

Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs*

Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby
School of Computing, University of Utah, Salt Lake City, UT 84112
{rpalmer,ganesh,kirby}@cs.utah.edu

ABSTRACT

Most distributed parallel programs in the high performance computing (HPC) arena are written using the MPI library. There is growing interest in using model checking for debugging these MPI programs. In this context, partial-order reduction has considerable potential for containing state explosion, given that MPI programs mostly interact through communication and synchronization commands. Unfortunately, there has been only limited successes in this area. In this paper, we first define the formal semantics for a non-trivial subset of MPI. We then prove independence theorems based on the formal semantics, giving us a semantically clear and general partial-order reduction approach for MPI. In particular, we can smoothly handle many constructs such as *wait* and *test*, which have not been considered while formally modeling MPI in past work, and hence were not characterized in terms of the dependency between MPI process actions that they induce. Another complication with MPI is that the notion of dependence between actions depends on knowing future actions yet to be explored during model checking. We show that the use of Flanagan and Godefroid's dynamic partial-order reduction algorithm helps determine dependency information with respect to future quite naturally. This paper presents the formal semantics of a subset of MPI and a dynamic partial-order reduction algorithm created using this semantics. Our initial results are encouraging.

Keywords

Partial-order Reduction, Concurrent Program Semantics, Transition Independence, Model Checking, MPI

1. INTRODUCTION

Virtually all supercomputers and computational cluster machines are programmed in MPI [12], where the embedding

*Supported in part by NSF award CNS00509379, Microsoft HPC Institutes Program, and SRC Contract 2005-TJ-1318.

language (C, C++, FORTRAN, etc.) specifies the intraprocess computations, and MPI function calls (over 130 in MPI 1.1 [12], and over 300 in MPI 2) provide a plethora of communication and synchronization commands. In fact, MPI is considered the *de facto* standard for distributed programming in the high performance computing (HPC) arena. While MPI programs avoid the pervasive global state that makes shared memory thread programs highly error prone, they are still extremely error-prone, as numerous studies show (e.g., [5]). Model checking can find deep-seated errors in MPI programs [15, 19]. However, containing state explosion is extremely important, especially given the fact that the intraprocess computations can interleave in an exponential number of ways. Partial-order reduction is a very natural approach to containing state explosion in this setting because MPI programs interact only when the MPI functions – such as various forms of send, receive, or barrier synchronization – are invoked. Unfortunately, the semantics of MPI are far from obvious; in fact, apart from an early LOTOS specification for a small subset of MPI [7], or the specification of some MPI operations including send, receive, etc. [20] using automata communicating through channels, there exists neither a more direct (e.g., transition system based) nor extensive formal characterization for MPI. The development of partial-order reduction methods for MPI is hampered by this lack of formalization. In this paper, we present a partial-order reduction method for a non-trivial subset of MPI, capitalizing on a high level formal semantic definition of MPI that we provide (full version in [14]), and capitalizing on dynamic information.

MPI communication commands include dozens of flavors of *sends* and *receives*. The *send* command always has to specify the receiver, but the *receive* command may or may not specify the sender. The former case is that of a *non-wildcard* receive, and the latter case a *wildcard* receive. Wildcard receives can match any sender targeting the process in which the receive occurs (barring *tag-matching*, a detail we suppress in this paper). In addition, one finds various forms of *wait* and *test* commands that help await (blocking semantics) or test (probe and return true/false without blocking) for the completion of asynchronous message transmittal. There are also many collective operations such as *barriers* and *reduces*.

Past work in the formal analysis of MPI includes work by Siegel and Avrunin [20, 22, 21, 18, 19] and our group [2, 13, 15, 14]. To make the contrast between past work and what

we propose here clearer, consider the two main commands supported by MPI, namely *send* and *receive*. In [20, 22], Siegel and Avrunin study “ordinary” flavors of these commands (known as `MPI_Send` and `MPI_Recv`). The MPI runtime system is allowed (but not required) to provide buffering for an `MPI_Send`. With buffering, the sender can post the message and asynchronously proceed; without buffering, a rendezvous-based message exchange happens. In [20, 22], Siegel and Avrunin show that one can analyze wildcard-free MPI programs for the absence of deadlocks by merely considering rendezvous-based executions, thus not inviting state explosion by modeling various degrees of the asynchronous sending.¹ In [18], the authors propose the following extension to their earlier result: in an MPI program with wildcards, if all senders that can potentially match a wildcard receive are known, then the same rendezvous-style communication can be forced. The knowledge about the senders comes in two ways: (i) all these sends are found as moves out of the current global state s being examined during depth-first search based explicit state enumeration model checking, or (ii) only some of the sends are found as moves out of s ; in this case it is assumed that the remaining sends will *never* be found offered in all future computations from s (thus, what is offered can be considered the *ample set* [4]). The authors propose a so-called *urgent algorithm* based on these ideas.

While shown effective on many case studies, the urgent algorithm only provides a partial answer to how partial-order reduction algorithms may be designed for MPI. First of all, the urgent algorithm was defined with respect to a formal model of MPI communication (introduced in [20]) that models only a few MPI functions. Many functions, especially the *deterministic* flavors of MPI *send* and *receive*, namely `MPI_Issend` and `MPI_Irecv` (frequently used during MPI program optimization) are not modeled. These commands behave in a deterministic manner because the associated communication buffers are explicitly provided by the user. They are termed *non-blocking* MPI calls, because after posting the send/receive, the computation advances without blocking, relying on a subsequent wait/test to check for message transmittal. Although a recent paper [19] considers non-blocking MPI commands, no partial-order reduction method is proposed for these non-blocking operations. In short, the approach taken in the urgent algorithm cannot be used as the basis of partial-order reduction for most MPI functions because: (i) a general understanding of the MPI semantics is required, and (ii) dependency on the future must be handled in a general manner. In this paper, we propose a partial-order reduction algorithm for MPI based on more general design principles. More specifically, we formalize the communication semantics of MPI, and then state and prove the classical notion of *independence* [4] among MPI program commands with respect to the formal semantics.

The behavior of wildcard receives (that future sends may induce a dependency on wildcard receives) is emblematic of a much more general issue with respect to MPI (and concurrent program analysis in general). It is well known that dependencies between operations will be precisely known only

¹We note that a very similar result was obtained in another setting by Manohar [11] and captured as the *Slack Elasticity* theorem.

at runtime. This is one of the main reasons why Flanagan and Godefroid devised the *dynamic* partial-order reduction (DPOR) method [6] for general concurrent software analysis. While the specific examples that motivated Flanagan and Godefroid pertained to unknown aliasing relationships and array range overlaps, we observe that the *same thinking* can be applied to the statically unknown (but dynamically known) dependency information in a communication oriented language such as MPI. We already foresee the possibility of handling many more dynamic features of MPI – such as `MPI_Cancel` – which allows a pending MPI operation to be canceled.

In the context of DPOR itself, our algorithm has two primary differences: (i) our DPOR algorithm is tailored for MPI operations in some natural ways, and (ii) there is no place in our DPOR algorithm where we need to perform full expansion of a state to deflect unsoundness. Regarding point (i), the MPI standard *requires* that correct MPI programs terminate at an `MPI_Finalize` call. Hence, the acyclic state-space requirement of MPI is checked by default in our algorithm, which keeps fingerprints of visited states in a hash table. For point (ii) we discuss this in detail with respect to the example shown in Figure 11 in Section 6.

