

An Approach to Formalization and Analysis of Message Passing Libraries*

Robert Palmer, Michael DeLisi,
Ganesh Gopalakrishnan, and Robert M. Kirby

School of Computing, University of Utah
{`rpalmer,delisi,ganesh,kirby`}@`cs.utah.edu`

Abstract. Message passing using libraries implementing the Message Passing Interface (MPI) standard is the dominant communication mechanism in high performance computing (HPC) applications. Yet, the lack of an implementation independent formal semantics for MPI is a huge void that must be filled, especially given the fact that MPI will be implemented on novel hardware platforms in the near future. To help reason about programs that use MPI for communication, we have developed a formal TLA+ semantic definition of the point to point communication operations to augment the existing standard. The proposed semantics includes 42 MPI functions, including all 35 point to point operations, many of which have not been formally modeled previously. We also present a framework to extract models from SPMD-style C programs, so that designers may understand the semantics of MPI by exercising short, yet pithy, communication scenarios written in C/MPI. In this paper, we describe (i) the TLA+ MPI model features, such as handling the explicit memory for each process to facilitate the modeling of C pointers, and some of the widely used MPI operations, (ii) the model extraction framework and the simplifications made to the model that help facilitate explicit-state model checking of formal semantic definitions, (iii) a customized model checker for MPI that performs much faster model checking, and features a dynamic partial-order reduction algorithm whose correctness is directly based on the formal semantics, and (iv) an error trail replay facility in the Visual Studio environment. Our effort has helped identify a few omissions in the MPI reference standard document. These benefits suggest that a formal semantic definition and exploration approach as described here must accompany every future effort in creating parallel and distributed programming libraries.

1 Introduction

Progress in high-performance scientific computing (HPC) is fundamental to scientific discovery in virtually all walks of life. The Message Passing Interface (MPI, [1]) library has become a *de facto* standard in HPC, and is being actively developed and supported through several implementations [2,3,4,5] designed to

* Supported in part by NSF award CNS-00509379, Microsoft HPC Institutes Program, and SRC Contract 2005-TJ-1318.

run on a plethora of architectural platforms. MPI is, however, a portable standard for overall behavior, and not performance. Therefore, MPI programs are often manually or automatically (e.g., [6]) re-tuned when ported to another hardware platform, for example by changing its basic primitives (e.g., `MPI_Send`) to specialized versions (e.g., `MPI_Isend`). The fact that MPI-1 supports over 128 primitives and MPI-2 supports over 300 is largely to facilitate such transformations.¹ In this context, it is crucial that the designers performing code tuning are aware of the very fine details of the MPI semantics. Unfortunately, such details are far from obvious. For illustration, consider the following MPI pseudo-code involving two processes:

```
P0: if(rank==0){MPI_Irecv(rcvbuf1, from 1); MPI_Irecv(rcvbuf2,from 1);..}
P1: if(rank==1){sendbuf1=6; sendbuf2=7;
      MPI_Issend(sendbuf1, to 0); MPI_Isend(sendbuf2, to 0);..}
```

Process 1 is designed to issue two *immediate mode* sends (the first being a synchronous-mode send) to process 0, while Process 0 is designed to post two immediate-mode receives. Consider some simple questions pertaining to the execution of this program:

1. *Is it guaranteed that `rcvbuf1` will eventually contain the message sent out of `sendbuf1`?* The answer is ‘yes,’ since MPI guarantees in-order message delivery.
2. *When can the buffers be accessed?* Since all sends and receives use the immediate mode, the handles that these calls return have to be tested for completion using an explicit `MPI_Test` or `MPI_Wait` (suppressed for brevity in our pseudo-code) before the associated buffers are allowed to be accessed (written to or even read from).
3. *Will the first receive always complete before the second?* No such guarantee exists (the second may complete first), as these are *immediate mode* receives which are guaranteed only to be *initiated* in program order.
4. *What is guaranteed about the matching receive when the first send completes?* It is guaranteed that this receive has been *posted*. This is because the first send is a *synchronous* send, which forces a rendezvous with the posting of the first receive.

The MPI reference standard [1] is an informal, non machine-readable document that offers English descriptions of the individual behaviors of MPI primitives. It does not support answering the above kinds of simple questions in any tractable and reliable way. Running test programs, using actual MPI libraries, to reveal answers to the above kinds of questions is also futile, given that various MPI implementations exploit the liberties of the standard by specializing the semantics in various ways.

In this paper, we present a formal, high-level, and executable standard specification for a non-trivial subset of MPI 1.1. In particular, our specification consists

¹ It is widely known that MPI programs use only about a dozen or so of the 300 MPI library calls - but the precise dozen chosen depends on the applications being programmed, as well as the hardware platform on which the program runs.

of 42 MPI 1.1 functions. We write this specification in TLA+ [7], a formal specification notation widely used in industry. Our specification is integrated with the verification framework described in this paper. The features of this framework are as follows:

1. It permits designers to explore the MPI semantics in the setting of MPI programs written in C by extracting a TLA+ model of the program, embedding the MPI calls, and linking it to our TLA+ models of MPI functions. The exploration happens through *model checking* [8], and not through concrete executions. Error traces produced by the model checker are, however, displayed in a user-friendly way by driving the Microsoft Visual Studio debugger to walk the original program code following the error trace.
2. The framework includes two model checkers: MPI-TLC, a model checker that works directly off the formal semantic definitions using the TLA+ model checker, TLC [9]; and MPIC [10], a model checker that embodies the communication semantics of MPI directly as C# program code.
3. The communication semantics of a small representative subset of MPI were incorporated into MPIC by faithfully following our TLA+ definitions. In addition, MPIC implements a *dynamic partial-order reduction algorithm* (DPOR) (adapted from [11]) for efficient state-space traversal. The DPOR algorithm avoids commuting independent actions, where the notion of independence was stated and manually proved using a simplified version of our MPI formal semantics.

