The Communication Semantics of the Message Passing Interface

Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby

UUCS-06-012

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

31 October 2006

Abstract

The Message Passing Interface (MPI) standard is a natural language document that describes a software library for interprocess communication. Automatic reasoning about the reactive nature of programs communicating via MPI libraries is not possible without also analyzing the library being used. Many distributed programs that use MPI are relatively brief compared to the libraries that implement MPI. A formal specification of the communication semantics of the MPI standard (i) enables modular automatic reasoning of MPI based parallel programs independent of the library implementation, (ii) provides a mathematically precise declaration of the natural language intent of the MPI specification, (iii) enables mathematical reasoning about libraries that implement the standard, and (iv) allows for reasoning about the standard itself. We have created such a specification of the point to point operations and present it in this report. We also discuss some preliminary efforts to accomplish (i) above.
**Disclaimer:** While the semantics have been proof-read once, the actual semantics document is continually evolving. We are developing a tool–MPIC–that can be used to verify programs against this semantic specification. When the MPIC tool is released there will be a new version. The MPIC tool will also have an accompanying technical report.

Although every effort has been made to correctly model the intent of the MPI 1.1 specification, we make no claim regarding the correctness of the model contained herein. Please notify the authors if a discrepancy is found.
The Communication Semantics of the Message Passing Interface *

Robert Palmer       Ganesh Gopalakrishnan       Robert M. Kirby

31 October 2006

Abstract

The Message Passing Interface (MPI) standard is a natural language document that describes a software library for interprocess communication. Automatic reasoning about the reactive nature of programs communicating via MPI libraries is not possible without also analyzing the library being used. Many distributed programs that use MPI are relatively brief compared to the libraries that implement MPI. A formal specification of the communication semantics of the MPI standard (i) enables modular automatic reasoning of MPI based parallel programs independent of the library implementation, (ii) provides a mathematically precise declaration of the natural language intent of the MPI specification, (iii) enables mathematical reasoning about libraries that implement the standard, and (iv) allows for reasoning about the standard itself. We have created such a specification of the point to point operations and present it in this report. We also discuss some preliminary efforts to accomplish (i) above.

1 Introduction

Standards documents are one of the powerful tools for developing portable, reusable, and correct implementations of complex systems. In almost all cases, they are initially created as semi-formal documents, often containing gaping holes and potentially ambiguous statements. Over time, thanks to the experience gained from

*Supported in part by NSF award CNS-0509379 and a grant from Microsoft Corporation.
the widespread use of systems built according to the standard, they evolve to become much more rigorous and coherent. Yet, without a mathematical (formal) description, they still leave much room for misinterpretation – often unfortunately coinciding with the increased scale of design and deployment of systems built according to the standard.

The IEEE Floating Point standard [6] is a resounding success story in this area. It was initially conceived as a standard that helped minimize the danger of non-portable floating point implementations. As a fortunate side effect of the infamous Intel Pentium division bug, it now has incarnation in various higher order logic specifications (e.g., [5]), and routinely finds applications in formal proofs of modern microprocessor floating point hardware circuits. We strongly believe that the MPI communication standard – one of the most widely used in high performance computing – has the vast potential of being solidified in a similar fashion.

MPI is already a success story in the area of software library standardization, in that a collection of primitives that support message passing based communication for high performance computing has been widely adopted. Unfortunately, the MPI standard [10] uses natural language descriptions, as well as examples to communicate definitions, semantics, and other important details. Experience shows that this can lead to errors that result from unstated assumptions, ambiguities, and unclear causal dependencies. Some of the recent additions to MPI, such as one-sided communication constructs, are so tricky to understand that even simple algorithms using them have been shown to be incorrect (e.g., [9, 12]). These errors can be progressively eliminated by relying on formal (mathematical) descriptions of MPI, and employing modern formal verification techniques such as model checking [3].

1.1 Related Work

Significant inroads have been made in formalizing the MPI standard. The earliest work we are aware of is that of Georgelin et. al. [4] where the authors create a LOTOS description of MPI_BSEND, MPI_SSEND, MPI_RSEND, and MPI_RECV along with the collective operations MPI_BROADCAST, MPI_GATHER, and MPI_SCATTER. The MPI_ANY_SOURCE and MPI_ANY_TAG wild-cards are modeled. The above operations are modeled using the channel primitives of the LOTOS description language. They then apply the model to verification of some MPI programs.

