

ANALYSIS OF MPI PROGRAMS

STEPHEN F. SIEGEL AND GEORGE S. AVRUNIN

ABSTRACT. We investigate the application of formal verification techniques to parallel programs which employ the Message Passing Interface (MPI). We develop a formal model of a subset of MPI, and then prove a number of theorems about that model which ameliorate or eliminate altogether the state explosion problem. As an example, we show that if one wishes to verify freedom from deadlock, it suffices to consider only synchronous executions.

1. INTRODUCTION

Message Passing is a widely-used paradigm for the communication between processes in a parallel or distributed system. The basic ideas have been around since the late 1960s, and by the early 1990s, several different and incompatible message-passing systems were being used to develop significant applications. The desire for portability and a recognized standard led to the creation of the *Message Passing Interface* (MPI), which defines the precise syntax and semantics for a library of functions for writing message-passing programs in a language such as C or Fortran (see [2] and [4]). Since that time, MPI has become widely adapted, with proprietary and open-source implementations available on almost any platform.

Developers of programs which use MPI have run into many of the problems typically encountered with concurrent programs. Programs deadlock; they behave non-deterministically due to scheduling or buffering choices made by the MPI implementation; bugs are extremely difficult to pin down or even reproduce. For this reason, formal analysis and verification techniques could prove very useful to MPI programmers.

In this paper we are interested in applying the techniques of Finite State Verification (FSV) to MPI programs. While FSV has been applied to many domains—including various protocols, programming languages, hardware, and other message-passing situations—we have not found in the literature any specific application of it to MPI programs. (See, however, [3] for an attempt to apply FSV to a parallel system which is used in implementing MPI.)

In fact, we will restrict our attention to a small subset of MPI. The advantage of this approach is that one can optimize techniques for the specific domain. Some FSV tools, such as the model checker SPIN ([1]), have very general constructs for modeling a wide range of synchronous and asynchronous communication mechanisms. It is not too difficult to translate, in a straightforward way, our subset of MPI into Promela (the input language for SPIN). The problem is that, for any realistic MPI program, the state space which SPIN must explore is so large as to

Key words and phrases. MPI, Message Passing Interface, parallel computation, formal methods, verification, analysis, finite-state verification, model checking, deadlock, concurrent systems, INCA, SPIN.

Draft version 1.34 of 2003/06/13.

render the analysis infeasible. This “state-space explosion” is a familiar problem in the FSV literature, and many different approaches have been explored to combat it.

One of the primary sources of state-explosion is the use of *buffered* message passing. When messages can be buffered, as they can in MPI, the state of the system must include not only the state of each local process, but also the state of the buffer(s) holding the messages. Moreover, as MPI does not specify any bounds on the sizes of the buffers, a finite-state model must impose an arbitrary upper bound on the number of messages in the buffers. In general it is difficult or impossible to know if the imposition of this bound is conservative: if one has verified that a property holds as long as the buffer size never exceeds, say, 4, how does one know that the property is not violated in an execution in which the buffer reaches size 5? This question, which in general is very difficult to answer, is an example of the kind of question that motivates our inquiry.

In this paper, we first develop a precise, formal model for programs which use a particular subset of MPI, and then prove a number of theorems about that model which ameliorate the state-explosion problem, or which show that certain properties follow automatically or can be verified in a very simple way. Our strategy is to take advantage of features which are peculiar to (our subset of) MPI, and to focus on specific properties or kinds of properties.

Our subset of MPI consists of the following functions: `MPI_SEND`, `MPI_RECV`, `MPI_SENDRECV`, and `MPI_BARRIER`. The precise semantics of these functions are discussed in Section 2. It will become clear in the subsequent sections that much of the complexity of the model derives from including `MPI_SENDRECV` in this list. This function allows a send and receive operation to happen concurrently; we model this by allowing the two operations to occur in either order, which adds another layer of non-determinism to the model. This requires us to develop a language in which we can talk about two paths being essentially the same if they differ only in the order in which they perform the two events within each send-receive.

In Section 3 we define precisely our notion of a model of an MPI program. Our model is suitable for representing the subset of MPI discussed above (though the representation of barriers is deferred until Section 10). In essence, our model consists of a state-machine for each process in the system, and these communicate via buffered channels which have fixed sending and receiving processes.

In Section 4 we give the execution semantics for the model. An execution is represented as a sequence of transitions in the state machines. Sections 5 and 6 establish some technical results which we will allow us to construct new executions from old; these will be used repeatedly in the proofs of the real applications, which begin in Section 7 with an application to the verification of freedom from deadlock.

If the property of concern is freedom from deadlock, and the program satisfies suitable restrictions, then Theorem 7.1 answers precisely the question on buffer depth mentioned above. The Theorem states that, if there exists a deadlocking execution of the program, then there must also exist a deadlocking execution in which all communication is synchronous. This means that if we are using SPIN, for example, to verify freedom from deadlock, we may let all the channels in our model have depth 0. This confirms the intuition underlying a common practice among developers of MPI software: that a good way to check a program for deadlocks is to

replace all the sends with synchronized sends, execute the resulting program, and see if that deadlocks.

The “suitable restrictions” required by Theorem 7.1 essentially amount to the requirement that the program contain no *wildcard receives*, i.e., that it uses neither `MPI_ANY_SOURCE` nor `MPI_ANY_TAG`. In fact, we give a simple counterexample to the conclusion of the Theorem with a program that uses a wildcard receive. This program can not deadlock if all communication is synchronous, but may deadlock if messages are buffered. This demonstrates that considerable complexity is added to the analysis by the use of wildcard receives. In fact, almost all of the results presented here require their absence; we hope, in future work, to develop techniques to handle the additional complexity introduced by wildcard receives.

In Section 8 we prove a similar result for *partial deadlock*: this is where some subset of the processes becomes permanently blocked, though other processes may continue to execute forever. Again, we show that it suffices to check only synchronous executions.

In Section 9, we consider properties such as assertions on whether certain states are reachable. We show one way to obtain an upper bound on channel depths for checking these properties, if certain conditions are met.

In Section 10, we show how one can represent barriers in our formal model. One of the common problems plaguing developers is the question “Is a barrier necessary at this point in the code?”. Adding barriers certainly limits the amount of process interleaving that can take place; but it also can take a huge toll on performance. Thus a technique which could tell developers whether or not the removal of a barrier could possibly lead to the violation of some correctness property would be very useful. While we certainly are not able to answer this question in general, we will take a step in that direction in Section 11. There, we give conditions under which the removal of all barriers from a program cannot possibly introduce deadlocks or partial deadlocks. In Section 12 we also show that if the program with barriers is deadlock-free, then in any execution all the channel buffers must be empty whenever all the processes are “inside” a barrier.

Section 13 deals with a particularly simple class of models of MPI programs, which we call *locally deterministic*. These are models in which there are not only no wildcard receives, but no non-deterministic local choices in the state machine for any process. The only possible states in one of these state machines with multiple departing transitions are those just before a receive operation, and the different transitions correspond to the various possible values that could be received.

If one begins with an MPI program with no wildcard receives and no non-deterministic functions (we call this a *locally deterministic program*), and if one fixes a choice of inputs and execution platform for the program, then one can always build a locally-deterministic model of that program. In some cases, it may also be possible to build a locally deterministic model of the program which is independent of the choices of inputs or platform.

In any case, given a locally deterministic model, we show that the analysis is greatly simplified. For, even though there may still exist many possible executions of the model—due to the interleaving and buffering choices allowed by MPI—these cannot differ by much. In particular, the same messages will always be sent on each channel, and the same paths will be followed in each state machine. The questions

concerning deadlock are shown to be easily answered for locally deterministic models: one need only examine a single execution of the model to verify freedom from deadlock on all executions of that model.

These last results have enormous practical implications. They imply, for example, that given a locally deterministic program, and a fixed platform, that one may check that the program is, say, deadlock-free on given input by executing it only once on that input, making any buffering and interleaving choices one likes. There is no need to explore the space of all possible program executions. Moreover, the values of any variables calculated and the control path taken by any process on that single execution will be the same on any execution (again, this is for a fixed choice of input). Hence for verification purposes, a locally deterministic program may be treated much like a sequential program.

2. MODELING ISSUES

2.1. Memory Model. For our purposes, an MPI program consists of a set of independent processes, each executing as a single thread in its own memory space, and communicating with other processes via the functions defined in the MPI specification ([2]). We assume throughout that there is no shared memory between any two processes in an MPI program. Hence the only way one process can affect another is by invoking one of the MPI functions. While the MPI specification does not specifically prohibit shared memory, neither does it provide any support for it, and it seems that most MPI programs make no use of it. Needless to say, the use of shared memory on top of MPI would greatly complicate the modeling issues.

2.2. Communicators. Each of the MPI functions described here takes, as one of its arguments, an MPI *communicator*. A communicator specifies a set of processes which may communicate with each other through MPI functions. Given a communicator with n processes, the processes are numbered 0 to $n - 1$; this number is referred to as the *rank* of the process in the communicator.

MPI provides a pre-defined communicator, `MPI_COMM_WORLD`, which represents the set of all processes. For simplicity, in this paper we will consider MPI programs which use only this communicator, though in the future we hope to expand our results to deal with arbitrary communicators.

2.3. Send and Receive. MPI provides many ways to send a message from one process to another. Perhaps the simplest is the *standard mode, blocking send* function, which has the form

`MPI_SEND(buf, count, datatype, dest, tag, comm).`

Here `buf` is the address of the first element in the sequence of data to be sent; `count` is the number of elements in the sequence, `datatype` indicates the type of each element in the sequence; `dest` is the rank of the process to which the data is to be sent; `tag` is an integer which may be used to distinguish the message; and `comm` is a handle representing the communicator in which this communication is to take place.

One may think of the message data as being bundled up inside an “envelope.” The envelope includes the rank of the sending process (the *source*), the rank of the receiving process (the *destination*), the tag, and the communicator.

Likewise, there are different functions to receive a message which has been sent; the simplest is probably the MPI *blocking receive*, which has the form

MPI_RECV(buf, count, datatype, source, tag, comm, status).

Here, `buf` is the address for the beginning of the segment of memory into which the incoming message is to be stored. The integer `count` specifies an upper bound on the number of elements of type `datatype` to be received. (At runtime, an overflow error will be generated if the length of the message received is longer than `count`.) The integer `source` is the rank of the sending process and the integer `tag` is the tag identifier. However, unlike the case of `MPI_SEND`, these last two parameters may take the *wildcard* values `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively. The parameter `comm` represents the communicator, while `status` is an “out” parameter used by the function to return information about the message that has been received.

An MPI receive operation will only select a message for reception if the message envelope *matches* the receive parameters in the following sense: of course, the destination of the message must equal the rank of the process invoking the receive, the source of the message must match the value of `source` (i.e., either the two integers are equal, or `source` is `MPI_ANY_SOURCE`), the tag of the message must match the value of `tag` (either the two integers are equal, or `tag` is `MPI_ANY_TAG`), and the communicator of the message must equal `comm`. It is up to the user to make sure that the data-type of the message sent matches the data-type specified in the receive. (In fact, the message tag is often used to facilitate this.)

2.4. Blocking and Buffering. The MPI implementation may decide to buffer an outgoing message. This means a send may complete before the receiving process has received the message. Between the time the message is sent, and the time it is received, the message may be thought of as existing in a *system buffer*. On the other hand, the implementation may decide to block the sending process, perhaps until the size of the system buffer becomes sufficiently small, before sending out the message. Finally, the system may decide to block the sending process until both of the following conditions are met: (a) the receiving process is at a matching receive, and (b) there is no pending message in the system buffer which also matches that receive. If the implementation chooses this last option, we say that it forces this particular send to be *synchronous*.

In fact, this last scenario is the only time the specification *guarantees* that a send which has been initiated will complete. We can also describe precisely the only situation in which a receive which has been initiated is guaranteed to complete. Here the situation is complicated by the fact that the specification allows for multiple threads to run simultaneously within a single process, so that it is possible for a single process to be in a state in which more than one receive has been initiated. If we ignore this situation, then a receive which has been initiated must complete if (a) there exists a matching message in the system buffer, or (b) there is no matching message in the system buffer, but another process initiates a matching send (and so the send and receive may execute synchronously).

2.5. Order. The specification imposes certain restrictions concerning the order in which messages may be received:

Messages are non-overtaking: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives.

These are the only guarantees made concerning order. So, for example, if two different processes send messages to the same receiver, the messages may be received in the order opposite that in which they were sent.

The order restrictions suggest natural ways to model process communication. For example, for each triple

(sender, receiver, tag)

we may create a message channel which behaves as a queue. (We are assuming here that the only communicator used is `MPI_COMM_WORLD`.) A send simply places the message into the appropriate queue. The modeling of a receive is a bit more complicated because of the non-determinism introduced by the wildcards. Without wildcards, a receive simply removes a message from the queue specified by the receive parameters. If the receive uses only the wildcard `MPI_ANY_SOURCE`, then it is free to choose non-deterministically among the queues for the various senders. However, if it uses the wildcard `MPI_ANY_TAG`, the situation is more complicated, since the specification prescribes that for a fixed sender, the receive choose the oldest matching message from among those with different tags. Hence if we allow the use of `MPI_ANY_TAG`, the queues alone may not contain enough information to model the state of the system buffer precisely.

One way to get around this would be to instead create a channel for each pair

(sender, receiver).

