Efficient Verification of Halting Properties for MPI Programs with Wildcard Receives

Stephen F. Siegel[*]

Laboratory for Advanced Software Engineering Research
Department of Computer Science, University of Massachusetts
Amherst MA 01003, USA
http://laser.cs.umass.edu
siegel@cs.umass.edu

Abstract. We are concerned with the verification of certain properties, such as freedom from deadlock, for parallel programs that are written using the Message Passing Interface (MPI). It is known that for MPI programs containing no "wildcard receives" (and restricted to a certain subset of MPI) freedom from deadlock can be established by considering only synchronous executions. We generalize this by presenting a model checking algorithm that deals with wildcard receives by moving back and forth between a synchronous and a buffering mode as the search of the state space progresses. This approach is similar to that taken by partial order reduction (POR) methods, but can dramatically reduce the number of states explored even when the standard POR techniques do not apply.

1 Introduction

It is well-known that finite-state verification techniques, such as model checking, suffer from the *state explosion problem*: the fact that the number of states of a concurrent system may—and often does—grow exponentially with the size of the system. Many different approaches have been studied to counteract this difficulty. These include partial order reduction (POR) methods, data abstraction, program slicing, and state compression techniques, to name only a few.

For the most part, these approaches have been formulated in very general frameworks. Their generality is both a strength and a weakness: the methods can be broadly applied, but may miss opportunities for reduction in specific situations. This observation has led to interest in more *domain-specific* approaches. The idea is to leverage knowledge of the restrictions imposed by a particular programming domain, or of common idioms used in the domain, in order to gain greater reductions than the generic algorithms allow. An example of this approach for concurrent Java programs is given in [2], where analysis that identifies common locking patterns, among other things, is exploited to dramatically improve the generic POR algorithms.

[*] Research supported by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement number DAAD190110564.

This paper is concerned with the domain of parallel programs that employ the Message Passing Interface (MPI). The MPI Standard [6, 7] specifies the syntax and semantics for a large library of message passing functions with bindings in C, C++, and Fortran. For many reasons—portability, performance, the broad scope of the library, and the wide availability of quality implementations—MPI has become the de facto standard for high-performance parallel computing. In addition, we focus on a particular class of properties of MPI programs, which we call *halting properties*: claims on the state of a program whenever execution halts, whether due to deadlock, or to normal termination. Freedom from deadlock is an example of a halting property; another would be an assertion on the values of variables when a program terminates.

Some explanation of the most essential MPI functions is required for what follows. The basic MPI function for sending a message to another process is MPI_SEND. To use it, one must specify the destination process and a message tag, in addition to other information. The corresponding function for receiving a message is MPI_RECV. In contrast to MPI_SEND, an MPI_RECV statement may specify its source process, or it may use the *wildcard* value MPI_ANY_SOURCE, indicating that this statement will accept a message from any source. Similarly, it may specify the tag of the message it wishes to receive, or it may use the wildcard value MPI_ANY_TAG. A receive operation that uses either or both wildcards is called a *wildcard receive.* The use of wildcards and tags allows for great flexibility in how messages are selected for reception.

Previous work has established that if a program (restricted to a certain subset of MPI) contains no wildcard receives, then a suitable model $\mathcal{M}$ of that program can be constructed with the following property: $\mathcal{M}$ is deadlock-free if, and only if, no synchronous execution of $\mathcal{M}$ can deadlock [8, Theorem 7.4]. This is exactly the kind of result we are after, as the need to represent all possible states of message channels is often a significant source of state explosion. Unfortunately, wildcard receives are common in actual MPI programs, and the theorem may fail if the hypothesis on wildcard receives is removed [8, Sec. 7.3].

The approach of this paper generalizes the earlier result in three ways. First, it shows that the hypothesis forbidding wildcard receives may be relaxed to allow the use of MPI_ANY_TAG, with no ill effects. Second, the range of properties is expanded to include all halting properties. But most importantly, it provides a model checking algorithm that deals with MPI_ANY_SOURCE by moving back and forth between a synchronous and a buffering mode as the search of the state space progresses. This approach is similar to that taken by POR methods, but can dramatically reduce the number of states explored even when the standard POR techniques do not apply.

The discussion proceeds as follows. Section 2 establishes the precise definition of a model of an MPI program, and of the execution semantics of such a model. The definition of a halting property and the statement of the main theorem are given in Sec. 3. Section 4 deals with consequences of the main theorem. These include a bounded model checking algorithm for halting properties; the consequences for programs that do not use MPI_ANY_SOURCE are also explored.

Section 5 discusses the relationship with the standard POR techniques. Results of an empirical investigation are presented in Sec. 6, and conclusions are drawn in Sec. 7. Appendix A contains proofs of the main theorem and two corollaries. Appendix B gives a description of the program and model for each example; complete MPI/C source code for the examples, as well as all the experimental results, can be downloaded from `http://laser.cs.umass.edu/~siegel/projects`.

2 Models of MPI Programs

For the purposes of this paper, an MPI program consists of a fixed number of concurrent processes, each executing its own code, with no shared variables, that communicate only through the MPI functions. The precise notion of a *model* of such a program is defined below. While there are many issues that arise in creating models from code, these are beyond the scope of this paper, and the reader is referred to [8] for a discussion of this subject and some examples. It is argued there that this notion of model suffices to represent MPI_SEND, MPI_RECV, MPI_SENDRECV (which concurrently executes one send and one receive operation), as well as the 16 collective functions, such as MPI_BCAST, MPI_ALLREDUCE, etc. The definition of receiving states here is slightly more general, in order to accommodate a new way to deal with tags, explained below.

2.1 Definition of a Model of an MPI Program

An *MPI context* is a 7-tuple $\mathcal{C} = (\mathsf{Proc}, \mathsf{Chan}, \mathsf{sender}, \mathsf{receiver}, \mathsf{msg}, \mathsf{loc}, \mathsf{com})$. The first two components are finite sets, representing the set of *processes*, and the set of communication *channels*, respectively. The next two components are functions from Chan to Proc; they define the sending and receiving process for each channel. The function msg assigns, to each $c \in \mathsf{Chan}$, a nonempty set $\mathsf{msg}(c)$; this is the set of messages that can be sent over channel c. The final two components are functions of Proc. For $p \in \mathsf{Proc}$, $\mathsf{loc}(p)$ is a finite set representing the set of *local events* for p, while $\mathsf{com}(p)$ is defined to be the set of *communication events* for p, namely, the set of send and receive symbols

$$\{c!x, d?y \mid c, d \in \mathsf{Chan}, x \in \mathsf{msg}(c), y \in \mathsf{msg}(d), \mathsf{sender}(c) = p = \mathsf{receiver}(d)\}.$$

Finally, for all $p, q \in \mathsf{Proc}$, we assume $\mathsf{loc}(p) \cap \mathsf{com}(q) = \emptyset$, and $p \neq q \Rightarrow \mathsf{loc}(p) \cap \mathsf{loc}(q) = \emptyset$.

An *MPI state machine for p under $\mathcal{C}$* is a 7-tuple

$$M = (\mathsf{States}, \mathsf{Trans}, \mathsf{src}, \mathsf{des}, \mathsf{label}, \mathsf{start}, \mathsf{End})$$

where States and Trans are sets, src and des are functions from Trans to States, label is a function from Trans to $\mathsf{loc}(p) \cup \mathsf{com}(p)$, $\mathsf{start} \in \mathsf{States}$, $\mathsf{End} \subseteq \mathsf{States}$, and, for each $u \in \mathsf{States}$, there exists $t \in \mathsf{Trans}$ with $\mathsf{src}(t) = u$ if, and only if,

$u \notin \mathsf{End}$. Finally, we require that every state u must fall into one of 5 categories, but before describing these, we define the following:

$$\begin{aligned}
R(u) &= \{(d,y) \mid d \in \mathsf{Chan}, y \in \mathsf{msg}(d), \exists t \in \mathsf{Trans}_p \colon \mathsf{src}(t) = u \wedge \mathsf{label}(t) = d?y\} \\
Q(u) &= \{d \in \mathsf{Chan} \mid \exists y \in \mathsf{msg}(d) \colon (d,y) \in R(u)\} \\
R_d(u) &= \{y \in \mathsf{msg}(d) \mid (d,y) \in R(u)\} \qquad\qquad (d \in Q(u)).
\end{aligned}$$

Now the 5 possibilities for u are as follows:

1. u is a *final state*: $u \in \mathsf{End}$,
2. u is a *local-event state*: the transitions departing from u are labeled by local events for p,
3. u is a *sending state*: there is precisely one transition departing from u and it is labeled by a send event for p,
4. u is a *receiving state*: the transitions departing from u are labeled by distinct receive events for p, or
5. u is a *send-receive state* (see Fig. 1): there is a $c \in \mathsf{Chan}$ with $\mathsf{sender}(c) = p$, an $x \in \mathsf{msg}(c)$, a state u', and states $v(d,y)$ and $v'(d,y)$ for all $(d,y) \in R(u)$, such that the following all hold:
   (a) u, u', and the $v(d,y)$ and $v'(d,y)$ are all distinct,
   (b) the set of transitions departing from u consists of one transition to u' whose label is $c!x$, and, for each $(d,y) \in R(u)$, one transition labeled $d?y$ to $v(d,y)$, and, furthermore, these are the only transitions terminating in u' or $v(d,y)$,
   (c) for each $(d,y) \in R(u)$, there is precisely one transition departing from $v(d,y)$, it is labeled $c!x$, and it terminates in $v'(d,y)$,
   (d) for each $(d,y) \in R(u)$, there is a transition from u' to $v'(d,y)$, it is labeled $d?y$, and these make up all the transitions departing from u', and
   (e) for each $(d,y) \in R(u)$, the only transitions terminating in $v'(d,y)$ are the one from u' and the one from $v(d,y)$.

Finally, a *model $\mathcal{M}$ of an MPI program* is a pair $(\mathcal{C}, M)$, where $\mathcal{C}$ is a context and M is a function that assigns, to each $p \in \mathsf{Proc}$, an MPI state machine M_p for p under $\mathcal{C}$, such that $\mathsf{States}_p \cap \mathsf{States}_q = \emptyset = \mathsf{Trans}_p \cap \mathsf{Trans}_q$ for $p \neq q$.

Given an MPI program, one may construct a model using one channel $c_{p,q}$, with $\mathsf{sender}(c_{p,q}) = p$ and $\mathsf{receiver}(c_{p,q}) = q$, for each $(p,q) \in \mathsf{Proc} \times \mathsf{Proc}$. To translate a receive statement r it suffices to specify the sets $Q(u)$ and $R_d(u)$ for the receiving state u corresponding to r. If r occurs in process q and specifies its

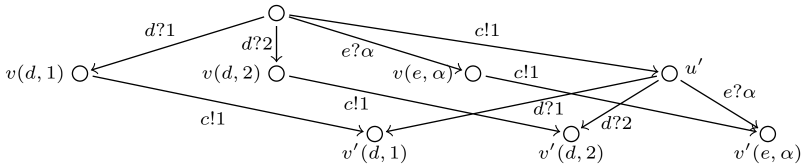


Fig. 1. A send-receive state u with $Q(u) = \{d, e\}$, $R_d(u) = \{1, 2\}$, $R_e(u) = \{\alpha\}$.

source p, then we let $Q(u) = \{c_{p,q}\}$. If r instead uses $\mathsf{MPI\_ANY\_SOURCE}$ then we let $Q(u) = \{c_{p,q} \mid p \in \mathsf{Proc}\}$. We may assume that the tags have been encoded in the messages, so that to each message x is associated an integer $\mathsf{tag}(x)$. Now if r specifies a tag t, we let

$$R_d(u) = \{x \in \mathsf{msg}(d) \mid \mathsf{tag}(x) = t\} \qquad\qquad (d \in\ Q(u)).$$

If instead r uses $\mathsf{MPI\_ANY\_TAG}$, we take $R_d(u) = \mathsf{msg}(d)$. We will see below that the execution semantics in effect allow a receive operation to choose nondeterministically among the receiving channels $Q(u)$, but, for a given $d \in Q(u)$, it must pick out the oldest message in d with a matching tag. This corresponds exactly to the requirements of the MPI Standard.

2.2 Execution Semantics of a Model of an MPI Program

Let $\mathbf{N} = \{0, 1, \ldots.\}$ and $\mathbf{N}^+ = \mathbf{N} \cup \{\infty\}$. A sequence $S = (x_1, x_2, \ldots)$ of elements of a set X may be either infinite or finite. We write $|S|$ for the length of S. If A is a subset of a set B, and S is a sequence of elements of B, then *the projection of S onto A* is the sequence that results by deleting from S all elements that are not in A. If S is any sequence and $n \in \mathbf{N}$, then S^n denotes the sequence obtained by truncating S after the n^{th} element.

Let $\mathcal{M}$ be a model of an MPI program. A *global state* σ of $\mathcal{M}$ is a pair of functions (u, α), where u assigns, to each $p \in \mathsf{Proc}$, a state $u_p \in \mathsf{States}_p$, and α assigns to each $c \in \mathsf{Chan}$ a finite sequence α_c of elements of $\mathsf{msg}(c)$. The sequence represents the *pending* messages for c: messages that have been sent but not yet received. We define $\mathsf{Pending}_c(\sigma) = \alpha_c$ and $\mathsf{state}_p(\sigma) = u_p$. The *initial state* of $\mathcal{M}$ is the global state for which $u_p = \mathsf{start}_p$ for all p, and α_c is empty for all c.

Suppose $\sigma = (u, \alpha)$ and $\sigma' = (u', \alpha')$ are global states of $\mathcal{M}$, $p \in \mathsf{Proc}$, $t \in \mathsf{Trans}_p$, and that $\mathsf{src}(t) = u_p$, $\mathsf{des}(t) = u'_p$, $u_q = u'_q$ for $q \neq p$, and one of the following holds:

1. $\mathsf{label}(t) \in \mathsf{loc}(p)$ and $\alpha = \alpha'$,
2. there exist $c \in \mathsf{Chan}$ and $x \in \mathsf{msg}(c)$ such that $\mathsf{label}(t) = c!x$, α'_c is obtained by appending x to the end of α_c, and $\alpha'_d = \alpha_d$ for $d \neq c$, or
3. there exist $d \in \mathsf{Chan}$ and $y \in \mathsf{msg}(d)$ such that $\mathsf{label}(t) = d?y$, y is the first element of the projection of α_d onto $R_d(u_p)$, α'_d is obtained by deleting the first occurrence of y from α_d, and $\alpha'_c = \alpha_c$ for $c \neq d$.

Then we call the triple $\tau = (\sigma, \sigma', t)$ a *simple global transition of $\mathcal{M}$*, and we define $\mathsf{label}(\tau) = \mathsf{label}(t)$.

Suppose now that σ, σ', and σ'' are global states, t_1, t_2 are transitions, $c \in \mathsf{Chan}$, $x \in \mathsf{msg}(c)$, $p = \mathsf{receiver}(c)$, and that the following all hold:

1. $\mathsf{label}(t_1) = c!x$ and $\mathsf{label}(t_2) = c?x$,
2. $\mathsf{Pending}_c(\sigma)$ contains no element of $R_c(\mathsf{state}_p(\sigma))$, and
3. (σ, σ', t_1) and (σ', σ'', t_2) are simple global transitions.

In this case we will refer to the 4-tuple $\tilde{\tau} = (\sigma, \sigma'', t_1, t_2)$ as a *synchronous global transition*, as it corresponds to a synchronous MPI communication: a message that is transferred directly from the sender to the receiver in one atomic step. We do *not* want to think of $\tilde{\tau}$ as "passing through" the intermediate state σ', but rather as leading directly from σ to σ''. In particular, since $\mathsf{Pending}_c(\sigma) = \mathsf{Pending}_c(\sigma'')$, $\tilde{\tau}$ leaves all of the channels unchanged. We define $\mathsf{label}(\tilde{\tau})$ to be the symbol $c!?x$.

The *state graph* of $\mathcal{M}$ is the ordered pair $\mathcal{G} = (\mathcal{S}, \mathcal{T})$, where $\mathcal{S}$ is the set of all global states, and $\mathcal{T}$ is the set of all (simple and synchronous) global transitions. Let $\mathsf{src}, \mathsf{des} \colon \mathcal{T} \to \mathcal{S}$ be the projections onto the first and second coordinates, respectively. These give $\mathcal{G}$ the structure of a directed graph.

An *event* α is any element of $\{\mathsf{label}(\tau) \mid \tau \in \mathcal{T}\}$. We say that α is *enabled* at the global state σ if there exists $\tau \in \mathcal{T}$ with $\mathsf{src}(\tau) = \sigma$ and $\mathsf{label}(\tau) = \alpha$.