More recent work by Siegel and Avrunin includes:
• In [13, 15] the authors create a mathematically precise model of MPI_SEND, MPI_RECV, MPI_SENDRECV, and MPI_BARRIER. The MPI_ANY_SOURCE and MPI_ANY_TAG wild-cards are expressly disallowed due to the additional non-determinism that are introduced with their use. The process universe is a fully connected graph where edges in the graph are a pair of FIFO channels. Synchronous communications are modeled as an interleaved execution of two processes where a send is followed immediately by the corresponding receive in an execution trace. A number of theorems and their proofs are presented with regards to synchronous communication in the proposed model.

• In [14] the authors model a simple MPI based 2D diffusion simulation and verify the model using both SPIN and INCA. Models of MPI_SEND, MPI_RECV, MPI_BARRIER, and MPI_SENDRECV are used in connection with the diffusion simulation. Some of the results from [13] are also presented.

• In [16] the authors verify the output of a distributed numerical program using a model checker and a sequential version of the same program.

Other work in modeling MPI related programs includes [12] where model checking is applied to a program using the one-sided locking routines of MPI 2.

We have also modeled MPI_SEND, MPI_RECV and MPI_BARRIER for use with both SPIN [2] and Zing [11].

1.2 Motivations

With several existing models of MPI one may naturally ask, why have another? To answer this question, consider the following points.

1. There are 35 operations related to point to point communication described in chapter 3 of the MPI 1.1 standard [10]. The most aggressive modeling effort we are aware of contains four (4) of these operations.

2. There is no tool based reasoning support for any mathematically rigorous model of MPI. The only rigorous model we are aware of is the one described in [13]. Models created in languages such as SPIN and Zing are
dependent upon the model checker implementation for a formal description of the language semantics.\textsuperscript{1}

3. Many of the errors that are seen in MPI programs are derived not in the use of the blocking sends, rather in the translation from the use of these simple send primitives to the more aggressive counterparts in the ongoing effort to optimize. None of the existing models have representations of these more aggressive operations.

4. A mathematically precise representation of a larger subset of the MPI operations is necessary to create an industrially useful tool for reasoning about programs that communicate using MPI libraries.

5. To serve as a specification, the MPI standard does not mandate implementation details beyond the function signatures and the existence of some symbols. No mention is made of how messages are to be transmitted from one process to another. To make a sufficiently complete model of any MPI program, these details must be filled in. Existing models make no distinction between what is specified in the standard and what is added to support tool based reasoning.

A mathematically precise specification the MPI standard can serve, not only to reason about programs that employ the MPI libraries, it can be used to reason about the various MPI library implementations and the standard itself.

1.3 A Formal Model

Our formal model of the MPI specification is expressed using the Temporal Logic of Actions [7, 1]. TLA is a formal logic containing standard ZF set theory, an action operator that induces a transition relation, and some limited temporal logic. It’s semantics are well understood by mathematicians and computer scientists, independent of any verification tool. Implementation details can be abstracted using set theoretic operations in combination with the action operator.

A program that uses MPI can be specified as a formula in TLA representing a distributed computation. The MPI operations are modeled as operators on variables in the formula. Comments accompany individual logical clauses referencing

\textsuperscript{1}While this may not necessarily be the case for LOTOS or INCA, any language developed as input for a verification tool that is undergoing active development could deviate from the previously published semantics.
The action operator of TLA induces a transition relation on logical formulas. Variables that are primed (i.e., $foo'$) indicate the value of that variable in the next state of the system. Every transition specifies the values of all variables in the next state. Transitions are total functions from valuations of variables to valuations of variables. New operators can be defined for any finite arity as a combination of the existing operators, user defined variables and constants, and parameters to the operator.

### 1.3.1 Transition Granularity

Using a TLA operator to represent an MPI operation implies that such operations will require exactly one transition to complete. Our model assumes that only one MPI operation will be applied in a given transition. Although 28 of the operations related to point to point communication are modeled in this way, we could see no way to model some of the point to point operations using only one transition. In particular, MPI\_SEND, MPI\_SSEND, MPI\_BSEND, MPI\_RSEND, MPI\_RECV, MPI\_SENDRECV, and MPI\_SENDRECV\_REPLACE could not be modeled in a single transition. The reason for this is quite simple: Each of these operations writes some variable and then waits for some other variable to be written by another process. The specification of which explicitly requires at least two (2) transitions. We will demonstrate in this paper how to model the remaining operations as a sequential composition of the provided operators.

