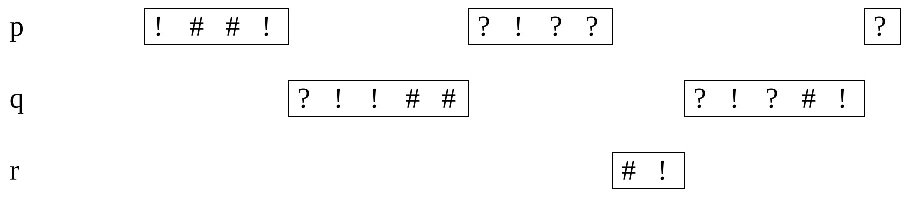

Figure 4: An Atomic Trace

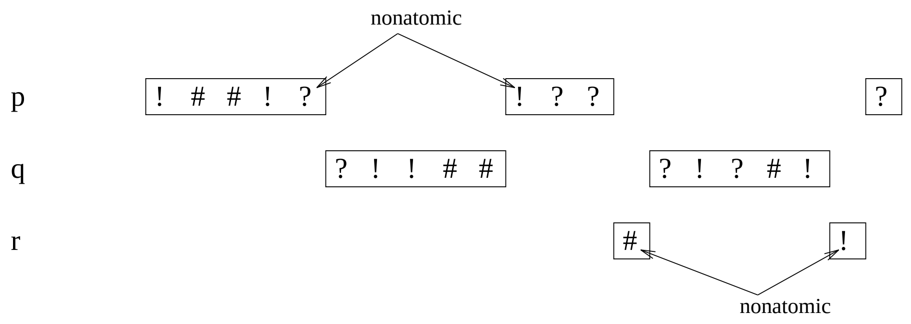

Figure 5: A Nonatomic Trace

to right. That is, the first atomic block is constructed by swapping to the left, all sends and local actions, for a given process, that occur prior to that process's next receive. Next, the second atomic block is constructed in the same way. Because the trace is finite, the required sequence of swaps is also finite and this procedure yields an atomic trace.

The first obligation corresponds to establishing (for all Q)

$$[wp.g.true \wedge wp.a.Q \Rightarrow wp.g.Q]$$

That is, since g is a trace in the computation, it must be enabled. Hence we can assume that g is enabled in proving that a is refined by g.

The second obligation corresponds to establishing

$$[wp.g.true \Rightarrow wp.a.true]$$

That is, if g is enabled, a is enabled as well.

We meet these obligations for each case for g, of which there are six to consider:

1. $(i\#; j!)$,
2. $(i!; j!)$,
3. $(i?; j!)$,
4. $(i\#; j\#)$,
5. $(i!; j\#)$, and
6. $(i?; j\#)$.

Case 1.

$$\begin{array}{ll} & wp.(j!; i\#).Q \\ \equiv & \quad \{ \text{ Definition of local actions, (2) } \} \\ & wp.(i\#; j!).Q \end{array}$$

The second obligation follows, with $Q \equiv true$.

Case 2.

$$\begin{array}{ll} & wp.(j!; i!).Q \\ \equiv & \quad \{ \text{ Definition of } wp \ \} \\ & wp.(j!; i!).true \ \wedge \ wlp.(j!; i!).Q \\ \equiv & \quad \{ \text{ Definition of ; } \} \\ & wp.j!.(wp.i!.true) \ \wedge \ wlp.(j!; i!).Q \\ \equiv & \quad \{ \text{ Sends are always enabled, (1) } \} \\ & wlp.(j!; i!).Q \\ \equiv & \quad \{ \text{ Sends commute, (7.2) } \} \\ & wlp.(i!; j!).Q \\ \equiv & \quad \{ \text{ Sends are always enabled, (1) } \} \end{array}$$

$$\begin{array}{ll} & wp.i!.(wp.j!.true) \wedge wlp.(i!;j!).Q \\ \equiv & \quad \{ \text{ Definition of ; } \} \\ & wp.(i!;j!).true \wedge wlp.(i!;j!).Q \\ \equiv & \quad \{ \text{ Definition of } wp \ \} \\ & wp.(i!;j!).Q \end{array}$$

Again, the second obligation follows, with $Q \equiv true$.

