Model Checking Nonblocking MPI Programs

Stephen F. Siegel*

Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
siegel@cis.udel.edu
http://www.cis.udel.edu/~siegel

Abstract. This paper explores a way to apply model checking techniques to parallel programs that use the *nonblocking* primitives of the Message Passing Interface (MPI). The method has been implemented as an extension to the model checker Spin called Mpi-Spin. It has been applied to 17 examples from a widely-used textbook on MPI. Many correctness properties of these examples were verified and in two cases nontrivial faults were discovered.

1 Introduction

Parallelism has proved remarkably effective at providing the high level of performance demanded by scientific computing. But parallel programming is notoriously difficult and, as the complexity of scientific applications increases, computational scientists find themselves expending an inordinate amount of effort developing, testing, and debugging their programs. Concerns about this level of effort—and the correctness of the resulting programs—have led to growing interest in new verification and validation approaches for scientific computing [6].

Model checking is a formal verification method that is widely-used in many hardware and software domains and in theory could be applied to scientific software. Yet significant hurdles must be overcome before model checking can be practically applied in the scientific domain. Among these is the fact that model checkers operate on a model of a program, rather than on the program itself. Hence techniques must be developed to construct finite-state models of scientific programs.

This paper describes a way to create finite-state models of programs that employ the "nonblocking" communication primitives of the *Message Passing Interface* (MPI) [3, 4]. MPI is a large message-passing library with subtle and complex semantics and has become the *de facto* standard for high-performance parallel computing. The nonblocking primitives provide a precise way to specify how computation and communication can be carried out concurrently in an MPI program. For example, one may specify that a communication task is to begin at one point in an MPI process and that the process should block at a subsequent point until that task has completed; computational code can be inserted between these two points to achieve the desired overlap. An algorithm expressed

* Supported by the U.S. National Science Foundation grant CCF-0541035.

B. Cook and A. Podelski (Eds.): VMCAI 2007, LNCS 4349, pp. 44–58, 2007.

in this way can be mapped efficiently to hardware architectures, common in high-performance computing, that utilize distinct, concurrently-executing components for communication and computation. Because of this, nonblocking communication is ubiquitous in MPI-based scientific software and is generally credited with playing a large role in the high level of performance that scientific computing has achieved.

While previous work applying model checking techniques to MPI programs has focused on various aspects of MPI, including the basic blocking point-to-point and collective functions [9, 8, 7, 11, 10], "one-sided" operations [5] and process management [2], none has dealt with nonblocking communication. There are two reasons that might explain this. First, the semantics of nonblocking communication are considerably more complex than those of blocking communication. The nonblocking semantics involve the introduction of types, constants, and a number of functions for creating and manipulating objects of those types, as well as complex rules prescribing how the MPI infrastructure is to carry out requests concurrently with program execution. Second, it is not obvious how to represent the state of a nonblocking MPI program in a way that is amenable to standard model checking techniques. MPI blocking communication operations map naturally to primitives provided by a model checker such as SPIN [1]: SPIN *channels* can be used to represent queues of buffered messages *en route* from one MPI process to another and the send and receive channel operations correspond closely to the blocking MPI send and receive functions. No SPIN data structure corresponds to an MPI nonblocking communication request nor supports the myriad operations upon it.

We proceed with a brief summary of the MPI nonblocking primitives (Sec. 2). This is followed by a detailed description of our approach for modeling nonblocking MPI programs for verification by standard explicit-state model checking techniques (Sec. 3). Discussion of a preliminary validation of the approach follows (Sec. 4): it has been implemented as an extension to SPIN called MPI-SPIN and has been applied to the 17 examples in the popular MPI textbook [12] dealing with nonblocking communication. Many correctness properties of these examples were verified and, in two cases, nontrivial faults were discovered.

2 Nonblocking Communication

The standard mode *blocking* function used to send a message from one MPI process to another is `MPI_Send`. Its arguments specify a communicator object that represents the communication universe in which the processes live, the *rank* of the destination process (an integer process ID relative to the communicator), the number and type of elements to send, their location in memory (the *send buffer*), and an integer tag. It blocks until the message has been completely copied out of the send buffer—either into a system buffer or directly into the receive buffer at the destination process. In particular, the MPI infrastructure *may* block the sender until the destination process is ready to receive the message synchronously.

The *nonblocking* version of this function is MPI_Isend. It takes the same arguments as MPI_Send but in addition it allocates and returns a handle r to a *request object*. This function initiates the sending of the message and does not block. A subsequent call to MPI_Wait on r blocks until the message has been completely copied out of the send buffer and then deallocates the request object. In particular, MPI_Send is equivalent to MPI_Isend followed immediately by MPI_Wait.