To model operations requiring more than one transition we adopt the same convention described in [13] noting that the sequential composition of MPI\_ISEND and MPI\_WAIT on a single process is semantically equivalent to MPI\_SEND and that a transition sequence with a MPI\_SEND followed immediately by the corresponding MPI\_RECV can be considered synchronous. Although it is possible to apply sequential and operator composition in such a way that all 35 operations can be derived using a minimal subset of the MPI operations, to facilitate cross referencing the MPI standard, all 28 single transition operations in our model are modeled separately.

### 1.4 What is and is not modeled

Before diving into all the details, it is important to note that not everything in MPI 1.1 is present in the model. In particular, the following are either not present, are
limited in their current modeling, or are currently only placeholders:

- **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 of the distributed simulation to verify reactive properties of nodes participating in the distributed simulation. Therefore we allow a placeholder for data such that it can be included when necessary.

- **Data manipulation operations.** There are many operations specified by MPI to pack and manipulate data. These are not currently modeled, but could be if there were sufficient interest.

- **Operations on communicators and topologies.** These are modeled to a limited extent to enable point to point communications on intra-communicators. We currently model the operations shown in Figure 2 in addition to the point to point operations of chapter 3 of MPI 1.1 shown in Figure 1. Such operations on communicators and topologies should be a strait forward extension of this work.

- **Implementation details.** To the greatest extent possible we have avoided asserting implementation details that might constrain an implementation. One obvious ramification of this omission is that modeling return codes of MPI operations is completely eliminated (see Pg 11 of [10]).

- **Transient buffering of messages created by the standard mode send (MPI_SEND, MPI_ISEND, MPI_SEND_INIT).** We require the system to either eventually buffer these send requests or to never buffer them. It is not clear at the time of this writing how to model the situation where a buffer may be available for some but not all of the program execution.

Our model includes the point to point operations shown in figure 1 as single transition TLA operators. The argument order and meaning is as specified in the MPI standard for each operator, adding the pid of the process that is applying the operator as the last argument.

## 2 Conventions

In TLA, the whitespace in the document is significant. Sequences of logical conjuncts can become quite large and are therefore formatted as bulleted lists, the bullet being the logical and (∧) or the logical or (∨) operator.
| 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_Isend | 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 |

Figure 1: Point to point operations included in the TLA specification.

| 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 |

Figure 2: Additional MPI operations modeled to enable tool based reasoning on MPI based parallel programs.
Our modeling is influenced by the desire to model SPMD style programs in connection with the TLA MPI specification. As such, all program variables are assumed to be arrays of variables (one for each process in the computation).

When specifying the next state of a variable, it is necessary to completely specify that next state. As an example suppose variable \( rank \in [0..(N - 1) \rightarrow 0..(N - 1)] \) is the rank variable declared by a process. When a process calls MPI_COMM_RANK the rank variable would be passed into the function. Our model of MPI_COMM_RANK requires that the \( rank \) be passed to the operator, not \( rank[pid] \). As such, we assume that any parameter that might be written by an operator is an array \( a : 0..(N - 1) \rightarrow \alpha \) where only the \( i^{th} \) element (i.e., \( a[i] \)) is ever accessed by the applying process.

When using the action operator, the value of all variables in the next state must be specified. The specification of the MPI operations includes sometimes many UNCHANGED commands which are short hand for \( f' = f \). The MPI operators completely specify all of the MPI variables. In addition, those user variables that may be changed by application of the operator are also either updated or marked as UNCHANGED.

Comments of the form \( n.m \) indicate the corresponding page (\( n \)) and line (\( m \)) numbers that require the particular feature. All comments are enclosed in shaded regions.

### 3 Data Structures

This section presents the elements of the model that are introduced to mathematically specify the constructs of MPI. Appendix A contains the entire model. We will refer to it throughout the remainder of the presentation.

#### 3.1 Constants

Symbols that are defined in the MPI standard are modeled as constant values. We have included the subset of symbols that are necessary for the point to point communications on intra-communicators.

In addition to these symbols, we introduce four (4) additional constants. These constant values are useful to (i) make an instantiated program model finite, and (ii) to provide some information that is implicitly available to the MPI system. The additional constants are:

- N. The number of processes in the distributed computation.
• MAX_COMM. The maximum number of communicators.
• TYPES. The set of strings representing user specified types.
• TAGS. The set of integers representing user specified tags.
• SEND_IS_BUFFERED. A flag to indicate whether a send can be buffered by the MPI system.