Case 3.

$$\begin{array}{ll} & wp.i?.true \wedge wp.(j!;i?).Q \\ \Rightarrow & \quad \{ \text{ Weak send-monotonicity, (5) } \} \\ & wp.i?.true \wedge wlp.(i?;j!).Q \\ \equiv & \quad \{ \text{ Sends are always enabled, (1) } \} \\ & wp.i?.(wp.j!.true) \wedge wlp.(i?;j!).Q \\ \equiv & \quad \{ \text{ Definition of ; } \} \\ & wp.(i?;j!).true \wedge wlp.(i?;j!).Q \\ \equiv & \quad \{ \text{ Definition of } wp \ \} \\ & wp.(i?;j!).Q \end{array}$$

Now our second obligation:

$$\begin{array}{ll} & wp.(i?;j!).true \\ \equiv & \quad \{ \text{ Definition of ; } \} \\ & wp.i?.(wp.j!.true) \\ \equiv & \quad \{ \text{ Sends are always enabled, (1) } \} \\ & wp.i?.true \\ \Rightarrow & \quad \{ \text{ Receive is enabled-stable, (3) } \} \\ & wp.(j!;i?).true \end{array}$$

Cases 4–6.

$$\begin{array}{ll} & wp.(j\#;iX).Q \\ \equiv & \quad \{ \text{ Definition of local actions, (2) } \} \\ & wp.(iX;j\#).Q \end{array}$$

The second obligation follows, with $Q \equiv true$.

Thus, each action swapping yields a trace refined by the original trace in G and present in G as well. Thus G is a refinement of the computation with no "out of order" pairs; that is, the corresponding atomic computation.

□

Corollary. Any computation in a synchronization system is a refinement of an atomic computation if that synchronization system has enabled-stable and send-monotonic receives, as well as commuting sends.

Proof. Any system that is send-monotonic is also weakly send-monotonic. Hence the theorem can be applied.

□

Aside. The proof of the theorem actually establishes the *equivalence* of the two computations. That is,

$$(A \sqsubseteq G) \wedge (G \sqsubseteq A)$$

The second conjunct follows from the observation that the traces in A are a subset of the traces in G. It is the first conjunct, however, in which we are usually interested.

7.4 Equivalence of Strong and Weak Send-monotonicity

The properties of send-monotonicity and weak send-monotonicity are closely related. Send-monotonicity is perhaps the easier of the two to verify, as it has fewer conjuncts. In this sense, the corollary is easier to apply than the theorem. The requirements stipulated in the corollary are stronger than those given in the theorem, so establishing the corollary can be applied suffices.

Unfortunately, the converse is not true. Establishing that the corollary requirements are not met does not say anything about the applicability of the theorem. In particular, establishing that receives are not send-monotonic does not establish whether or not the theorem can be applied. Fortunately, in the restricted (but common) case of *deterministically terminating* receive actions, the two properties are equivalent. Thus, for deterministically terminating receive actions, if a system does not have (strongly) send-monotonic receives, the theorem cannot be applied.

A receive is said to be deterministically terminating exactly when any state from which it is not guaranteed to terminate, is a state from which it is guaranteed to not terminate. More formally, this property is defined by:

$$[\neg wp.p?.true \Rightarrow wlp.p?.false] \qquad (6)$$

The converse is true for nonmiraculous receives; that is, receives for which $wp.S.false$ is false. (See Appendix A for the proof). For such receives, the above definition of deterministic termination can be rephrased as:

$$[\neg wp.p?.true \equiv wlp.p?.false] \qquad (7)$$

For any given global state, a deterministically terminating receive is either enabled (*i.e.* guaranteed to terminate), or guaranteed not to terminate.

The following result formally states the equivalence of send-monotonicity and weak send-monotonicity for deterministically terminating receives.

Equivalence. For deterministically terminating receives:

$$\begin{array}{ll} & [wp.(q!;p?).Q \Rightarrow wlp.(p?;q!).Q] \\ \equiv & [wp.p?.true \wedge wp.(q!;p?).Q \\ & \qquad\qquad \Rightarrow wlp.(p?;q!).Q] \end{array}$$

Proof. The forward direction is true trivially. It remains to be shown, then, that send-monotonicity follows from weak send-monotonicity when the receive is deterministically terminating.

$$\begin{array}{ll} & wp.(q!;p?).Q \\ \Rightarrow & \quad \{ \text{ Weak send-monotonicity, (5) } \} \\ & wlp.(p?;q!).Q \vee \neg wp.p?.true \\ \Rightarrow & \quad \{ \text{ Deterministic. terminat., (6) } \} \\ & wlp.(p?;q!).Q \vee wlp.p?.false \\ \Rightarrow & \quad \{ \text{ Monotonicity of } wlp \ \} \\ & wlp.(p?;q!).Q \vee wlp.p?.(wlp.q!.false) \\ = & \quad \{ \text{ Definition of ; } \} \\ & wlp.(p?;q!).Q \vee wlp.(p?;q!).false \\ \Rightarrow & \quad \{ \text{ Property of } wlp \ \} \\ & wlp.(p?;q!).(Q \vee false) \\ = & \quad \{ \text{ Identity of } \vee \ \} \\ & wlp.(p?;q!).Q \end{array}$$

□

8 Examples of Theorem Application

8.1 Shared Memory

In this section we describe some shared memory synchronization paradigms to which our theorem applies. Recall from our model of computation that only synchronization actions can access the shared state. Thus, we do not permit sharing of general state that is accessible through actions that are not a send or receive.