The receive operations MPI_Recv and MPI_Irecv work in an analogous way. In particular, MPI_Irecv initiates the receiving of a message and the subsequent call to MPI_Wait blocks until the incoming message has been completely copied into the receive buffer, from either a system buffer or directly from the send buffer. The receive request will only be paired with a message whose destination, tag, and communicator fields match the source, tag, and communicator fields of the receive, respectively. Unlike sends, the source and tag arguments for the receiving functions can take the *wildcard* values MPI_ANY_SOURCE and MPI_ANY_TAG, specifying that the receive will accept a message from any source, and/or with any tag, respectively.

MPI makes certain guarantees concerning how receives and messages are paired (or "matched") [3, Sec. 3.5]. Fix two processes p and q. A receive r posted from q cannot be paired with a message emanating from p if there is an earlier-posted unpaired message from p to q that matches r. Similarly, a message s emanating from p cannot be paired with a receive posted from q if there is an earlier-posted unpaired receive from q that matches s.

These strictly negative guarantees are complemented by the following positive ones. If s is an unpaired send request posted by p and r is an unpaired receive request posted by q, and r and s match, then (1) s will complete unless r is paired with another message and completes, and (2) r will complete unless s is paired with another receive request and completes. In particular, at least one of r, s will complete.

The function MPI_Test can be invoked on r to determine whether r has completed without blocking; it sets a boolean flag to 0 if r has not completed, else it sets this flag to 1 and proceeds as MPI_Wait.

MPI_Request_free can be invoked on r to indicate that the request object should be deallocated as soon as the request completes (in which case no subsequent call to MPI_Wait is necessary).

A number of MPI functions operate on arrays (r_i) of request handles. The function MPI_Waitany takes such an array and blocks until at least one request has completed. It then chooses one of the completed requests, returns its index i, and proceeds as if MPI_Wait were invoked on r_i. MPI_Waitall blocks until all requests in the array have completed and then proceeds as if MPI_Wait were invoked on all r_i. MPI_Waitsome blocks until at least one has completed and then invokes MPI_Wait on all that have completed and returns the subset of indices of all completed requests. The functions MPI_Testany, MPI_Testall, and MPI_Testsome work in an entirely analogous way but never block.

The function MPI_Probe takes source, tag, and communicator arguments and blocks until it determines there is an incoming message that matches these parameters. However, it does not consume the message (as a receive operation would) but simply returns certain information about the message which can then be used in a subsequent receive operation. MPI_Iprobe is similar but returns a flag instead of blocking. MPI guarantees that if a send request is posted with parameters matching those passed to MPI_Probe, then MPI_Probe will eventually return, though there can be a delay between the posting and the return of the probe. Similarly, repeated calls to MPI_Iprobe must eventually return true if a matching send is posted.

MPI_Cancel is invoked on r to attempt to cancel the request. The cancellation may or may not succeed. If it does succeed then any receive buffer involved in the canceled communication should remain unchanged; if it does not then execution should proceed as if MPI_Cancel were never called. A subsequent call to MPI_Test_canceled on the status object of r is used to determine whether or not the cancellation succeeded.

Persistent requests are created by calling MPI_Send_init or MPI_Recv_init. The arguments are similar to those for MPI_Isend and MPI_Irecv but, unlike ordinary requests, a persistent request r is *inactive* until *started* by invoking MPI_Start on r. After invoking MPI_Wait (or one of the other completion operations) on r, the request object is not deallocated but is returned to the inactive state until it is re-started. A persistent request is deallocated by invoking MPI_Request_free. MPI_Startall starts all persistent requests in an array.

An example of the use of nonblocking communication is given in the MPI/C code of Fig. 1, which is extracted from [12, Ex. 2.18]. In this program, multiple producers repeatedly send messages to a single consumer. The consumer posts receive requests for each producer in order of increasing rank, and then waits on each request in a cyclic order. After a receive request completes, the message is consumed and another receive request is posted for that producer. Note that overlap between computation and communication is achieved because the consumer may consume a message from a producer while the MPI infrastructure carries out the requests to receive data from other producers.

3 Modeling Approach

We now describe our notion of a model of an MPI program that consists of a fixed number of processes and uses the functions described in Sec. 2. For this work, we make a few simplifying assumptions: the only communicator is MPI_COMM_WORLD, each process is single-threaded, there is no aliasing of request handles, and no non-zero error codes are returned by the MPI functions. In future work we expect to eliminate each of these assumptions.