Roadmap: Section 2 contains examples illustrating MPI and our contributions. Section 3 presents a simplified goto based modeling language for MPI, the semantics of transitions in that language, along with a number of theorems regarding the independence of transitions under the intended semantics of MPI. Section 4 gives the dynamic partial-order reduction algorithm that we have developed for use with MPI based programs, and provides a proof sketch that a number of interesting properties are preserved. Section 5 takes aim at the 2D diffusion example of [21]. We adapt this example to our MPI communication primitives, and are able to demonstrate the advantages of DPOR. We discuss preliminary results and integration into a practical analysis framework. Section 6 compares some interesting facets of our work with existing related work. Section 7 gives some future directions and concludes.

2. MOTIVATING EXAMPLES

We provide a few examples of MPI programs, partly to give the reader a basic understanding of MPI, partly for illustration later, and partly to denote issues which our approach cannot yet handle. Most MPI programs employ the single program multiple data (SPMD) style. The subset of MPI operations we model² includes `MPI_Issend`, `MPI_Irecv`, `MPI_Wait`, `MPI_Test`, and `MPI_Barrier`; we also allow wildcard receive operations. We will omit the prefix “MPI.” in the remainder of this paper.

The N processes in an MPI distributed computation can be differentiated based on the unique *process rank* (integers in $\{0, \dots, N - 1\}$) that the MPI runtime system assigns to each MPI process. Messages are addressed by process rank (“rank”) or through wildcards (denoted by $*$) that stand for `ANY_SOURCE`. Consider the simple communication pattern where all processes send a message to the *root* process

²In our approach, the more familiar operations such as `MPI_Send` are modeled in terms of these more primitive operations.

```

1 if(rank != 0){
2   h = Issend 0 (addrof x)
3   Wait h
4 } else {
5   for(int i = 0; i < N; ++i){
6     h = Irecv i (addrof x)
7     Wait h
8   }
9 }

```

Figure 1: Deadlock caused by a mismatched handshake.

```

1 if(rank==0){
2   count = 1
3   while(count < N){
4     h = Irecv * (addrof x) // Receive rank of
5     Wait h                // completed proc
6     count++
7   }
8   assert(x == 3);
9 } else {
10  //do some work
11  if(done){
12    h = Issend 0 (addrof rank) // Send rank to proc 0
13    Wait h
14  }
15 }

```

Figure 2: An assertion violation caused by non-deterministic message order.

(rank 0) implemented in Figure 1. Here, `Issend 0` means “send to 0” while `Irecv i` means “receive from i.” This code will deadlock as it contains a mismatched communication: waiting (through the `wait h` command) for an `Irecv` with source rank 0 to complete before executing the `Irecv` operations for other ranks.

Figure 2 shows a second example where a process can receive from multiple potential sources in any order. Here the developer expected the process with `rank == 3` to always finish last. However unless this is guaranteed using some other synchronization mechanism, it will be possible to violate this assertion. This example underscores the importance of model checking based verification of MPI programs, as opposed to random-testing which can miss such bugs.

A third example shown in Figure 3 contains an error specific to MPI – although it is tempting to program in this manner. MPI makes no guarantee about process fairness. In this example, while the user may have intended for a communication to eventually force the termination of the while loop, it may in fact continue forever. Liveness checking is future work for us, however, if this example creates a cycle our algorithm will detect the cycle and report it as an error.

Again, considering Figure 1, *which actions in this program are independent?* It turns out that the `Issend` operations from line 2 are dependent only on the `Wait` corresponding to the `Irecv` operations on line 7 when `i` in the `Irecv` on line 6 is equal to the rank of the sending process! Similarly the `Wait` on line 3 is dependent only on the `Irecv` of line 6 when `i` is equal to the rank of the sending process. All other program actions are independent. All this information is a simple and direct consequence of our formal semantics, as we show later. Without a formal semantics, finding such

```

1 if(rank != 0){
2   h = Issend 0 (addrof x)
3 } else {
4   h = Irecv * (addrof x)
5   flag = Test h
6   while(!flag){
7     // do some work that
8     // does not involve x
9     flag = Test h
10  }
11 }

```

Figure 3: The while loop may never terminate.

```

1 if(rank%3==0){
2   h = Irecv * (addrof x);
3   i = Irecv * (addrof y);
4   flag = Test h;
5   Wait i;
6 } else {
7   if((rank+2)%3==0){
8     h = Issend (rank+2)%rank (addrof x);
9   } else {
10    h = Issend (rank+1)%rank (addrof x);
11  }
12  Wait h;
13 }

```

Figure 4: The role of dynamic information in POR for MPI.

dependencies becomes difficult.

Our next example (Figure 4) demonstrates the need to use dynamic information in the computation of dependencies. Let this program be instantiated for $N = 6$ processes. This program causes ranks 1 and 2 to send a message to rank 3; ranks 4 and 5 also send to rank 0. The messages can be received in either order—meaning that the value in `x` on rank 0, after the `Wait h`; executes, could be either from rank 5 or rank 6. Moreover, since `Test` is used on line 4 it is possible that one of the messages is not received by this part of the program. Although dependencies exist, our dynamic partial-order reduction algorithm naturally forms clusters of independence e.g., as in [3]; however, unlike in their work, the clusters will not be statically and *a priori* determined.

3. LANGUAGE EXECUTION AND COMMUNICATION SEMANTICS

In this section we define the execution semantics for a simple goto-based program modeling language for MPI called MPIC (our much more detailed semantics of MPI appears in [14] that is cross-referenced with the natural language standard [12]). The grammar for this language is shown in Figure 5. We have followed the C convention in modeling Boolean operations using Integers. Non-zero values are true; zero is false. For communication we have chosen a representative operation from each of the point-to-point operation groups.

The communication operations were chosen for their applicability in deterministic optimization. A program that implements message packaging and queuing would use `Issend` and `Irecv` to indicate to the MPI subsystem that additional buffering is unnecessary—thereby increasing performance. `Wait` and `Test` both complete communications. `Wait`

```

const ::= int
var ::= intvar
intexp ::= const
        | var
        | intexp arithop intexp
        | intexp logicop intexp
        | unaryop intexp
labexp ::= label
        | 'if' intexp 'then' label 'else' label
exp ::= intexp
      | '(' exp ')'
localinstr ::= var '=' exp
            | 'goto' labexp
            | 'assert' intexp
comminstr ::= int '=' 'issend' intexp
            | int '=' 'irecv' intexp
            | bool '=' 'test' intvar
            | 'wait' intvar
            | 'barrier_init'
            | 'barrier_wait'
instr ::= localinstr ';' instr
        | localinstr
        | comminstr ';' instr
        | comminstr
        | label ':' instr
prog ::= (instr)+

```

Figure 5: The Grammar of MPIC.

$$\begin{aligned}
\textit{Match} \quad & ((src_1, dest_1, _, type_1, completed_1), \\
& (src_2, dest_2, _, type_2, completed_2)) = \\
& \wedge dest_1 = dest_2 \\
& \wedge completed_1 = \textit{false} \\
& \wedge completed_2 = \textit{false} \\
& \wedge \vee src_1 = src_2 \\
& \quad \vee \wedge type_1 = \textit{recv} \\
& \quad \quad \wedge src_1 = * \\
& \quad \vee \wedge type_2 = \textit{recv} \\
& \quad \quad \wedge src_2 = * \\
& \wedge \vee \wedge type_1 = \textit{send} \\
& \quad \wedge type_2 = \textit{recv} \\
& \quad \vee \wedge type_1 = \textit{recv} \\
& \quad \quad \wedge type_2 = \textit{send} \\
\textit{Completed} \quad & (_, _, _, c) = c \\
\textit{Type} \quad & (_, _, _, t, _) = t \\
\textit{Address} \quad & (_, _, a, _, _) = a
\end{aligned}$$

Figure 6: Helper functions.

blocks until the communication completes, whereas `Test` is always enabled and returns in a flag whether the communication has completed. `Barrier` is used to conservatively represent all of the collective communication operations of MPI—any of which may have a synchronizing effect (including the actual `Barrier` function supported by MPI that is always synchronizing).