Now a receive must be able to pull out the oldest item in the channel with a matching tag. While this is certainly possible (and easily expressed in modeling languages such as Promela), it does complicate the modeling of the state enormously. For this reason, we have chosen—in this paper—to ignore the possibility of `MPI_ANY_TAG`. In fact, while our model can deal precisely with `MPI_ANY_SOURCE`, almost all of the results presented here require that one does not use either wildcard.

2.6. Variations. The send and receive operations described above are called *blocking* for the following reason: after the return of a call to `MPI_SEND`, one is guaranteed that all the data has been copied out of the send buffer—either to the system buffer or directly to the receiving process. Therefore it is safe to reuse the memory in the send buffer, as this can no longer affect the message in any way. Likewise, upon the return of an invocation of `MPI_RECV`, one is guaranteed that the incoming message has been completely copied into the receive buffer. Therefore it is safe to immediately read the message from that buffer. Hence `MPI_SEND` “blocks” until the system has completed copying the message out of the send buffer, and `MPI_RECV` “blocks” until the system has completed copying the message into the receive buffer.

MPI also provides *nonblocking* send and receive operations. A non-blocking send may return immediately—even before the message has been completely copied out

of the send buffer. A nonblocking send operation returns a “request handle”; a second function (`MPI_WAIT`) is invoked on the handle, and this will block until the system has completed copying the message out of the buffer. By decomposing the send operation in this way, one may achieve a greater overlap between computation and communication, by placing computational code between the send and the wait. The non-blocking receive is similar.

In addition, both the blocking and non-blocking sends come in several *modes*. We have described the *standard* mode, but there are also the *synchronous* mode, *buffered* mode, and *ready* mode variants. The synchronous mode forces the send to block until it can execute synchronously with a matching receive, the buffered mode always chooses to buffer the outgoing message (as long as sufficient space is available in the system buffer), and the ready mode can be used as long as a matching receive will be invoked before the send is (else a runtime error is generated).

In this paper we are concerned only with the blocking operations and the standard mode send; we hope to extend these results to the more general non-blocking operations and the other send modes in future work. Nevertheless, the results here may still have some bearing on MPI programs which use the buffered and synchronous mode blocking sends. For any MPI program which satisfies a property \mathcal{P} on all possible executions must necessarily satisfy \mathcal{P} on all possible executions after any or all of the `MPI_SEND` statements are replaced with their synchronous or buffered mode variants. This is because the specification always allows an `MPI_SEND` to choose to synchronize or buffer. In fact, a program which depends upon specifying the buffered or synchronous mode for correctness should probably be considered “unsafe,” and it may be reasonable to expect programmers to design and verify their programs using the standard mode, and only after that change to one of the other two modes (perhaps for reasons of performance).

The ready mode is special: it is possible that a correct program will break if a standard mode send is replaced with a ready mode send, so the results of this paper will not directly apply to this case.

2.7. Send-Receive. One often wishes to have two processors exchange data: process 0 sends a message to process 1 and receives from process 1, while process 1 sends to 0 and receives from 0. More generally, one may wish to have a set of processors exchange data in a cyclical way: process 0 sends to 1 and receives from n , 1 sends to 2 and receives from 0, ..., n receives from $n - 1$ and sends to 0. In both cases, each process must execute one send and one receive. One must be careful, however: if this is coded so that each process first sends, then receives, the program will deadlock if the MPI implementation chooses to synchronize all the sends. While there are ways to get around this, the situation occurs frequently enough that MPI provides a special function that executes a send and a receive in one invocation, without specifying the order in which the send and receive are to occur. The semantics are defined so that execution of this function is equivalent to having the process fork off two independent threads—one executing the send, the other the receive. The function returns when both have completed.

This function has the form

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag,  
             rcvbuf, rcvcount, rcvtype, source, rcvtag, comm, status)
```

The first 5 parameters are the usual ones for send, the next 5 together with `status` are the usual ones for receive, and `comm` specifies the communicator for both the send and receive.

2.8. **Barriers.** The function

MPI_BARRIER(comm)

is used to force all processes in the communicator `comm` to synchronize at a certain point. Upon invocation, we say that a process has *entered* the barrier. That process is then blocked until all the processes (in the communicator) are “inside” the barrier (i.e., have invoked but not yet returned from a call to the barrier function). Once this happens, the function may return, and we say that the process *exits* the barrier.

3. MODELS

In this section, we describe precisely what we mean by a *model of an MPI program*.

3.1. **Context.** First, we describe what we call a *context*. This is a tuple

$$\mathcal{C} = (\text{Proc}, \text{Chan}, \text{sender}, \text{receiver}, \text{msg}, \text{loc}, \text{comm}).$$

We describe each of these components in turn.

First, `Proc` is a finite set. It represents the set of *processes* in the MPI program.

Second, `Chan` is also a finite set, representing the set of communication *channels*. Next, `sender` and `receiver` are functions from `Chan` to `Proc`. The idea is that each channel is used exclusively to transfer messages from its sender process to its receiver process. (Recall from Section 2 that, in creating a model from code, we intend to make one channel for each triple (`sender`, `receiver`, `tag`). It is because of the tags that we allow the possibility of having more than one channel from one process to another.)

Next, `msg` is a function which assigns, to each $c \in \text{Chan}$, a (possibly infinite) nonempty set $\text{msg}(c)$, representing the set of messages which can be sent over channel c .

Now `loc` is a function which assigns, to each $p \in \text{Proc}$, a (possibly infinite) set $\text{loc}(p)$, representing the set of *local events* for process p . We require that

$$p \neq q \Rightarrow \text{loc}(p) \cap \text{loc}(q) = \emptyset.$$

Finally, `comm` is the function which assigns, to each $p \in \text{Proc}$, the set consisting of all triples (c, send, x) , where $c \in \text{Chan}$ and $\text{sender}(c) = p$, and $x \in \text{msg}(c)$, together with all triples $(d, \text{receive}, y)$, where $d \in \text{Chan}$ and $\text{receiver}(d) = p$, and $y \in \text{msg}(d)$. This is the set of all *communication events* for p . We abbreviate (c, send, x) as $c!x$ and $(d, \text{receive}, y)$ as $d?y$. The first represents the event in which process p inserts message x into channel c , and the second represents the event in which p removes message y from d .

We also require that $\text{loc}(p) \cap \text{comm}(q) = \emptyset$ for all $p, q \in \text{Proc}$, and we define

$$\text{event}(p) = \text{loc}(p) \cup \text{comm}(p).$$

That completes the definition of a *context* \mathcal{C} .

3.2. MPI State Machines. Suppose we are given a context \mathcal{C} as above and a $p \in \text{Proc}$. We will define what we mean by an *MPI state machine for p under \mathcal{C}* . This is a tuple

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

We define the components of M in order.

First, **States** is a (possibly infinite) set. It represents the set of possible *states* in which process p may be during execution.

Second, **Trans** is a (possibly infinite) set, representing the set of possible transitions from one state of p to another during execution. Now, **src** and **des** are just functions from **Trans** to **States**. These are interpreted as giving the *source state* and *destination state*, respectively, for a transition.

Next, **label** is a function from **Trans** to $\text{event}(p)$.

Finally, **start** is an element of **States**, and **End** is a (possibly empty) subset of **States**. The former represents the initial state of process p , and elements of the latter represent the state of p after process termination. We allow the possibility that $\text{start} \in \text{End}$. Furthermore, we require that there is no $t \in \text{Trans}$ with $\text{src}(t) \in \text{End}$.

We are not quite finished with the definition of *MPI state machine*. We also require that for each $u \in \text{States}$, exactly one of the following hold:

- (i) The state $u \in \text{End}$. We call this a *final state*.
- (ii) The state $u \notin \text{End}$, there is at least one $t \in \text{Trans}$ with $\text{src}(t) = u$, and, for all such t , $\text{label}(t) \in \text{loc}(p)$. In this case we say u is a *local state*.
- (iii) There is exactly one $t \in \text{Trans}$ such that $\text{src}(t) = u$, and $\text{label}(t)$ is a communication event of the form $c!x$. We say u is a *sending state*.
- (iv) There is a nonempty subset R of **Chan** with $\text{receiver}(d) = p$ for all $d \in R$, such that the restriction of **label** to the set of transitions departing from u is a 1-1 correspondence onto the set of events of the form $d?y$, where $d \in R$ and $y \in \text{msg}(d)$. We say u is a *receiving state*.
- (v) (See Figure 1.) There is a $c \in \text{Chan}$, a nonempty subset R of **Chan**, an $x \in \text{msg}(c)$, a state u' , and states $v(d, y)$, and $v'(d, y)$, for all pairs (d, y) with $d \in R$ and $y \in \text{msg}(d)$, such that the following all hold:
 - For all $d \in R$, $\text{sender}(c) = p$ and $\text{receiver}(d) = p$.
 - The states u, u' , and the $v(d, y)$ and $v'(d, y)$ are all distinct.
 - The set of transitions departing from u consists of one transition to u' whose label is $c!x$, and, for each (d, y) , one transition labeled $d?y$ to $v(d, y)$. Furthermore, these are the only transitions terminating in u' or $v(d, y)$.
 - For each (d, y) , there is precisely one transition departing from $v(d, y)$, it is labeled $c!x$, and it terminates in $v'(d, y)$.
 - For each (d, y) , 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' .
 - For each (d, y) , the only transitions terminating in $v'(d, y)$ are the two already mentioned: one from u' and one from $v(d, y)$.

We call u a *send-receive state*. Notice that u' is a receiving state, and each $v(d, y)$ is a sending state.

This completes the definition of an *MPI state machine for p under \mathcal{C}* .

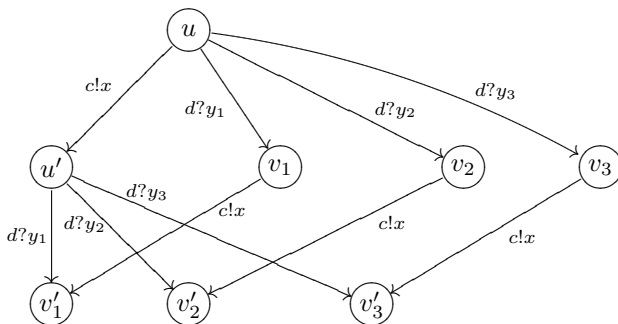


FIGURE 1. A send-receive state u with one receiving channel d .

Finally, a *model of an MPI program* is a pair $\mathcal{M} = (\mathcal{C}, M)$, where \mathcal{C} is a context and M is a function which assigns, to each $p \in \text{Proc}$, an MPI state machine

$$M_p = (\text{States}_p, \text{Trans}_p, \text{src}_p, \text{des}_p, \text{label}_p, \text{start}_p, \text{End}_p)$$

for p under \mathcal{C} , such that

$$\text{States}(p) \cap \text{States}(q) = \emptyset = \text{Trans}(p) \cap \text{Trans}(q)$$

for $p \neq q$. We will leave off the subscript “ p ” for the maps src_p , des_p , and label_p , since each transition is in precisely one M_p so there is no need to specify p .

A *global state* of \mathcal{M} is a map U which assigns to each $p \in \text{Proc}$ an element $U_p \in \text{States}_p$.

We say that \mathcal{M} *has no wildcard receives* if, in each M_p , the sets R for the receiving or send-receive states all have cardinality one.

3.3. Building Models From Code. Let us briefly describe how our model relates to actual program code. The idea of representing an ordinary sequential program as a finite state machine goes back to the dawn of computer science. The basic idea is that one creates a state for every possible combination of values of all variables, including the program counter, which represents a position in the program code. If there are functions then the call stack must also be included in the state, and if there is a heap then this may also have to be represented. On the other hand, by means of abstraction, one may include less information in the state and therefore reduce the total number of states in the model.

In creating a model from code, one usually wishes to employ abstraction in such a way that the resulting model is still *conservative*. This means that every execution of the actual program is represented by an execution of the model, though there may be additional executions of the model which do not correspond to actual executions of the program. If one proves that a property holds on every execution in a conservative model, then one is guaranteed that the property holds on every execution of the program. On the other hand, if an execution of the model is found which violates the property, there is the possibility that that execution is *spurious*, i.e., it does not correspond to an actual execution of the program.

A state in an MPI state machine M_p represents the local state of process p —the values of its variables and position counter, etc. A local transition represents a change in state in p that does not involve any communication with other processes—for example, the assignment of a new value to a variable. The labels on the local

transitions do not play a significant role in this paper. One could, for example, just use a single label for all the local transitions in a process. On the other hand, if one wishes to reason about particular local transitions in a correctness property, one could use different local labels for those transitions so that they could be referenced in the property specification.

There are several reasons why we allow *local non-determinism* in our models, i.e., a local state with more than one departing transition. One of these is the result of abstraction. Suppose, for example, that we have chosen to abstract away a variable x from the state, i.e., the local state no longer contains a component for the value of x . Suppose now that there is a local state s , which represents a point in the code just before a statement of the following form:

```
if (x == 0) then {...} else {...}.
```

If we want our model to be conservative then we must allow two transitions to depart from s —one for each branch in this conditional statement—since this local state does not “know” the value of x .

Other cases in which we might have multiple local transitions departing from a state include the case where we wish the model to simultaneously represent the same program operating on various inputs, and the case where the program code uses an actual non-deterministic function, such as a function which returns a random value.

All of this is standard, and so we concentrate now on what is particular to the representation of MPI communication in our model.

As we have already pointed out, we create one channel in the model for each triple

$$(\text{sender}, \text{receiver}, \text{tag}),$$

where **sender** and **receiver** are any processes for which the former might possibly send a message to the latter, and **tag** is any tag which might be used in such a communication. The order restrictions imposed by the MPI specification imply that this channel should behave like a queue: the messages enter and exit it in first-in, first-out order.

The sets $\text{msg}(c)$ are likely candidates for abstraction. A typical MPI program might send blocks of 100 floating point numbers across c . In this case we could, in theory, use no abstraction at all, and take $\text{msg}(c)$ to be the set of all vectors of length 100 of floating point numbers. We would then have a fully precise representation of the messages being sent along c . While it might be useful to reason theoretically about such a model, the size of the state space would probably make the model intractable for automated finite-state verification. On the other end of the abstraction spectrum, we could take $\text{msg}(c)$ to contain a single element, say $\text{msg}(c) = \{1\}$. If the property we wish to verify cannot be influenced by the actual data in the messages, this may be a perfectly reasonable abstraction. Of course, there are many choices between these two extremes, and finding appropriate abstractions for the program and property of interest is part of the art of model creation.

A sending state represents a point in code just before a send operation. At that point, the local state of the process invoking the send contains all the information needed to specify the send exactly: the value to be sent, the process to which the information is to be sent, and the tag. After the send has completed, the state of this process is exactly as it was before, except for the program counter, which

has now moved to the position just after the send statement. That is why there is precisely one transition departing from the sending state.

A receiving state represents a point in code just before a receive operation. Unlike the case for send, this process does not know, at that point, what value will be received. The value received will of course change the state of this process—the variable or buffer into which the data is stored will take on a new value or values. Hence a transition to a different state must be included for every possible value that could be received. If this is a wildcard receive (because `MPI_ANY_SOURCE` is used as the source parameter in the code), then we must also allow a different set of transitions for each of the possible senders.

A send-receive state represents a point in code just before a send-receive statement. According to the MPI specification, the send and receive operations may be thought of as taking place in two concurrent threads; we model this by allowing the send and receive to happen in either order. If the send happens first, this process then moves to a receiving state, whereas if the receive happens first, the process moves to one of the sending states. After the second of these two operations occurs, the process moves to a state which represents the completion of the send-receive statement. Notice that there is always a “dual path” to this state, in which the same two operations occur in the opposite order.

4. EXECUTION SEMANTICS

We will represent executions of a model of an MPI program as sequences of transitions. For simplicity, our representation will not distinguish between the case where a send and receive happen synchronously, and the case where the receive happens immediately after the send, with no intervening events. It is clear that in the latter case, the send and receive *could* have happened synchronously, as long as the sending and receiving processes are distinct.

4.1. Definitions. A sequence $S = (x_1, x_2, \dots)$ may be either infinite or finite. We write $|S|$ for the length of S ; we allow $|S| = \infty$ if S is infinite. By the *domain* of S , we mean the set of all integers which are greater than or equal to 1 and less than or equal to $|S|$. For i in the domain of S , define $S|_i = x_i$.

If S and T are sequences, and S is a prefix of T , we write $S \subseteq T$; this allows the possibility that $S = T$. If S is a proper prefix of T , we write $S \subset T$.

We also say T is an *extension* of S if $S \subseteq T$, and T is a *proper extension* of S if $S \subset T$.

If $S \subseteq T$ then we define a sequence $T \setminus S$ as follows: if $S = T$ then we let $T \setminus S$ be the empty sequence. Otherwise, S must be finite, and if $T = (x_1, x_2, \dots)$, we let

$$T \setminus S = (x_{|S|+1}, x_{|S|+2}, \dots).$$

If A is a subset of B , and S is a sequence of elements of B , then *the projection of S onto A* is the sequence which results by deleting from S all elements which are not in A .

If S is any sequence and n is a non-negative integer, then S^n denotes the sequence obtained by truncating S after the n^{th} element. In other words, if $|S| \leq n$, then $S^n = S$, otherwise, S^n consists of the first n elements of S .

We now define the execution semantics of a model of an MPI program. Fix a model $\mathcal{M} = (\mathcal{C}, M)$ as in Section 3.

Let $S = (t_1, t_2, \dots)$ be a (finite or infinite) sequence of elements of Trans , and $c \in \text{Chan}$. Let $(c!x_1, c!x_2, \dots)$ denote the projection of

$$(\text{label}(t_1), \text{label}(t_2), \dots)$$

onto the set of events which are sends on c . Then define

$$\text{Sent}_c(S) = (x_1, x_2, \dots).$$

This is the sequence of messages which are sent on c in S . The sequence $\text{Received}_c(S)$ is defined similarly as the sequence of messages which are received on c in S .

We say that S is *c-legal* if for all n in the domain of S ,

$$\text{Received}_c(S^n) \subseteq \text{Sent}_c(S^n).$$

This is exactly what it means to say that the channel c behaves like a queue. If S is *c-legal*, we define

$$\text{Queue}_c(S) = \text{Sent}_c(S) \setminus \text{Received}_c(S).$$

This represents the messages remaining in the queue after the last step of execution in S .

Suppose next that we are given a sequence $\pi = (t_1, t_2, \dots)$ in Trans_p for some $p \in \text{Proc}$. We say that π is a *path through M_p* if π is empty, or if $\text{src}(t_1) = \text{start}_p$, and $\text{des}(t_i) = \text{src}(t_{i+1})$ for all $i \geq 1$ for which $i + 1$ is in the domain of π .

If $\pi = (t_1, \dots, t_n)$ is a finite path through M_p , we let

$$\text{terminus}(\pi) = \begin{cases} \text{des}(t_n) & \text{if } n \geq 1 \\ \text{start}_p & \text{otherwise.} \end{cases}$$

Given any sequence S of transitions, and $p \in \text{Proc}$, we let $S \downarrow_p$ denote the projection of S onto Trans_p .

Finally, we define an *execution prefix* of \mathcal{M} to be a sequence S of transitions such that (i) for each $p \in \text{Proc}$, $S \downarrow_p$ is a path through M_p , and (ii) for each $c \in \text{Chan}$, S is *c-legal*.

Given a finite execution prefix S , the *terminal state of S* is the global state $\text{terminus}(S)$ defined by $\text{terminus}(S)_p = \text{terminus}(S \downarrow_p)$.

4.2. Synchronous executions. Let $S = (t_1, t_2, \dots)$ be an execution prefix and $c \in \text{Chan}$. We say that S is *c-synchronous* if, for all i in the domain of S for which $\text{label}(t_i)$ is a send, say $c!x$, the following all hold:

- $i + 1$ is in the domain of S .
- $\text{label}(t_{i+1}) = c?x$.
- $\text{sender}(c) \neq \text{receiver}(c)$ or $\text{src}(t_i)$ is a send-receive state.

We say that S is *synchronous* if S is *c-synchronous* for all $c \in \text{Chan}$.

The idea is the following: we are representing executions of a model of an MPI program as sequences of transitions. For simplicity, our representation does not distinguish between the case where a send and receive happen synchronously, and the case where the receive happens immediately after the send, with no intervening events. In general, in the latter case, the send and receive *could* have happened synchronously. The exception to this rule is the somewhat pathological case in which a process sends a message to itself and then receives the message. If this is done with an `MPI_SEND` followed by an `MPI_RECEIVE`, then that send can never happen synchronously, as it is impossible for the process to be at two positions at once.

However, if it is done with an `MPI_SENDRECV`, it may happen synchronously since the send and receive are thought of as taking place in two independent processes. This is the reason for the third item in the list above.

4.3. SRC equivalence and associated synchronous prefixes. Let \mathcal{M} be a model of an MPI program, $p \in \text{Proc}$, and $\pi = (t_1, t_2, \dots)$ a path through M_p .

Suppose that k and $k + 1$ are in the domain of π and that $u = \text{src}(t_k)$ is a send-receive state. Then one of t_k, t_{k+1} is labeled by a send and the other by a receive. Assume also that the sending and receiving channels are distinct. (If the sending and receiving channels are the same, then we are in the case where a process is sending a message to itself using the same sending and receiving tag.) We define the *dual path* \bar{t}_{k+1}, \bar{t}_k by

$$\begin{aligned} u &= \text{src}(\bar{t}_{k+1}) \\ \text{des}(\bar{t}_{k+1}) &= \text{src}(\bar{t}_k) \\ \text{label}(\bar{t}_i) &= \text{label}(t_i) \quad (i \in \{k, k + 1\}). \end{aligned}$$

This is just the path around the send-receive diamond that performs the same send and receive, but in the opposite order. Now let π' be the sequence obtained from π by replacing the subsequence (t_k, t_{k+1}) with the dual path $(\bar{t}_{k+1}, \bar{t}_k)$. Then π' is also a path, since the two subsequences start at the same state, and end at the same state.

If ρ is also a path through M_p , and ρ can be obtained from π by applying a (possibly infinite) set of transformations like the one above, we say that π and ρ are *equivalent up to send-receive commutation*, or *src-equivalent*, for short, and we write $\pi \sim \rho$. Clearly, src-equivalence is an equivalence relation on the set of paths through M_p . It is also clear that, if $\pi \sim \rho$, then $\text{terminus}(\pi) = \text{terminus}(\rho)$.

Lemma 4.1. *Let S and T be execution prefixes for \mathcal{M} . Let $c \in \text{Chan}$, $p = \text{sender}(c)$, and $q = \text{receiver}(c)$. Suppose $S \downarrow_p \sim T \downarrow_p$ and $S \downarrow_q \sim T \downarrow_q$. Then $\text{Sent}_c(S) = \text{Sent}_c(T)$, $\text{Received}_c(S) = \text{Received}_c(T)$, and $\text{Queue}_c(S) = \text{Queue}_c(T)$.*

Proof. This is clear from the definition of src-equivalence, since the transformations do not change the projections of the event-sequences onto any channel. \square

We now define some relations on the set of paths through M_p .

Definition 4.2. If π and ρ are paths through M_p , we say $\pi \preceq \rho$ if π is a prefix of a sequence which is src-equivalent to ρ . We write $\pi \prec \rho$ if $\pi \preceq \rho$ but $\pi \not\sim \rho$.

The following is easily verified:

Lemma 4.3. *Let π, π', ρ, ρ' , and σ be paths through M_p . Then*

- (1) *If $\pi \sim \pi'$ and $\rho \sim \rho'$ then $\pi \preceq \rho \iff \pi' \preceq \rho'$.*
- (2) *$\pi \preceq \pi$.*
- (3) *If $\pi \preceq \rho$ and $\rho \preceq \sigma$ then $\pi \preceq \sigma$.*
- (4) *If $\pi \preceq \rho$ and $\rho \preceq \pi$ then $\pi \sim \rho$.*

In other words, \preceq induces a partial order on the set of src-equivalence classes of paths through M_p .

Definition 4.4. If π and ρ are paths through M_p , we say that π and ρ are *comparable* if $\pi \preceq \rho$ or $\rho \preceq \pi$.

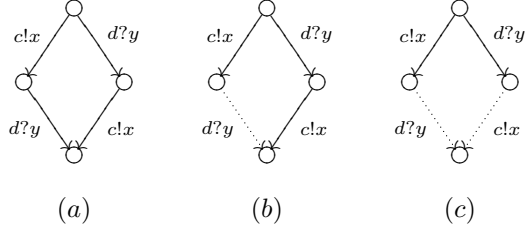


FIGURE 2. Compatible paths. In each case, ρ , the path on the left (excluding the dotted arrows), is compatible with σ , the path on the right. In (a), $\rho \sim \sigma$. In (b), $\rho \prec \sigma$. In (c), ρ and σ are non-comparable compatible paths. In all cases, π may be taken to be either of the paths from the top node to the bottom node of the diamond.