Experimental results from MPIC are provided in Figure 11. A more detailed coverage of MPIC or our DPOR algorithm are outside the scope of this paper, but may be found in [10].²

The questions raised on Page 165 can be answered by writing an MPI program such as the one in Figure 1 and analyzing this program using our framework. The four questions can be answered, in order, as follows:

1. Assert that the data read by process 0 is: `rcvbuf1 == 6 && rcvbuf2 == 7`. If it is possible under the semantics for other values to be assigned to these two variables, then the TLC model checker will find the violation.
2. Move the assertions mentioned in the response to the previous question to any other point before the corresponding `wait`s. The model checker then finds violations—meaning that the data cannot be accessed on the receiver until after the `wait`. If one adds an assignment to the variable being transmitted, i.e., after the `MPI_Isend` yet before the `MPI_Wait`, the model checker discovers the violation as the wrong value will be passed to the receiver.
3. We can reverse the order of the `MPI_Wait` commands. If the model checker does not find a deadlock then it is possible for the operations to complete in either order.
4. To answer this question, we employ the program in Figure 1. The MPI semantics for immediate mode ready send requires the corresponding receive to

² The entire modeling framework described in this paper may be downloaded from http://www.cs.utah.edu/formal_verification/verification_environment

```

1 #include "mpi.h"
2
3 int main(int argc, char** argv)
4 {
5     int rank, size, data1, data2, data3, flag;
6     MPI_Request req1, req2, req3;
7     MPI_Status stat;
8     MPI_Init(&argc, &argv);
9     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10    MPI_Comm_size(MPI_COMM_WORLD, &size);
11    if(rank == 0){
12        data1 = 0;
13        data2 = 0;
14        MPI_Irecv(&data1, 1, MPI_INT, 1,
15                0, MPI_COMM_WORLD, &req1);
16        MPI_Irecv(&data2, 1, MPI_INT, 1,
17                1, MPI_COMM_WORLD, &req2);
18        MPI_Irecv(&data3, 1, MPI_INT, 1,
19                2, MPI_COMM_WORLD, &req3);
20    } else {
21        data1 = 7;
22        data2 = 6;
23        MPI_Issend(&data1, 1, MPI_INT, 0,
24                 1, MPI_COMM_WORLD, &req1);
25    }
26    if(rank == 1){
27        MPI_Wait(&req1, &stat);
28        MPI_Irsend(&data2, 1, MPI_INT, 0,
29                 0, MPI_COMM_WORLD, &req2);
30        MPI_Irsend(&data3, 1, MPI_INT, 0,
31                 2, MPI_COMM_WORLD, &req3);
32    } else {
33        MPI_Wait(&req2, &stat);
34    }
35    if(rank == 0){
36        MPI_Wait(&req1, &stat);
37    } else {
38        MPI_Wait(&req2, &stat);
39    }
40    MPI_Finalize();
41    return 0;
42 }

```

Fig. 1. The C program used to answer Question 4 on Page 165

be posted before the `MPI_Irsend`. We cause the tag of the messages to force the second `MPI_Irecv` to match the `MPI_Issend`. We execute the `MPI_Wait` corresponding to the `MPI_Issend` and then post two `MPI_Irsend` operations. Now we observe that the model checker (in performing a breadth first search) finds the first `MPI_Irsend` posts without error, but the second `MPI_Irsend` violates the semantics. Thus we conclude that when the `MPI_Wait` of process 1 returns, process 0 is guaranteed to have executed the second `MPI_Irecv`, but is not guaranteed to have executed any further.

1.1 Related Work

The idea of writing formal specifications of standards and building executable environments is a vast area. The IEEE Floating Point standard [12] was initially conceived as a standard that helped minimize the danger of non-portable floating point implementations, and now has incarnations in various higher order logic specifications (e.g., [13]), finding routine applications in *formal proofs* of modern microprocessor floating point hardware circuits. Formal specifications using TLA+ include Lamport's Win32 Threads API specification [14] and the RPC Memory Problem specified in TLA+ and formally verified in the Isabelle theorem prover by Lamport, Abadi, and Merz [15]. In [16], Jackson presents a lightweight object modeling notation called Alloy, which has tool support [17,18] in terms of formal analysis and testing based on Boolean satisfiability methods.

Each formal specification framework solves modeling and analysis issues specific to the object being described. In our case, we were initially not sure how to handle the daunting complexity of MPI nor how to handle its modeling, given that there has only been very limited effort in terms of formal characterization of MPI. The architecture of our framework that incorporates solutions that finally worked are described in Section 2.