Given a path $T = (\tau_1, \tau_2, \ldots)$ in $\mathcal{G}$, we define the *atomic length of* T to be $||T|| = \sum_i \epsilon(\tau_i)$, where $\epsilon(\tau) = 1$ if τ is simple and $\epsilon(\tau) = 2$ if τ is synchronous. This is sometimes a more natural measure of length than $|T|$. A *trace* of $\mathcal{M}$ is any path in $\mathcal{G}$ originating in the initial state. Finally, If T originates in the global state σ and $c \in \mathsf{Chan}$, we define

$$\mathsf{maxlen}_c(T) = \max_i \{|\mathsf{Pending}_c(\sigma)|, |\mathsf{Pending}_c(\mathsf{des}(\tau_i))|\}.$$

3 The Main Theorem

The main theorem concerns *halting properties* so we first explain what these are. In general, a concurrent program is considered to be in a halted state if every process has become permanently blocked. A receive statement in an MPI program blocks, as one would expect, as long as there is no pending message that matches the parameters of that statement. However, the circumstances under which a sending statement blocks are more subtle. Typically, one would assume that each channel c has some fixed size $\nu(c) \in \mathbf{N}$, and declare that a send on c blocks whenever the length of c equals $\nu(c)$. The MPI Standard, however, imposes no such bounds, but instead declares that a send *may* block at any time, unless the receiving process is at a state from which it can receive the message synchronously. We thus make the following definition for a model $\mathcal{M}$:

Definition 1. A global state σ of $\mathcal{M}$ is *potentially halted* if no receive, local, or synchronous event is enabled at σ.

We use the word "potentially" because a program in such a state may or may not halt, depending on the particular choices made by the MPI implementation.

For any predicate f on the global states of $\mathcal{M}$, and any subgraph $\mathcal{H}$ of $\mathcal{G}$ that contains the initial state σ_0, let $\Pi(\mathcal{H}, f)$ denote the statement *for all states σ reachable in $\mathcal{H}$ from σ_0, $f(\sigma)$*. Let phalt be the predicate defined by $\mathsf{phalt}(\sigma) \Leftrightarrow \sigma$ is potentially halted.

Definition 2. A *halting predicate* is a state predicate f of the form $\mathsf{phalt} \Rightarrow q$, where q is any state predicate. A *halting property* is a statement of the form $\Pi(\mathcal{H}, f)$, where f is a halting predicate.

An example of a halting property is given by taking $q = \mathsf{false}$, the predicate that holds at no state. For this q, $\Pi = \Pi(\mathcal{G}, f)$ states that $\mathcal{M}$ never halts. One could also take $q = \mathsf{term}$, the predicate that is true when all processes are at final states. Then Π states that whenever $\mathcal{M}$ halts, all processes have terminated, i.e., $\mathcal{M}$ is deadlock-free. More generally, one could take q to be the predicate term_Σ that holds when all processes in a certain subset Σ are at final states. One could also let q be the conjunction of term_Σ with another predicate—for example, a predicate that holds when variables in the processes in Σ, whose values are encoded in the local states, have particular values. In this case Π would say that whenever the program halts, all processes have terminated and the variables have the specified values.

To motivate what follows, consider the model of [8, Fig. 5] (the "Chansize Deadlocker" of Appendix B for $n = 1$). Suppose we try to verify freedom from deadlock for this model by considering only synchronous executions. Then we only explore the sequence $(c!?1, e!?1, d!?1)$, which terminates normally, and miss the deadlocking sequence $(c!1, e!?1, d!?1)$. We can try to explain why we missed the deadlock in the following way. At the initial state, process $p = \mathsf{receiver}(c)$ is at a wildcard receive u with $Q(u) = \{c, d\}$. At this state, c is ready to receive a message (synchronously) but d is not. By pursuing only synchronous communication, we never get to see the state in which p is at u and a receive on d is enabled.

The solution is to consider all enabled events (not just synchronous ones) whenever a process p is at a wildcard receive u, unless u has become "urgent." By this we mean that for each $c \in Q(u)$, either a (synchronous or buffered) receive on c is enabled or we know that a receive on c can never become enabled. Note that once a receive on c becomes enabled, it will remain enabled until p executes a transition, since p is the only process which may remove a message from c. Since no receive event can be enabled at a potentially halted state σ, the only way we can arrive at σ is if p eventually executes. Now if u is urgent, no new events in p can become enabled, and so one of the currently enabled events in p must occur if the system is to arrive at σ. Since those events are *independent* of events in other processes, we might as well explore the paths that result from scheduling each of those enabled events immediately. (If two events are independent then neither can disable the other and the effect of applying one and then the other does not depend on the order in which they are applied.) Local event states are similar, but they are always urgent since the local events are always enabled. The following definitions attempt to make all of this precise:

Definition 3. Let σ be a global state of $\mathcal{M}$, $p \in \mathsf{Proc}$, and $u = \mathsf{state}_p(\sigma)$. We say p is *at an urgent state in* σ if either u is a local event state, or all of the following hold:

1. u is a receiving or send-receive state,
2. for all $d \in Q(u)$, either
   (a) there is an event of the form $d?y$ or $d!?y$ enabled at σ, or
   (b) there is no such event enabled and $\mathsf{state}_{\mathsf{sender}(d)}(\sigma)$ is a final state,
   and

3. there is at least one $d \in Q(u)$ for which 2(a) holds.

We define $\mathsf{Urgent}(\sigma)$ to be the set of all $p \in \mathsf{Proc}$ such that p is at an urgent state in σ. Finally, we say that σ is *urgent* if $\mathsf{Urgent}(\sigma) \neq \emptyset$.

Definition 4. A global transition τ is *urgent for process* p if τ has the form (σ, σ', t) or (σ, σ', t', t), where $p \in \mathsf{Urgent}(\sigma)$, $t \in \mathsf{Trans}_p$, and $\mathsf{label}(t)$ is either a local event or a receive.

Condition 2(b) of Definition 3 can be relaxed somewhat: all that is really required is that $\mathsf{sender}(d)$ be in a state from which it can never reach a send on d. However, the version that we have stated has the advantage that it is very easy to check. Also, note that the third condition guarantees there is at least one enabled event at an urgent state.

We now fix a total order on Proc. The reason for this will become clear: we do not have to consider all urgent transitions departing from an urgent state, but only those for a single process, and so we will just choose the least one.

Definition 5. Let τ be a global transition and $\sigma = \mathsf{src}(\tau)$. We say that τ is *ample* if either σ is not urgent, or τ is urgent for the minimal element of $\mathsf{Urgent}(\sigma)$. We say that a path $(\tau_1, \tau_2, \ldots)$ in $\mathcal{G}$ is *ample* if τ_i is ample for all i.

The word "ample" comes from the notion of "ample set" in POR, where it plays essentially the same role. We let $\tilde{\mathcal{T}}$ be the set of all ample transitions and $\tilde{\mathcal{G}} = (\mathcal{S}, \tilde{\mathcal{T}})$. Now we can state the main theorem:

Theorem 1. *Given any path S in $\mathcal{G}$ from a global state σ_0 to a potentially halted global state σ, there exists a path T from σ_0 to σ in $\tilde{\mathcal{G}}$ such that $||T|| = ||S||$, $|T| \leq |S|$, and $\mathsf{maxlen}_c(T) \leq \mathsf{maxlen}_c(S)$ for all $c \in \mathsf{Chan}$. In particular $\Pi(\mathcal{G}, f) \Leftrightarrow \Pi(\tilde{\mathcal{G}}, f)$ for any halting predicate f.*

In light of the discussion above, it should come as no surprise that the proof of Theorem 1 (Appendix A.1) relies on many of the restrictions imposed by our domain and property. For example, the fact that each channel has an exclusive receiving process was used to show that once a receive event becomes enabled, it must remain enabled until that process executes. The knowledge that the property could be violated only if no receive were enabled was also used. The fact that a sending state has exactly one outgoing transition also comes into play: if the sending state had outgoing transitions on two different channels then a synchronous event that was enabled on one channel could become disabled if the sending process were to send on the other channel. These arguments withstand the introduction of send-receive states only because the specific structure of those states guarantees that the send event is independent of the receive events. Remove any of these domain-specific restrictions, and Theorem 1 may fail.