Suppose S is any finite execution prefix of \mathcal{M} . Consider the set consisting of all synchronous execution prefixes S' with $S' \downarrow_p \preceq S \downarrow_p$ for all $p \in \text{Proc}$. This set is finite, and it is not empty, as it contains the empty sequence as a member. Let T be an element of this set of maximal length. We say that T is an *associated synchronous prefix* for S .

5. COMPATIBLE PREFIXES

Let \mathcal{M} be a model of an MPI program. Let ρ and σ be paths through M_p for some $p \in \text{Proc}$. We say ρ and σ are *compatible* if there exists a path π through M_p such that $\rho \preceq \pi$ and $\sigma \preceq \pi$ (see Figure 2). The following is not hard to verify:

Lemma 5.1. *Suppose $\rho = (s_1, s_2, \dots)$ and $\sigma = (t_1, t_2, \dots)$ are compatible paths through M_p . If $|\rho| \neq |\sigma|$, then $\rho \prec \sigma$ or $\sigma \prec \rho$. If $|\rho| = |\sigma|$ then either $\rho \sim \sigma$ or ρ is finite, say $|\rho| = n$, and all of the following hold:*

- (i) $\text{terminus}(\rho^{n-1}) = \text{terminus}(\sigma^{n-1})$ is a send-receive state.
- (ii) $\rho^{n-1} \sim \sigma^{n-1}$.
- (iii) One of s_n, t_n is a send, and the other a receive.
- (iv) If $\pi = (s_1, \dots, s_n, \bar{t}_n)$ or $\pi = (t_1, \dots, t_n, \bar{s}_n)$, where \bar{s}_n and \bar{t}_n are chosen so that (s_n, \bar{t}_n) is the dual path to (t_n, \bar{s}_n) , then $\rho \preceq \pi$ and $\sigma \preceq \pi$.

The last case in the Lemma describes precisely the non-comparable compatible paths.

We say that two execution prefixes S and T of \mathcal{M} are *compatible* if $S \downarrow_p$ is compatible with $T \downarrow_p$ for all $p \in \text{Proc}$.

Lemma 5.2. *Let \mathcal{M} be a model of an MPI program with no wildcard receives. Let $S = (s_1, \dots, s_n)$ be a finite execution prefix and T an arbitrary execution prefix for \mathcal{M} . Suppose S^{n-1} is compatible with T and s_n is a send or receive. Then S is compatible with T .*

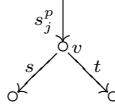
Proof. Let $s = s_n$. Say $s \in \text{Trans}_p$. Clearly $T \downarrow_r$ and $S \downarrow_r$ are compatible for all $r \neq p$. Write

$$\begin{aligned} S \downarrow_p &= (s_1^p, \dots, s_j^p, s) \\ T \downarrow_p &= (t_1^p, t_2^p, \dots). \end{aligned}$$

Let $\sigma = (s_1^p, \dots, s_j^p)$.

By hypothesis, σ is compatible with $T \downarrow_p$. If $T \downarrow_p \preceq \sigma$ then $T \downarrow_p \preceq \sigma \preceq S \downarrow_p$ and so $T \downarrow_p$ and $S \downarrow_p$ are compatible and we are done. So there remain two possibilities to consider: (a) $\sigma \prec T \downarrow_p$, and (b) σ and $T \downarrow_p$ are non-comparable compatible paths.

Consider first case (a). Then there is a proper extension τ of σ which is src-equivalent to $T \downarrow_p$. Let $t = \tau|_{j+1}$ and $v = \text{src}(t) = \text{src}(s)$. It suffices to show that $S \downarrow_p$ and τ are compatible. As this is certainly the case if $s = t$, we will assume $s \neq t$, and we have the following configuration:

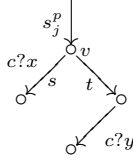


Suppose now that s and t are both receives. Since there are no wildcard receives, s and t must be receives on the same channel c ; say $\text{label}(s) = c?x$ and $\text{label}(t) = c?y$, where $c \in \text{Chan}$ and $x, y \in \text{msg}(c)$. Say that s is the i^{th} receive on c in S . Then t is also the i^{th} receive on c in τ . Hence

$$y = \text{Received}_c(\tau)|_i = \text{Received}_c(T)|_i = \text{Sent}_c(T)|_i,$$

Similarly, $x = \text{Sent}_c(S^{n-1})|_i$. However, $S^{n-1} \downarrow_q$ and $T \downarrow_q$ are compatible, where $q = \text{sender}(c)$. So $\text{Sent}_c(S^{n-1})$ and $\text{Sent}_c(T)$ are prefixes of a common sequence. This means $x = y$, and so $s = t$, a contradiction.

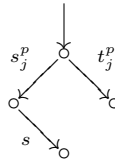
Suppose instead that v is a send-receive state and that s is a receive and t a send. If $|\tau| = j + 1$ then $S \downarrow_p$ and τ are non-comparable compatible paths, and we are done. So suppose $|\tau| \geq j + 2$. Then $\text{label}(\tau|_{j+2}) = c?y$ and $\text{label}(s) = c?x$ for some $c \in \text{Chan}$ and $x, y \in \text{msg}(c)$:



If s is the i^{th} receive on c in S then $\tau|_{j+2}$ is the i^{th} receive on c in τ , and, arguing as in the paragraph above, we arrive at $x = y$. This means $S \downarrow_p \preceq \tau$ and hence that $S \downarrow_p$ and τ are compatible.

Suppose instead that s is a send and t a receive. If $|\tau| = j + 1$ then $S \downarrow_p$ and τ are non-comparable compatible paths and we are done. If not, then $\tau|_{j+2}$ is labeled $c!x$ and so $S \downarrow_p \preceq \tau$ and again we are done.

We now turn to case (b). By Lemma 5.1, $\text{src}(s_j^p) = \text{src}(t_j^p)$ is a send-receive state and $|T \downarrow_p| = j$:



Suppose first that s_j^p is the send and t_j^p the receive. Then we must have $\text{label}(s) = c?x$ and $\text{label}(t_j^p) = c?y$ for some $c \in \text{Chan}$ and $x, y \in \text{msg}(c)$. Arguing as before, we have $x = y$, whence $T \downarrow_p \preceq S \downarrow_p$. If instead s_j^p is the receive and t_j^p is the send

then $\text{label}(s) = \text{label}(t_j^p)$ and $T \downarrow_p \preceq S \downarrow_p$ in this case as well. In either case, we have shown that $T \downarrow_p$ and $S \downarrow_p$ are compatible, completing the proof. \square

6. UNIVERSALLY PERMITTED EXTENSIONS

Let $S = (s_1, \dots, s_m)$ be a finite execution prefix for \mathcal{M} . A finite execution prefix $T = (s_1, \dots, s_m, \dots, s_n)$ extending S is said to be *universally permitted* if for all i such that $m < i \leq n$ and s_i is a send, say $\text{label}(s_i) = c!x$, then $\text{Queue}_c(T^{i-1})$ is empty and $\text{label}(s_{i+1}) = c?x$. The idea is that the universally permitted extensions are precisely the ones that must be allowed in *any* legal MPI implementation. Note that S is a universally permitted extension of itself.

Notice also that if S is synchronous, then a universally permitted extension of S is the same thing as a synchronous extension of S .

We also observe that the set of sequences which may be appended to S to create universally permitted extensions depends only on the states $\text{terminus}(S)_p$ and the sequences $\text{Queue}_c(S)$ (and not on the history of how one arrived at those states and queues). From this observation, it follows that if S and S' are two finite execution prefixes such that $S \downarrow_p \sim S' \downarrow_p$ for all $p \in \text{Proc}$, then there is a 1-1 correspondence between the universally permitted extensions of S and those of S' , with the property that if T corresponds to T' under this correspondence, then $T \setminus S = T' \setminus S'$.

Suppose now that we are given a fixed, finite, execution prefix T , and a second execution prefix S which is compatible with T . We will often have a need to extend S , in a universally permitted way, so that it maintains compatibility with T . The following Proposition shows that, if we extend S far enough in this way, then we reach a point where any further universally permitted extension *must* be compatible with T . An even stronger statement can be made if T is synchronous. Like most of the results in this paper, we require that there are no wildcard receives.

Proposition 6.1. *Let \mathcal{M} be a model of an MPI program with no wildcard receives. Let S and T be compatible finite execution prefixes for \mathcal{M} . Then there is a universally permitted finite extension S' of S , with the property that any universally permitted extension of S' is compatible with T . Moreover, if T is synchronous, then S' may be chosen so that $T \downarrow_p \preceq S' \downarrow_p$ for all $p \in \text{Proc}$.*

Let us look at an example using the model illustrated in Figure 3. We will take

$$T = (2, 3, 2, 3, 1, 9, 10, 9, 10, 8, 6, 7).$$

To summarize T , first Process 0 sends two messages on c and terminates, then Process 2 sends two messages on d and terminates, then Process 1 chooses the d branch and receives one message on d . Suppose that

$$S = (2, 3, 9, 10).$$

Clearly, S is compatible with T , as $S \downarrow_p \prec T \downarrow_p$ for all p . Now, there are many universally permitted extensions of S which are not compatible with T , for example

$$(2, 3, 9, 10, 5, 4).$$

However, let us consider the following universally permitted extension of S :

$$S' = (2, 3, 9, 10, 6, 7, 9, 10, 7, 2, 8).$$

We have $S' \downarrow_0 \prec T \downarrow_0$, $T \downarrow_1 \prec S' \downarrow_1$, and $S' \downarrow_2 = T \downarrow_2$. Clearly, no extension of S' could receive any messages on c , and therefore no universally permitted extension

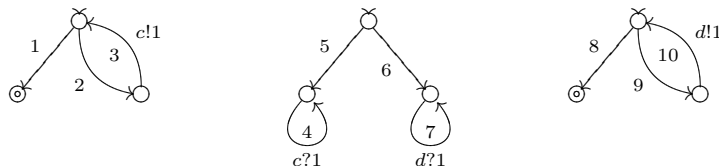


FIGURE 3. A Model of an MPI Program with 3 Processes. Edges with no label are local. Process 0 (left) chooses to either terminate, or to move to a state from which it will send a message to Process 1 on channel c , then return to its start state and repeat. Process 2 (right) does the same for channel d . Process 1 (middle) chooses, once and for all, whether to move to a state from which it will loop forever receiving messages on c , or to do that for d .

of S' could send any more messages on c . This means that no universally permitted extension of S' can have any additional transitions in Process 0. Since in the other two processes, S' has already “covered” T , any universally permitted extension of S' will be compatible with T . So S' is the sort of prefix whose existence is guaranteed by the Proposition.

The second part of the Proposition says that if T were synchronous, then there would exist an S' which “covers” T on every process.

The idea behind the proof of Proposition 6.1 will be to choose an S' which maximizes its “coverage” of T . To make this precise, we introduce the following function. Suppose ρ and σ are compatible paths through some M_p and ρ is finite. We want to measure the part of σ that is not “covered” by ρ . Recall from Lemma 5.1 that if $|\sigma| > |\rho|$ we must have $\pi \prec \sigma$, while if $|\sigma| \neq |\rho|$, are src-equivalent except possibly for the last transition in each. Define

$$d_\sigma(\rho) = \begin{cases} \max(0, |\sigma| - |\rho|) & \text{if } |\sigma| \neq |\rho| \\ 1 & \text{if } |\sigma| = |\rho| \text{ but } \sigma \not\approx \rho \\ 0 & \text{if } \sigma \sim \rho. \end{cases}$$

Figure 4 shows the graph of $d_\sigma(\rho)$ as a function of the length of ρ for the case where $|\sigma| = 4$. Note that $d_\sigma(\rho)$ is a nonincreasing function of $|\rho|$.

It follows from this that if ρ' is also compatible with σ , and $\rho \preceq \rho'$, then $d_\sigma(\rho) \geq d_\sigma(\rho')$.

We now turn to the proof of Proposition 6.1. Let S and T be as in the statement of the Proposition. For any execution prefix R which is compatible with T , and $p \in \text{Proc}$, define

$$d_p(R) = d_{\sigma(p)}(R \downarrow_p),$$

where $\sigma(p) = T \downarrow_p$. Define

$$d(R) = \sum_{p \in \text{Proc}} d_p(R).$$

$$d_\sigma(\rho)$$

$$|\rho|$$

FIGURE 4. $d_\sigma(\rho)$ as a function of $|\rho|$

Next, consider the set of all universally permitted finite extensions of S which are compatible with T , and let S' be an element of that set which minimizes the function d . (The set of such extensions is non-empty, as S is a universally permitted extension of itself.) It follows from the previous paragraph that if R is any extension of S' which is compatible with T then $d_p(R) \leq d_p(S')$ for all $p \in \text{Proc}$; so if R is also universally permitted then in fact we must have $d_p(R) = d_p(S')$ for all p .

Let $\tilde{S} = (s_1, s_2, \dots)$ be a universally permitted extension of S' . We will assume \tilde{S} is not compatible with T and arrive at a contradiction.

Let n be the greatest integer such that \tilde{S}^n is compatible with T . Let $\sigma = \tilde{S}^n \downarrow_p$, where $s_{n+1} \in \text{Trans}_p$. Clearly $n \geq |S'|$ since S' is compatible with T . By Lemma 5.2, s_{n+1} must be a local transition. From this and Lemma 5.1, it also follows that $\sigma \preceq T \downarrow_p$. Therefore σ is a prefix of a sequence $\tau = (t_1, t_2, \dots)$ which is src-equivalent to $T \downarrow_p$. We must have $|\tau| > |\sigma|$ since τ cannot be compatible with $\tilde{S}^{n+1} \downarrow_p$. So s_{n+1} and $t = t_{|\sigma|+1}$ are distinct local transitions departing from the same state.

Consider the sequence $R = (s_1, \dots, s_n, t)$. Then R is an execution prefix, it is a universally permitted extension of S' , and it is compatible with T . However, $d_p(R) \leq d_p(S') - 1$, a contradiction, completing the proof of the first part of the Proposition.