Figure 6 provides four helper functions that are used in the remainder of our presentation. Communications between processes are determined using meta-data provided in the request. When two requests can form a communication we say they *Match*. The predicate *Match* evaluates to true when two message requests, one sending and the other receiving, can be combined to create a communication. The functions *Completed*, *Type*, and *Address*, provide access to the corresponding fields of the request tuple.

A state is a tuple (c, p) where c is the collective context and p is a function mapping process identifiers (pids in \mathbb{N}) onto the state of each process. Processes are individually modeled by a local store l and global store g . Each local store l_i of process i is a function from addresses in \mathbb{N} onto values also in \mathbb{N} . The global store can only be accessed via MPI operations. Each global store g_i is a function from pids to a set of request tuples. A request is a five-tuple $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{B} \times \mathbb{B}$ indicating the source, destination, memory address to be read or written, message type (being either send or receive), and an indication whether the message has been completed. The state of collective operations such as `Barrier` are recorded in the collective context c . The collective context c is modeled using a tuple $\{vacant, in, out\} \times \mathcal{P}(\{i : i \in 0..N - 1\})$, representing the state of the collective operation and the processes that are currently participating. The parameter N is constant in our system and represents the number of processes participating in the distributed computation.

Each of the rules in Figures 7 and 8 manipulate the state tuple. Operations with multiple rules model the disjoint nature of the transition type. The `Barrier` operation is modeled using six rules implementing two transition types: an entrance and an exit. The reachable state space, viewed as a unary predicate Σ , is recursively defined by the execution of these rules parameterized over the transition relation of a given program and some fixed number of processes N . The transition relation is defined using two functions. The *proc* function returns the AST node for a given pc value. Sequential control flow is modeled by the *next* function. The $pc \in \mathbb{N}$ is held in the local store of each process.

Although suppressed for simplicity in this presentation, data is transmitted between processes in the `Wait` (2) and `Test` (2) rules.

Again consider the example of Figure 1. This program uses `Assignment`, `Goto`, `Issend`, `Irecv`, and `Wait`. We assume the existence of a suitable evaluator E for expressions in our language and that control flow is simplified into conditional `goto` statements. The rule for `Assignment` says that if there is a state (c, p) such that *proc* maps to an assignment for the current pc of process i , in the next state the system updates the value of the pc with $next(pc)$, and assigns the evaluated value of expression e in the current state to the memory location referenced by x . `Goto` is very similar except the expression evaluated is assigned to the pc in the next state. `Issend` and `Irecv` are again similar with the exception that x contains a numeric handle to the request that is created by the execution of this operation.

Before we elaborate on the semantics of `Wait`, we should mention a bit about requests. The natural language MPI standard [12] uses the term request extensively without giving a formal definition to it. Send and receive operations, regardless of their type (we show only `Issend` and `Irecv` in this paper), all produce or activate requests; `Wait` and `Test` operations consume or deactivate and possibly deallocate requests depending on the flavor of operation and request. In our work we have defined a request to be a five-tuple that contains the rank of the sender and receiver, the address of memory to be read or written, the type of message being requested (either send or receive), and whether or not

Assignment:

$$\frac{\Sigma(c, p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{assign } x \ e)}{\Sigma(c, p[i \mapsto (l[pc \mapsto \text{next}(pc)], E[(\text{addrof } x), p(i)] \mapsto E[e, p(i)]], g)])}$$

Goto:

$$\frac{\Sigma(c, p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{goto } e)}{\Sigma(c, p[i \mapsto (l[pc \mapsto E[e, p(i)]], g)])}$$

Assert:

$$\frac{\Sigma(c, p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{assert } e) \wedge E[e, p(i)] = \text{true}}{\Sigma(c, p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], g)])}$$

Barrier_init (1):

$$\frac{\Sigma((\text{vacant}, \emptyset), p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{barrier_init})}{\Sigma((in, \{i\}), p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], g)])}$$

Barrier_init (2):

$$\frac{\Sigma((in, s), p) \wedge p(i) = (l, g) \wedge i \notin s \wedge \text{proc}(l(pc)) = (\text{barrier_init})}{\Sigma((in, s \cup \{i\}), p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], g)])}$$

Barrier_wait (1):

$$\frac{\Sigma((in, s), p) \wedge p(i) = (l, g) \wedge s = \{i : i \in 0..(N-1)\} \wedge s \neq \{i\} \wedge \text{proc}(l(pc)) = (\text{barrier_wait})}{\Sigma((out, s \setminus i), p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], g)])}$$

Barrier_wait (2):

$$\frac{\Sigma((in, s), p) \wedge p(i) = (l, g) \wedge s = \{i : i \in 0..(N-1)\} \wedge s = \{i\} \wedge \text{proc}(l(pc)) = (\text{barrier_wait})}{\Sigma((\text{vacant}, \emptyset), p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], g)])}$$

Barrier_wait (3):

$$\frac{\Sigma((out, s), p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{barrier_wait}) \wedge i \in s \wedge s \setminus i \neq \emptyset}{\Sigma((out, s \setminus i), p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], g)])}$$

Barrier_wait (4):

$$\frac{\Sigma((out, s), p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{barrier_wait}) \wedge i \in s \wedge s \setminus i = \emptyset}{\Sigma((\text{vacant}, \emptyset), p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], g)])}$$

Issend:

$$\frac{\Sigma(c, p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{issend } x \ \text{dest } \text{addr})}{\Sigma(c, p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], E[(\text{addrof } x), p(i)] \mapsto (|Dom(g)| + 1)], g[(|Dom(g)| + 1) \mapsto (i, E[\text{dest}, p(i)], E[\text{addr}, p(i)], \text{send}, \text{false}])])}$$

Irecv:

$$\frac{\Sigma(c, p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{irecv } x \ \text{src } \text{addr})}{\Sigma(c, p[i \mapsto (l[pc \mapsto \text{next}(l(pc))], E[(\text{addrof } x), p(i)] \mapsto (|Dom(g)| + 1)], g[(|Dom(g)| + 1) \mapsto (E[\text{src}, p(i)], i, E[\text{addr}, p(i)], \text{recv}, \text{false}])])}$$

Figure 7: Semantic definitions of Assignment, Assert, Goto, Barrier_init, Barrier_wait, Issend, and Irecv.