In [19], Georgelin and Pierre specify some of the MPI functions in LOTOS [20]. In [21], Siegel and Avrunin describe a finite state model of a limited number of MPI point-to-point operations. This finite state model is embedded in [22]. In [23], the authors support a limited partial-order reduction method – one that handles wild-card communications in a restricted manner, as detailed in [10]. In [24], additional ‘non-blocking’ MPI primitives are modeled in Promela. Our own past efforts in this area are described in [25,26,27,28]. None of these efforts: (i) approach the number of MPI functions we handle, (ii) have the same style of high level specifications (TLA+ is much closer to mathematical logic than finite-state Promela or LOTOS models), (iii) have a model extraction framework starting from C/MPI programs, (iv) incorporate a dynamic partial-order reduction algorithm that handles the difficulties of wildcard communications more generally, and (v) have a practical way of displaying error traces in the user’s C code. Section 3 describes the architecture of our implementation.

In the act of writing our formal specification, we noticed serious omissions in the English standard (confirmed by experts [29]). While these omissions were found largely by luck, the *opposite* problem – namely, that of our specification itself not correctly implementing the intent of the MPI English standard writers – needs much more care to avoid. We have taken some precautions to avoid such errors. First, our specification is organized for *easy traceability*: many clauses in our specification are cross-linked with [1] to particular page/line numbers of [1]. Second, the “formal semantic calculator” provided by our approach using familiar programming and debugging environments (*e.g.*, TLC, Phoenix, and Visual Studio) may help engage expert MPI users (who may not be formal methods experts) into experimenting with our semantic definitions.

More work is needed to exploit the full potential of formal semantic definitions, as well as a framework such as ours. One can state and prove theorems that link concepts spread across multiple pages, as is the case with the current reference document [1]. These, and other concluding remarks are provided in Section 4.

2 Communication Semantics Model of MPI

We have tried to make this section intuitive even for those not familiar with MPI: they may focus on the higher level points that we have expressed, as these issues are bound to arise in any such endeavor as this.

The TLA+ model of MPI is intended to capture the semantic details that are both explicitly and tacitly referenced in the natural language standard, while abstracting away the implementation specific issues that are not specified. Our model broadly implements the architecture shown in Figure 2. We preserve the MPI API such that application of an MPI operation has the same external interface as an MPI procedure call in C. The main pieces of the model are point-to-point operations, collective operations, and constants.

Point-to-point and collective operations are coupled using a communicator. We model the communicator as a context and a group (MPI additionally has topologies and attributes, which we consider to be future work). The context

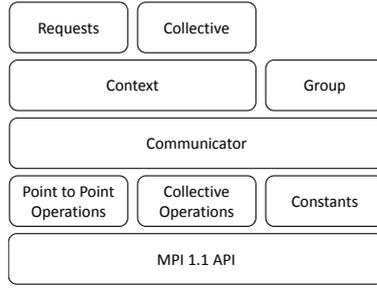


Fig. 2. TLA+ MPI model architecture

| | | |
|-------------------|------------------|-------------------|
| MPI_Get_count | MPI_Request_free | MPI_Test_canceled |
| MPI_Buffer_attach | MPI_Waitany | MPI_Send_init |
| MPI_Buffer_detach | MPI_Testany | MPI_Bsend_init |
| MPI_Isend | MPI_Waitall | MPI_Ssend_init |
| MPI_Ibsend | MPI_Testall | MPI_Rsend_init |
| MPI_Issend | MPI_Waitsome | MPI_Recv_init |
| MPI_Irsend | MPI_Testsome | MPI_Start |
| MPI_Irecv | MPI_Iprobe | MPI_Startall |
| MPI_Wait | MPI_Probe | |
| MPI_Test | MPI_Cancel | |

Fig. 3. Point-to-point operations included in the TLA+ specification

houses all information about messages that are currently available for communication. Groups define the set of processes allowed to access a communicator and their respective ranks (used for message addressing).

2.1 Modeling Approach

The MPI standard [1] contains some 128 operations that provide a rich collection of communication options. A full 35 of these operations are dedicated to pair-wise exchanges of messages between processes. Our model contains those operations that we could represent using exactly one TLA+ atomic transition (primed variables equated to unprimed variables, as in Figures 6 and 7). The operations included are shown in Figure 3. We model the remaining seven operations as sequential compositions of those shown in Figure 3. Thus MPI_Ssend becomes MPI_Isend and MPI_Wait issued in that order. Similarly, MPI_Sendrecv becomes MPI_Isend, MPI_Irecv, and two MPI_Wait operations issued sequentially, and so on. The reason for this decision is that the additional overhead involved in modeling these operations *directly* would significantly complicate our model. For example, consider the additional information needed to model MPI_Ssend directly. For doing this, we would require, for each process, a map from the program counter (pc) to the next operation to be performed when MPI_Ssend is enabled. In this manner, we can determine when a corresponding

| | | |
|---------------|----------------|------------------|
| MPI_Barrier | MPI_Group_size | MPI_Group_rank |
| MPI_Comm_size | MPI_Comm_rank | MPI_Comm_compare |
| MPI_Init | MPI_Finalize | MPI_Initialized |
| MPI_Abort | | |

Fig. 4. Additional MPI operations modeled to enable tool-based reasoning on MPI based parallel programs

MPI_Recv could be executed by the receiving process, and then cause both processes to jointly execute their state transition steps. However, since there is no restriction on what type of receive could be matched with MPI_Ssend (it could be MPI_Recv, MPI_Irecv, MPI_Sendrecv, *etc.*), nor are there restrictions on the blocking nature of the receives (some block the receiving process while others do not), supporting each of the variants becomes quite laborious, in addition to resulting in unreadable model descriptions.