4 Consequences of the Main Theorem

4.1 The Urgent Algorithm

In general, the number of reachable states in $\mathcal{G}$ or $\tilde{\mathcal{G}}$ may be very large (or even infinite). So it is common practice to place upper bounds on the channel sizes, or the search depth, in order to reach a conclusive result on at least a bounded region of the state space. For these reasons we define the following concepts. Let $\nu : \mathsf{Chan} \rightarrow \mathbf{N}^+$ and $m \in \mathbf{N}^+$. Let $\mathcal{T}_{\nu,m}$ be the set of all global transitions that occur in traces T that satisfy (i) $||T|| \leq m$, and (ii) for all global states σ through which T passes, and all $c \in \mathsf{Chan}$, $|\mathsf{Pending}_c(\sigma)| \leq \nu(c)$. We let $\mathcal{G}_{\nu,m} = (\mathcal{S}, \mathcal{T}_{\nu,m})$.

Let $\mathcal{T}^\flat_{\nu,m}$ be the set of all $\tau \in \mathcal{T}_{\nu,m}$ such that τ is ample and

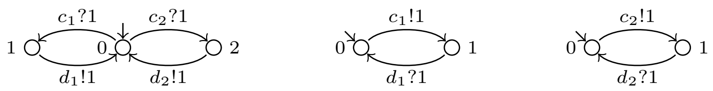
$$\text{if } \mathsf{label}(\tau) = c!?x \text{ for some } c, x \text{ then } \sigma \text{ is urgent or } |\mathsf{Pending}_c(\sigma)| = \nu(c), \quad (1)$$

where $\sigma = \mathsf{src}(\tau)$. Condition (1) is not strictly necessary, but it may provide some additional reduction. The idea is that when σ is not urgent, it would be redundant to consider synchronous transitions since we are already pursuing all buffered sends and receives. An exception is made if a channel is full since then a buffered send would not be enabled. Let $\mathcal{G}^\flat_{\nu,m} = (\mathcal{S}, \mathcal{T}^\flat_{\nu,m})$. The following consequence of Theorem 1 is proved in Appendix A.2:

Corollary 1. *Given any path in $\mathcal{G}_{\nu,m}$ from a global state σ_0 to a potentially halted global state σ, there exists a path in $\mathcal{G}^\flat_{\nu,m}$ from σ_0 to σ. In particular, $\Pi(\mathcal{G}_{\nu,m}, f) \Leftrightarrow \Pi(\mathcal{G}^\flat_{\nu,m}, f)$ for any halting predicate f.*

If States_p, Trans_p, and $\nu(c)$ are finite for all $p \in \mathsf{Proc}$ and $c \in \mathsf{Chan}$, then $\mathcal{T}_{\nu,m}$ and $\mathcal{T}^\flat_{\nu,m}$ are finite as well. It follows from Corollary 1 that we can verify a halting property in this case by performing a depth-first search of $\mathcal{G}^\flat_{\nu,m}$. Specifically, algorithm Urgent of Fig. 2 will find all reachable states in $\mathcal{G}_{\nu,m}$ for which f does not hold. We assume $\mathsf{Proc} = \{p_1, \ldots, p_n\}$ and $p_1 < \cdots < p_n$. The search is initiated by setting the global variable R to the empty set and calling *search*$(\sigma_0, 0)$, where σ_0 is the initial state. Function *urgent_transitions*(σ, p) returns the set of all $\tau \in \mathcal{T}$ such that $\mathsf{src}(\tau) = \sigma$ and τ is urgent for p. Function *standard_transitions*(σ,ν) returns the set of all $\tau \in \mathcal{T}$ that satisfy (i) $\mathsf{src}(\tau) = \sigma$, (ii) $|\mathsf{Pending}_c(\mathsf{des}(\tau))| \leq \nu(c)$ for all c, and (iii) $\mathsf{label}(\tau) = c!?x \Rightarrow |\mathsf{Pending}_c(\sigma)| = \nu(c)$. There is no need to specify ν for *urgent_transitions* since an urgent transition can never increase the length of a channel.

Example. In a model of a *client-server* system with n clients ($n \geq 1$), $\mathsf{Proc} = \{0, 1, \ldots, n\}$ with the natural order, $\mathsf{Chan} = \{c_1, d_1, \ldots, c_n, d_n\}$, $\mathsf{msg}(c) = \{1\}$ for all $c \in \mathsf{Chan}$, and $\mathsf{sender}(c_i) = i = \mathsf{receiver}(d_i)$, $\mathsf{receiver}(c_i) = 0 = \mathsf{sender}(d_i)$ for $1 \leq i \leq n$. For $n = 2$, the state machines for processes 0 (the server), 1, and 2, are respectively:



Let us see how the Urgent algorithm applies to this system for any ν and $m = \infty$. We start with the initial state: this state is urgent for process 0, so we explore the states resulting from the global transitions labeled $c_i!?1$ for all i. For any such i, the resulting state has process 0 in local state i, process i in local state 1, and all other processes and channels unchanged. This state is urgent

```
function bounded_ample($\sigma$)    /* returns $\{\tau \in \mathcal{T}^{\flat}_{\nu,m} \mid \mathsf{src}(\tau) = \sigma\}$ */
    for $i = 1$ to $n$ do
        if $p_i \in \mathsf{Urgent}(\sigma)$ then return urgent_transitions($\sigma, p_i$) end if
    end for;
    return standard_transitions($\sigma$,$\nu$)
end function;

procedure search($\sigma, n$)
    if $n > m$ then return end if;
    $R := R \cup \{\sigma\}$;
    if not $f(\sigma)$ then report_violation() end if;
    for all $\tau \in$ bounded_ample($\sigma$) do
        if $\mathsf{des}(\tau) \notin R$ then search($\mathsf{des}(\tau), n + \epsilon(\tau)$) end if
    end for all
end procedure
```

Fig. 2. The Urgent Algorithm: depth-first search of $\mathcal{G}^{\flat}_{\nu,m}$.

for i, and so we explore the single transition $d_i!?1$. This returns us to the initial state, which is already in R. Hence the algorithm explores a total of $n+1$ global states, and $2n$ transitions. Notice also that, in this case, the search does not explore any buffered communication, even though process 0 contains a wildcard receive.

4.2 Source-Specific Models and Synchronous Traces

We say that $\mathcal{M}$ is *source-specific* if for every receiving and send-receive state u in $\mathcal{M}$, $|Q(u)| = 1$; this corresponds to an MPI program which never uses MPI_ANY_SOURCE (though it may use MPI_ANY_TAG). We say that a path in $\mathcal{G}$ is *synchronous* if it consists solely of local and synchronous transitions.

Let $\mathcal{M}$ be any model and σ a global state of $\mathcal{M}$. If σ is urgent, then clearly σ cannot be potentially halted. Now if $\mathcal{M}$ is source-specific, the converse is also true. For if there is some $c \in \mathsf{Chan}$ and $x \in \mathsf{msg}(c)$ for which $c?x$ or $c!?x$ is enabled at σ, then $p = \mathsf{receiver}(c) \in \mathsf{Urgent}(\sigma)$, since $Q(\mathsf{state}_p(\sigma)) = \{c\}$.

Now suppose $\mathcal{M}$ is source-specific and T is a trace terminating in a potentially halted state σ. By Theorem 1, there exists an ample trace $\tilde{T} = (\tau_1, \ldots, \tau_n)$ terminating in σ, with $n \leq |T|$ and $||\tilde{T}|| = ||T||$. Let $\sigma_k = \mathsf{des}(\tau_k)$ for $1 \leq k \leq n$ and let σ_0 be the initial state. Let i be the least integer for which σ_i is potentially halted. For $0 \leq j < i$, σ_j is not potentially halted, which as we have seen means that σ_j is urgent. This implies that τ_{j+1} is a local event, synchronous, or receive transition. But τ_{j+1} cannot be a receive: if it were, there would have to be a preceding send. In other words, $\tilde{T}^i$ is synchronous. We have proved:

Corollary 2. *Let $\mathcal{M}$ be a source-specific model of an MPI program and T a trace terminating in a potentially halted state σ. Then there exist $i \in \mathbf{N}$ and an ample trace $\tilde{T}$ terminating in σ such that $|\tilde{T}| \leq |T|$, $||\tilde{T}|| \leq ||T||$, and $\tilde{T}^i$ is synchronous and terminates in a potentially halted state.*

This leads to the following, which generalizes [8, Theorem 7.4], and is proved in Appendix A.3. Note that 0 is used to denote the function on Chan which is identically 0. Also, all of the examples of halting predicates given in Sec. 3 satisfy the condition on q.