3.2 Variables

Variables are functions. Functions need not have homogeneous domains or ranges. The elements of the domains or ranges need not be numbers (they could be other functions, or strings, or values). The variables in the model are group, communicator, requests, initialized, bufsize, message_buffer, and collective.

Functions therefore model data structures such as records, arrays, and sequences. Functions can represent a sequence in that elements can be modified, added, or deleted in the range or domain using the action operator. For example, a sequence \(\langle 3, 2, 1 \rangle\) can be modeled by the function

\[
s(x) = \begin{cases} 
3 & \text{if } x = 1, \\
2 & \text{if } x = 2, \\
1 & \text{if } x = 3, \text{ and} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

If we wish to append \(\langle 4, 5 \rangle\) to this sequence we would let \(\langle 3, 2, 1 \rangle \circ \langle 4, 5 \rangle = \langle 3, 2, 1, 4, 5 \rangle\) as

\[
s'(x) = \begin{cases} 
4 & \text{if } x = 4 \\
5 & \text{if } x = 5 \\
s(x) & \text{otherwise}
\end{cases}
\]

As a shorthand we write \(x = a..b\) for \(x = \{y \in \mathbb{N} : a \leq y \leq b\}\). If \(x\) is a set, TLA denotes \(\text{SUBSET}x\) to be the power-set or the set of all possible subsets of \(x\).

3.2.1 Groups and Communicators:

A group is a set of integers representing process IDs \(\text{members} \in \text{SUBSET}(0..(N-1))\) and the size of \(\text{members} \), \(\text{size} = |\text{members}|\). If foo is a Group then
\texttt{foo.members} is the set of pids in the group and \texttt{foo.size} is the number of elements in \texttt{foo.members} (i.e., \texttt{foo[\text{members}]} \in SUBSET(0..N-1) and \texttt{foo[size]} = |\texttt{foo(members)}|).

A ranking function and inverse ranking function are maps \(\text{ranking} : 0..(N - 1) \to 0..(N-1), \text{invranking} : 0..(N-1) \to 0..(N-1)\) such that \(\forall k \in \text{Dom(ranking)} : \exists n \in 0..(N-1) : \text{ranking}[k] = n \land \text{invranking}[n] = k \land \forall m \in \text{Dom(ranking)} : \text{ranking}[k] = \text{ranking}[m] \Rightarrow k = m\). A ranking and inverse ranking function are associated with each group.

A communication universe is a record containing a group handle \texttt{group} and a collective context handle \texttt{collective}. Groups and Communicators are referenced by handles on processes. Thus the mapping from handles to group or communicator records may be different on any process.

### 3.2.2 The collective context

Each communicator has a collective context associated with it. The collective context is not directly accessible to the user program, only through the handle in the associated communicator.

Our model currently includes \texttt{MPI\_BARRIER} as two transitions: \texttt{MPI\_BARRIER\_INIT} and \texttt{MPI\_BARRIER\_WAIT}. The collective context is a record having

\[
\begin{align*}
\text{participants} & \to x \in SUBSET(0..(N - 1)) \\
\text{root} & \to 0..(N - 1) \\
\text{type} & \to \{"barrier"\} \\
\text{state} & \to \{"in", "out", "vacant"\}
\end{align*}
\]

All processes in the communicator’s group must participate in the collective communication. Collective operations operate under a simple state machine. When no process is in the communication the state is “vacant” and the participants set is empty. As processes enter the operation their pid is added to the participants set and the first process changes the state from “vacant” to “in” and sets the type of the communication to “barrier”. Processes are only allowed to enter the communication when the state is “in”. \texttt{MPI\_BARRIER\_INIT} performs the addition of a process to the participant set when the state is “vacant” or “in” and the process is not represented in the set of participants; blocking the process applying this operator otherwise.

When all processes in the group are in the participant set then the state of the operation changes from “in” to “out” and processes are allowed to exit. \texttt{MPI\_BARRIER\_WAIT}
blocks the calling process until the state is “out”, removes the process applying
the operator from the participant set, and sets the state to “vacant” if the process
is the last to leave the communication.

Additional collective operations can be implemented by adding additional col-
lective message types to the range of collective.type and appropriate checks on
the parameters that are passed to the operators.

3.2.3 Requests

The set of requests represent the point to point contexts of all communicators.
Messages are paired only if they have the same communicator handle (which in
our model are unique across space and time).