Wait (1):

$$\frac{\Sigma(c, p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{wait } e) \wedge (E[e, p(i)] = 0 \vee (E[e, p(i)] \in \text{Dom}(g) \wedge \text{Completed}(g(E[e, p(i)]))))}{\Sigma(c, p[i \mapsto (l[pc \mapsto \text{next}(l(pc)), E[(\text{addrof } e), p(i)] \mapsto 0], g)])}$$

Wait (2):

$$\frac{\Sigma(c, p) \wedge p(i) = (l_i, g_i) \wedge \text{proc}(l_i(pc)) = (\text{wait } e) \wedge E[e, p(i)] \in \text{Dom}(g_i) \wedge p(j) = (l_j, g_j) \wedge \text{Match}(g_j(k), g_i(E[e, p(i)])) \wedge m < k \Rightarrow \neg \text{Match}(g_j(m), g_i(E[e, p(i)]))}{\Sigma(c, p[i \mapsto (l_i[pc \mapsto \text{next}(l_i(pc)), E[(\text{addrof } e), p(i)] \mapsto 0], g_i[E[e, p(i)] \mapsto g_i(E[e, p(i)])[false/true]], j \mapsto (l_j, g_j[k \mapsto g_j(k)[false/true]])])}$$

Test (1):

$$\frac{\Sigma(c, p) \wedge p(i) = (l, g) \wedge \text{proc}(l(pc)) = (\text{test } e_1 e_2) \wedge (E[e_1, p(i)] = 0 \vee (E[e_1, p(i)] \in \text{Dom}(g) \wedge \text{Completed}(g(E[e_1, p(i)]))))}{\Sigma(c, p[i \mapsto (l[pc \mapsto \text{next}(l(pc)), E[(\text{addrof } e_1), p(i)] \mapsto 0, E[(\text{addrof } e_2), p(i)] \mapsto \text{true}], g)])}$$

Test (2):

$$\frac{\Sigma(c, p) \wedge p(i) = (l_i, g_i) \wedge \text{proc}(l_i(pc)) = (\text{test } e_1 e_2) \wedge E[e_1, p(i)] \in \text{Dom}(g_i) \wedge \neg \text{Completed}(g_i(E[e_1, p(i)])) \wedge p(j) = (l_j, g_j) \wedge \text{Match}(g_j(k), g_i(E[e_1, p(i)])) \wedge m < k \Rightarrow \neg \text{Match}(g_j(m), g_i(E[e_1, p(i)]))}{\Sigma(c, p[i \mapsto (l_i[pc \mapsto \text{next}(l_i(pc)), E[e_1, p(i)] \mapsto 0, E[e_2, p(i)] \mapsto \text{true}], g_i[E[e_1, p(i)] \mapsto g_i(E[e_1, p(i)])[false/true]], j \mapsto (l_j, g_j[k \mapsto g_j(k)[false/true]])])}$$

Test (3):

$$\frac{\Sigma(c, p) \wedge p(i) = (l_i, g_i) \wedge \text{proc}(l_i(pc)) = (\text{test } e_1 e_2) \wedge E[e_1, p(i)] \in \text{Dom}(g_i) \wedge \neg \text{Completed}(g_i(E[e_1, p(i)])) \wedge p(j) = (l_j, g_j) \wedge \neg \text{Match}(g_j(k), g_i(E[e_1, p(i)]))}{\Sigma(c, p[i \mapsto (l[pc \mapsto \text{next}(l(pc)), E[e_2, p(i)] \mapsto \text{false}], g)])}$$

Figure 8: Semantic definitions of Wait and Test.

this particular request has been completed. Moreover, we ignore deallocation and deactivation and focus only on the producer/consumer relationship that exists between `Issend`, `Irecv`, `Wait`, and `Test` in this restricted context. A model that admits more of these operations (such as `Ssend_init` and `Start`) would have to take into account the additional complexity (cf. our TLA+ model of MPI [14]).

A process completes the communication initiated by an `Issend` or `Irecv` by executing a `Wait` or `Test`. The `Wait` operation is paired with an `Issend` or `Irecv` on a particular process via a request handle returned by the `Issend` or `Irecv`. The execution semantics are similar to `Assignment` as it requires that some process i be positioned to execute a `Wait` operation in the current state. The additional restriction for `Wait` (1) is that either the request handle passed to the `Wait` evaluates to 0 ($E[e, p(i)] = 0$), a special value outside $Dom(g)$ used to represent `REQUEST_NULL`, or if the handle is valid then the request has been completed by the `Wait` or `Test` of another process. In this case the pc of the process is updated to $next(pc)$ and the handle is set to 0. The additional restriction for `Wait` (2) is that the request handle evaluate to a value in $Dom(g_i)$ and there must be some other request on process j that will *Match* the request $g(E[e, p(i)])$. The final clause, in connection with the deterministic structure of a single process, enforces the program order matching requirement for requests imposed by MPI. In this case the pc and handle are updated as before. In addition, the global store is modified to reflect that the communication has completed. The data is also moved from sender to receiver in this step although tacit in our presentation.

3.1 Assumptions and Properties of Interest

Some well-formedness assumptions are made of MPI programs that are handled by our DPOR algorithm. It is assumed that a process does not access the buffer passed to an `Issend` or `Irecv` until after the `Wait` or `Test` returns and if `Test` is used, the flag returned by `Test` is true. More formally, for any process i , if a is an address such that $\exists r : (Address(g_i(r)) = a) \wedge \neg Completed(g_i(r))$ then it is assumed that no other action of process i will read or write $l_i(a)$. This assumption is necessary for correctness according to the MPI standard [12]. An example that violates this assumption is: `h = irecv 3 (addr of h)`. A second assumption is that programs cannot explicitly reference the program counter of any process, i.e., statements of the form `pc = exp` and `v = irecv 2 (addr of pc)` are a violation of this assumption. It is possible to check these assumptions, with modest or no over-approximations, using existing static analysis methods.

Only certain properties are preserved by our reduction algorithm. In particular, under this execution semantics, the reduction algorithm proposed here preserves (i) deadlocks, (ii) cycles³, and (iii) local assertions, more formally, assertions on $Dom(l_i)$ for each process i that are invariant under stuttering.

3.2 Independence

³The MPI standard requires all processes to eventually call `Finalize`, after which there can be no more communication via the MPI library. Therefore, all cycles are considered errors.

We can now discuss the independence properties of the transition semantics presented in Figures 7 and 8. For completeness we restate the definition of independence from [8]. For any state $\sigma \in \Sigma$, the set of transitions enabled in σ , $enabled(\sigma)$ is defined as $enabled(\sigma) = \{t_i \mid t_i(\sigma) \in \Sigma\}$. A transition t_i of process i and a transition t_j of process j where $i \neq j$ are independent (i.e., $I(t_i, t_j)$) if

$$\forall \sigma \in \Sigma : t_i, t_j \in enabled(\sigma) \Rightarrow (t_i \in enabled(t_j(\sigma)) \wedge t_i(t_j(\sigma)) = t_j(t_i(\sigma)))$$

We say that two transitions t_i and t_j are dependent if $\neg I(t_i, t_j)$.

With the transition semantics defined we are able to state and prove a number of theorems regarding the independence of the different transition types. For brevity we will only provide proof sketches, keeping the details for Appendix A.