Single-assignment Variables. A single-assignment variable is a shared variable that can be written (assigned) at most once. It is initially undefined, and a process that attempts to read an undefined single-assignment variable suspends execution until the variable is written. Writing and reading these variables correspond to send and receive actions in our model of computation.

These actions meet the requirements for the corollary:

receive enabled-stable:
Once defined, a single-assignment variable cannot be made undefined. Thus, a read operation, once enabled, remains enabled.

receive send-monotonic: Any state in which a write followed by a read terminates and returns a value is also a state in which the read followed by the write either does not terminate (*i.e.* they involve the same variable) or returns the same value (*i.e.* they involve different variables).

sends commute: Two writes can be executed in either order. In a correct program, these writes must be to different variables and so cannot be interfering.

Single-assignment variables are extremely restrictive. The required send-monotonicity property for our theorem allows an anticipated read to return the same or stronger result (if it is enabled). An anticipated read with single-assignment variables, however, returns exactly the same result (if it is enabled). This observation suggests a weaker notion of single-assignment variables, we call monotonic-assignment variables, or monotonic variables for short.

Monotonic Variables. Like single-assignment variables, monotonic variables are initially undefined. A process that attempts to read such an undefined variable suspends. Unlike single-assignment variables, however, monotonic variables can be assigned multiple times. Each assignment must be guaranteed to increment the value of the variable. Also unlike single-assignment variables, the read operation of monotonic variables is not deterministic. Reading a monotonic variable returns a value equal to or less than the value of the variable.

Such a monotonic variable meets the requirement of the corollary:

receive enabled-stable: Once defined, a monotonic variable cannot be made undefined. (In this respect, it is similar to a single-assignment variable). Thus, a read operation, once enabled, remains enabled.

receive send-monotonic: If a read is now anticipated, then the result (if the read terminates) is stronger, since the range of values that can be returned is smaller.

sends commute: The write operations that increment the value of the monotonic variable can be performed in either order.

Notice that the last requirement prevents monotonic variables from having, for example, both an increment and a double operation, since these two operations, though both monotonic, do not commute.

Semaphores. A semaphore is a shared nonnegative integer counter. A process can increment the value of the semaphore (the send action). This action never suspends. A process can also decrement the value of the semaphore (the receive action). This action is enabled only when the value of the semaphore is strictly greater than 0.

Unlike the previous examples, once a receive action is enabled, it is not guaranteed to remain so until it is executed. If the value of a semaphore is 1, all the decrement (receive) actions are enabled. But once one of them completes, the rest are no longer enabled.

Nevertheless, the send and receive actions clearly meet the requirements for the theorem:

receive enabled-stable: If the decrement action is enabled, the value of the semaphore is greater than 0. Performing an increment on the semaphore does not make the decrement unenabled.

receive send-monotonic: When an increment followed by a decrement is guaranteed to terminate in some set of states, then either the decrement followed by the increment would terminate in the same set of states or it would not terminate.

sends commute: Since addition commutes, the order of increment actions is not important.

The requirement for send-monotonicity forces us to restrict decrement actions to those that are not sensitive to the particular value of the semaphore. In particular, we disallow a decrement action that returns the current value of the semaphore. On the other hand, we do allow a decrement action that returns nondeterministically any value less than or equal to the current value. Such a decrement is still send-monotonic.

8.2 Distributed Memory

In this section we describe some distributed memory synchronization paradigms to which our theorem applies.

Blocking Receives. A simple paradigm for communication in distributed-memory systems is that of message passing over directed point-to-point ordered first-in first-out channels. A send action appends a message to a channel, and a receive action removes a message from the channel. A receive action is enabled exactly when there is at least one message that has been sent but not received.

These send and receive actions meet the requirements for the corollary:

receive enabled-stable: If a receive action is enabled, there is a message that has been sent but not received. Another send action cannot remove this message, so the receive remains enabled.

receive send-monotonic: If a receive following a send terminates returning a particular message, then anticipating the receive causes it either to not terminate or to return the same message, since channels are first-in first-out.

sends commute: Channels are point-to-point, so sends from different processes yield the same global state regardless of the order in which they are executed. Notice that this is not the case if there is a merge operation at the destination.