Additional supporting operations included in the model are shown in Figure 4. Each of the operations has the same parameters in the same order as the MPI standard, with two additions. First, there is no way for TLA+ operations to query the system to discover which process is executing, short of having a globally visible state element. Therefore, the PID of the process executing an MPI call is passed as a parameter, which appears after the parameters specified in the standard. We also have not determined a graceful way to provide return values of MPI function calls. The return address is, therefore, also provided as a parameter (although handling return values other than MPI_SUCCESS remains as future work).

2.2 What Is Not Modeled

It is important to point out that we have not modeled all of the semantics of MPI in our work. In addition to the restrictions stated in the previous section, we have not modeled the following items.

Data: Most data. Data, such as arrays of floating point values, objects, *etc.*, could be modeled using TLA+. It is, however, not necessary in most cases to retain the actual data values of a distributed computation to verify reactive properties of the participating nodes. Therefore we allow a placeholder for data in our formal model in such a way that it can be included when necessary. We currently *do* allow for the preservation of data values, if they are used in `assert` statements. Similarly, there are many data manipulation operations, and also operations to pack data. These are not currently modeled.

Operations on communicators and topologies: Operations on communicators and topologies are modeled to a limited extent to enable point-to-point communications on intracommunicators. We currently model the operations shown in Figure 4 in addition to the point-to-point operations of Chapter 3 of MPI 1.1 shown in Figure 3. Operations on communicators and topologies are planned to be modeled in the next version of our semantics.

Implementation details: To the greatest extent possible we have avoided asserting implementation-specific details in our formal semantics. One obvious ramification of this omission is that modeling return codes of MPI operations is completely eliminated (cf. [1, Page 11]).

Handling Implementation-dependent Buffer Availability: As far as the standard mode sends (*e.g.*, MPI_Send, MPI_Isend, MPI_Send_init) go, we require the system to either eventually buffer these requests or to not buffer them at all. The standard allows for an implementation to switch between these policies in a time-varying manner; we do not know how to attain such generality without complicating our semantics drastically.

2.3 Modeling Granularity to Preserve the Corner Cases

A formal model for a communications library must model at the right level of granularity in order to not mask corner cases. In order to achieve this objective, we introduced three additional rules that are allowed to interleave with the actions of an individual processes. These rules facilitate message pairing, message buffering, and message transmission.

Figure 5 shows the interleaved rule that transmits messages from one process to another. This rule is enabled when there exists process *i*, and request *j* on process *i* such that the request is started, is globally active, has not been canceled, has not been transmitted, and has already been paired with another request on some other process. It is necessary to pair and transmit messages separately because *there is*

```

1 Transmit ==
2   /\ \E i \in 0..(N-1) :
3     \E j \in 1..Len(requests[i]) :
4       LET m == requests[i][j] IN
5         /\ m.started
6         /\ m.globalactive
7         /\ \not m.canceled
8         /\ \not m.transmitted
9         /\ m.match /= <<>>
10        /\ requests' = [requests EXCEPT ![i] =
11                       [@ EXCEPT ![j] =
12                        [@ EXCEPT !.transmitted = TRUE]]]
13        /\ IF \not requests[m.match[1]][m.match[2]].transmitted
14           THEN
15             IF m.message.state = "recv"
16               THEN Memory' = [Memory EXCEPT ![i] = [@ EXCEPT ![m.message.addr] =
17                               Memory[m.match[1]][requests[m.match[1]][m.match[2]].message.addr]]]
18             ELSE Memory' = [Memory EXCEPT ![m.match[1]] =
19                               [@ EXCEPT ![requests[m.match[1]][m.match[2]].message.addr] =
20                               Memory[i][m.message.addr]]]
21           ELSE
22             UNCHANGED <<Memory>>
23
24        /\ IF m ctype = "bsend"
25           THEN
26             message_buffer' = [message_buffer EXCEPT ![i] = @ - 1]
27           ELSE
28             UNCHANGED << message_buffer >>
29        /\ UNCHANGED << group, communicator, bufsize, initialized, collective >>

```

Fig. 5. Message transmission

no requirement for ordering of message completion in the MPI standard [1]. Consider the case where two messages are sent from process 1 to process 2 where the first message is very large and the second message is very small. The MPI standard requires that the first message sent be matched with the first receive posted in program order on both processes. However this makes no statement about *when* the messages will complete. In our example, it should be possible for the smaller message to complete first. The use of a separate transmit rule allows us to facilitate the modeling of MPI_Cancel which is used to cancel pending MPI messages. Further discussions are provided in Section 2.5.

Continuing with Figure 5, the final three conjuncts in the model of MPI_Wait define the values of `Memory`, `requests`, and the `message_buffer` in the next state. In MPI, the event marking the completion of the transmission on the sender side may become visible before the event on the receiver side, or vice versa. Therefore, in our model, only one request is updated to show that the transmitting step has completed. We do move some data between processes. We currently have abstracted the programs modeled such that the value in only *one memory location* can be transmitted between processes. We also have abstracted the notion of buffering such that a counting semaphore tracks the number of messages that can be buffered using the explicit space provided by the user — rather than modeling the number of bytes being sent per message.