THEOREM 1. *If t_i is an Assignment, Goto, Assert, or Barrier transition for a given process i and t_j is any transition of process j where $i \neq j$ then $I(t_i, t_j)$.*

To prove this theorem for Assignment, Goto, and Assert it is sufficient to note that disjoint memory spaces are read and written. For each of the Barrier transitions, the key observation is that the execution of other processes cannot disable the enabled transition into or out of a Barrier. In both cases independence is with respect to observable behavior.

Corresponding theorems for `Issend`, `Irecv`, `Wait`, and `Test` require some additional proof machinery. Figure 9 gives the definition for the `Complete` predicate and pseudo-code for the `Index` operation. The `Complete` predicate returns true for two transitions when they could be used to cause a communication between two processes. In the pseudo-code, $post(t)$ is the state generated by executing transition t , $proc(t)$ is the process that executed t . The `Index` operation takes a transition t as an argument and returns the handle of the request referenced by t .

THEOREM 2. *If t_i is an Issend, Irecv, Wait, or Test of process i and t_j is any transition of another process j where $i \neq j$ and $\neg Complete(t_i, t_j)$ then $I(t_i, t_j)$.*

Intuitively, `Complete` indicates when two transitions could result in a communication under the right execution interleaving order. This theorem says that transitions that cannot result in a communication are independent with respect to the observable behavior. This is the case again because the transitions involved read from and write to disjoint memory spaces.

The consequence of the above two theorems is that all of the program's non-communication actions are independent and many of the communication actions are also independent. Moreover, we have given a simple predicate (i.e., `Complete`) that can be evaluated during model checking to determine whether two MPI communication operations are in fact dependent. Using this predicate, the questions from Section 2

```

Index(t)  $\equiv$ 
IF t is a Wait or Test
THEN
  return the evaluated handle argument
ELSE
  IF t is an Issend or Irecv
  return the value written into the handle address
  ELSE
  return the null request value

Complete( $\alpha$ ,  $\beta$ )  $\equiv$ 
Let ( $c_1$ ,  $p_1$ ) = post( $\alpha$ ) in
Let ( $l_1$ ,  $g_1$ ) =  $p_1$ (proc( $\alpha$ )) in
Let ( $c_2$ ,  $p_2$ ) = post( $\beta$ ) in
Let ( $l_2$ ,  $g_2$ ) =  $p_2$ (proc( $\beta$ )) in
Let i = Index( $\alpha$ ) in
Let j = Index( $\beta$ ) in
IF  $\neg(i = \text{null}) \wedge \neg(j = \text{null})$ 
THEN
  Match( $g_1(i)$ ,  $g_2(j)$ )
ELSE
  false

```

Figure 9: The Complete operation.

pertaining to the independence of actions in Example 1 can be answered.

4. DYNAMIC PARTIAL-ORDER REDUCTION FOR MPI

As discussed in Section 2, traditional partial-order reduction techniques have prohibitive limitations in the setting of MPI for the following reasons: (i) Addressing is most likely computed at run time as a function of the pid or rank of a process. (ii) In the presence of wildcard receive operations it becomes difficult to know if all possible Issends will be interleaved without fully expanding the state. (iii) The dependencies are only between operations that **Complete**, therefore, one needs to compute the trajectory of the handle returned by the Issend or Irecv.

The algorithm we propose and have implemented appears in Figure 10. The algorithm adds pids of processes with enabled transitions to the **interleave** set arbitrarily in the forward direction (ε is the choose operator). A state at **top**(s) on line 24 is popped from the search stack s when that state has no enabled transitions or when there are no more transitions to try from **nextinterleaved**. The **nextinterleaved** function iterates through the enabled transitions for each process in **interleave**(q), returning each in turn. When a state q is about to be popped from s , the transition executed to generate q , namely **tran**(q), is compared to transitions in the search stack using the **Complete** predicate. The state v is the state before the closest dependent transition. In v we schedule the process **proc**(q) that executed to generate q .

The correctness of this algorithm depends upon the types of properties we are trying to preserve by the reduction. As stated in Section 3.1, we are interested in verifying local assertions, the absence of deadlocks, and the absence of cycles.

THEOREM 3. *If a local assertion is violated, a deadlock exists, or a cycle exists in the full state space, then it will be explored by the algorithm of Figure 10.*

```

q, q' : state
s      : stack
h      : state set

1  q := initial state
2  interleave(q) := { $\varepsilon$  p :  $\exists t_p \in \text{enabled}(q)$ }
3  push(s, q)
4  h := {q}
5
6  while size(s) > 0
7    q := top(s)
8    if |interleave(q)| > 0
9       $t_p := \text{nextinterleaved}(q)$ 
10      $q' := t_p(q)$  (* Local assertions are checked here. *)
11     if  $q' \notin h$ 
12        $h := h \cup \{q'\}$ 
13       interleave( $q'$ ) := { $\varepsilon$  p :  $\exists t_p \in \text{enabled}(q')$ }
14       push(s,  $q'$ )
15     else (* Cycles are detected here. *)
16       if  $\exists i \in \text{Dom}(s) : s[i] = q'$ 
17         report an error and exit
18       end if
19     end if
20   else
21     if  $\exists i \in \text{Dom}(s) : \text{Complete}(\text{tran}(s[i]), \text{tran}(q))$ 
22        $v := \text{pre}(s[\max(i \in \text{Dom}(s) : \text{Complete}(\text{tran}(s[i]), \text{tran}(q)))])$ 
23       interleave( $v$ ) := interleave( $v$ )  $\cup$  {proc(tran( $q$ ))}
24     end if
25   pop(s)
26   end if
27 end while
28

```

Figure 10: Dynamic partial-order reduction based depth first search.

PROOF. (Sketch) The proof proceeds in two parts. The first part shows that the transitions explored by the algorithm form persistent sets at each state—thereby preserving local assertions and deadlocks. We note that if no transitions are executed from a given state by the algorithm, then there must be no enabled transitions at that state. This empty set is therefore persistent trivially. If the enabled set is non-empty then any transition that is enabled and not independent of the transition selected but not executed in a given state will eventually be executed by the algorithm. This means that the stack will be searched when that transition is backed off of and the process executing the transition will be scheduled before the dependent transition.

The second part of the proof shows that a cycle may be delayed but will not be ignored. Since all cycles are considered errors when the cycle is eventually closed it is reported at line 18. \square

Appendix A provides a complete description of our proof.⁴

5. EXPERIMENTS USING DPOR FOR MPI IN MPIC

To demonstrate the effectiveness of the specialization of partial-order reduction for MPI primitives, consider the two dimensional diffusion simulation described in [21]. Here the authors model the MPI primitives using Promela and attempt to model check for a 4×4 grid (16 processes). They report that the model checker runs out of memory. It is not

⁴The appendix may be removed and cited separately in the final version of this paper as space constraints require.

```

1 if(rank == 0){
2   h = Irecv * (addrof x);
3 } else {
4   h = Issend 0 (addrof x);
5 }
6 Wait h;

```

Figure 11: A non-deterministic receive operation.

clear whether the authors attempt to use the partial-order reduction implemented in SPIN.