Now suppose that T is synchronous. By replacing S with S' , we may assume that S and T are compatible and that any universally permitted extension R of S is compatible with T and satisfies $d_p(R) = d_p(S)$ for all $p \in \text{Proc}$. Write $S = (s_1, \dots, s_n)$ and $T = (t_1, t_2, \dots)$.

We wish to show $T \downarrow_p \preceq S \downarrow_p$ for all p . So suppose this is not the case, and let k be the greatest integer such that $T^k \downarrow_p \preceq S \downarrow_p$ for all p . Now let $t = t_{k+1}$ and let p be the element of Proc for which $t \in \text{Trans}_p$, and we have $T^{k+1} \downarrow_p \not\preceq S \downarrow_p$. We will arrive at a contradiction by showing there exists a universally permitted extension R of S with $d_p(R) < d_p(S)$.

For each $r \in \text{Proc}$ there is a path

$$\sigma_r = (s_1^r, \dots, s_{n(r)}^r)$$

through M_r which is src-equivalent to $S \downarrow_r$ such that for all $r \neq p$,

$$T^k \downarrow_r = T^{k+1} \downarrow_r = (s_1^r, \dots, s_{m(r)}^r)$$

for some $m(r) \leq n(r)$, and such that

$$T^{k+1} \downarrow_p = (s_1^p, \dots, s_{m(p)}^p, t).$$

We will consider first the case that $m(p) = n(p)$.

Suppose t is a send, say $\text{label}(t) = c!x$. Then $\text{label}(t_{k+2}) = c?x$, as T is synchronous. Moreover, $\text{Queue}_c(T^k)$ is empty. Let $q = \text{receiver}(c)$. If $p = q$ then $\text{src}(t)$ is a send-receive state with the same sending and receiving channel, but let us assume for now that $p \neq q$. Let $u = \text{src}(t_{k+2})$, so that $u = \text{terminus}(T^k)_q$. Say that t is the i^{th} send on c in T^{k+1} . Then there are $i - 1$ sends on c in S , and therefore no more than $i - 1$ receives on c in S . This implies $m(q) = n(q)$: if not, there would be at least i receives on c in S . Hence $\text{Queue}_c(S^n)$ is empty. Now, whether or not $p = q$, let

$$R = (s_1, \dots, s_n, t, t_{k+2}).$$

Then R is a universally permitted extension of S with $d_p(R) < d_p(S)$.

If t is local, we may take $R = (s_1, \dots, s_n, t)$.

Suppose t is a receive, say $\text{label}(t) = c?x$, and say t is the i^{th} receive on c in T . Then t_k must be the matching send, i.e., t_k must be the i^{th} send on c in T and $\text{label}(t_k) = c!x$. Let $q = \text{sender}(c)$. Since $T^k \downarrow_q \preceq S \downarrow_q$, there must be at least i sends on c in S , and $\text{Sent}_c(S)|_i = x$. As there are $i - 1$ receives on c in $S \downarrow_p$, we may conclude that $\text{Queue}_c(S^n)$ begins with x . So

$$R = (s_1, \dots, s_n, t),$$

will suffice.

Now we turn to the case where $m(p) < n(p)$. Since $T^{k+1} \downarrow_p$ and $S \downarrow_p$ are compatible and neither $T^{k+1} \downarrow_p \preceq S \downarrow_p$ nor $S \downarrow_p \preceq T^{k+1} \downarrow_p$, Lemma 5.1 implies $n(p) = m(p) + 1$ and one of $s = s_{m(p)+1}^p$, t is a send, and the other, a receive.

Suppose s is the send and t the receive. Then there is a receive transition \bar{t} with $\text{label}(\bar{t}) = \text{label}(t)$ and $\text{src}(\bar{t}) = \text{des}(s)$. Arguing as in the case in which $n(p) = m(p)$, we see that the extension $R = (s_1, \dots, s_n, \bar{t})$ is universally permitted, and satisfies $d_p(R) < d_p(S)$.

If, on the other hand, s is the receive and t the send, then t_{k+2} must be the receive matching t . Let \bar{t} be the transition departing from $\text{des}(s)$ (so $\text{label}(\bar{t}) = \text{label}(t)$). Arguing just as in the $m(p) = n(p)$ case we see that we may take

$$R = (s_1, \dots, s_n, \bar{t}, t_{k+2}),$$

completing the proof of Proposition 6.1.

Corollary 6.2. *Let \mathcal{M} be a model of an MPI program with no wildcard receives, and let T be a finite execution prefix for \mathcal{M} . Then there exists a finite synchronous execution prefix S for \mathcal{M} , with the property that any synchronous extension of S is compatible with T .*

Proof. Apply Proposition 6.1 to the empty sequence and T , and recall that for a synchronous prefix, an extension is universally permitted if, and only if, it is synchronous. \square

7. DEADLOCK

The main result of this section shows that, under certain hypotheses on the MPI program, to verify that the program is deadlock-free it suffices to consider only synchronous executions.

7.1. Definitions. Let $\mathcal{M} = (\mathcal{C}, M)$ be a model of an MPI program, $\Sigma \subseteq \text{Proc}$, and let S be a finite execution prefix. We say that S is *potentially Σ -deadlocked* if $\text{terminus}(S)_p \notin \text{End}_p$ for some $p \in \Sigma$ and S has no universally permitted proper extension. This is equivalent to saying that all of the following hold:

- (i) For some $p \in \Sigma$, $\text{terminus}(S)_p \notin \text{End}_p$.
- (ii) For all $p \in \text{Proc}$, $\text{terminus}(S)_p$ is not a local state.
- (iii) For each $p \in \text{Proc}$ for which $\text{terminus}(S)_p$ is a receiving or a send-receive state: for all $c \in \text{Chan}$ for which there is a transition departing from $\text{terminus}(S)_p$ labeled by a receive on c : $\text{Queue}_c(S)$ is empty, and, letting $q = \text{sender}(c)$, no transition departing from $\text{terminus}(S)_q$ is labeled by a send on channel c .

Notice that it may be possible to extend a potentially deadlocked execution prefix—but not in a way which is universally permitted. For example, if $\text{terminus}(S)_p$ is a sending state for some p , we can always append the send transition onto S , and the result will be an execution prefix. On the other hand, given a potentially deadlocked prefix, there is always some legal MPI implementation for which that prefix will represent an execution which actually deadlocks. Specifically, that implementation will be the one that allows all sends to occur until the last step, when it suddenly decides it will force sends to synchronize.

We say that S is *absolutely Σ -deadlocked* if $\text{terminus}(S)_p \notin \text{End}_p$ for some $p \in \Sigma$ and there is no proper extension of S to an execution prefix. This is equivalent to requiring that (i)–(iii) above and

- (iv) For all $p \in \text{Proc}$, U_p is not a sending or send-receive state.

all hold. Clearly, in any legal MPI implementation, if such a sequence occurs, it must result in deadlock.

We remark that both of these definitions of deadlock depend only on knowing the src-equivalence class of $S \downarrow_p$ for each $p \in \text{Proc}$. For we have already observed, in Section 5, that the universally permitted extensions of an execution prefix depend only on that information. It is clear that the set of arbitrary extensions also depends only on that information.

We say that \mathcal{M} is *Σ -deadlock-free* if it has no potentially Σ -deadlocked execution prefix. This means that under any MPI implementation, no execution of the program can ever deadlock. Finally, we say that \mathcal{M} is *synchronously Σ -deadlock-free* if it has no potentially Σ -deadlocked synchronous execution prefix.

The role of the set Σ in these definitions arises from the fact that, for some systems, we may not wish to consider certain potentially deadlocked prefixes as problematic. For example, if one process p represents a server, then often p is designed to never terminate, but instead to always be ready to accept requests from clients. In this case we probably would not want to consider an execution in which every process other than p terminates normally to be a deadlock. For such a system, Σ might be taken to be all processes other than the server.

7.2. The Deadlock Theorem. Our main result concerning deadlock is the following:

Theorem 7.1. *Let \mathcal{M} be a model of an MPI program with no wildcard receives. Let Σ be a subset of Proc . Then \mathcal{M} is Σ -deadlock-free if, and only if, \mathcal{M} is synchronously Σ -deadlock-free.*

Proof. If \mathcal{M} is Σ -deadlock-free then, by definition, it has no execution prefix which is potentially Σ -deadlocked. So it suffices to prove the opposite direction.

So suppose \mathcal{M} is synchronously Σ -deadlock-free, and that T is a finite execution prefix with $T_p \notin \text{End}_p$ for some $p \in \Sigma$. We must show there exists a universally permitted proper extension T' of T .

By Corollary 6.2, there is a synchronous finite execution prefix S with the property that any synchronous extension of S is compatible with T .

By hypothesis, either $S \downarrow_p \in \text{End}_p$ for all $p \in \Sigma$, or there exists a synchronous proper extension S' of S . If the former is the case then we must have $|S \downarrow_p| > |T \downarrow_p|$ for some $p \in \Sigma$, by compatibility. If the latter is the case, then replace S with S' and repeat this process, until $|S \downarrow_r| > |T \downarrow_r|$ for some $r \in \text{Proc}$; this must eventually be the case as the length of S is increased by at least 1 in each step.

Hence there is a finite synchronous execution prefix S , compatible with T , and an $r \in \text{Proc}$ for which $|S \downarrow_r| > |T \downarrow_r|$. Now apply Proposition 6.1 to conclude there exists a finite, universally permitted extension T' of T with the property that $S \downarrow_p \preceq T' \downarrow_p$ for all $p \in \text{Proc}$. We have

$$|T \downarrow_r| < |S \downarrow_r| \leq |T' \downarrow_r|,$$

so T' must be a proper extension of T . □

7.3. Counterexample with wildcard receive. Theorem 7.1 fails if we allow \mathcal{M} to have wildcard receives. Consider the example with three processes illustrated in Figure 5. It is not hard to see that the synchronous execution prefixes for this model are all prefixes of the sequence $\{4, 1, 5, 6, 7, 3\}$, and none of these is potentially deadlocked. However, the non-synchronous execution prefix $\{4, 5, 6, 7, 2\}$ is potentially deadlocked.

8. PARTIAL DEADLOCK

8.1. The Partial Deadlock Theorem. Let $\mathcal{M} = (\mathcal{C}, M)$ be a model of an MPI program, and let S be a finite execution prefix. Let Σ be a subset of Proc .

We say S is *potentially partially Σ -deadlocked* (or Σ -ppd, for short) if for some $p \in \Sigma$, $\text{terminus}(S)_p \notin \text{End}_p$ and there is no universally permitted proper extension S' of S with $|S' \downarrow_p| > |S \downarrow_p|$. The idea is that there is some MPI implementation for which process p will never be able to move after its last step in S . As before, that is the implementation which allows all sends to occur in S , and thereafter decides to force all sends to be synchronous.

We say that \mathcal{M} is *free of partial Σ -deadlock* if it has no execution prefix which is Σ -ppd. Clearly, if \mathcal{M} is free of partial Σ -deadlock then it is Σ -deadlock-free.

Finally, we say that \mathcal{M} is *synchronously free of partial Σ -deadlock* if it has no synchronous execution prefix which is Σ -ppd.

Theorem 8.1. *Let \mathcal{M} be a model of an MPI program with no wildcard receives. Let Σ be a subset of Proc . Then \mathcal{M} is free of partial Σ -deadlock if, and only if, \mathcal{M} is synchronously free of partial Σ -deadlock.*

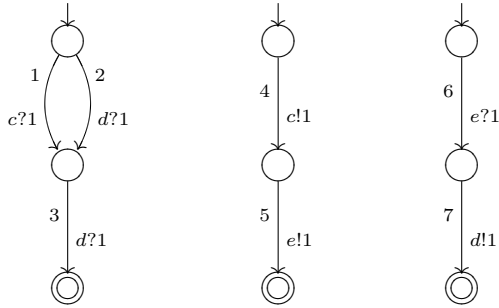


FIGURE 5. Counterexample to Deadlock Theorem with wildcard receive

Proof. If \mathcal{M} is free of partial Σ -deadlock then, by definition, it has no execution prefix which is Σ -ppd. So it suffices to prove the opposite direction.

So suppose T is a finite execution prefix, $p \in \Sigma$, and $T_p \notin \text{End}_p$. We must show there exists a universally permitted proper extension T' of T with $|T \downarrow_p| < |T' \downarrow_p|$.

By Corollary 6.2, there is a synchronous finite execution prefix S with the property that any synchronous extension of S is compatible with T .

By hypothesis, either $S \downarrow_p \in \text{End}_p$, or there exists a synchronous proper extension S' of S in which $|S \downarrow_p| < |S' \downarrow_p|$. If the former is the case then we must have $|S \downarrow_p| > |T \downarrow_p|$, by compatibility. If the latter is the case, then replace S with S' and repeat this process, until $|S \downarrow_p| > |T \downarrow_p|$; this must eventually be the case as the length of $S \downarrow_p$ is increased by at least 1 in each step.

Hence there is a finite synchronous execution prefix S , compatible with T , for which $|S \downarrow_p| > |T \downarrow_p|$. Now apply Proposition 6.1 to conclude there exists a finite, universally permitted extension T' of T with the property that $S \downarrow_r \preceq T' \downarrow_r$ for all $r \in \text{Proc}$. We have

$$|T \downarrow_p| < |S \downarrow_p| \leq |T' \downarrow_p|,$$

which completes the proof. \square

9. APPLICATION TO CHANNEL DEPTHS

As we have seen, one of the important questions facing the analyst of an MPI program is the upper bound to be placed on the depth of the communication channels. If a property has been verified under the assumption that the size of the message buffers never exceeds, say, 4, how do we know that a violation will not be found with 5?