2.4 A Complete Definition: MPI_Wait

Figures 6 and 7 contain the TLA+ model definition of MPI_Wait, commonly used to complete communications. As with all MPI operations (except for MPI_Initialized), MPI_Init must have been called prior to the application of this operation. The model checks this as an assertion on line 3 of the operation. The comments are of two types: *regular* and *cross references* into the natural language version of the standard. The cross references are numbered as “page.line” following the TLA+ comments ($\backslash*$), and allow our assertions to be traced. We now examine a few aspects of the specification of MPI_Wait. The main conjunct in the specification causes the `group`, `communicator`, `bufsize`, `message_buffer`, `initialized`, and `finalized` to remain unchanged in the next state. It then considers two cases: when the request is the special MPI_REQUEST_NULL value, or when it is a non-null request handle. For the non-null case, the operation becomes enabled when (i) the request is locally active — meaning it has not been previously completed by some wait or test, and (ii) the request indicates that the message has been transmitted, canceled, or buffered. In this case, if the source and destination referenced in the request are non-null, the memory of the executing process is updated to indicate that the message has completed by filling the fields of the status object (lines 16–22). Otherwise, the status fields are set to reflect that the completion has occurred on a request referencing MPI_PROC_NULL. In either case the request handle is appropriately set, and we also mark the status fields in memory.

The request sequence for the executing process must also be updated (lines 34–58. When a communication between processes i and j is initiated by i using

```

1 MPI_Wait(request, status, return, proc) ==
2 LET r == requests[proc][Memory[proc][request]] IN
3 /\ Assert(initialized[proc] = "initialized", \* 200.10-200.12
4      "Error: MPI_Wait called with proc not in initialized state.")
5      \* 41.32-41.39 The request handle is not the null handle.
6 /\ \/ /\ Memory[proc][request] /= MPI_REQUEST_NULL
7      /\ r.localactive
8      /\ \/ /\ r.message.src /= MPI_PROC_NULL \* The message src is not null
9      /\ r.message.dest /= MPI_PROC_NULL \* The message dest is not null
10      \* 41.32 - Blocks until complete
11      /\ \/ r.transmitted \* The message was transmitted or
12      \/ r.canceled \* canceled by the user program or
13      \/ r.buffered \* buffered by the system
14      /\ Memory' =
15          [Memory EXCEPT ![proc] = \* 41.36
16            [@ EXCEPT ![Status_Canceled(status)] =
17              /\ r.canceled
18              /\ \lnot r.transmitted, \* 54.46
19              ![Status_Count(status)] = r.message.numelements,
20              ![Status_Source(status)] = r.message.src,
21              ![Status_Tag(status)] = r.message.msgtag,
22              ![Status_Err(status)] = r.error,
23              ![request] = \* 41.32-41.35, 58.34-58.35
24              IF r.persist
25              THEN @
26              ELSE MPI_REQUEST_NULL]]
27      \/ /\ \/ r.message.src = MPI_PROC_NULL
28      \/ r.message.dest = MPI_PROC_NULL
29      /\ Memory' = [Memory EXCEPT ![proc] = \* 41.36
30        [@ EXCEPT ![Status_Canceled(status)] = r.canceled,
31          ![Status_Count(status)] = 0,
32          ![Status_Source(status)] = MPI_PROC_NULL,
33          ![Status_Tag(status)] = MPI_ANY_TAG,
34          ![Status_Err(status)] = 0,
35          ![request] = \* 41.32-41.35, 58.34-58.35
36          IF r.persist
37          THEN @
38          ELSE MPI_REQUEST_NULL]]

```

Fig. 6. The first half of the TLA+ model of MPI_Wait. The rest of MPI_Wait is shown in Figure 7.

a buffered send (such as MPI_Send) or when using MPI_Cancel, it is possible for the Wait to become enabled before the matching request is posted on process j . This is apparent when $r.match = \langle \langle \rangle \rangle$ on line 34. In the true case, the previously paired request is marked globally inactive, in addition to the local request being marked locally inactive and globally inactive. In the false case, only the local request is marked locally inactive. Again, the status fields are marked as required by the standard.

2.5 Issues Raised by Modeling

While creating the model we became aware of some specific issues that had not been discussed in the MPI natural language version of the standard. The following descriptions are helpful in understanding the issues identified. MPI_Probe takes a process rank j and some additional message envelope information, and becomes enabled when there is a matching request posted on process j . MPI_Cancel takes a request handle as an argument and attempts to cancel the corresponding

```

39  /\ requests' =
40  IF r.match /= << >>
41  THEN
42  [requests EXCEPT ![proc] = \* 58.34
43  [@ EXCEPT
44  ![Memory[proc][request]] =
45  IF r.persist
46  THEN
47  IF requests[r.match[1]][r.match[2]].localactive
48  THEN [@ EXCEPT !.localactive = FALSE,
49  !.globalactive = FALSE]
50  ELSE [@ EXCEPT !.localactive = FALSE]
51  ELSE
52  IF requests[r.match[1]][r.match[2]].localactive
53  THEN [@ EXCEPT !.localactive = FALSE,
54  !.globalactive = FALSE,
55  !.deallocated = TRUE]
56  ELSE [@ EXCEPT !.localactive = FALSE,
57  !.deallocated = TRUE]],
58  ![r.match[1]] =
59  [@ EXCEPT ![r.match[2]] =
60  IF requests[r.match[1]][r.match[2]].localactive
61  THEN requests[r.match[1]][r.match[2]]
62  ELSE [@ EXCEPT !.globalactive = FALSE]]]
63  ELSE
64  [requests EXCEPT ![proc] = \* 58.34
65  [@ EXCEPT ![Memory[proc][request]] =
66  IF r.persist
67  THEN [@ EXCEPT !.localactive = FALSE]
68  ELSE [@ EXCEPT !.localactive = FALSE,
69  !.deallocated = TRUE]]]
70  \/ /\ \/ Memory[proc][request] = MPI_REQUEST_NULL \* 41.40-41.41 The
71  \/ /\ Memory[proc][request] /= MPI_REQUEST_NULL \* request handle is
72  /\ \!not r.localactive \* null or the request is not active
73  /\ Memory' = [Memory EXCEPT ![proc] = \* 41.36
74  [@ EXCEPT ![Status_Canceled(status)] = FALSE,
75  ![Status_Count(status)] = 0,
76  ![Status_Source(status)] = MPI_ANY_SOURCE,
77  ![Status_Tag(status)] = MPI_ANY_TAG,
78  ![Status_Err(status)] = 0]
79  /\ UNCHANGED << requests >>
80  /\ UNCHANGED << group, communicator, bufsize, message_buffer,
81  initialized, collective >>