A message is represented by the envelope that includes all information needed
to pair and transmit point to point communication operations. We model messages
as a record (i.e., a function having character strings as elements of the domain) as
follows:

\[
\begin{align*}
\text{data} & \rightarrow \text{Buffers} \\
\text{src} & \rightarrow 0..(N - 1) \cup \{\text{MPI\_ANY\_SOURCE}\} \\
\text{dest} & \rightarrow 0..(N - 1) \\
\text{msgtag} & \rightarrow \text{TAGS} \cup \{\text{MPI\_ANY\_TAG}\} \\
\text{dtype} & \rightarrow \text{TYPES} \cup \{\text{MPI\_TYPES}\} \\
\text{num} & \rightarrow N \\
\text{universe} & \rightarrow 0..(\text{MAX\_COMM} - 1) \\
\text{state} & \rightarrow \{“send”, “recv”\}
\end{align*}
\]

Where \text{Buffers} is a placeholder for future inclusion of data in a model.

A request is the bookkeeping information needed to manage messages within
a process. Request objects are required to be opaque to the user process and
are therefore represented by a function \text{requests} : N \rightarrow \text{Request} where the
set \text{Request} is the set of all possible request objects. The request handle is the
element of the domain of the \text{requests} function which returns the associated
request object.

With \text{Seq} (\mathbb{N}) as the set of all sequences of natural numbers, we model request
objects as records as follows:

- error → \( \mathbb{N} \)
- active → \{TRUE, FALSE\}
- transmitted → \{TRUE, FALSE\}
- buffered → \{TRUE, FALSE\}
- started → \{TRUE, FALSE\}
- canceled → \{TRUE, FALSE\}
- deallocated → \{TRUE, FALSE\}
- ctype → \{"send", "bsend", "ssend", "rsend", "recv"\}
- persist → \{TRUE, FALSE\}
- match → \( \text{Seq}(\mathbb{N}) \)
- message → Messages

A new request is appended to the requests function as described above. Each request record is accessible by the user process through its associated handle until that record is marked as deallocated either by successful application of a message completion operator such as MPI_WAIT or MPI_REQUEST_FREE. The handles are set to MPI_REQUEST_NULL at this time and become unaccessible to the user process.

### 3.2.4 Message buffers and buffer size

Users may wish to provide buffer space to the MPI system and allow the MPI system to manage that buffer space. Calls to MPI_BSEND, MPI_IBSEND, and MPI_BSEND_INIT use this buffer that is specified through MPI_BUFFER_ATTACH.

Not modeling data, the buffers are represented by a counting semaphore to track resource availability. Only one buffer can be attached to the MPI system for a process at a time. We approximate the use of the buffer space as follows. The user specifies how many messages can be stored in the buffer by the call to MPI_BUFFER_ATTACH. When a message is activated one buffer slot is consumed until the message is transmitted or canceled. Accordingly, MPI_BUFFER_DETACH blocks the process applying the operator until all buffered messages have either been transmitted or canceled.

\[
\begin{align*}
message_buffer : & \quad 0..(N - 1) \rightarrow \mathbb{N} \\
bufsize : & \quad 0..(N - 1) \rightarrow \mathbb{N}
\end{align*}
\]
The message_buffer variable is a function that represents the counting semaphore for each process. The bufsize variable is a function that represents the maximum values for each of the associated message_buffer variables.

### 3.3 Statuses

MPI operations return information to the user program in two ways. The first is the return value of a function. We do not model this. The second way information is returned to the user program is via the status object.

We model a status as a record with members as follows:

- **state** \(\rightarrow \{\text{"defined"}, \text{"undefined"}, \text{"empty"}\}\)
- **MPI\_SOURCE** \(\rightarrow 0..(N - 1) \cup \{\text{MPI\_PROC\_NULL}, \text{MPI\_ANY\_SOURCE}\}\)
- **MPI\_TAG** \(\rightarrow \text{TAGS} \cup \{\text{MPI\_ANY\_TAG}\}\)
- **MPI\_ERROR** \(\rightarrow \mathbb{N}\)
- **count** \(\rightarrow \mathbb{N}\)
- **canceled** \(\rightarrow \{\text{TRUE, FALSE}\}\)

### 4 Collective Communications

### 5 Closing the model for use with model checking

Many things are left and specified by MPI. Among these are details on how messages are communicated between processes. So far we’ve introduced the request, status, and communicator records. Using the temporal logic of actions we now have sufficient structure in our model to specify the pairing, buffering, transmitting, and completing of messages.