To handle this program we made a few modifications. First we changed the pseudo-code shown in their paper into C. We then transformed the program so that all of the Send and Recv operations were the corresponding Issend and Irecv operations followed by a Wait. Optimization came next—we moved the MPI operations such that setting up buffers for communication could overlap the sending and receiving of buffers.⁵

From the program text we extract a model automatically using the Microsoft Phoenix compiler [16]. We then automatically simplified the model by inlining and slicing such that only the communication skeleton is preserved. We are then able to verify this example generating only 7×10^5 states in about one minute on an ordinary laptop computer (2 GHz, 1GB memory). We could not handle this example without partial-order reduction. Additional experiments to collect statistics are in progress. The next section discusses one more example which elucidates the inner workings of our algorithm even more.

6. DISCUSSIONS AND ADDITIONAL RELATED WORK

Our algorithm is similar in many respects to the algorithm proposed in [6], with one notable difference. In the Godefroid and Flanagan work, when a process p is added to the backtrack set $backtrack(v)$ for some state v , the algorithm checks to insure that p is enabled. If p is disabled, a check is performed to find some transition $t \in enabled(v)$ such that p becomes enabled in $t(v)$. If no such t can be found v is fully expanded. Our algorithm does not require this check as a result of the semantics of MPI. In particular, only Wait and the Barrier operations can become disabled. We have already shown that Barrier transitions are independent of transitions of other processes.

For Wait, consider the execution sequence in Figure 11, instantiated for three processes. If the communication execution order is Irecv; Issend₁; Wait_{Issend₁}; Issend₂; Wait_{Irecv}; then the problem is that the Irecv operation may not form a communication with the Issend of process 2 because Wait_{Issend₁} will force a match and disable the ability of Wait_{Irecv} to form a communication non-deterministically under our execution semantics. However after executing the algorithm, the following interleaved sets would be produced (shown between transitions):

{0, 1, 2}	In this state we force the Issend ₂ to happen first in the subsequent exploration sequence
Irecv	
{1, 0}	The Wait _{Irecv} is disabled
Issend ₁	
{1, 0}	The Wait _{Irecv} has no other choice and the Issend ₂ does not complete the Issend ₁ operation so process 2 is not scheduled
Wait _{Issend₁}	
{2, 0}	The Wait _{Irecv} is still forced to match Issend ₁
Issend ₂	
{0}	
Wait _{Irecv}	

Since the Issend₂ is eventually forced to happen first—meaning above the receive, then it remains to show that some interleaving will either have Wait_{Issend₂} or Wait_{Irecv} before Wait_{Issend₁}. We know this is the case because Wait_{Irecv} and Wait_{Issend₁} are guaranteed to happen in both orders whenever they appear along any path because Complete(Wait_{Irecv}, Wait_{Issend₁}).

6.1 Other Related Work

This paper has focused on the presentation of a partial-order reduction algorithm that is enabled by the communication semantics of MPI. Other models of MPI exist including [20] where the authors build a model of MPI from first principles. They then propose an urgent scheduling for MPI operations included in their model that preserve a halting properties. They have implemented many of the MPI primitives—including a number of non-blocking operations in the SPIN [19] model checker. The approaches are difficult to compare because they work for different subsets of MPI. It is also not clear how or whether partial-order reduction is being used in their current implementation.

Other previous work in formalizing MPI such as [7, 14, 20, 13, 2] do not implement the semantics proposed directly in a model checker. Rather these models serve to augment the program model in a library format.

There are several model checkers that have partial-order reduction such as SPIN [10], Zing [1], Verisoft [9], Bogor [17] and perhaps others. In each of these, the reduction is not tailored to MPI.

Partial-order reduction has been studied extensively—a survey of which is beyond the scope of this paper. This work is most closely related to the dynamic partial-order reduction algorithm of Godefroid and Flanagan [6]. The two primary differences being (i) our algorithm is tailored for MPI operations, and (ii) there is no place in our algorithm where full expansion is used to deflect unsoundness.

7. FUTURE DIRECTIONS

One problem that faces software model checking is getting a reasonable model to verify from some piece of source code. We have mentioned in Section 5 that we have a framework that helps in this regard although details are not included in this paper. Static analysis to help reduce the size of the models checked is an important area of future work.

There is immense potential for additional research in this area. There are many more MPI operations that require

⁵This program code can be downloaded from our web-site at www.cs.utah.edu/formal_verification/verification_environment.

a formal semantic characterization, to show independence. We conjecture that these proofs are mechanizable. Once the independence theorems are known they can be used in model checking to eliminate unnecessary redundant exploration by the model checker.

More significantly, the approach of formalizing communication libraries and building formal semantics based partial-order reduction algorithms may emerge to be a viable approach in analyzing other library based parallel and distributed programs.

8. REFERENCES

- [1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV 2004: 16th International Conference on Computer Aided Verification, Boston, Massachusetts, July 2004*, LNCS. Springer-Verlag, 2004.
- [2] Steven Barrus, Ganesh Gopalakrishnan, Robert M. Kirby, and Robert Palmer. Verification of MPI programs using SPIN. Technical Report UUCS-04-008, The University of Utah, 2004.
- [3] Twan Basten, Dragan Bosnacki, and Marc Geilen. Cluster-based partial-order reduction. *Automated Software Eng.*, 11(4):365–402, 2004.
- [4] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [5] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of mpi programs with intel message checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 78–82, 2005. ISBN:1-59593-117-1.
- [6] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM Press.
- [7] Philippe Georgelin, Laurence Pierre, and Tin Nguyen. A formal specification of the MPI primitives and communication mechanisms. Technical report, LIM, 1999.
- [8] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer-Verlag, 1996.
- [9] Patrice Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.
- [10] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [11] Rajit Manohar and Alain J. Martin. Slack elasticity in concurrent computing. In *MPC '98: Proceedings of the Mathematics of Program Construction*, pages 272–285, London, UK, 1998. Springer-Verlag.
- [12] MPI: A Message-Passing Interface Standard. <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>.
- [13] Robert Palmer, Steven Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss: A framework for verifying scientific computing software. In *SoftMC: Workshop on Software Model Checking*, number 953 in ENTCS, August 2005.
- [14] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby. The communication semantics of the message passing interface. Technical Report UUCS-06-012, The University of Utah, October 2006.
- [15] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Formal verification of programs that use mpi one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192/2006 of LNCS, pages 30–39, Berlin/Heidelberg, 2006. Springer.
- [16] <http://research.microsoft.com/phoenix>.
- [17] Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267–276. ACM, 2003.
- [18] Stephen F. Siegel. Efficient verification of halting properties for mpi programs with wildcard receives. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005*, pages 413–429, Paris, January 2005. Springer-Verlag.
- [19] Stephen F. Siegel. Model checking nonblocking mpi programs. In B. Cook and A. Podelski, editors, *VMCAI 07*, number 4349 in LNCS, pages 44–58. Springer-Verlag, 2007.
- [20] Stephen F. Siegel and George Avrunin. Analysis of mpi programs. Technical Report UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst, 2003.
- [21] Stephen F. Siegel and George S. Avrunin. Verification of mpi-based software for scientific computation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, volume 2989 of LNCS, pages 286–303, Barcelona, April 2004. Springer.
- [22] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free MPI programs for verification. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 95–106, Chicago, June 2005.
- [23] Antti Valmari. A stubborn attack on state explosion. In *CAV 91: Computer Aided Verification*, pages 156–165. Springer-Verlag, 1991.

APPENDIX

A. PROOFS

A.1 Proof of independence theorems

For each of the proofs to follow we assume the requirements of Section 3.1 hold.