The results on deadlock show that, in certain circumstances, we are justified in assuming all communication is synchronous. For a model checker such as SPIN, that means we may use channels of depth 0, which may greatly reduce the size of the state space that SPIN will explore. In this section we attempt to gain some control on the channel depths for other kinds of properties. The following result

could be applicable to a property which is an assertion on what types of states are reachable.

Theorem 9.1. *Let \mathcal{M} be a model of an MPI program with no wildcard receives, and $\Sigma \subseteq \text{Proc}$. Suppose \mathcal{M} is free of partial Σ -deadlock. Let T be a finite execution prefix of \mathcal{M} such that, for all $p \in \text{Proc}$, $\text{terminus}(T)_p$ is not an immediate successor to a send-receive state. Then there exists an execution prefix S of \mathcal{M} satisfying all of the following:*

- (1) $S \downarrow_p \sim T \downarrow_p$ for all $p \in \Sigma$.
- (2) $S \downarrow_p \preceq T \downarrow_p$ for all $p \in \text{Proc} \setminus \Sigma$.
- (3) For all $c \in \text{Chan}$ for which $\text{receiver}(c) \in \Sigma$, if $|\text{Queue}_c(T)| = 0$ then S is c -synchronous, while if $|\text{Queue}_c(T)| > 0$ then

$$|\text{Queue}_c(S^i)| \leq |\text{Queue}_c(T)|$$

for all i in the domain of S .

Proof. We claim there exists a finite synchronous execution prefix \tilde{S} which is compatible with T and satisfies $T \downarrow_p \preceq \tilde{S} \downarrow_p$ for all $p \in \Sigma$. To see this, first note that by Corollary 6.2, there is a finite synchronous execution prefix R with the property that any synchronous extension of R is compatible with T . Suppose $p \in \Sigma$ and $T \downarrow_p \not\preceq R \downarrow_p$. By assumption, there is a synchronous extension R' of R for which $|R' \downarrow_p| > |R \downarrow_p|$. We may replace R with R' and repeat this process, until $T \downarrow_p \preceq R \downarrow_p$. Now we continue extending R in this way for each $p \in \Sigma$ and we let \tilde{S} be the resulting sequence, which clearly satisfies the claim.

Since we have assumed that $\text{terminus}(T)_p$ is not an immediate successor to a send-receive state, compatibility implies that for all $p \in \text{Proc}$, $T \downarrow_p \preceq \tilde{S} \downarrow_p$ or $\tilde{S} \downarrow_p \preceq T \downarrow_p$.

We construct the sequence S as follows. First let S be a copy of \tilde{S} , and fix a $p \in \text{Proc}$. Let

$$m(p) = \min\{|\tilde{S} \downarrow_p|, |T \downarrow_p|\}.$$

Delete from S all transitions in Trans_p which occur after the $m(p)^{\text{th}}$ transition in Trans_p . Repeat this process for each $p \in \text{Proc}$; we will show that the resulting sequence S has the required properties.

First we must show that S is indeed an execution prefix. It is clear that $S \downarrow_p$ is a path through M_p for each p , and that, if $p \in \Sigma$, $S \downarrow_p \sim T \downarrow_p$. Now fix a $c \in \text{Chan}$ and we must show that S is c -legal. To do this we argue as follows: let

$$\begin{aligned} r &= |\text{Received}_c(T)| \\ s &= |\text{Sent}_c(T)| \\ m &= |\text{Received}_c(\tilde{S})| = |\text{Sent}_c(\tilde{S})|. \end{aligned}$$

Now, if we project the sequence of labels of elements of \tilde{S} onto the set of events involving c , the result is a sequence of the form

$$\tilde{C} = (c!x_1, c?x_1, c!x_2, c?x_2, \dots, c!x_m, c?x_m),$$

as \tilde{S} is synchronous. Now let

$$\begin{aligned} r' &= \min\{r, m\} \\ s' &= \min\{s, m\}. \end{aligned}$$

If we project the sequence of labels of elements of S onto the set of events involving c , the result is the sequence C obtained from \tilde{C} by deleting all the receive events after the r' -th such event, and deleting all the send events after the s' -th such event. But since $r \leq s$, we have $r' \leq s'$. This means that

$$C = (c!x_1, c?x_1, \dots, c!x_{r'}, c?x_{r'}, c!x_{r'+1}, \dots, c!x_{s'}),$$

i.e., C begins with r' send-receive pairs, followed by a sequence of $s' - r'$ sends, which is clearly c -legal. Moreover, if $s' = r'$ then S is c -synchronous, while if not then

$$|\text{Queue}_c(S^i)| \leq s' - r'$$

for all i in the domain of S .

Now if $\text{receiver}(c) \in \Sigma$, then $r' = r$, whence

$$s' - r' \leq s - r = |\text{Queue}_c(T)|.$$

So if $|\text{Queue}_c(T)| = 0$ then $s' = r'$ and, as we have seen, this implies that S is c -synchronous. If $|\text{Queue}_c(T)| > 0$, then for all i in the domain of S we have

$$|\text{Queue}_c(S^i)| \leq s' - r' \leq |\text{Queue}_c(T)|,$$

as claimed. \square

We outline here one way Theorem 9.1 could prove useful. Suppose that R is a set of global states, and one wishes to verify the property \mathcal{P} which says that no state in R is reachable. One could prove \mathcal{P} in a series of steps. In the first step, one should show that the model is free of partial deadlock (perhaps using Theorem 8.1 to reduce to the synchronous case). Next, one must somehow show that, if there exists a violation to \mathcal{P} , then there exists a violating execution prefix T in which, at the end of execution of T , all the queues have size no greater than some fixed integer d . Finally, one may verify \mathcal{P} while bounding the channel depths by d . Theorem 9.1 justifies this last step, for, if there were a violation to \mathcal{P} , then by the Theorem there would have to be one for which the channel depths never exceed d at any stage.

10. BARRIERS

Let $\mathcal{M} = (\mathcal{C}, M)$ be a model of an MPI program. Suppose B is a set of states in \mathcal{M} , and we wish to insert “barriers” before each of these states. We assume that B does not contain any start_p . Also, we assume that B contains no immediate successors of send-receive states, nor the immediate successors of those states. (These are the u' , v , and v' states of part (v) in the definition of MPI State Machine, Section 3.2.) We exclude such states because we do not allow a barrier statement before the process begins, nor “inside” an `MPI_SENDRECV` statement. We call such a set B a *barrier-acceptable* state set for \mathcal{M} .

Given \mathcal{M} and B as above, we now describe how to create a new model $\mathcal{M}^B = (\mathcal{C}^B, M^B)$ which represents the model \mathcal{M} with barriers added just before the states in B . Write

$$\mathcal{C}^B = (\text{Proc}^B, \text{Chan}^B, \text{sender}^B, \text{receiver}^B, \text{msg}^B, \text{loc}^B, \text{comm}^B).$$

We define $\text{Proc}^B = \text{Proc} \cup \{\beta\}$, where β is some object not in Proc . We call β the *barrier process*. For all $p \in \text{Proc}$, we add two new channels ϵ_p and ξ_p to Chan , to form Chan^B . Channel ϵ_p sends messages from p to β saying that p is ready to enter

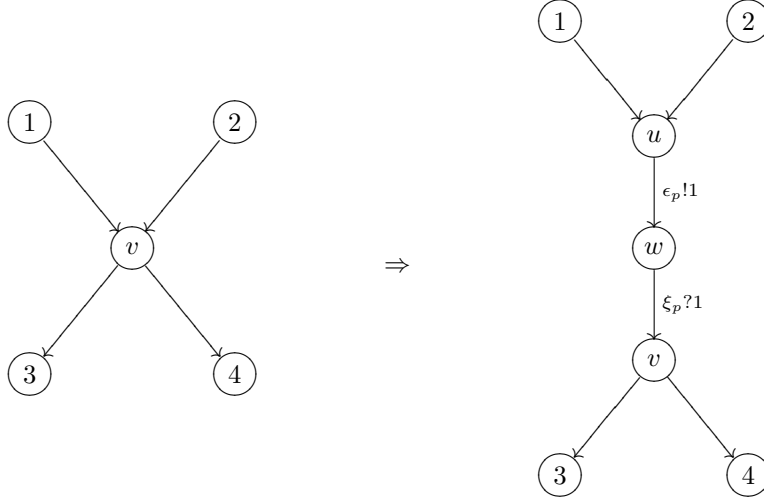


FIGURE 6. Insertion of barrier before state v .

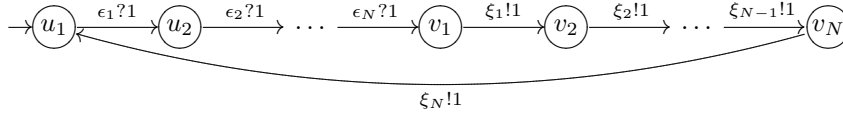


FIGURE 7. The barrier process M_β .

the barrier. Channel ξ_p sends messages from β to p telling p it may exit the barrier. We let $\text{msg}^B(\epsilon_p) = \text{msg}^B(\xi_p) = \{1\}$.

Now we modify each state machine M_p to produce the state machine M_p^B as follows. We will add some states and transitions, so $\text{States}_p \subseteq \text{States}_p^B$ and $\text{Trans}_p \subseteq \text{Trans}_p^B$. (See Figure 6.) Specifically, for each state $v \in B$, we add two new states u, w , and two new transitions t_1, t_2 . The functions src^B and des^B are defined so that t_1 leads from u to w , and t_2 leads from w to v , and so that any transition in M_p which terminates in v is redirected to terminate in u in M_p^B . Set $\text{label}^B(t_1) = \epsilon_p!1$ and $\text{label}^B(t_2) = \xi_p?1$.

The state w of M_p^B will be referred to as a *barrier state*.

Now we describe the state machine M_β^B for the barrier process. We let $M_\beta = M_\beta^B$ to simplify the notation. (See Figure 7.) To do this, choose an order on Proc , say $\text{Proc} = \{p_1, \dots, p_N\}$, and let $\epsilon_i = \epsilon_{p_i}$ and $\xi_i = \xi_{p_i}$, where $p = p_i$. Now M_β has states u_i and v_i , for $1 \leq i \leq N$. The start state is u_1 . For each i , there is a transition s_i , departing from u_i , with $\text{label}(s_i) = \epsilon_i?1$. For $i < N$ this terminates in u_{i+1} , while s_N terminates in v_1 . For each i there is also a transition t_i departing from v_i , labeled $\xi_i!1$. For $i < N$ this terminates in v_{i+1} , while t_N terminates in u_1 . Finally, let $\text{End}_\beta = \emptyset$; the barrier process never terminates.

We say that a global state U of \mathcal{M}^B is *inside a barrier* if U_β is the state v_1 of M_β .

Proposition 10.1. *Let S be a finite execution prefix for \mathcal{M}^B terminating inside a barrier. Then for all $p \in \text{Proc}$, $\text{terminus}(S)_p$ is a barrier state. Moreover, there exists $k \geq 1$ such that for all $p \in \text{Proc}$, if $c = \epsilon_p$ then $\text{Received}_c(S)$ and $\text{Sent}_c(S)$ both have length k , while if $c = \xi_p$ the two sequences have length $k-1$. In particular, $\text{Queue}_c(S)$ is empty if $c = \epsilon_p$ or $c = \xi_p$ for some $p \in \text{Proc}$.*

Proof. For $p \in \text{Proc}$, let $l(p)$ be the number of messages received on ϵ_p in S , and $\bar{l}(p)$ the number of messages sent on ϵ_p in S . Let $m(p)$ and $\bar{m}(p)$ be the analogous numbers for ξ_p .

Let $k = l(p_1)$. By examining M_β (see Figure 7), it is clear that $l(p) = k$ and $\bar{m}(p) = k-1$ for all $p \in \text{Proc}$. Considering the construction of M_p^B , we see that, for all p , $\bar{l}(p) \leq m(p) + 1$, and equality holds if, and only if, $\text{terminus}(S)_p$ is a barrier state. Hence

$$(1) \quad k = l(p) \leq \bar{l}(p) \leq m(p) + 1 \leq \bar{m}(p) + 1 = k - 1 + 1 = k.$$

So we must have equality throughout in equation (1), which completes the proof. \square

Proposition 10.2. *Let $S = (s_1, \dots, s_n)$ be a finite execution prefix for \mathcal{M}^B . Suppose for some $p \in \text{Proc}$, and some $n' < n$, $s_{n'}$ is labeled $\epsilon_p!1$, s_n is labeled $\xi_p?1$, and for all k , $n' < k < n$, $s_k \notin \text{Trans}_p^B$. Then for some k , $n' \leq k < n$, S^k terminates inside a barrier.*

Proof. Let l be the number of messages received on ϵ_p in S , \bar{l} the number sent on ϵ_p , and m and \bar{m} the analogous numbers for ξ_p . Define $l', \bar{l}', m', \bar{m}'$ to be the analogous numbers for the sequence $S^{n'-1}$.

We know that $l \leq \bar{l}$ and $m \leq \bar{m}$, since the number of messages received on a channel can never exceed the number sent. We know that $\bar{l} = m$ because it is clear from the construction of M_p^B that the number of $\epsilon_p!1$ events must equal the number of $\xi_p?1$ events in any path which does not terminate in a barrier state. We also know that $\bar{m} \leq l$ since in the cyclic path through M_β the transition labeled $\epsilon_p?1$ precedes the one labeled $\xi_p!1$. Putting these together yields

$$l \leq \bar{l} = m \leq \bar{m} \leq l,$$

which implies

$$(2) \quad l = \bar{l} = m = \bar{m}.$$

Similar reasoning yields

$$(3) \quad l' = \bar{l}' = m' = \bar{m}'.$$

Now, in the sequence $S' = (s_{n'}, \dots, s_n)$ there occur only two events in M_p^B ; one of these is $\epsilon_p!1$ and the other $\xi_p?1$. Hence

$$\begin{aligned} \bar{l} &= \bar{l}' + 1 \\ m &= m' + 1. \end{aligned}$$

Putting these together with equations (2) and (3), we have

$$\begin{aligned} l &= \bar{l} = \bar{l}' + 1 = l' + 1 \\ \bar{m} &= m = m' + 1 = \bar{m}' + 1. \end{aligned}$$

This implies that in S' there is exactly one occurrence of $\epsilon_p?1$ and one occurrence of $\xi_p!1$. Moreover, since $l' = \bar{m}'$, $\text{terminus}(S^{n'-1})_p$ lies on the path joining v'_p to u_p , where v'_p is the immediate successor to v_p in M_β (see Figure 7). This means that in S' , the $\epsilon_p?1$ must occur before the $\xi_p!1$. But any path in M_β for which this is the case must pass through v_1 . \square

11. REMOVAL OF BARRIERS

A common question that arises with MPI programs is “when is it safe to remove a barrier?” In this section, we show that, under appropriate hypotheses, at least the removal of barriers can not introduce deadlocks into the program.

Theorem 11.1. *Let \mathcal{M} be a model of an MPI program with no wildcard receives, and let B be a barrier-acceptable state set for \mathcal{M} , and Σ a subset of Proc . Suppose \mathcal{M}^B is Σ -deadlock-free (resp., free of partial Σ -deadlock). Then \mathcal{M} is Σ -deadlock-free (resp., free of partial Σ -deadlock).*