Corollary 3. *Suppose $\mathcal{M}$ is a source-specific model of an MPI program, and q is a state predicate satisfying $q(\sigma) \Rightarrow q(\sigma')$ for any simple global transition (σ, σ', t). Let f denote the predicate $\mathsf{phalt} \Rightarrow q$, $\nu\colon \mathsf{Chan} \to \mathbf{N}^+$, and $m \in \mathbf{N}^+$. Then $\Pi(\mathcal{G}_{\nu,m}, f) \Leftrightarrow \Pi(\mathcal{G}^\flat_{0,m}, f)$.*

5 The Relationship to Partial Order Reduction

We follow the presentation of POR techniques in [1, Chap. 10]. The goal is to show that an arbitrary trace can be transformed into a representative form. At each state σ in the representative trace, the transition departing from σ is chosen from a specific "ample" subset of all transitions enabled at σ. (The word *transition* in this context corresponds to a set of our global transitions.) If the choice of ample sets satisfies certain conditions then the reduced state graph, consisting of just the ample transitions, is guaranteed to be stutter-equivalent to the full state graph, and hence can be used to verify any LTL$_{-X}$ property [1, Cor. 2 and Thm. 12].

One of these conditions is the following: a transition that is dependent on a transition in the ample set for a given state cannot be executed without a transition in the ample set occurring first. Unfortunately, because we have included the synchronous transitions in our state graph, our definition of *ample* does not satisfy this condition. Consider, for example, a client-server system with one client. At the initial state, the sole ample transition is the one labeled $c!?1$. But the path $c!1, c?1$ is also possible and both $c!1$ and $c?1$ are dependent on $c!?1$.

It could be argued that we would avoid this problem if we never introduced the synchronous transitions in the first place. A process at a receiving state would then be urgent when all the receiving channels have a pending message. This would in fact lead to a correct algorithm, and is similar to the algorithm suggested for message passing systems in [1, Sec. 10.5.2] and the one used by Spin [5] (though other differences, discussed below, still arise). However, this algorithm would miss many of the opportunities for reduction. Consider a client server system with n clients. The system would not be in an urgent state until every client had sent at least one request; in particular we would explore all possible states of the n request channels for which at least one channel is empty.

Another condition concerns *invisibility.* A transition is invisible if it never changes the value of a predicate referred to by the property (e.g., phalt). The issue arises in the following context. Suppose we have a transition α that is independent of transitions β_i:

$$\begin{tikzcd}
s = s_0 \arrow[r, "\beta_0"] \arrow[d, "\alpha"'] & s_1 \arrow[r, "\beta_1"] \arrow[d, "\alpha"'] & s_2 \arrow[r, "\beta_2"] \arrow[d, "\alpha"'] & \cdots \arrow[r, "\beta_{n-1}"] & s_n \arrow[d, "\alpha"'] \\
r_0 \arrow[r, "\beta_0"] & r_1 \arrow[r, "\beta_1"] & r_2 \arrow[r, "\beta_2"] & \cdots \arrow[r, "\beta_{n-1}"] & r_n = r
\end{tikzcd}$$

If the current state of the depth-first search is s, then instead of investigating both sequences of states $\sigma = (s_0, \ldots, s_n, r)$ and $\rho = (s, r_0, \ldots, r_n)$, we would like to ignore σ and pursue only ρ. If α is invisible, then ρ and σ are stuttering equivalent, so σ may be safely ignored. In our case, α would be an urgent transition, which might be any local event, receive, or synchronous transition. Since all of these may change phalt from false to true, α is not necessarily invisible. Why is this not a problem for our method? Because an urgent transition can only be enabled at a state that is not potentially halted, which is to say that phalt must be false at all the s_i. Since our goal is to find all potentially halted states, we are justified in ignoring the s_i. Notice that a send transition can change phalt from true to false, and so it really would be a mistake to allow sends to be urgent.

A third condition states that the reduced graph cannot contain a cycle in which some transition in the full graph remains enabled throughout but is not included in the reduced graph. The point is to avoid the situation where a visible transition is delayed forever due to the insertion of an infinite number of invisible ample transitions. Enforcing the condition typically requires a modification to the depth-first search algorithm that involves checking whether a new state is currently on the search stack. The whole issue does not arise in our case: our method of transforming an arbitrary violating trace to an ample one never involves inserting new transitions; it only permutes those that are already there.

6 Experimental Results

The eight scalable C/MPI programs used for our empirical investigation are described in Appendix B. They range from standard toy concurrency examples to more complex programs from a well-known book on MPI [3]. For each, we constructed by hand an abstract model appropriate for verifying freedom from deadlock. These models were encoded as certain Java classes that can be read by the MPI-Optimized Verifier (Mover), a Java tool developed for this project. Given the model and an object describing a halting property, Mover can either (A_1) execute a generic depth-first search of the state space to verify the property or report any violations, (A_2) execute the Urgent algorithm to do the same, or (A_3) produce a Promela model that can be used by Spin [4] to do the same.

The processes and channels in the Promela model correspond exactly to those in the MPI model. There are no variables in the Promela, other than the channels. The local states of a process are encoded by labeled positions in the code. States with multiple departing transitions are encoded using the Promela selection construct (`if`...`fi`). A never claim is inserted corresponding to the LTL formula `<>!(univenabled || terminated)`, where `univenabled` is defined to hold whenever a synchronous, local, or receive event is enabled (the definition refers to the lengths of the channels and the positions of the local processes), and `terminated` is defined to hold when all terminating processes are at final states. It might seem appropriate to use Spin's `xr` and `xs` declarations, which declare a process to have exclusive read or write access to a channel and provide information to help the POR algorithm. However, this is not allowed, as the never claim makes reference to all the channels, and in fact an attempt to use those declarations causes Spin to flag the error. This is Spin's way of recognizing that the communication events may not be invisible with respect to the property.

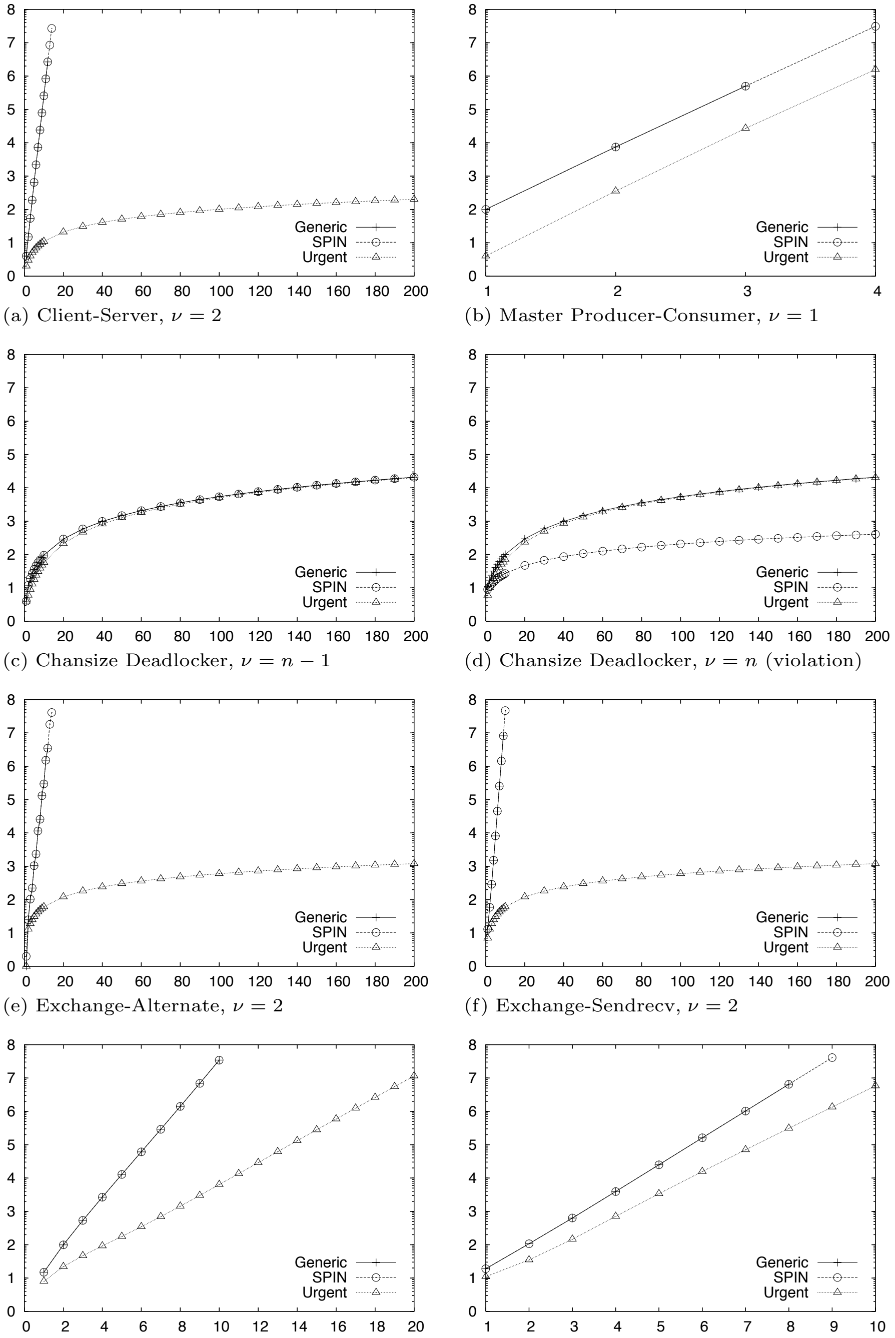
(A different way to use Spin to verify freedom from deadlock for MPI programs is described in [9]. In that approach, every send is immediately followed by a non-deterministic choice between blocking until the channel becomes empty and proceeding without blocking. Freedom from deadlock can then be checked in the usual way with Spin, i.e., without a never claim. While we have not carried out an extensive comparison, it appears that the state-explosion is much worse for that approach than for the approach presented here, due to all the new states introduced by the non-deterministic choices.)