Our model consists of a particular kind of guarded transition system for each process and a global array of *communication records* representing buffered messages and outstanding requests. The execution semantics are defined so that, at any global state, either an enabled transition from one process or a transition corresponding to an action by the MPI infrastructure may be selected for execution.

```
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
if (rank != size-1) { /* producer code */
 while (1) {
  /* produce data */
   MPI_Send(buffer->data, buffer->datasize, MPI_CHAR, size-1 tag, comm);
 }
} else { /* consumer code */
 for (i=0; i < size-1; i++)
  MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
   &(buffer[i].req));
 for (i=0; ; i=(i+1)%(size-1)) {
  MPI_Wait(&(buffer[i].req), &status);
  /* consume data */
  MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
   &(buffer[i].req));
 }
}
```

Fig. 1. Code excerpt from [12, Ex. 2.18], *Multiple producer, single consumer*

We now sketch how this can be made precise. We first fix values for the following parameters: the number $n \geq 1$ of MPI processes, an upper bound $b \geq 0$ on the total number of buffered messages that may exist at any one time, and an upper bound $r \geq 0$ on the total number of outstanding requests that may exist at any one time. We consider it an error if the outstanding request bound can be exceeded. On the other hand, if a send is posted after the buffer bound has been reached, execution can proceed but the MPI infrastructure will not be allowed to buffer messages. The difference in how our model treats these two bounds stems from the different roles these concepts play in MPI. The MPI Standard states that each request object consumes some system resources and so there must be some limit on the number of outstanding requests. (The precise limit is implementation-dependent but is expected to be reasonably high.) Furthermore, a function that allocates a new request, such as `MPI_Isend`, will not block if this limit has been reached—instead, an error occurs. On the other hand, a correct MPI implementation should never report an error if it has insufficient space to buffer messages; at worst, the send operations will not complete until they can be paired with matching receives or sufficient buffer space becomes available.

We begin with the definition of *communication record*, then describe the transition system for a single process, and finally define the global model and execution semantics.

3.1 Communication Records

A *communication record* is an 11-tuple

$$(\mathsf{core}, \mathsf{source}, \mathsf{dest}, \mathsf{datatype}, \mathsf{count}, \mathsf{tag}, \mathsf{data}, \mathsf{handle}, \mathsf{status}, \mathsf{freeable}, \mathsf{match}).$$

For each of these components (or *fields*) we give a description and a default value. The symbol '$-$' will denote the appropriate default value wherever it appears. We let C denote the set of all communication records. The *null* element of C is the one for which all fields have their default values; it is also the default value for C.

The field core (or *core state*), captures the most essential information about the object: whether the record is for a request or message, a send or receive request, whether it has been canceled, completed, or matched, and so on. The core state is completely specified by the values of 9 boolean flags that answer the questions given in Fig. 2a. With a few exceptions, these are self-explanatory. A request is *active* if it is either (1) a nonpersistent request that has not been canceled or completed, or (2) a persistent request that has been started and has not been canceled or completed since last being started. A send request or message is *visible* if it can be detected by a probe on the receiver side.

At first glance it appears there could be as many as 2^9 distinct core states. But it is clear that many of the combinations are not possible, and in fact a simple reachability analysis reveals that only a small number (24, including a special *null* value) can occur. This analysis, carried out with SPIN, considers all ways in which a communication record can be created, modified, and destroyed by the 13 types of primitive state transformations described in this paper (Fig. 4). The 24 reachable core states are enumerated in Fig. 2b and the transitions between them are depicted in Fig. 3. The default value is s_0.

The integer fields source, dest, count, and tag mean exactly what one would expect; the special wildcard values may be used for the source and tag fields of receive requests. The default values are all 0.

The datatype field specifies the type of the elements comprising the message. We assume there is a fixed, finite set of datatypes numbered $0, 1, \ldots, d - 1$ and that for each i we are given size(i), the size (in bytes) of datatype i. In our implementation, there are several integer types of various sizes, an *empty* type of size 0, and a *symbolic* type (of size 4) used to model floating point values as symbolic expressions. There is no reason this could not be extended in many ways, including to incorporate MPI derived datatypes. The default value is 0.

For requests, the data field is an integer referring to the location of the start of the send or receive buffer. We will see below that the local memory of a process is modeled as a finite sequence of bytes; this integer refers to the index in that sequence. For messages, this integer instead encodes the sequence of bytes comprising the message. We assume there is a fixed procedure to losslessly encode any sequence of bytes into an integer, and decode the integer back into the byte sequence. The default is 0.

Our modeling approach requires that for each process, a unique integer ID be associated to each variable that will hold a request handle (i.e., each variable of type `MPI_Request` in MPI/C). It is assumed that there is at most one variable containing any given handle. (While aliasing of handles is allowed in MPI, this feature is rarely used. One could incorporate aliasing into our approach using

R: Is this a request?
B: Is this a buffered message?
P: Is this a persistent request?
S: Is this a send request?
A: Is this an active request?
C: Is this a request that has completed successfully?
V: Is this a visible (but unmatched) send request or buffered message?
M: Is this a matched (but incomplete) send request or buffered message?
X: Is this a request that has been successfully canceled?