LEMMA 1. *If t_i is an Assignment, Goto, or Assert transition for a given process i and t_j is any transition of process j where $i \neq j$, then $I(t_i, t_j)$.*

PROOF. To show the enabledness condition, suppose t_i is an Assignment, Goto, or Assert transition of process i that is enabled at s . Now suppose that t_i is disabled in $t_j(\sigma)$. Then the action of process j changed the local store l_i of process i (as the enabling condition for t_i is only dependent on l_i). Since none of the semantic rules allows such a modification we have a contradiction. Now suppose that t_j is disabled in $t_i(\sigma)$. We can see that t_i is only able to write the local store of process i . Therefore it is not possible to disable any action of process j , including t_j .

Now to show the commutativity property, again suppose that t_i is a transition, as before, of process i enabled at σ and t_j is a transition of process j where $t_i, t_j \in \text{enabled}(\sigma)$. Since we have $t_i \in \text{enabled}(t_j(\sigma)) \wedge \Sigma t_j(\sigma)$ and $t_j \in \text{enabled}(t_i(\sigma)) \wedge \Sigma t_i(\sigma)$ clearly there some $\sigma_1 = t_i(t_j(\sigma))$ and $\sigma_2 = t_j(t_i(\sigma))$. We conclude $\sigma_1 = \sigma_2$ as the modified stores are disjoint. \square

LEMMA 2. *If t_i is a Barrier transition for a given process i and t_j is any transition of process j where $i \neq j$ then $I(t_i, t_j)$.*

PROOF. To show the enabledness condition, suppose t_i is a Barrier transition of process i , t_j is a transition of another process, and $t_i, t_j \in \text{enabled}(\sigma)$ for some $\sigma \in \Sigma$. Now suppose for contradiction that executing t_j causes t_i to become disabled. The enabledness conditions of t_i is only dependent upon actions of process i and the state of the collective context. The collective context changes only in response to all processes entering or exiting the barrier. Therefore we can conclude that t_i was disabled by an action of process i , a contradiction. Now suppose that t_j is disabled in $t_i(\sigma)$ Since t_i only modifies the collective context of σ and the $l_i(pc)$ of process i then t_j must be a Barrier operation. If t_j is a Barrier_init that is enabled in σ and disabled in $t_i(\sigma)$ then t_i must have added j to the collective context—a contradiction. If t_j is a Barrier_wait that is enabled in σ and disabled in $t_i(\sigma)$ then t_i must have removed j from the collective context—a contradiction.

Now to show the commutativity condition, suppose t_i and t_j are Barrier transitions of processes i and j respectively (all other transitions commute trivially). Further suppose $t_i, t_j \in \text{enabled}(\sigma)$ for some state σ . We can conclude that t_i and t_j are both the same type of Barrier transition (either init or wait). In this case we note that the set s is unordered and the change in the collective context state would be the same after executing in either order. Therefore $t_i(t_j(\sigma)) = t_j(t_i(\sigma))$. \square

One interesting consequence of Lemma 2 is that it is unnecessary to examine more than one execution interleaving order of processes entering and exiting barrier operations. This can greatly reduce the cost of analysis when barriers are heavily used to force lock-step execution.

DEFINITION 1. *Transitions t_1 is said to complete t_2 if $\text{Complete}(t_1, t_2)$ evaluates to true.*

LEMMA 3. *If t_i is an Issend, Irecv of process i and t_j is any transition of another process j where $i \neq j$ and $\neg \text{Complete}(t_i, t_j)$ then $I(t_i, t_j)$.*

PROOF. To show enabledness, suppose t_i is an Issend or Irecv of process i and t_j is any transition of process j where $t_i, t_j \in \text{enabled}(\sigma)$ for some $\sigma \in \Sigma$. Suppose to the contrary that $t_i \notin \text{enabled}(t_j(\sigma))$. Then t_j must have written $l_i(pc)$. We have assumed that this is not possible. Now suppose that $t_j \notin \text{enabled}(t_i(\sigma))$. Since t_i only writes the local and global stores of process i , it is not possible to disable any transition of another process and in particular t_j .

To show commutativity, since $\neg \text{Complete}(t_i, t_j)$, regardless of the transition type of t_j , under the well-formedness assumptions, t_i and t_j write and read disjoint memory locations. Therefore $t_i(t_j(\sigma)) = t_j(t_i(\sigma))$. \square

LEMMA 4. *If t_i is a Wait of process i and t_j is any transition of another process and $\neg \text{Complete}(t_i, t_j)$ then $I(t_i, t_j)$.*

PROOF. To show enabledness, suppose t_i is a Wait of process i and t_j is any transition of process j where $t_i, t_j \in \text{enabled}(\sigma)$ for some $\sigma \in \Sigma$. Suppose to the contrary that $t_i \notin \text{enabled}(t_j(\sigma))$. Then t_j must have modified some of the enabling conditions of the Wait. We know that t_j could only have written to the local or global store of process i if it were completing the message between processes j and i (implying t_j is a Wait or Test transition). However, we have assumed $\neg \text{Complete}(t_i, t_j)$, meaning that if t_j were a Wait or Test, the corresponding request would not Match the request of t_i and therefore would reference disjoint memory space under our well-formedness assumptions. Since no other transition could change the enabling condition of t_i we have $t_i \in \text{enabled}(t_j(\sigma))$. Now suppose $t_j \notin \text{enabled}(t_i(\sigma))$. Then t_i must have written $l_j(pc)$ or t_j was a Wait on a request of process i . Under our well-formedness assumptions, process i could not write to $l_j(pc)$. Similarly t_j only accesses the memory area referenced by t_i if $\text{Complete}(t_i, t_j)$, which we have assumed not to be the case.

To show commutativity, suppose t_i is a Wait of process i , and t_j is any transition of another process and $t_i, t_j \in \text{enabled}(\sigma)$ for some $\sigma \in \Sigma$. Since we have $\neg \text{Complete}(t_i, t_j)$, we know that t_j does not post or complete another request that could cause communication between process i and process j . $t_i \in \text{enabled}(\sigma)$ means we must consider two cases: (i) When the communication has already happened, and (ii) when the communication has not yet happened but the corresponding request necessary to complete this communication has been posted.

For case (i), the Wait operation updates only the local and global stores of process i . Under our assumptions any other operation would manipulate disjoint memory space. For case (ii), the Wait operation non-deterministically chooses between posted requests that *Match* the one being waited on that are not *Completed* in the current state. Upon choosing a request on process k to form a communication the Wait marks the local and remote requests, transfers data from sender to receiver, and writes the local store of process i . If t_j is any transition except for a Wait or Test on the request of process k commutativity is trivial. Otherwise we have assumed $\neg\text{Complete}(t_i, t_j)$ so t_j could not be a Wait or Test on the request of process k . \square

LEMMA 5. *If t_i is a Test of process i and t_j is any transition of another process j where $i \neq j$ and $\neg\text{Complete}(t_i, t_j)$ then $I(t_i, t_j)$.*

PROOF. To show enabledness, it need only be the case that other processes are not allowed to write $l_i(pc)$ (which we have assumed to be the case). Now suppose $t_j \notin t_i(\sigma)$. Then t_i must have written $l_j(pc)$ or t_j was a Wait on a request of process i . Neither case is possible under our semantics and assumptions.