```

Fig. 7. The second half of the TLA+ model of MPIWait

communication. The standard says the message may still be complete, and it is up to the user to program appropriately. A third operation MPIRsend, and variants, requires the matching receive operation to have been previously posted, barring which the operation is in error. In this context, here are some specific issues we identified:

- There are numerous ways that MPIProbe and MPICancel can interact, resulting in an undefined system state. In particular, any time a message is probed successfully, it is not specified whether it is still possible for the message to be canceled or if the message must at that point be delivered.
- MPICancel also creates an undefined system state when used with ready mode send (MPIIrsend). Consider the following execution trace: “MPIIrecv; MPIIrsend; MPICancel; ...” If the ready send is successful, can the receive still be canceled?

- Continuing with Cancel, what happens if the null request is canceled?
- The MPI system allows the user to specify a buffer for outgoing messages. To ensure that all buffered messages have been sent, the user must call `MPI_Buffer_detach`. What is the state of the system when no buffer has been specified and `MPI_Buffer_detach` is called?

It is encouraging to note that even a few weeks invested in the process of writing a formal semantics forced us to conduct a thorough walk-through of the MPI standard, spotting the above omissions.

3 Modeling Framework

We have developed a modeling framework based on the Microsoft Phoenix [30] compiler which allows developers to insert a compilation phase between existing compiler phases in the process of lowering a program from language independent MSIL to device specific assembly. We place our phase at the point where the input program has been simplified into a single static assignment form, with a homogenized pointer referencing style, where the instructions are still device independent. Our phase reads the Phoenix intermediate representation and builds from it a state-transition system (the MPIC IR) for each function, similar in spirit to a control flow graph. Control locations in the program are represented by states, and program statements are represented using transitions.

The architecture of the verification framework is shown in Figure 8. From the MPIC IR, we can output different formats, including TLA+, Dot[31], and MPIC. The framework integrates both TLC and a new model checker MPIC to perform the verification tasks. If an error is found, the error trail is then made available to the verification environment, and can be used by our tool to drive the Visual Studio debugger to replay the trace to the error. The remainder of this section describes the simplification and replay capabilities of our framework. We report the MPIC tool primarily in [10].

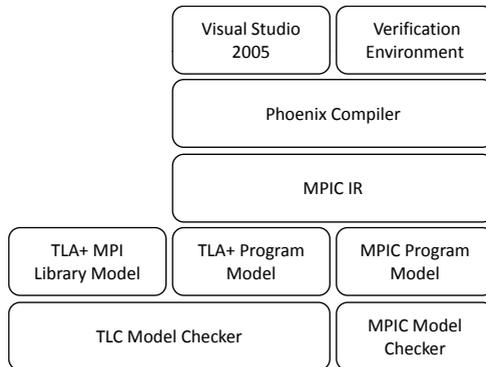


Fig. 8. System architecture

3.1 Simplification

From the extracted state-transition format, it would be possible to emit a TLA+ model directly. However, the TLA+ model would have to have sufficient mechanisms to handle function calls and returns. Although this is possible with TLA+ [32]—the scientific computing applications we have considered would not benefit from the additional functionality. As such we propose the following sequence of transformations, intended to reduce the complexity of model checking while preserving the properties of interest, before applying model checking based analysis. The simplifications are as follows:

- Inline all user defined functions: We assume (i) that all parameters are pass by value, (ii) that there are no function pointers, and (iii) there is no recursion.
- Remove operations foreign to the model checking framework: Examples include `printf`.
- Slice the model with respect to communications and user assertions: The cone of influence of variables is computed using a chaotic iteration over the program graph, similar to what is described in [33].
- Eliminate redundant counting loops: This is a heuristic to handle loops that occur frequently in MPI programs.