We applied all three approaches to each of the examples, increasing system size n until $n = 200$ or we ran out of memory. In each case we recorded the numbers of states and transitions explored, and the time and memory used. We used the Java2 SDK 1.4.2 with options `-Xmx1900M` and Spin 4.2.0, with options `-DCOLLAPSE -DMEMLIM=2800 -DSAFETY`; the maximum search depth also had to be increased in some cases. The experiments were run on a Linux box with a 2.2 GHz Xeon processor and 4 GB of memory. In the one case where a deadlock was found, the searches were stopped after finding the first counterexample.

Figures 3 and 4 show the number of states explored. We first observe that the numbers for A_1 and A_3 are exactly equal in all cases where both searches completed. Since A_1 explores all reachable states, this means that Spin's POR algorithm (on, by default) made no difference in the number of states explored. This is not surprising, since there are no invisible events for the algorithm to exploit. For the one case where a violation exists, Spin did find the violation much sooner than either Mover algorithm (Fig. 3(d)). This appears to be just a fluke related to process ordering: we ran the same problem but reversed the order in which the processes were declared (for both tools), and the results were almost exactly reversed.

For the Client-Server, Producer-Consumer, and the two exchange examples, the performance of A_2 was the most impressive, reducing the complexity class from one that is apparently exponential to one that is linear. For Monte Carlo and Master-Slave, both functions appear to be exponential, but the exponent for the A_2 function is lower (significantly so for Master-Slave), allowing it to scale further. In one case (Fig. 3(c)), the use of A_2 makes almost no difference, but there the number of reachable states was quadratic to begin with so there was not much room for improvement. The Master Producer-Consumer proved the most difficult: there seemed to be a small constant reduction but no approach could scale beyond $n = 4$.

For Producer-Consumer, we give on one graph (Fig. 4, left) the results for various values of ν. This graph demonstrates the impact of channel size on state explosion for systems that can buffer many messages. For $\nu = 0$, however, the number of reachable states for the system of size n is just $n + 1$, and A_2 searches that number of states for any value of ν, since the system contains no wildcard



(a) Client-Server, $\nu = 2$

(b) Master Producer-Consumer, $\nu = 1$

(c) Chansize Deadlocker, $\nu = n - 1$

(d) Chansize Deadlocker, $\nu = n$ (violation)

(e) Exchange-Alternate, $\nu = 2$

(f) Exchange-Sendrecv, $\nu = 2$

(g) Master-Slave, $\nu = 2$

(h) Monte Carlo, $\nu = 2$

Fig. 3. Graphs of $y = \log_{10}(f(n))$, where $f(n)$ is the number of states explored for the system of size n, with channel size bound ν.

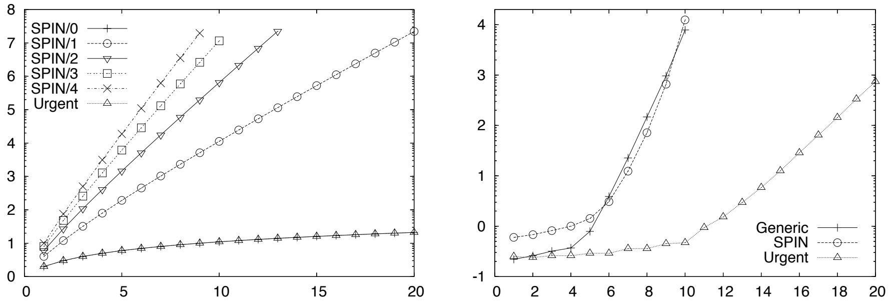


Fig. 4. Producer-Consumer states for $\nu \in \{0, 1, \ldots, 4\}$ ($\log_{10}$ of number of states, left), and Master-Slave time ($\log_{10}$ of number of seconds, right).

receives. We also give the time for the Master-Slave example; typical of these examples, the pattern is similar to that for the number of states.

In summary, the Urgent algorithm often dramatically reduced the number of states explored. It can never increase that number, as long as the search is carried to completion, nor did it appear to have a significant impact on the time required to complete the search. In contrast, the POR algorithm implemented in Spin had no effect on the number of states explored.

7 Conclusions and Future Work

We have presented a POR-like optimization to the standard model checking algorithm for verifying halting properties of MPI programs. The algorithm seeks to control state explosion by limiting the number of transitions explored that involve buffering messages. We have demonstrated its effectiveness on several scalable examples, including some with wildcard receives.

A better validation of effectiveness would utilize more “realistic” examples. There is no guarantee that scaling our simple examples presents the same kind of challenge to the Urgent algorithm that an actual production-level MPI code would. Due to the difficulty of creating models by hand, this task would benefit from an automated MPI model extractor. We intend to develop such a tool, and use it to verify not only freedom from deadlock, but also other halting properties. For example, we would like to model the arithmetic computations performed by an MPI program symbolically, and check that at termination the program has arrived at the correct arithmetic result.

Finally, the study of domain-specific approaches may also shed light on the general framework. The standard POR definition of *independence*, for example, was shown to be inadequate for Java programs, and so the authors of [2] presented a generalization of the POR framework that introduced the concept of *conditional independence*. It would be interesting to see if the standard POR framework could be extended to incorporate the idea of switching between a synchronous and a buffering mode, generalizing our MPI-specific approach.

References

1. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
2. Dwyer, M.B., Hatcliff, J., Robby, Ranganath, V.P.: Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. Formal Methods in System Design **25** (2004) 199–240
3. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA (1999)
4. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Boston (2004)
5. Holzmann, G.J., Peled, D.: An improvement in formal verification. In Hogrefe, D., Leue, S., eds.: Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994. Volume 6 of IFIP Conference Proceedings. Chapman & Hall (1995) 197–211
6. Message Passing Interface Forum: MPI: A Message-Passing Interface standard, version 1.1. `http://www.mpi-forum.org/docs/` (1995)
7. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/` (1997)
8. Siegel, S.F., Avrunin, G.S.: Modeling MPI programs for verification. Technical Report UM-CS-2004-75, Department of Computer Science, University of Massachusetts (2004)
9. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In Graf, S., Mounier, L., eds.: Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings. Volume 2989 of Lecture Notes in Computer Science. Springer-Verlag (2004) 286–303

A Proofs

A.1 Proof of Theorem 1

The following assert the independence of certain transitions in a model $\mathcal{M}$ of an MPI program. They are easily established from the definitions:

Lemma 1. *Suppose $\mathcal{M}$ has global states σ_0, σ_1, and σ_2, and local transitions t_1 and t_2 that lie in distinct processes, such that $(\sigma_0, \sigma_1, t_1)$ and $(\sigma_1, \sigma_2, t_2)$ are global transitions. Assume furthermore that if $\mathsf{label}(t_2) = c?x$ for some $c \in \mathsf{Chan}$ and $x \in \mathsf{msg}(c)$, then x occurs in $\mathsf{Pending}_c(\sigma_0)$. Then $\mathcal{M}$ has a global state σ_1' for which $(\sigma_0, \sigma_1', t_2)$ and $(\sigma_1', \sigma_2, t_1)$ are global transitions.*

Lemma 2. *Suppose $p \in \mathsf{Proc}$, u is a send-receive state in States_p, $t_1, t_2 \in \mathsf{Trans}_p$, $c, d \in \mathsf{Chan}$, $x \in \mathsf{msg}(c)$, $y \in \mathsf{msg}(d)$, and $\sigma_0, \sigma_1, \sigma_2$ are global states such that all of the following hold:*

1. *$\{\mathsf{label}(t_1), \mathsf{label}(t_2)\} = \{c!x, d?y\}$,*
2. *$\mathsf{state}_p(\sigma_0) = u$,*
3. *$(\sigma_0, \sigma_1, t_1)$ and $(\sigma_1, \sigma_2, t_2)$ are global transitions for $\mathcal{M}$, and*
4. *if $c = d$ then y occurs in $\mathsf{Pending}_c(\sigma_0)$.*

Then there exists a global state σ_1', and transitions $t_1', t_2' \in \mathsf{Trans}_p$ such that $\mathsf{label}(t_i') = \mathsf{label}(t_i)$ and $(\sigma_0, \sigma_1', t_2')$ and $(\sigma_1', \sigma_2, t_1')$ are global transitions for $\mathcal{M}$.