(a) Core state flags

ID	name	R	B	P	S	A	C	V	M	X
s_0	*NullState*	·	·	·	·	·	·	·	·	·
s_1	*InvisibleSendReq*	✓	·	·	✓	✓	·	·	·	·
s_2	*VisibleSendReq*	✓	·	·	✓	✓	·	✓	·	·
s_3	*MatchedSendReq*	✓	·	·	✓	✓	·	·	✓	·
s_4	*CompleteSendReq*	✓	·	·	✓	✓	✓	·	·	·
s_5	*CanceledSendReq*	✓	·	·	✓	·	·	·	·	✓
s_6	*UnmatchedRecvReq*	✓	·	·	·	✓	·	·	·	·
s_7	*MatchedRecvReq*	✓	·	·	·	✓	·	·	✓	·
s_8	*CompleteRecvReq*	✓	·	·	·	✓	✓	·	·	·
s_9	*CanceledRecvReq*	✓	·	·	·	·	·	·	·	✓
s_{10}	*InactiveSendPreq*	✓	·	✓	✓	·	·	·	·	·
s_{11}	*InvisibleSendPreq*	✓	·	✓	✓	✓	·	·	·	·
s_{12}	*VisibleSendPreq*	✓	·	✓	✓	✓	·	✓	·	·
s_{13}	*MatchedSendPreq*	✓	·	✓	✓	✓	·	·	✓	·
s_{14}	*CompleteSendPreq*	✓	·	✓	✓	✓	✓	·	·	·
s_{15}	*CanceledSendPreq*	✓	·	✓	✓	·	·	·	·	✓
s_{16}	*InactiveRecvPreq*	✓	·	✓	·	·	·	·	·	·
s_{17}	*UnmatchedRecvPreq*	✓	·	✓	·	✓	·	·	·	·
s_{18}	*MatchedRecvPreq*	✓	·	✓	·	✓	·	·	✓	·
s_{19}	*CompleteRecvPreq*	✓	·	✓	·	✓	✓	·	·	·
s_{20}	*CanceledRecvPreq*	✓	·	✓	·	·	·	·	·	✓
s_{21}	*InvisibleMessage*	·	✓	·	·	·	·	·	·	·
s_{22}	*VisibleMessage*	·	✓	·	·	·	·	✓	·	·
s_{23}	*MatchedMessage*	·	✓	·	·	·	·	·	✓	·

(b) Reachable core states

	Prod$_0$	Prod$_1$	Cons	MPI	c_0	c_1	c_2	c_3	c_4
0					–	–	–	–	–
1			irecv$_0$		v_1	–	–	–	–
2		isend			v_2	v_1	–	–	–
3				reveal$_1$	v_3	v_1	–	–	–
4				upload$_1$	v_5	v_1	v_4	–	–
5	isend				v_6	v_5	v_1	v_4	–
6				reveal$_0$	v_7	v_5	v_1	v_4	–
7				match$_0$	v_5	v_4	v_8	v_9	–
8			irecv$_1$		v_5	v_{10}	v_4	v_8	v_9
9				match$_1$	v_4	v_8	v_9	v_{11}	v_{12}
10		wait			v_8	v_9	v_{11}	v_{12}	–
11		isend			v_2	v_8	v_9	v_{11}	v_{12}
12				synch$_0$	v_2	v_{11}	v_{12}	v_{13}	v_{14}
13			wait$_0$		v_2	v_{11}	v_{12}	v_{13}	–
14			irecv$_0$		v_2	v_1	v_{11}	v_{12}	v_{13}
15				download$_1$	v_2	v_1	v_{15}	v_{13}	–
16			wait$_1$		v_2	v_1	v_{13}	–	–
17	wait				v_2	v_1	–	–	–

(c) An execution prefix for program of Fig. 1

	core	source	dest	handle	match
v_0	s_0	–	–	–	–
v_1	s_6	0	2	0	–
v_2	s_1	1	2	0	–
v_3	s_2	1	2	0	–
v_4	s_4	1	2	0	–
v_5	s_{22}	1	2	–	–
v_6	s_1	0	2	0	–
v_7	s_2	0	2	0	–
v_8	s_3	0	2	0	0
v_9	s_7	0	2	0	–
v_{10}	s_6	1	2	1	–
v_{11}	s_{23}	1	2	–	1
v_{12}	s_7	1	2	1	–
v_{13}	s_4	0	2	0	–
v_{14}	s_8	0	2	0	–
v_{15}	s_8	1	2	1	–