To prove the Theorem, we first define a new model \mathcal{N} . This is obtained by modifying \mathcal{M}^B in a fairly simple way which we will now describe. We will only change the values of the sender and receiver functions for certain channels, and we will change the label function for certain transitions, but everything else, including the state set, the transition set, and the src and des functions, will be exactly the same in the two models.

First, we change the direction of each channel ξ_p , so that in the new model, this channel carries messages from p to β , just like ϵ_p . Second, we change the labeling function so that any transition that was labeled by $\xi_p?1$ in \mathcal{M}^B is labeled by $\xi_p!1$ in \mathcal{N} , and similarly, any transition that was labeled by $\xi_p!1$ in \mathcal{M}^B is labeled by $\xi_p?1$ in \mathcal{N} .

Hence in \mathcal{N} , the original processes send messages to the barrier process to both enter and exit the barrier. Now, for arbitrary executions, this arrangement will not necessarily function as a barrier, since the messages may all be buffered. However, for synchronous executions, it will function as a barrier in exactly the same way that this worked in \mathcal{M}^B . In our proof, \mathcal{N} will act as a bridge between \mathcal{M} and \mathcal{M}^B .

There is a 1-1 correspondence ψ from the set of synchronous execution prefixes of \mathcal{N} to the set of those of \mathcal{M}^B , defined as follows: suppose $S = (t_1, t_2, \dots)$ is a synchronous execution prefix of \mathcal{N} . Suppose for some i , t_i and t_{i+1} are labeled by $\xi_p!1$ and $\xi_p?1$, respectively. By replacing each such subsequence (t_i, t_{i+1}) with (t_{i+1}, t_i) , we obtain a synchronous execution prefix $\psi(S)$ for \mathcal{M}^B . The definition of the inverse of ψ is clear. It is also clear that

$$S \subseteq T \iff \psi(S) \subseteq \psi(T)$$

and that $|S \downarrow_p| = |\psi(S) \downarrow_p|$ for all $p \in \text{Proc}^B$. It follows from these observations that S is potentially Σ -deadlocked (resp., Σ -ppd) if, and only if, $\psi(S)$ is. We may conclude that \mathcal{N} is Σ -deadlock-free (resp., free of partial Σ -deadlock) if, and only if, \mathcal{M}^B is, as Theorems 7.1 and 8.1 reduce these questions to the synchronous case.

We now turn to the relationship between \mathcal{M} and \mathcal{N} . We define a map θ from the set of all finite execution prefixes of \mathcal{M} to the set of those of \mathcal{N} , as follows: suppose $S = (t_1, \dots, t_n)$ is a finite execution prefix for \mathcal{M} . For each i for which $\text{des}(t_i) \in B$, insert, immediately after t_i , the appropriate transitions labeled $\epsilon_p!1$ and $\xi_p!1$. This guarantees that we get paths through each \mathcal{N}_p , and the resulting

sequence is still c -legal for each channel c since we only inserted send transitions. Now, at the end of this prefix, there may be messages in the queues for the ϵ_p and ξ_p , so we append the longest possible sequence of transitions from Trans_β so that the resulting sequence is still an execution prefix (i.e., we just let the barrier process run until it reaches a state in which the channel for the unique outgoing transition has an empty message queue). This results in an execution prefix $\theta(S)$ for \mathcal{N} .

Observe that, by construction, $\text{terminus}(\theta(S))_p = \text{terminus}(S)_p$ for all $p \in \text{Proc}$. In particular, $\text{terminus}(\theta(S))_p$ does not have an outgoing barrier transition (i.e., a transition labeled by a communication event involving an ϵ_p or a ξ_p).

We next define a map ϕ in the opposite direction—from the set of all finite execution prefixes of \mathcal{N} to the set of those of \mathcal{M} —by simply deleting all the barrier transitions. It is clear that $\phi \circ \theta$ is the identity, and that, if $S \subseteq T$ are prefixes for \mathcal{N} , then $\phi(S) \subseteq \phi(T)$. Furthermore, if T is a universally permitted extension of S then $\phi(T)$ is a universally permitted extension of $\phi(S)$.

Now suppose \mathcal{N} is Σ -deadlock-free, and we wish to show the same of \mathcal{M} . Suppose S is a finite execution prefix of \mathcal{M} such that $\text{terminus}(S)_p \notin \text{End}_p$ for some $p \in \Sigma$. Then $\text{terminus}(\theta(S))_p \notin \text{End}_p$ as well. Hence there exists a universally permitted proper extension T of $\theta(S)$. Moreover, the first transition t in $T \setminus \theta(S)$ must lie in Trans_q for some $q \in \text{Proc}$, since, by the construction of $\theta(S)$, the barrier process is blocked at the end of $\theta(S)$. As we have seen, t cannot be a barrier transition. It follows that

$$\phi(T) \supset \phi(\theta(S)) = S,$$

so $\phi(T)$ is a universally permitted proper extension of S . Hence \mathcal{M} is Σ -deadlock-free.

Suppose instead that \mathcal{N} is free of partial Σ -deadlock, and we wish to show the same of \mathcal{M} . In this case we may choose T as above but with the additional requirement that $|T \downarrow_p| > |\theta(S) \downarrow_p|$. Again, it must be the case that the first transition t of $T \downarrow_p \setminus \theta(S) \downarrow_p$ is not a barrier transition. Hence

$$|\phi(T) \downarrow_p| > |\phi(\theta(S)) \downarrow_p| = |S \downarrow_p|,$$

and this shows that \mathcal{M} is free of partial Σ -deadlock, completing the proof of Theorem 11.1.

12. EMPTINESS OF CHANNELS INSIDE BARRIERS

We have already seen, in Proposition 10.1, that inside a barrier, the barrier channels ξ_p and ϵ_p must be empty. Now we will show that, under suitable restrictions on the model, all channels must be empty inside a barrier.

Theorem 12.1. *Let \mathcal{M} be a model of an MPI program with no wildcard receives, and B a barrier-acceptable state set for \mathcal{M} . Let Σ be a nonempty subset of Proc . Suppose \mathcal{M}^B is Σ -deadlock-free. Let S be a finite execution prefix for \mathcal{M}^B which terminates inside a barrier. Let T be an associated synchronous prefix for S . Then $S \downarrow_p \sim T \downarrow_p$ for all $p \in \text{Proc}^B$. In particular, $\text{Queue}_c(S)$ is empty for all $c \in \text{Chan}^B$.*

Proof. By Proposition 10.1, there is an integer k such that for each $p \in \text{Proc}$, there are exactly k occurrences of $\epsilon_p?1$ and k occurrences of $\epsilon_p!1$ in S , and $k - 1$ occurrences of $\xi_p?1$ and $\xi_p!1$.

For each $r \in \text{Proc}^B$, there is a path

$$\pi_r = (s_1^r, \dots, s_{n(r)}^r) \sim S \downarrow_r$$

through M_r^B and an integer $m(r)$ in the domain of π_r such that

$$T \downarrow_r = (s_1^r, \dots, s_{m(r)}^r).$$

Clearly we cannot have $\text{terminus}(T)_r \in \text{End}_r$ for any r , since no $\text{terminus}(S)_r$ is an end state. Nor can we have $\text{terminus}(T)_r$ is a local state, since then we would have $n(r) > m(r) + 1$ and $s_{m(r)+1}^r$ is a local transition, and we could append this transition to T .

Suppose $m(\beta) = n(\beta)$. Then Propositions 10.1 and 10.2 imply $m(r) = n(r)$ for all $r \in \text{Proc}$, so $T \downarrow_r \sim S \downarrow_r$ for all r , as required.

So let us assume that $m(\beta) < n(\beta)$. We will arrive at a contradiction.

If T were potentially deadlocked then we would have $\text{terminus}(T)_r \in \text{End}_r$ for some r , since Σ is non-empty. As this is not the case, T cannot be potentially deadlocked. So there must exist $p, q \in \text{Proc}^B$, and $c \in \text{Chan}^B$, such that $\text{terminus}(T)_p$ has an outgoing transition t labeled $c!x$ and $\text{terminus}(T)_q$ has an outgoing transition t' labeled $c?x$. Clearly $m(p) < n(p)$, since π_p terminates at a receiving state. We claim that $m(q) < n(q)$ as well. For if not, then $m(q) = n(q)$, $c = \xi_q$ and $p = \beta$. So β is at the state with an outgoing transition labeled $\xi_p!1$. Moreover, since $m(q) = n(q)$, there are $k - 1$ occurrences of $\xi_q?1$ in $T \downarrow_q$ and therefore $k - 1$ occurrences of $\xi_p!1$ in $T \downarrow_\beta$. This implies $m(\beta) \geq n(\beta)$, a contradiction.

Hence either $s_{m(p)+1}^p$ is labeled $c!x$, or $\text{terminus}(T)_p$ is a send-receive state and $s_{m(p)+2}^p$ is labeled $c!x$. Similarly, either $s_{m(q)+1}^q$ is labeled $c?y$, or $\text{terminus}(T)_q$ is a send-receive state and $s_{m(q)+2}^q$ is labeled $c?y$ for some $y \in \text{msg}(c)$. (We know that the receiving channel must be c , since there are no wildcard receives.) Since $\text{Received}_c(S)$ is a prefix of $\text{Sent}_c(S)$, and T is synchronous, we must have $x = y$. Hence if we append t and then t' to T , the result is a synchronous execution prefix T' which is longer than T and satisfies $T' \preceq S$, a contradiction. \square

13. LOCALLY DETERMINISTIC MODELS

We say that a model \mathcal{M} of an MPI program is *locally deterministic* if it has no wildcard receives, and, for every local state u , there is precisely one transition t with $\text{src}(t) = u$.

Theorem 13.1. *Suppose \mathcal{M} is a locally deterministic model of an MPI program. Then there exists an execution prefix S for \mathcal{M} with the following property: if T is any execution prefix of \mathcal{M} , then for all $p \in \text{Proc}$, $T \downarrow_p \preceq S \downarrow_p$. In particular, any two execution prefixes for \mathcal{M} are compatible.*

We construct the execution prefix S described in the statement of Theorem 13.1 using the following inductive procedure. Pick a total order on Proc , say $\text{Proc} = \{p_1, \dots, p_N\}$. To begin, set $S = ()$, the empty sequence, and let $p = p_1$.

We define a sequence S' as follows. If there is no $s \in \text{Trans}_p$ such that the sequence obtained by appending s to S is an execution prefix, let $S' = S$. Otherwise, we pick one such s as follows. Let U be the terminal state of S . If U_p is a local state, by hypothesis there is only one outgoing transition, so s is uniquely defined. The same is true if U_p is a sending state. If U_p is a receiving state, then there is precisely one channel c such that there is one outgoing transition, labeled $c?x$, for each $x \in \text{msg}(c)$. Only one of these can possibly be appended to S to yield a new execution prefix—this is the one corresponding to the element x that is the first element in the message channel queue for c after the last step of S . Finally, if U_p

is a send-receive state, let s be the transition departing from U_p labeled by a send. Now let S' be the sequence obtained by appending s to S .

Now let $S = S'$, $p = p_2$, and repeat the paragraph above. Continue in this way, and after p_N cycle back to p_1 . Continue cycling around the processes in this way. If one ever passes through N processes in a row without adding a transition to S , then there can be no further extension of S (either because $U_p \in \mathbf{End}_p$ for all p , or because S is absolutely deadlocked) and the process stops with S finite. Otherwise, this yields a well-defined infinite execution prefix S .

Before we turn to the proof of Theorem 13.1, we make two observations concerning the execution prefix $S = (s_1, s_2, \dots)$ defined above.

First, it cannot be the case that for some $p \in \mathbf{Proc}$, $S \downarrow_p$ is finite and terminates at a local, sending, or send-receive state. For, in any of these cases, the first time p is reached in the cyclic schedule after reaching this state, the local or send transition would be available to append to S .

Second, suppose for some $p \in \mathbf{Proc}$, $S \downarrow_p$ is finite and terminates at a receiving state u . Let n be the least integer such that $s_i \notin \mathbf{Trans}_p$ for all $i > n$. Let c be the receiving channel for the receive transitions departing from u . Then $\mathbf{Queue}_c(S^i)$ is empty for all $i \geq n$. For, if at some point the queue became non-empty, then the next time after that point when p is reached in the cyclic schedule, one of the receive transitions would be available to append to S .

We now turn to the proof of Theorem 13.1. Suppose there is an execution prefix $T = (t_1, t_2, \dots)$ such that for some $p \in \mathbf{Proc}$, $T \downarrow_p \not\leq S \downarrow_p$. It is not hard to see that there must exist $n > 0$ such that

$$(4) \quad T^n \downarrow_p \not\leq S \downarrow_p$$

for some p . Choose n so that it is the least integer with this property, and replace T with T^n . Clearly (4) holds if p is the element of \mathbf{Proc} for which $t_n \in \mathbf{Trans}_p$.

To summarize, we now have the following situation: there exist a finite execution prefix $T = (t_1, \dots, t_n)$, a $p \in \mathbf{Proc}$, and for each $r \in \mathbf{Proc}$, a path

$$\pi_r = (s_1^r, s_2^r, \dots)$$

through M_r and a non-negative integer $m(r)$ in the domain of π_r such that:

$$(5) \quad \pi_r \sim S \downarrow_r \text{ for all } r \in \mathbf{Proc}$$