The channels described above are not implementable in practice because they are instantaneous: there is no delay between sending a message and it being delivered. A message-passing layer that has arbitrary but finite delay can be modeled as another process in the computation that is responsible for shuttling messages from origin to destination. This process performs a fair merge implicitly. Such a system meets the requirements for the corollary.

A slightly weaker model that still meets the requirement of the corollary is that of unordered channels with blocking receive. In this case the receive returns any message that has been delivered but not yet received.

Weak Probes. A weak probe is an action that queries the message-passing layer. It returns false when there is no delivered message to receive. Otherwise, it returns true or false nondeterministically. This nondeterminism distinguishes this action from a traditional probe.

These probes (and the usual message-passing sends) meet the requirement for the corollary:

receive enabled-stable: Weak probes are always enabled. Hence, they are enabled-stable.

receive send-monotonic: If a send followed by a weak probe is guaranteed to return a value, that value must be false. There must be no message pending. In this case, reversing the order of the actions has no effect.

sends commute: This argument is the same as given above.

8.3 Failure Cases

In this section we explore the limits of the theorem by illustrating three synchronization systems for which it does not apply. For each example we specify how the system fails to meet the necessary requirements. We also describe how a program could be written to detect the difference between a general and an atomic computation (thus proving that the former does not refine the latter).

```
p :  ! x to q          q :  repeat
     ! y to q                    skip
                            until (? x from p )
                            if (? y from p )
                                 then ok
                                 else error
```

Figure 6: Program with General Probes

General Probes. General probes give more information than the weak probes described above. In addition to returning false when there is no pending message, they are also guaranteed to return true when there is a pending message. These probes are not send-monotonic because there is a state such that a send followed by a probe is guaranteed to return true, whereas reversing these actions returns false; namely, the state where the channel is empty (for a send and a probe on this empty channel).

On the other hand, general probes are clearly deterministically terminating. Therefore, send-monotonicity is equivalent to weak send-monotonicity. Since probes are not send-monotonic, they are not weakly send-monotonic. Hence the theorem cannot be applied.

Figure 6 gives a program that illustrates the difference between atomic and general computations in this system. This program is guaranteed to terminate in the state *ok* for atomic computations, but not for general computations.

Stack-based Channels. Channels are commonly modeled either as queues or as sets. For the former, a receive action removes a message from the head of the queue, and for the later a receive action removes an arbitrary element of the set. We could envision, however, a channel modeled as a stack, where receive actions remove messages from the top, where they are also inserted by send actions.

Such a system, however, would not satisfy the requirements of the theorem. In particular, the receive is not send-monotonic. Due

```
p :  ! 1 to q        q :  ? x from p
     ! 2 to q             if (x = 2)
                               then ok
                               else error
```

Figure 7: Program with Stack-based Channels

```
p :  ! x             q :      ? x → error
     ! x                   [] ? y → ok
     ! y
```

Figure 8: Program with Gate Synchronization

to the stack nature of the channel, the message received is the most recent message sent. Thus, if a receive is anticipated prior to a send, an entirely different message could be received. Since the receive is deterministically terminating, however, this is sufficient to show that the theorem cannot be applied.

Figure 7 gives a program that illustrates the difference between atomic and general computations in this system. This program is guaranteed to terminate in the state *ok* for atomic computations (since process q must receive the value 2), but not for general computations (since process q could receive a value before p has completed all its sends).

Gate Synchronization. In this paradigm of synchronization, processes share variables known as gates. A gate is in one of two states: open or closed. Initially a gate is closed. Two operations are permitted on gate: a wait and a toggle. A process executing a wait on a closed gate suspends. A wait can only complete when the gate is open. A toggle has the effect of closing a gate if it is open and opening a gate if it is closed.

The receive action in this system (*i.e.* the wait) is not enabled-stable. That is, if it is enabled there is no guarantee that it remains enabled if preceded by a send (*i.e.* a toggle).

Figure 8 gives a program that illustrates the difference between atomic and general computations in this system. This program is guaranteed to terminate in the state *ok* for atomic computations, but not for general computations.

8.4 Theorem vs. Corollary

The corollary is more restrictive than the theorem. It imposes the requirement that the receives be send-monotonic, which is more stringent than the corresponding requirement in the theorem (that of weak send-monotonicity). Therefore, there are systems to which the theorem applies (*i.e.* have weakly send-monotonic receives) but to which the corollary does not.

As a simple example of such a system, consider receives that are nondeterministic when they are not enabled. In particular, if there is no pending message, then a receive either returns an arbitrary value or does not terminate. Notice that such a receive is not dangerous since it is not enabled, and so cannot be selected for execution. Nevertheless, it is not send-monotonic.