(d) Communication record values used in prefix

Fig. 2. Communication records

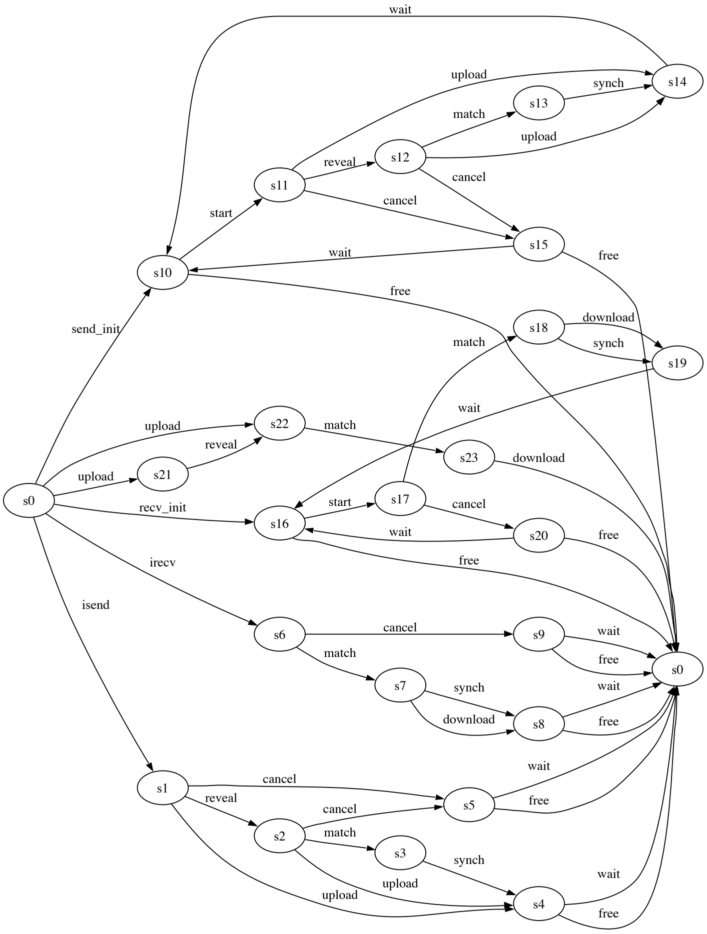

Fig. 3. Transitions between communication record core states

techniques similar to those for modeling references to heap-allocated data in Java or C, but we have chosen to defer this for future work and concentrate here on issues particular to nonblocking communication.) The integer handle field thus specifies the unique handle variable referring to that request. It is not used for messages. The default is 0.

The status field is used only for completed receive requests. It is a 4-tuple giving the source, tag, count and *status type* of the received message. (The source and tag information is redundant unless wildcards were used in the receive.) The status type can be either *undefined* (the default), *canceled* (the request was successfully canceled), *normal* (the message was successfully received), or *empty*. The last case is used in MPI to signify certain exceptional scenarios. In the default value, the status is *undefined*, and the source, tag, and count are all 0.

The boolean field freeable is 1 for a request that can be deallocated as soon as it completes (because of a user call to MPI_Request_free). Otherwise it has the default value 0. It is not used for messages.

A send request or message that has been matched with a receive request will have its integer match field set to the handle of the receive request. Since the rank of the receiver is the dest field, and we are assuming unique references to request objects, this uniquely determines the matching receive request. The match field is not used in receive requests, or in messages or send requests that have not been paired. The default is 0.

3.2 Local Process Model

A *local process model of rank R with global buffer bound b and global request bound r* is a tuple $L = (Q, q_0, T, h, l)$ where Q is a set of *local control states*, $q_0 \in Q$ is the *initial control state*, $T \subset Q \times E \times Q$ is a set of *local transitions* (the *event set E* is defined below), and h and l are nonnegative integers specifying, respectively, the number of request handle variables available to the process and the size, in bytes, of the local memory (excluding the request handle variables).

Let $X = \{0, \ldots, 255, \mathsf{UNDEF}\}$; these are the possible values for a unit of the local memory. Let $Y = \{0, \ldots, h-1, \mathsf{UNDEF}, \mathsf{NULL}\}$; these are the possible values for a request handle variable. The set $W = X^l \times Y^h$ represents all possible states of the process memory. A *local state of L* is an element of $Q \times W$. The *initial state of L* has control state q_0 and all local memory and request variables set to UNDEF.

The event set E consists of ordered pairs $\langle \gamma, \phi \rangle$, where $\gamma \colon W \times C^{b+r} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ is a *guard* specifying when the transition is enabled and $\phi \colon W \times C^{b+r} \rightarrow W \times C^{b+r}$ is a *transformation function* describing the change to the local state and communication record array effected by the transition. A transformation that modifies the communication record array is required to fall into one of the 8 categories of Fig. 4a. Each of these transformations is specified by certain parameters that are functions on W; these parameters represent the expressions that occur as arguments in the corresponding MPI function. For example, at a state with process memory w, the isend transformation modifies the communication record array by inserting the record

$$(s_1, R, \mathit{dest}(w), \mathit{dtype}(w), \mathit{count}(w), \mathit{tag}(w), \mathit{buf}(w), \mathit{req}(w), -, -, -).$$

The only change to the process memory W is to set the value of the request handle variable in position $\mathit{req}(w)$ to $\mathit{req}(w)$. A wait transformation on a completed or canceled nonpersistent request removes the record from the array, sets the value of the request handle variable to NULL, sets the status object at position $\mathit{status}(w)$ in local memory to the appropriate value, and so on.