To show commutativity, the proof is essentially the same as that for Wait with a third case: (iii) When the communication has not yet happened and the corresponding request necessary to complete this communication has not been posted. Since we have $\neg\text{Complete}(t_i, t_j)$, we know that t_j does not post or complete another request that could cause communication between process i and process j . Therefore the Test operation will write only the local store l_i of process i . Thus making commutativity with all other transition types trivial. \square

The final theorem of this section combines all of the lemmas as a convenience for the proofs in the next section.

THEOREM 4. *For all transitions t_1, t_2 , and states $\sigma \in \Sigma$, $t_1, t_2 \in \text{enabled}(\sigma)$ and $\neg\text{Complete}(t_1, t_2) \Rightarrow I(t_1, t_2)$.*

PROOF. We conclude this theorem from Lemmas 1, 2, 3, 4, 5, the definition of *Complete* and *I*. \square

A.2 Proof of correctness of the reduction algorithm.

We begin with two definitions from [8].

DEFINITION 2. *The set of transitions T enabled in state q is persistent in state q if and only if all nonempty sequences of transitions*

$$q = q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} q_3 \cdots \xrightarrow{t_{n-1}} q_n \xrightarrow{t_n} q_{n+1}$$

from q in A_G and including only transitions $t_i \notin T, 1 \leq i \leq n$, t_n is independent in q_n with all transitions in T .

DEFINITION 3. *A set T_s of transitions is a conditional stubborn set in state s if T_s contains at least one enabled transition, and if for all transitions $t \in T_s$, the following condition holds: for all sequences $q = q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} q_3 \cdots \xrightarrow{t_{n-1}} q_n \xrightarrow{t_n} q_{n+1}$, of transitions such that t and t_n are dependent in q_n , at least one of the t_1, \dots, t_n is also in T_s .*

The proof will show that for all states visited by the algorithm that if the set of enabled transitions is non-empty, then the set of transitions explored by the algorithm at that state is a conditional stubborn set [23].

LEMMA 6. *When backing off of a state created by a communication operation of process p , p is added to the interleave set of the pre-state of the nearest dependent transition in the search stack.*

PROOF. When *Complete* evaluates to true we do not know whether the two transitions compared are Independent. Lines 21–25 of Figure 10 clearly show the algorithm performing a search through s for the nearest transition such that *Complete* evaluates to true. When such a state v is found, p is added to $\text{interleave}(v)$, thus making all enabled transitions of p at v available through $\text{nextinterleaved}(v)$. By executing p at v , the algorithm will again search through s to find the nearest transition such that *Complete* evaluates to true. This will continue until no such transition is found. We have thereby over-approximated the dependence relation, guaranteeing that p will not be scheduled in states in s only when we can show that the two transitions are in fact Independent.

\square

THEOREM 5. *At line 26 the set of transitions T explored by the algorithm of Figure 10 is persistent in $\text{top}(s)$.*

PROOF. Let $\text{top}(s)$ be denoted q . Also, let the set of transitions executed from q be denoted T .

There are two cases: $T = \emptyset$, and $T \neq \emptyset$. If $T = \emptyset$ then $\text{enabled}(q) = \emptyset$ on lines 2 or 14. When $T = \text{enabled}(q)$ the set is trivially persistent in q .

It is shown in [8] that a conditional stubborn set [23] in some state q is persistent in q . We will show that when $T \neq \emptyset$, at line 27, T is a conditional stubborn set and therefore persistent in q .

Let w be the least sequence such that $s.w \in A_G$, for all $1 \leq i < n$, the transitions $t_i \in w$ are not in T , and $\text{last}(w) = t_n$ is dependent with some transition $t \in T$. All transitions t_i for $1 \leq i < n$ are independent with the transitions of processes in $\text{interleave}(q)$ therefore they are moves of different processes. Now consider two cases: $\text{proc}(t_n) = \text{proc}(t)$ and $\text{proc}(t_n) \neq \text{proc}(t)$. Since all transitions in w are of processes different from $\text{proc}(t)$ then some transition in w must have written $l_{\text{proc}(t)}(pc)$ (a contradiction) or $t = t_n$. For the other case, $\text{proc}(t_n) \neq \text{proc}(t)$. Since *Complete*(t_n, t) we can

conclude that both t_n and t are an Issend, Irecv, Wait, or Test operation. If t_n is an Issend, Irecv, or Test, t_n will eventually be executed through some maximal execution of the algorithm from σ as they cannot be disabled. Therefore by Lemma 6 we have that $\text{proc}(t_n) \in \text{interleave}(q)$ and $t_n \in T$, a contradiction. If t_n is a Wait, t_n could become disabled by the execution of t where $\text{Complete}(t, t_n)$. We conclude t is a Wait or Test transition associated with an Issend—the execution of which precludes the Wait or Test associated with the matching Irecv from choosing non-deterministically between available requests. However, the algorithm will schedule the process with the enabled Wait or Test (let this transition be t') on the Irecv at q because it completes the Wait or Test for the Issend—thus trying the completing operations in either order. Now $t_n \in \text{enabled}(t'(q))$ which is explored by the algorithm and cannot become disabled. Therefore t_n will eventually be explored. By Lemma 6, when $\text{post}(t_n)$ is about to be popped from the search stack, $\text{Complete}(t', t_n)$ will evaluate to true, causing $\text{proc}(t_n)$ to be added to $\text{interleave}(q)$ and t_n to be executed. Therefore we have $t_n \in T$. \square

The MPI standard requires that all processes eventually call `MPI_Finalize`. A cycle in the full state space will prohibit processes from reaching `MPI_Finalize` and therefore is an error. The algorithm shown in Figure 10 checks for cycles while performing the model checking. It is therefore desirable to show that the algorithm actually detects the presence of cycles.

THEOREM 6. *The algorithm of Figure 10 discovers a cycle in the reduced state space if and only if there is a cycle in A_G .*

PROOF. In the \Rightarrow direction, the proof is trivial since all transitions explored by the algorithm are in A_G . For the \Leftarrow direction, suppose there exists some cycle in A_G that is not explored by the algorithm. Let w represent the least sequence in A_G that contains the unexplored cycle be as follows:

$$w = q_0 \xrightarrow{t_0} q_1 \xrightarrow{t_1} q_2 \cdots \xrightarrow{t_n} q_n$$

where there exists some $0 \leq i < n$ such that $q_i = q_n$ for some $q_i \in w$. Now let j be the maximal transition index such that the algorithm explores the prefix of w , $t_1 \cdots t_j$ (the algorithm explores at least the initial state q_0). Theorem 5 shows that in q_j the set of transitions that are explored by the algorithm are a persistent set in q_j . The next transition t_{j+1} may not be explored by the algorithm at q_j because t_{j+1} is enabled but independent of all other transitions explored from q_j . Therefore we have that t_{j+1} remains enabled in the next state and conclude that the algorithm eventually explores t_{j+1} . Let q' be the state generated by the algorithm upon executing t_{j+1} . Now there are two cases: either (i) q' closes the cycle, detected by lines 17–19 of the algorithm, or (ii) there are more transitions in w . This process can iterate until the final transition in w . At this point every transition in w has been explored by the algorithm and the final state is a revisit in the search stack as all of the state transitions necessary to create the cycle have been taken by the algorithm. \square