$$(6) \quad T^{n-1} \downarrow_r = T \downarrow_r = (s_1^r, \dots, s_{m(r)}^r) \text{ for all } r \in \mathbf{Proc} \setminus \{p\}$$

$$(7) \quad T^{n-1} \downarrow_p = (s_1^p, \dots, s_{m(p)}^p)$$

$$(8) \quad T \downarrow_p \not\leq \pi_p.$$

We will obtain a contradiction.

Let $n(r)$ denote the length of π_r , allowing the possibility that $n(r) = \infty$. Let $u = \mathbf{src}(t_n)$.

Suppose u is a local or a sending state. Then there is a unique transition departing from u . As we have seen, it is not possible that $n(p)$ is finite and $\mathbf{terminus}(\pi_p) = \mathbf{terminus}(S \downarrow_p) = u$. So we must have $n(p) > m(p)$ and $s_{m(p)+1}^p = t_n$. But this means $T \downarrow_p$ is a prefix of π_p , contradicting (8).

Suppose u is a send-receive state and that t_n is a send, say $\mathbf{label}(t_n) = c!x$. Since it is not possible that π_p terminates at a send-receive state, we must have $n(p) > m(p)$, and, considering (8), $s_{m(p)+1}^p$ is a receive. However, this means that $\mathbf{des}(s_{m(p)+1}^p)$ is a sending state, so $n(p) > m(p) + 1$ and $s_{m(p)+2}^p$ is a send

transition labeled $c!x$. Now let π'_p be the sequence obtained from π by replacing $s_{m(p)+1}^p, s_{m(p)+2}^p$ with the dual path. Then $\pi'_p \sim \pi_p$ and $T \downarrow_p$ is a prefix of π'_p , contradicting (8).

Suppose now that t_n is a receive, say $\text{label}(t_n) = c?x$. We claim that for some $i \in \{1, 2\}$, $n(p) \geq m(p) + i$ and $s_{m(p)+i}^p$ is a receive on c . For if not, then $S \downarrow_p$ is finite and

$$\text{Received}_c(\pi_p) \subset \text{Received}_c(T),$$

whence

$$\begin{aligned} \text{Received}_c(S) &= \text{Received}_c(S \downarrow_p) = \text{Received}_c(\pi_p) \\ &\subset \text{Received}_c(T) \\ &\subseteq \text{Sent}_c(T) \\ &= \text{Sent}_c(T \downarrow_q) \\ &\subseteq \text{Sent}_c(\pi_q) \\ &= \text{Sent}_c(S \downarrow_q) \\ &= \text{Sent}_c(S). \end{aligned}$$

In other words, $\text{Received}_c(S)$ is a proper prefix of $\text{Sent}_c(S)$. Moreover, $\text{terminus}(S \downarrow_p) = \text{terminus}(\pi_p)$ is a receiving state on channel c . We have seen this is not possible, so our claim must hold.

Now let $q = \text{sender}(c)$ and let k be the length of $\text{Received}_c(T)$. Then

$$\begin{aligned} x = \text{Received}_c(T)|_k &= \text{Sent}_c(T)|_k = \text{Sent}_c(\pi_q)|_k = \text{Sent}_c(S \downarrow_q)|_k \\ &= \text{Sent}_c(S)|_k = \text{Received}(S)|_k. \end{aligned}$$

Hence $\text{label}(s_{m(p)+i}^p) = c?x$.

Now, if $i = 1$, then $T \downarrow_p$ is a prefix of π_p , while if $i = 2$, $T \downarrow_p$ is a prefix of a sequence $\pi'_p \sim \pi_p$. In either case, we have $T \downarrow_p \preceq \pi_p$, a contradiction, completing the proof of Theorem 13.1.

Corollary 13.2. *Suppose \mathcal{M} is a locally deterministic model of an MPI program, and $\Sigma \subseteq \text{Proc}$. Then \mathcal{M} is Σ -deadlock-free if, and only if, there exists a synchronous execution prefix T such that either T is infinite or T is finite and $\text{terminus}(T)_p \in \text{End}_p$ for all $p \in \Sigma$.*

Proof. If \mathcal{M} is Σ -deadlock-free then for any synchronous execution prefix T , either $\text{terminus}(T)_p \in \text{End}_p$ for all $p \in \Sigma$ or T has a proper synchronous extension. So we may construct the required T inductively by beginning with the empty sequence and applying this fact repeatedly.

Now suppose such a prefix T exists and we wish to show that \mathcal{M} is Σ -deadlock-free. By Theorem 7.1, it suffices to show that \mathcal{M} has no synchronous execution prefix S which is potentially Σ -deadlocked. So suppose S is a finite execution prefix with $\text{terminus}(S)_p \notin \text{End}_p$ for some $p \in \Sigma$. We must show that S has a proper synchronous extension.

If T is infinite, then there is an $n > |S|$ such that T^n is synchronous. Thus, for some $r \in \text{Proc}$,

$$|T^n \downarrow_r| > |S \downarrow_r|.$$

By Theorem 13.1, S and T^n are compatible. So by Proposition 6.1, there exists a synchronous extension S' of S with the property that $T^n \downarrow_q \preceq S' \downarrow_q$ for all $q \in \text{Proc}$.

Hence

$$|S \downarrow_r| < |T^n \downarrow_r| \leq |S' \downarrow_r|,$$

which shows that S' is a proper extension of S , as required.

If T is finite and $\text{terminus}(T)_r \in \text{End}_r$ for all $r \in \Sigma$, then we may apply Proposition 6.1 directly to S and T to conclude there is a finite synchronous extension S' of S with $T \downarrow_q \preceq S' \downarrow_q$ for all $q \in \text{Proc}$. This implies, in particular, that $\text{terminus}(S')_p \in \text{End}_p$, which shows that S' must be a proper extension of S . \square

Corollary 13.3. *Suppose \mathcal{M} is a locally deterministic model of an MPI program, and $\Sigma \subseteq \text{Proc}$. Then \mathcal{M} is free of partial Σ -deadlock if, and only if, there exists a synchronous execution prefix T such that for each $p \in \Sigma$, either $T \downarrow_p$ is infinite or $T \downarrow_p$ is finite and $\text{terminus}(T \downarrow_p) \in \text{End}_p$.*

Proof. If \mathcal{M} is free of partial Σ -deadlock then for any synchronous execution prefix T and $p \in \Sigma$, either $\text{terminus}(T)_p \in \text{End}_p$ or T has a synchronous extension T' with $|T' \downarrow_p| > |T \downarrow_p|$. So we may construct the required T inductively by beginning with the empty sequence and then cycling repeatedly through all elements of Σ , applying this fact.

Now suppose such a prefix T exists and we wish to show that \mathcal{M} is free of partial Σ -deadlock. By Theorem 8.1, it suffices to show that \mathcal{M} has no synchronous execution prefix S which is Σ -ppd. So suppose S is a finite execution prefix, $p \in \Sigma$, and $\text{terminus}(S)_p \notin \text{End}_p$. We must show that S has a synchronous extension S' with $|S \downarrow_p| < |S' \downarrow_p|$.

If $T \downarrow_p$ is infinite, then there is an $n > 0$ such that T^n is synchronous and

$$|T^n \downarrow_p| > |S \downarrow_p|.$$

By Theorem 13.1, S and T^n are compatible. So by Proposition 6.1, there exists a synchronous extension S' of S with the property that $T^n \downarrow_q \preceq S' \downarrow_q$ for all $q \in \text{Proc}$. Hence

$$|S \downarrow_p| < |T^n \downarrow_p| \leq |S' \downarrow_p|,$$

as required.

If $T \downarrow_p$ is finite and $\text{terminus}(T \downarrow_p) \in \text{End}_p$, then for some $n > 0$, T^n is synchronous and $\text{terminus}(T^n \downarrow_p) \in \text{End}_p$. By Proposition 6.1, there is a finite synchronous extension S' of S with $T^n \downarrow_q \preceq S' \downarrow_q$ for all $q \in \text{Proc}$. This implies, in particular, that $\text{terminus}(S')_p \in \text{End}_p$, which shows that $|S' \downarrow_p| > |S \downarrow_p|$, as required. \square

13.1. Consequences. Keep in mind that “locally deterministic” is a property of a model of a program, and not of the program itself. It is certainly possible to have two models (even conservative ones) of the same program for which one of those models is locally deterministic and the other is not. This might depend, for example, on what kinds of abstraction each model employs.

Let us describe a case where we can always create a locally deterministic model, and see what conclusions we can draw from this. Suppose we are given an actual MPI program, in a language such as C or Fortran. Suppose this program only makes use of the subset of MPI we are considering here (i.e., `MPI_Send`, `MPI_Receive`, `MPI_Sendrecv`, and `MPI_Barrier`). Assume that the program uses neither `MPI_ANY_SOURCE` nor `MPI_ANY_TAG`. And finally, assume it uses no non-deterministic functions.

This last condition requires clarification. For in one sense, any numerical operation, such as addition of floating point numbers, may be non-deterministic, since

these operations may differ from platform to platform. However, it is almost always the case that, for a given platform, the numerical operations are deterministic functions, in the sense that, given the same inputs repeatedly, they will always return the same output. (The platform may also specify other information not included in the program itself, such as the number of bits used to represent a floating point number.) So we will allow numerical operations in our program; what we want to prohibit are random function or any other function whose behavior is not a function solely of the system state.

Now, given (a) an MPI program satisfying the conditions above, (b) a fixed platform, and (c) the inputs to the program, we may consider the “full precision” model. This is the model in which there is a state for every possible combination of values for all variables (including the program counter, the call stack, etc.) in each process. Our assumptions ensure that this model will be locally deterministic. So suppose, for example, we want to check freedom from deadlock. Let us say we execute the program once on the given platform and with the given inputs and that the program terminates normally. Then we may conclude that, if run again on that platform and with the same inputs, the program will never deadlock, and will always terminate normally. Furthermore, we may conclude that it will always produce the same output. The execution steps may be interleaved differently on different executions, but for each local process, the path followed will be exactly the same, and the messages sent and received will be the same, on every execution.

Of course, it is possible that when given other inputs the program may deadlock. This may happen, for example, if the program branches on a condition which is a function of one of the inputs. However, it is sometimes the case that we can use the methods of this section to prove that a program is deadlock-free on any input. We may be able to do this by abstracting away the values of the inputs when building the (still conservative) model. If there are no branches which rely on the input values then with suitable abstractions the model we build may be locally deterministic. In this case we need only check that one execution of this model is deadlock-free, and then we have shown that any execution of the program, with any inputs, will not deadlock.

The concept of “platform” may be broadly construed. For example, one platform might consist of a particular C compiler with a particular operating system and machine architecture. However, we may also consider theoretical platforms: for example, the platform in which the values of all floating point variables are considered to be actual real numbers, integer variables actual integers, and so on. It may not be unreasonable to require that the program behave correctly on this kind of platform. Another example would be a platform in which all computations are performed symbolically. In this scenario, all input and initial values may be represented by symbols, and then an operation such as addition of two values x and y would return a tree structure of the form $(+ x y)$. Such a representation would allow one to reason about very specific aspects of the calculations performed by the program. In some cases it may be possible to build locally deterministic models using this platform.

Of course, it is still possible that the MPI program might behave differently on two different platforms, due to differences in the numerical operations. Of course, the same could be true of even a sequential program. The results of this Section simply do not address this sort of non-determinism. We have addressed instead

the source of non-determinism that arises from the choices available in buffering messages, and in interleaving the execution steps of the various processes. What we have shown is that, under appropriate hypotheses on the model, this source of non-determinism can have no bearing on the eventual outcome of execution of that program.

REFERENCES

- [1] Holzmann, Gerard J., *The Model Checker* SPIN, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279–295.
- [2] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard (version 1.1). Technical report, 1995. <http://www.mpi-forum.org>.
- [3] Shumsky Matlin, Olga, Ewing L. Lusk, and William McCune, *SPINning Parallel Systems Software*, in Bosnacki, Dragan and Stefan Leue (Eds.), Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11–13, 2002, Proceedings, pp. 213–220. Lecture Notes in Computer Science 2318, Springer, 2002.
- [4] Snir, Marc, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra, MPI—The Complete Reference: Volume 1, The MPI Core, second edition, The MIT Press, Cambridge, 2000.

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF MASSACHUSETTS, AMHERST, MA 01003
E-mail address: `siegel@cs.umass.edu`

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF MASSACHUSETTS, AMHERST, MA 01003
E-mail address: `avrunin@cs.umass.edu`