Each MPI function described in Sec. 2 can be modeled using suitable choices of guards and primitive transformations. For example, an MPI_Isend at control state q is modeled with two outgoing transitions t_1 and t_2. The first leads to an “error” trap state, indicating that the outstanding request bound has been

transformation	corresponding MPI function
isend(*buf*, *count*, *dtype*, *dest*, *tag*, *req*)	`MPI_Isend`
irecv(*buf*, *count*, *dtype*, *source*, *tag*, *req*)	`MPI_Irecv`
wait(*req*, *status*)	`MPI_Wait`
cancel(*req*)	`MPI_Cancel`
send_init(*buf*, *count*, *dtype*, *dest*, *tag*, *req*)	`MPI_Send_init`
recv_init(*buf*, *count*, *dtype*, *source*, *tag*, *req*)	`MPI_Recv_init`
free(*req*)	`MPI_Request_free`
start(*req*)	`MPI_Start`

(a) Primitive state transformations effected by an MPI process

transformation	effect summary
match(i, j)	match send request/message with receive request
upload(i)	copy data from send to system buffer
download(i)	copy data from system to receive buffer
synch(i)	copy data from send to receive buffer
reveal(i)	make invisible send request/message visible

(b) Primitive state transformations effected by the MPI infrastructure

Fig. 4. The 13 primitive MPI state transformations

violated, and has guard γ_1, which holds iff the communication record array contains r requests. Transition t_2 leads to the state for the next point of control, has guard $\neg\gamma_1$, and a transformation of the isend type described above.

The more complex MPI functions can be translated using more states and some of the local memory. Say, for example, we wish to translate a call to `MPI_Waitany` on the array of request handles that starts with the k-th handle and has length m. To do this, we introduce an intermediate state q', and add transitions $t_1 = (q, \langle\gamma_1, \phi_1\rangle, q')$, $t_2 = (q', \langle\gamma_2, \phi_2\rangle, q')$, and $t_3 = (q', \langle\gamma_3, \phi_3\rangle, q'')$, where q'' is the state for the next point of control. The guard γ_1 holds iff there exists j such that $k \leq j < k+m$ and the communication record array contains a request from process R with handle j that has completed or been canceled. The transformation ϕ_1 sets some scratch variable i (residing in some part of the local memory reserved for this purpose) to the least such j. The guard γ_2 holds iff there exists j such that $i < j < k+m$ and the array contains a request from process R with handle j that has completed or been canceled. The transformation ϕ_2 sets i to the least such j. The guard γ_3 is true and ϕ_3 is a wait transformation on the request with handle i. The effect of all this is to wait until at least one request has completed or been canceled and then nondeterministically choose one of them and apply wait to it.

3.3 Global Model

Finally, a *model of a nonblocking MPI program with* n *processes, global buffer bound* b*, and global request bound* r is an n-tuple $M = (L_0, \ldots, L_{n-1})$, where for each i, L_i is a local process model of rank i with bounds b and r. Let W_i denote the set of all local states for L_i. A *global state* of M is an element

$$(w_0, \ldots, w_{n-1}, c_0, \ldots, c_{b+r-1}) \in W_0 \times \cdots \times W_{n-1} \times C^{b+r}.$$

The initial state is one for which each w_i is initial and all c_j are null. An execution of M is a sequence of global states, starting with the initial state, such that a *global transition* exists between each pair of consecutive states. A global transition corresponds to the execution of an enabled local transition or an *MPI infrastructure transition.*

The MPI infrastructure transitions correspond to the 5 transformations in Fig. 4b. Given a global state, one match transition is enabled for each pair (i, j) for which all of the following hold: (1) $0 \leq i, j < b + r$, (2) c_i is an unmatched receive request and c_j is an unmatched send request or buffered message, (3) the parameters of c_i and c_j "match" in the MPI sense, and (4) pairing c_i and c_j would not violate the ordering rules of the MPI Standard. The effect of the transition is to change the two entries in the communication record array to indicate the two records are matched. An upload transition models the completion of a send request by copying the message data from the send buffer into some system buffer. One such transition is enabled for each send request as long as the number of buffered messages is less than b. The effect is to complete the send request record and create a new record for a buffered message. A download transition models copying a message from a system buffer to the receive buffer; this results in changing the local state of the receiver appropriately, deleting the record for the message, and completing the receive request record. A synch transition corresponds to copying the message directly from the send to the receive buffer and completes both requests. A reveal transition makes an invisible send request or message visible; it is only enabled if all preceding send requests/messages emanating from the same sender and destined for the same receiver are already visible.