This might be a cause for concern since we have only modified the semantics of an action that cannot be selected, and yet the corollary is no longer applicable! However, our equivalence result of weak and strong send-monotonicity is predicated on the receive being deterministically terminating. Since the receive action as described above is not deterministically terminating, the equivalence does not hold and we should check the requirements given by the theorem instead of those given by the corollary. Doing this we see that the system is still weakly send-monotonic and hence the theorem still applies.

9 Conclusions

Atomic blocks are critical to reasoning about the correctness of parallel and distributed systems. Many paradigms exist for explicitly creating critical sections of atomically executing instructions, but these synchronization bottlenecks are expensive and should be avoided when possible. A theorem has existed for considerable time in the distributed-systems community that receive actions can be considered atomic with subsequent sends, since a send action is insensitive to the state of the message-passing layer. Treating distributed programs in this manner has simplified formal and informal arguments of correctness.

The contribution of this paper is a novel validation of this useful theorem and its extension to systems that exhibit nondeterministic termination. Our emphasis has been on a precise characterization of the synchronization primitives required for application of the theorem. Armed with a formal definition of these properties we have verified that the theorem is applicable to a variety of real synchronization systems, ranging from point-to-point message passing over ordered channels to shared-memory monotonic-assignment variables.

A Deterministically Terminating, Nonmiraculous Receives

If a receive action is nonmiraculous (that is, satisfies $wp.?.false = false$), then the property expressed in (6) is equivalent to the seemingly stronger property (7).

We must show:

$$[wlp.p?.false \Rightarrow \neg wp.p?.true]$$

Proof.

$$\begin{array}{ll}
 & wlp.p?.false \\
\Rightarrow & \quad \{ \text{ Weakening } \} \\
 & \neg wp.p?.true \lor wlp.p?.false \\
= & \quad \{ \text{ Law of Excluded Middle, and Identity of } \land \ \} \\
 & (\neg wp.p?.true \lor wp.p?.true) \\
 & \ \land (\neg wp.p?.true \lor wlp.p?.false) \\
= & \quad \{ \text{ Distribution of } \lor \text{ over } \land \ \} \\
 & \neg wp.p?.true \\
 & \ \lor (wp.p?.true \land wlp.p?.false) \\
= & \quad \{ \text{ Definition of } wp \ \} \\
 & \neg wp.p?.true \lor wp.p?.false \\
= & \quad \{ \text{ Receive is nonmiraculous, so } [wp.p?.false \equiv false] \ \} \\
 & \neg wp.p?.true \lor false \\
= & \quad \{ \text{ Identity of } \lor \ \} \\
 & \neg wp.p?.true
\end{array}$$

□

Acknowledgments

Thanks to several anonymous CASCONreferees for pointing out some related work and for many suggestions that improved the paper. Thanks as well to the Compositional Systems Research Group at Caltech for comments on earlier drafts.

The author was funded in part by an IBM Cooperative Computer Science Fellowship.

About the Author

Paul Sivilotti is an Assistant Professor in the Department of Computer and Information Science at The Ohio State University. He received his B.Sc.H. degree from Queen's University in 1991 and his M.S. and Ph.D. degrees from Caltech in 1993 and 1998 respectively. He has been a recipient of an NSERC '67 Fellowship as well as an IBM Cooperative Computer Science Fellowship. He can be reached by e-mail at `paolo@cis.ohio-state.edu`.

References

[1] R. J. R. Back. On correct refinements of programs. *Journal of Computer and System Sciences*, 23:49–68, 1981.

[2] R. J. R. Back. A method for refining atomicity in parallel algorithms. In G. Goos and J. Hartmanis, editors, *PARLE '89*, volume II, pages 199–216, Berlin, June 12–16 1989. Available as LNCS 366, Springer-Verlag.

[3] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hass Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[4] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[5] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, 1990.

[6] Thomas W. Doeppner, Jr. Parallel program correctness through refinement. In

Fourth ACM Symposium on Principles of Programming Languages, pages 155–169, 1133 Avenue of the Americas, New York, New York 10036, January 17–19 1977. ACM.

[7] Leslie Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4(2):59–68, 1990.

[8] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report 44, Digital Systems Research Center, Palo Alto, California, May 1989.

[9] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall Software Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[10] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18(12):717–721, December 1975.

[11] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Specification. Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, 1992.

[12] Jayadev Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 4(1):37–43, January 1982.

[13] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):7–30, July 1988.

[14] Joseph M. Morris. A theoretical basis for stepwise refinement and the progamming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.