3.2 Program Modeling

Our model of MPI is intended to capture the semantics while abstracting away the possible implementation details. However there are some implementation details retained that are common to all present-day computer systems, and that are implied by the standard. The first of these is the notion of memory: it is assumed that each process operates in a disjoint memory space. As such, we allocate an array of TLA+ variables that represent the local store of each process. More formally, memory is modeled as a function $memory : \mathbb{N} \rightarrow \mathbb{N}$ where allocated addresses are mapped onto values. Variable names are represented by an array of address that use symbols, (*i.e.*, strings) for indices. These are again functions that map strings onto addresses. The mention of memory brings to the fore the first of several abstractions that are imposed on the model. Only values in \mathbb{N} are considered valid memory contents. With an explicit notion of memory and addresses, it is possible to have explicit pointers in the model. This we support, allowing for arbitrary dereferences. We also never allocate address 0, allowing for null pointer dereference violations to be discovered.

It is possible to allocate memory using the operator in Figure 9. This operator updates the function representing process memory by changing the function for process `i` such that there are `size` new memory locations at the end, each having uninitialized memory contents. The operator also writes the address of the first uninitialized location into the memory location of the pointer.

Many constants are used by MPI and consequently in our model. Since the model is automatically extracted from the program while it is being compiled, it is necessary that the constants used in our model match those used by the

```

1 AllocateMemory(ptr, pid, size) ==
2  /\ Memory' = [i \in 0..(N-1) |->
3     IF i = pid
4     THEN [j \in 1..(Len(Memory[i]) + size) |->
5        IF j <= Len(Memory[i])
6        THEN
7            IF ptr = j
8            THEN Len(Memory[pid]) + 1
9            ELSE Memory[i][j]
10         ELSE "uninitialized memory space"]
11     ELSE Memory[i]]

```

Fig. 9. Memory allocation in TLA+

```

\∇ /\ pc[pid] = state_pc
   /\ pc' = [pc EXCEPT ![pid] = next_pc]
   /\ guard
   /\ action
   /\ UNCHANGED << variables not mentioned in the action >>

```

Fig. 10. Transition template for TLA+ program model

implementation of MPI used with the program being analyzed. Constants are provided in a separate TLC configuration file. These constant definitions generally match the values used in the corresponding C header files (`mpi.h`). Since not all values can be used (*e.g.*, no floating point values, *etc.*) we make manual changes to the configuration and corresponding header files when necessary.

The individual transitions are formatted as shown in Figure 10, combined with the initial values of the memory array and the map from variable names to their addresses and written to disk. The constants, program model, and MPI model are then given to the TLC model checker.

Error Trail Generation. In the event that the model contains an error, an error trail is produced by the model checker and returned to the verification environment. To map the error trail back onto the actual program we observe the changes in the error trail to variable values that appear in the program text. For each such change, we step the Visual Studio debugger until the corresponding value of the variable in the debugger matches. We also observe which process moves at every step in the error trail and context switch between processes in the debugger at corresponding points. When the error trail ends, the debugger is within a few steps of the error with the process that causes the error scheduled.

4 Examples

We have applied our semantic evaluation framework to a small number of examples and show the results of a few verification tasks in this section. The two tables shown in Figure 11 shows the number of states generated / execution time for the following examples: (i) an example code from [34], (ii) the 2D

| | MPI-TLC | MPIC without DPOR | MPIC with DPOR |
|--------------|---------|-------------------|----------------|
| Trap | 724/2 | 331/0 | 66/0 |
| Diffusion 2D | timeout | timeout | 19,807,253/623 |
| Scenario 4 | 310/2 | N/A | N/A |

Fig. 11. Number of states generated / execution time (seconds)

diffusion example from [35], and (iii) the last scenario described in Section 1, namely “*What is guaranteed about the matching receive when the first send completes?*” Each of the experiments was run on a dual core 2GHz processor with 2GB of memory. When TLC was applied, two worker threads were used.

The Trap example from [34] computes the integral under a curve by application of the trapezoidal integration rule. The program is written in the SPMD style and is typical of “textbook examples” in this area. We verify the example as written for the absence of deadlocks and the default assertions provided by the respective model checkers for two model processes.

The Diffusion 2D example computes the diffusion of a substance through a two dimensional grid of cells. We could not verify the pseudo-code given in [35] because we require actual C program code. To facilitate this requirement, we implemented the program as described. We then optimized the code to overlap the preparation for communication with the actual communication operations. This is accomplished by changing the program to communicate via immediate mode synchronous sends and immediate mode receives (`MPI_Issend` and `MPI_Irecv`) coupled with `MPI_Wait` and then moving the message initiations as far from the completions as possible. We then were able to verify this code using MPIC using dynamic partial-order reduction, for the absence of deadlocks and the default set of assertions for 4 model processes.

The final example requires an additional MPI procedure, namely `MPI_Irsend`, which requires that the matching receive be posted before the “ready” mode send can be posted. We cause the first send to match the second receive using the tag field of the message. We then post the ready mode send immediately after the `MPI_Wait` corresponding to the `MPI_Issend`. We post a second ready mode send that can match only the third receive. Successful posting of the first `MPI_Irsend` implies that the receiver is guaranteed to be beyond that program point. Failed posting of the second `MPI_Irsend` implies that no guarantee can be made about further progress: thus the receiver is guaranteed to have posted the corresponding receive and no more (Figure 1). This verification task requires only two model processes.

5 Concluding Remarks

To help reason about programs that use MPI for communication, we have developed a formal TLA+ semantic definition of the point-to-point communication operations to augment the existing standard. We described this formal specification, as well as our framework to extract models from SPMD-style C programs.