An execution prefix for the example of Fig. 1 is described in Figs. 2c and 2d. In each row (other than 0) of Fig. 2c there is a transition from either one of the three processes or the MPI infrastructure. This is followed by a description of the state of the communication record array after executing the transition. The v_i refer to entries in the table of Fig. 2d. This table contains one entry for each communication record value occurring in the prefix and gives the values for the 5 most essential fields of each. The subscripts on the transitions from the consumer and the MPI infrastructure refer to the rank of the sending process.

3.4 Order

We have seen that both process and infrastructural transitions may insert, delete, and modify entries in the communication record array, but we have not yet discussed the way in which the entries of this array are ordered. It is clear that the order must reflect some information concerning the temporal order in which the requests were generated, in order to prevent violations to the MPI matching rules. On the other hand, if we maintain this temporal ordering in its full precision, we risk creating unnecessary distinctions between states and an explosion in their number. The trick is to keep track of just as much "history" as is required to prevent violations of the MPI matching rules, and no more.

Our approach is to maintain the communication record array in such a way that the $b+r$ entries always occur in the following order: (1) the send requests and messages that need to be matched (i.e., those with core state s_1, s_2, s_{11}, s_{12}, s_{21}, or s_{22}), (2) the receive requests that need to be matched (s_6, s_{17}), (3) all other non-null records, and (4) all null records. These sections are further refined as follows. Within section 1, all records with source 0 occur first, followed by those with source 1, and so on. Within each of these subsections, those with destination 0 occur first, followed by those with destination 1, and so on. Within each of these subsubsections, the records occur according to the order in which the requests were posted. Within section 2, all records with destination 0 occur first, followed by those with destination 1, and so on. Within each of these subsections, the records occur according to the order in which the requests were posted. Notice that, for receives, the further division by source is not possible because of the possible use of `MPI_ANY_SOURCE`. Within section 3, the records are placed in any canonical order. (In our implementation, each communication record value is assigned a unique integer ID; the canonical order is that of increasing ID.)

Each primitive MPI transformation is engineered to preserve this order. For example, in line 4 of Fig. 2c, an `upload` transition applied to the send request v_3 that was in section 1, at position 0, causes the send request to be completed (v_4) and moved to section 3, in position 2. A new record for a buffered message (v_5) is inserted at the original position of the send request.

4 Validation

We have implemented the approach of Sec. 3 as an extension to Spin called Mpi-Spin. The core of the implementation is a C library for manipulating communication records. The library provides functions corresponding to the primitive MPI state transformations of Fig. 4. Because the memory required to store a single communication record is quite large, the library employs a "flyweight" pattern which (1) assigns a unique integer ID to each communication record value it encounters, and (2) stores a single copy of the record in a hash table. By using these IDs, the communication record array can be represented as an integer array in the Promela model. The library functions that operate on the array are incorporated into the Promela model using Spin's embedded C code facility. The user can access these functions through preprocessor macros defined in a header file. There is one macro for each of the MPI primitives discussed in this paper, and their syntax corresponds closely to the syntax for the C bindings of MPI, making it particularly easy to create models of C/MPI programs (Fig. 5). The MPI infrastructure events are incorporated into the model through an additional "daemon" process that, at each state, nondeterministically selects one of the enabled infrastructure events for execution.

```
active proctype consumer() {
  MPI_Request req[NPRODUCERS];
  byte i = 0;

  MPI_Init(Pconsumer, Pconsumer->_pid);
  do
  :: i < NPRODUCERS ->
     MPI_Irecv(Pconsumer, RECV_BUFF, COUNT, MPI_POINT,
               Pconsumer->i, TAG, &Pconsumer->req[Pconsumer->i]);
     i++
  :: else -> i = 0; break
  od;
  do
  :: MPI_Wait(Pconsumer, &Pconsumer->req[Pconsumer->i],
              MPI_STATUS_IGNORE);
     MPI_Irecv(Pconsumer, RECV_BUFF, COUNT, MPI_POINT, Pconsumer->i,
               TAG, &Pconsumer->req[Pconsumer->i]);
     i = (i + 1)%NPRODUCERS
  od;
  MPI_Finalize(Pconsumer)
}
```

Fig. 5. MPI-SPIN source for model of consumer process of Fig. 1

By default, MPI-SPIN checks a number of generic properties that one would expect to hold in any correct MPI program. These include (1) the program cannot deadlock, (2) there are never two outstanding requests with buffers that intersect nontrivially, (3) the total number of outstanding requests never exceeds the specified bound r, (4) when `MPI_Finalize` is called there are no request objects allocated for and there are no buffered messages destined for the calling process, and (5) the size of an incoming message is never greater than the size of the receive buffer. In addition, MPI-SPIN can check application-specific properties formulated as assertions or in linear temporal logic.

MPI-SPIN includes some primitives that do not correspond to anything in MPI, but are useful for modeling MPI programs. For example, there is a type `MPI_Symbolic` (together with a number of operations on that type) that can be used to represent floating-point expressions *symbolically.* Previous work [11] showed how symbolic techniques can be used to verify that a parallel program computes the same result as a trusted sequential version of the program *on any input.* Another primitive, `MPI_POINT`, represents an "empty" MPI datatype that can be used to abstract away data completely; this is particularly useful for constructing a model of the MPI communication skeleton of a program, as in Fig. 5.