We now turn to the proof of Theorem 1.

Let S be the given path, and $N = |S|$. We will show by induction on m that for $0 \leq m \leq N$, there exists a path T from σ_0 to σ such that $|T| \leq N$, T^m is ample, $||T|| = ||S||$, and $\mathsf{maxlen}_c(T) \leq \mathsf{maxlen}_c(S)$ for all $c \in \mathsf{Chan}$. For $m = 0$, we may take $T = S$. The case $m = N$ is the desired conclusion.

Suppose $0 \leq m < N$ and the inductive hypothesis holds for m. Write $T = (\tau_1, \ldots, \tau_n)$ and $\sigma_i = \mathsf{des}(\tau_i)$. We must construct a trace $\tilde{T}$ for which the inductive hypothesis holds for $m+1$. If $m \geq n$ then we may take $\tilde{T} = T$, so assume $m < n$.

If σ_m is not urgent we may take $\tilde{T} = T$, so assume σ_m is urgent. Let p be the minimal element of $\mathsf{Urgent}(\sigma_m)$ and $u = \mathsf{state}_p(\sigma_m)$. By definition, either (i) u is a local event state, or (ii) u is a receiving or send-receive state.

Suppose u is a local event state. Then there exists k with $m < k \leq n$ such that $\mathsf{label}(\tau_k)$ is a local event in process p and, for $m < i < k$, τ_i does not involve p. For if this were not the case, a local event would be enabled at σ, and σ would not be potentially halted. Now we may use Lemma 1 to move this local event to the left until it is in position $m + 1$. (If a synchronous transition is encountered along the way, it may be replaced by its two simple parts, each of which commutes with the local event, and then the two parts can be merged back to the synchronous transition.) Hence there exists a trace

$$\tilde{T} = (\tau_1, \ldots, \tau_m, \tau_k', \tau_{m+1}', \ldots, \tau_{k-1}', \tau_{k+1}', \ldots, \tau_n') \tag{2}$$

that terminates in σ and for which $\mathsf{label}(\tau_i') = \mathsf{label}(\tau_i)$ for $m < i \leq n$. As $\mathsf{label}(\tau_k')$ is a local event in process p, $\tilde{T}^{m+1}$ is ample, as required. Clearly $||\tilde{T}|| = ||T||$, $|\tilde{T}| = |T|$, and the maxlen_c are also unchanged, so the inductive step is established for this case.

Suppose instead u is a receiving or send-receive state. Again, we know that one of the channels d in $Q(u)$ must eventually receive a message in T, else a receive or synchronous event would be enabled at σ, and σ would not be potentially halted. However, if u is a send-receive state, it is possible that the send event for that state takes place before the receive on d. Hence there exist an integer k, $d \in Q(u)$, and $y \in \mathsf{msg}(d)$ such that $m < k \leq n$, $\mathsf{label}(\tau_k) \in \{d?y, d!?y\}$, and there is at most one i, $m < i < k$, for which τ_i involves process p, and if there is such an i then u is a send-receive state and $\mathsf{label}(\tau_i) \in \{c!x, c!?x\}$, where $c!x$ is the label of the send transition departing from u.

If the projection of $\mathsf{Pending}_d(\sigma_m)$ onto $R_d(u)$ is nonempty then that projection must begin with y. In this case, we must have $\mathsf{label}(\tau_k) = d?y$, since the synchronous event can only take place if there are no pending messages. Now we may use Lemma 1 and Lemma 2 to move the $d?y$ in position k leftward to position $m + 1$ and produce the desired trace $\tilde{T}$, just as in the case of the local event state. We again have $||\tilde{T}|| = ||T||$, $|\tilde{T}| = |T|$, and $\tilde{T}^{m+1}$ is ample. Moreover, since this transformation has only caused a receive to take place earlier in the sequence, the maxlen_c cannot have increased. Hence the inductive step is established in this case as well.

So suppose $\mathsf{Pending}_d(\sigma_m)$ does not contain an element of $R_d(u)$. It then follows from the definition of *urgent* that $v = \mathsf{state}_q(\sigma_m)$ is a sending or send-receive state with outgoing send transition labeled $d!y$, where $q = \mathsf{sender}(d)$. Hence for some j, $m < j \leq k$, $\mathsf{label}(\tau_j) \in \{d!y, d!?y\}$. Furthermore, there is at most one i, $m < i < j$, for which τ_i involves process q, and if there is such an i then v is a send-receive state and $\mathsf{label}(\tau_i) \in \{e?z, e!?z\}$, where $e?z$ is the label of the receive transition departing from v.

If $j = k$ then $\mathsf{label}(\tau_j) = \mathsf{label}(\tau_k) = d!?y$, while if $j < k$ then $\mathsf{label}(\tau_j) = d!y$ and $\mathsf{label}(\tau_k) = d?y$.

Assuming $j < k$, the two Lemmas allow us to move the $d!y$ in position j leftward to position $m+1$, and then we may move the $d?y$ in position k leftward to position $m+2$. Now the transitions in positions $m+1$ and $m+2$ can be combined into a single synchronous transition labeled $d!?y$.

If $j = k$, the synchronous transition τ_j may be replaced by its two simple parts and the argument in the paragraph above may be applied.

In any case, the resulting trace $\tilde{T}$ has an ample prefix of length $m+1$, $||\tilde{T}|| = ||T||$, and $|\tilde{T}|$ is either $|T|$ or $|T|-1$. To see that we have never increased maxlen_d, we argue as follows. The same transformation on the sequence of events is accomplished if we first move the receive $d?y$ leftward to meet the send $d!y$, then combine these two into a synchronous transition, then move the entire synchronous transition leftward to position $m+1$. Since moving a receive to the left cannot increase channel length, and a synchronous transition has no effect on any channel, the result cannot increase maxlen_d. □

A.2 Proof of Corollary 1

Let S be the given path. We will show by induction on i that there exists a path $T = (\tau_1, \tau_2, \ldots)$ from σ_0 to σ in $\mathcal{G}_{\nu,m}$ for which $||T|| = ||S||$ and $\tau_j \in \mathcal{T}^\flat_{\nu,m}$ for $1 \leq j \leq i$. For $i = 0$, we can take $T = S$, while the case $i = 2||S||$ is the desired result, since $|T| \leq 2||T|| = 2||S||$.