We discuss how the framework incorporates high level formal specifications, and yet allows designers to experiment with these specifications, using model checking, in a familiar debugging environment. Our effort has helped identify a few omissions in the original MPI reference standard document. The experience gained so far suggests that a formal semantic definition and exploration approach as described here must accompany every future effort in creating parallel and distributed programming libraries.

Our future plans include overcoming the limitations of our current framework in terms of handling communication topologies. Another area where formal semantic definitions can help is in extensions of MPI to support different levels of threading. As pointed out in [36], even short MPI programs which employ threading can have nasty corner cases. Formal specifications, as well as direct execution methods for these specifications can have maximal impact in these areas, in that we will not be capacity limited in terms of model checking, and yet be able to shed light on the semantic intricacies, and pitfalls to avoid.

References

1. The Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1995), <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>
2. Gropp, W., Lusk, E.L., Doss, N.E., Skjellum, A.: A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing* 22(6), 789–828 (1996)
3. Microsoft: Microsoft windows compute cluster 2003 (2006), www.microsoft.com/windowsserver2003/ccs/faq.msp
4. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) *EuroPVM/MPI 2003*. LNCS, vol. 2840, pp. 379–387. Springer, Heidelberg (2003)
5. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary*, pp. 97–104 (2004)
6. Danalis, A., Kim, K.Y., Pollock, L., Swamy, M.: Transformations to parallel codes for communication-computation overlap. In: *SC 2005: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, p. 58. IEEE Computer Society, Los Alamitos (2005)
7. Lamport, L.: *Specifying concurrent systems with TLA* (1999)
8. Clarke, E., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (1999)
9. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Pierre, L., Kropf, T. (eds.) *CHARME 1999*. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)
10. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs. In: *PADTAD 2007* (2007), http://www.cs.utah.edu/formal_verification/verification_environment

11. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 110–121. ACM Press, New York (2005)
12. IEEE: IEEE Standard for Radix-independent Floating-point Arithmetic, ANSI/IEEE Std 854–1987 (1987)
13. Harrison, J.: Formal verification of square root algorithms. *Formal Methods in System Design* 22(2), 143–154 (2003); Guest Editors: Gopalakrishnan, G., Hunt, W., Jr.
14. Lamport, L.: The Win32 Threads API Specification (1996), <http://research.microsoft.com/users/lamport/tla/threads/threads.html>
15. Abadi, M., Lamport, L., Merz, S.: A tla solution to the rpc-memory specification problem. In: *Formal Systems Specification*, pp. 21–66 (1994)
16. Jackson, D.: Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies* 11(2), 256–290 (2002)
17. Jackson, D., Schechter, I., Shlyahter, H.: Alcoa: The alloy constraint analyzer. In: ICSE 2000: Proceedings of the 22nd international conference on Software engineering, pp. 730–733. ACM Press, New York (2000)
18. Jackson, D.: Alloy: A new technology for software modeling. In: Katoen, J.-P., Stevens, P. (eds.) *TACAS 2002*. LNCS, vol. 2280. Springer, Heidelberg (2002)
19. Georgelin, P., Pierre, L., Nguyen, T.: A formal specification of the MPI primitives and communication mechanisms. Technical report, LIM (1999)
20. Eijk, P.V., Diaz, M. (eds.): *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA (1989)
21. Siegel, S.F., Avrunin, G.: Analysis of mpi programs. Technical Report UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst (2003)
22. Holzmann, G.: The model checker SPIN. *IEEE Transactions on Software Engineering* 23(5), 279–295 (1997)
23. Siegel, S.F., Avrunin, G.S.: Modeling wildcard-free MPI programs for verification. In: *SIGPLAN Symposium, A.C.M.* (ed.) *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Chicago, pp. 95–106 (2005)
24. Siegel, S.F.: Model Checking Nonblocking MPI Programs. In: Cook, B., Podelski, A. (eds.) *VMCAI 2007*. LNCS, vol. 4349, pp. 44–58. Springer, Heidelberg (2007)
25. Barrus, S., Gopalakrishnan, G., Kirby, R.M., Palmer, R.: Verification of MPI programs using SPIN. Technical Report UUCS-04-008, The University of Utah (2004)
26. Palmer, R., Barrus, S., Yang, Y., Gopalakrishnan, G., Kirby, R.M.: Gauss: A framework for verifying scientific computing software. In: *SoftMC: Workshop on Software Model Checking*. ENTCS, vol. 953 (2005)
27. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Thakur, R., Gropp, W.: Formal verification of programs that use MPI one-sided communication. In: Mohr, B., Träff, J.L., Worringer, J., Dongarra, J. (eds.) *PVM/MPI 2006*. LNCS, vol. 4192, pp. 30–39. Springer, Berlin, Heidelberg (2006)
28. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: The communication semantics of the message passing interface. Technical Report UUCS-06-012, The University of Utah (2006)
29. Gropp, W.D.: Personal communication (2006)
30. Microsoft: Phoenix academic program (2007), <http://research.microsoft.com/phoenix>

31. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz – open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, p. 483. Springer, Heidelberg (2002)
32. Lamport, L.: A +CAL user’s manual (2006),
<http://research.microsoft.com/users/lamport/tla/p-manual.pdf>
33. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
34. Pacheco, P.S.: Parallel programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
35. Siegel, S.F., Avrunin, G.S.: Verification of mpi-based software for scientific computation. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 286–303. Springer, Heidelberg (2004)
36. Gropp, W., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringer, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)