We applied MPI-SPIN to Examples 2.17–2.33 of [12], attempting to verify generic and application-specific properties of each. (The source code for MPI-SPIN and all input and output for these experiments are available at `http://www.cis.udel.edu/ siegel/projects`.) The symbolic technique was applied to various configurations of the Jacobi iteration examples (2.17, 2.27, 2.32; the sequential version is Ex. 2.12). Ex. 2.17 is one of the cases for which Mpi-Spin discovered a fault. The problem occurs when the number of matrix columns is less than twice the number of processes. In this case, on at least one process two send requests will be posted using the same buffer: the single column stored on that process. For configurations outside of that range, equivalence with the sequential program was verified successfully. One of the larger configurations for Ex. 2.17 involved $N = 11$ matrix columns distributed over $n = 4$ processes, $k = 2$ loop iterations, $r = 16$, and $b = 0$; its verification resulted in searching 256,905 states and consumed 30 MB of RAM. The configuration with $N = 7$, $n = 3$, $k = 2$, $r = 12$, $b = 6$ required 65,849 states and 8 MB.

For each of the producer-consumer systems (2.18, 2.19, 2.26, 2.28, 2.33) the following were checked: (p_0) freedom from deadlock and standard assertions, (p_1) every message produced is eventually consumed, (p_2) no producer becomes permanently blocked, and (p_3) for a fixed producer, messages are consumed in the order produced. Again, various configurations were used in each case; one of the largest involved the verification of p_0 for Ex. 2.18, with $n = 8$, $r = 14$, and $b = 0$, which resulted in 1.8 million states and consumed 235 MB. Some of the properties were and some were not expected to hold on particular systems and, in general, the expected result was obtained for each property-system pair. An exception was Ex. 2.19. In this program, the second `for` loop in Fig. 1 is replaced with

```
i = 0;
while(1) {
  for (flag=0; !flag; i= (i+1)%(size-1)) {
    MPI_Test(&(buffer[i].req), &flag, &status);
  }
  /* consume data */
  MPI_Irecv(bufer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
            &buffer[i].req);
}
```

The idea is that the busy-wait loop allows the consumption of messages in whatever order they arrive, rather than enforcing a cyclic order. However, while checking p_0, Mpi-Spin discovered that `i` is erroneously incremented after the call to `MPI_Test` sets `flag` to `true` and before exiting the loop. This causes the consumer to consume from and repost to the wrong producer and can lead to a violation of the outstanding request bound (and other errors). After correcting this problem, the expected results were obtained.

These preliminary experiments were encouraging in several ways: (1) the tool was able to achieve a conclusive result on all of the examples to which it was applied, including some of nontrivial size, (2) the resources consumed were not excessive, at least by the standards of model checking, and (3) the tool discovered two nontrivial faults that had survived two editions of a widely-used text. However, these examples were admittedly small, and the true viability of the approach will only become apparent as we attempt to scale it to larger and more realistic scientific programs. This will be the focus of our future work.

References

1. G. J. Holzmann. *The* SPIN *Model Checker.* Addison-Wesley, Boston, 2004.
2. O. S. Matlin, E. Lusk, and W. McCune. SPINning parallel systems software. In D. Bosnacki and S. Leue, editors, *Model Checking of Software: 9th International SPIN Workshop, Grenoble, France, April 11–13, 2002, Proceedings*, volume 2318 of *LNCS*, pages 213–220. Springer-Verlag, 2002.
3. Message Passing Interface Forum. MPI: A Message-Passing Interface standard, version 1.1. `http://www.mpi-forum.org/docs/`, 1995.
4. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/`, 1997.
5. S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Formal verification of programs that use MPI one-sided communication. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, LNCS. Springer, 2006.
6. D. E. Post and L. G. Votta. Computational science demands a new paradigm. *Physics Today*, pages 35–41, Jan. 2005.
7. S. F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, January 17–19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 413–429, 2005.
8. S. F. Siegel and G. S. Avrunin. Modeling MPI programs for verification. Technical Report UM-CS-2004-75, Department of Computer Science, University of Massachusetts, 2004.
9. S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. In S. Graf and L. Mounier, editors, *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 286–303. Springer-Verlag, 2004.
10. S. F. Siegel and G. S. Avrunin. Modeling wildcard-free MPI programs for verification. In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming: PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA*, pages 95–106. ACM Press, 2005.
11. S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In L. Pollock and M. Pezzé, editors, *ISSTA 2006: Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis*, pages 157–168, Portland, ME, 2006.
12. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference, Volume 1: The MPI Core.* MIT Press, second edition, 1998.