So suppose the inductive hypothesis holds for i and we wish to construct a path $\tilde{T}$ demonstrating that it holds for $i+1$. If $i \geq |T|$ we can take $\tilde{T} = T$, so assume $i < |T|$. By applying Theorem 1 to the path $(\tau_{i+1}, \ldots)$, we may assume that T is ample. Now if $\tau = \tau_{i+1}$ satisfies (1) we may take $\tilde{T} = T$, so assume $\mathsf{label}(\tau) = c!?x$, $\sigma' = \mathsf{src}(\tau)$ is not urgent, and $|\mathsf{Pending}_c(\sigma')| < \nu(c)$. In this case, we form $\tilde{T}$ by replacing τ with the two simple transitions α, β labeled $c!x$, $c?x$, respectively. Now α is ample, since σ' is not urgent, it satisfies (1), and at $\mathsf{des}(\alpha)$, the number of pending messages in c is at most $\nu(c)$. Moreover, $||\tilde{T}|| = ||T||$. □

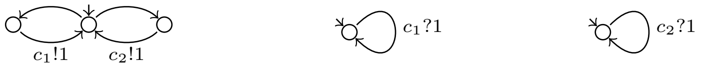
A.3 Proof of Corollary 3

Clearly $\Pi(\mathcal{G}_{\nu,m}, f) \Rightarrow \Pi(\mathcal{G}^\flat_{0,m}, f)$, so suppose $\Pi(\mathcal{G}_{\nu,m}, f)$ does not hold. Then there is a trace of atomic length at most m terminating in a state σ which is potentially halted but for which $q(\sigma)$ does not hold. By Corollary 2, there is an ample trace $T = (\tau_1, \ldots)$ terminating in σ with $||T|| \leq m$, and an integer i such that T^i is synchronous and terminates in a potentially halted state σ'. Since $\mathsf{src}(\tau_j)$ is urgent for $j \leq i$, condition (1) holds for T^i, so T^i is a path in $\mathcal{G}^{\flat}_{0,m}$. If $q(\sigma')$, then by our assumption on q, we would have $q(\sigma)$, a contradiction. Hence $f(\sigma')$ does not hold, and so $\Pi(\mathcal{G}^{\flat}_{0,m}, f)$ does not hold. □

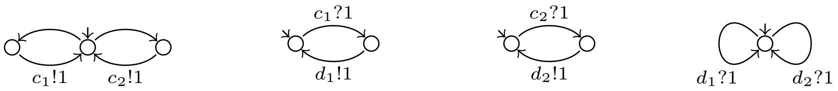
B The Examples

Client-Server. See Sec. 4.1.

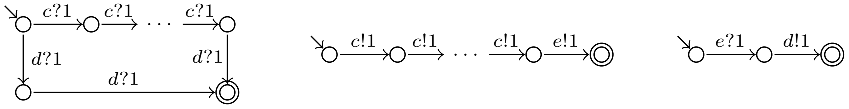
Producer-Consumer. A system of 1 producer and n consumers. The producer repeatedly chooses a consumer and sends to it. The consumers repeatedly receive. State machines for $n = 2$ (unlabeled edges are local event transitions):



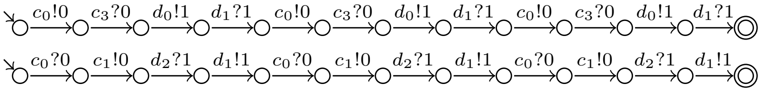
Master Producer-Consumer. A producer-consumer system with 1 "master producer," n producers, and 1 consumer. The master chooses a producer randomly and sends it a message, and then repeats. Each producer, after receiving a message from the master, passes the message on to the consumer. The consumer receives the messages using a wildcard receive. State machines for $n = 2$:



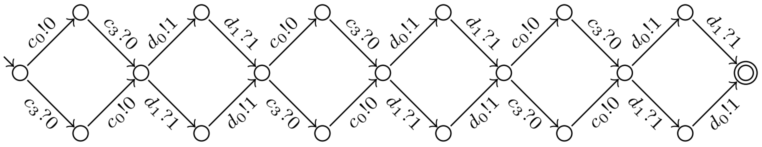
Chansize Deadlocker. For any $n \geq 1$, this provides an example of an MPI program that may deadlock if channel size $\nu \geq n$, but is deadlock-free if $\nu < n$. This generalizes the example given in [8, Fig. 5]. For all n, the system consists of three processes; but the number of consecutive $c!1$ transitions equals n:



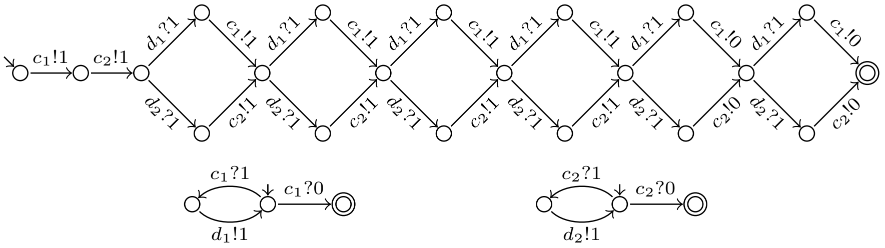
Exchange-Alternate. Suppose n processes are arranged cyclically and each has a single value. We wish for each process to obtain the values of its left and right neighbors. This can be accomplished in two "exchanges." In the first exchange, each process must send its value to its right neighbor, and receive the value from its left neighbor. In the second exchange, the role of right and left are reversed. An issue arises in coding the exchanges. If each process first sends, and then receives, the program may deadlock if the MPI infrastructure chooses to synchronize all the sends. There are several well-known solutions to this problem. In this solution, based on the code of [3, Fig. 4.12], the deadlock is avoided by having the processes of odd rank first receive and then send, while the processes of even rank first send and then receive. The pair of exchanges is repeated 3 times. For $n = 4$, the state machines for processes 0 and 1 are as follows:



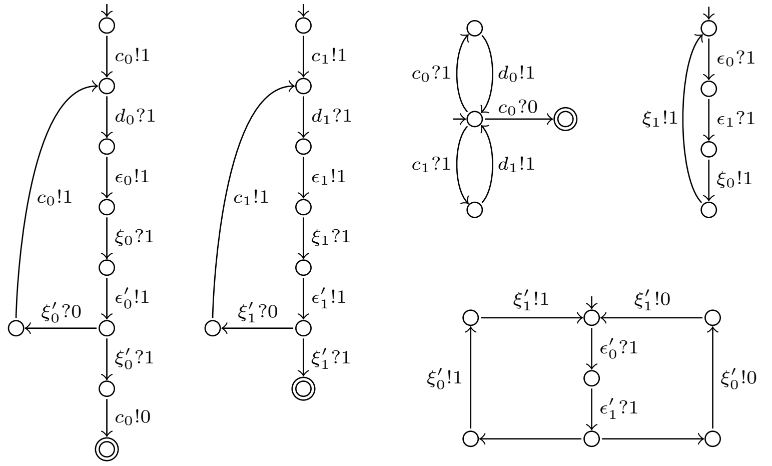
Exchange-Sendrecv. Another solution to the exchange problem, in which the deadlock is avoided by coding each exchange using MPI_SENDRECV(cf. [3, Fig. 4.14]). For $n = 4$, the state machine for process 0 (the others are similar):



Master-Slave. Based on the program of [3, Sec. 3.7], which employs a master-slave architecture to parallelize matrix multiplication. The system consists of one master process, and n slave processes. We assume there are $3n$ tasks to be performed. The master begins by sending out a task (represented in the model by "1") to each slave. It then waits at a wildcard receive for results to come in. After receiving a result from (say) slave i, it then sends out the next task to slave i, and waits for the next result. The master continues in this way until all tasks have been handed out. It then receives the outstanding results and as each comes in, the master sends a termination message (represented by "0") to the slave, and finally, the server terminates. State machines for $n = 2$:



Monte Carlo. Based on the program of [3, Figs. 3.15–3.18], which uses a Monte Carlo algorithm to estimate π. The system consists of n worker processes, and a random number server process. A worker begins by sending a request to the server for a random number. After receiving the random number, it performs a local computation. Next, all the workers engage in a collective call to MPI_ALLREDUCE. Another local computation is performed and then a second call to MPI_ALLREDUCE takes place. At this point each worker compares the result returned by the second reduction call to a constant $\delta > 0$. If the result is less than δ, the worker terminates, although the first worker first sends a termination message to the server. Otherwise, the worker sends another request to the server and loops back to receive the response. In the model, the two calls to MPI_ALLREDUCE are handled by two *coordinator* processes. The second of these two chooses non-deterministically between returning a “0” or a “1” to all the workers; the value returned represents the boolean value of the predicate that the sum computed is less than δ. For $n = 2$, the state machines for the two workers, the server, and the two coordinators are, respectively, as follows:


$c_0!1$
$d_0?1$
$\epsilon_0!1$
$c_0!1$
$\xi_0?1$
$\epsilon'_0!1$
$\xi'_0?0$
$\xi'_0?1$
$c_0!0$
$c_1!1$
$d_1?1$
$\epsilon_1!1$
$c_1!1$
$\xi_1?1$
$\epsilon'_1!1$
$\xi'_1?0$
$\xi'_1?1$
$c_0?1$
$d_0!1$
$c_0?0$
$c_1?1$
$d_1!1$
$\epsilon_0?1$
$\xi_1!1$
$\epsilon_1?1$
$\xi_0!1$
$\xi'_1!1$
$\xi'_1!0$
$\epsilon'_0?1$
$\xi'_0!1$
$\xi'_0!0$
$\epsilon'_1?1$