#### 5.1 Completing messages

##### 5.1.1 Envelope matching

Envelopes match according to the operator Match shown in appendix A.

##### 5.1.2 Pairing messages

Messages are paired together as a send request and a receive request. Program order must be observed on both the send and receive process when matching two
requests. To enforce this policy, the operator that performs message pairing specifies the earliest active message in the sequence that has not been canceled, transmitted, or paired previously. Operator Pair contains the logic of this operation. Pairs of messages that have been started, not matched, not canceled, not transmitted, and where one message is a send and the other message is a receive can be paired. In addition we require messages to have matching envelopes.

5.1.3 Transmitting messages

Once messages are paired appropriately they may complete in any order. Thus it is not enough to model communication as the pairing of messages. The Transmit operator contains the logic involved in passing data from one process to another. Only messages that have been started, have not been canceled, have not previously been transmitted, and have been previously paired can be transmitted. The request is updated to reflect that the corresponding message has been transmitted.

5.1.4 Buffering messages

Message buffering can happen under two circumstances. The first is when the user specifically requests MPI to buffer the outgoing messages using commands such as MPI_IBSEND. These messages may be buffered at any time after the message is started and before the message has transmitted. The operator Buffer_ibsend contains the logic to mark requests when messages have been buffered appropriately. Thereby allowing the sending process to continue when the corresponding message completion operator is applied.

When using MPI_SEND this system may choose to buffer the outgoing message. We allow this to happen at any time after the message is posted up until the message is transmitted or canceled. However it may also be the case that the MPI system will never buffer such a message. The operator Buffer_send performs this operation.

It is possible, from the user’s perspective, for the message to be buffered and transmitted before the user program regains control. For this reason we allow the message to be buffered up to the point where the message is actually transmitted or canceled by the user.
6 Modeling MPI Programs

There are many ways to model programs in TLA. The +CAL tool makes it significantly easier to take this step [8]. We will describe a similar modeling paradigm that suits our needs.

In modeling MPI programs in connection with the TLA MPI specification, we assume for simplicity that all programs are written in the SPMD style. Although this is not required, it is required that all variables be declared as arrays as described in section 2.

It is also convenient to assume that all programs make only MPI function calls, although adding procedure calls is a relatively trivial extension. Closing the environment and making available other standard system procedures is an important area of research but is beyond the scope of this work.

6.1 Sequential execution

Let PC be an array \([0..(N - 1) \rightarrow Labels]\) such that each process \(i \in 0..(N - 1)\) in the distributed computation has a program counter represented by \(PC[i]\). The transition relation of a sequential program can be specified as a disjunct of conjuncts where each conjunct has (i) a current PC guard, (ii) the specified next PC after executing the conjunct, and (iii) an action associated with the current PC that modifies the state – perhaps only the PC itself.

All control statements can be modeled using the explicit PC and an IF construct provided by TLA.

6.2 Multiple step MPI procedures

As mentioned before, when using a multi-step MPI operator these can be compiled into some sequence of single-step operators. We present possible solutions for the seven contained in MPI that are not present in our TLA model.

The MPI operations can be modeled using a sequence of transitions with \(proc\) being the pid of the process, \(\text{"in"}^2\) being the starting PC of the call to MPI_SEND, \(\text{"intermediate"}\) being the middle PC, and \(\text{"out"}\) being the return PC, and the variable \(\text{req} \in [0..(N - 1) \rightarrow Request]\). We also consider the status variable \(\text{stat} : [0..(N - 1) \rightarrow Status]\) as defined in the appendix.

\(^2\)Strings are valid PC values in TLA. Recall that the PC is a function whose domain is the set of pids and the range is in this case a string.
6.2.1 MPI\texttt{SEND}

Applying MPI\texttt{SEND} in a program having sequential execution can be implemented follows:

\begin{align*}
\forall &\land pc[\texttt{proc}] = \text{“in”} \\
\land &pc = [pc \text{ EXCEPT } ![\texttt{proc}] = \text{“intermediate”}] \\
\land &\texttt{MPI_Isend}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{dest}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\
\forall &\land pc[\texttt{proc}] = \text{“intermediate”} \\
\land &pc’ = [pc \text{ EXCEPT } ![\texttt{proc}] = \text{“out”}] \\
\land &\texttt{MPI_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc})
\end{align*}

6.2.2 MPI\texttt{BSEND}

Applying MPI\texttt{BSEND} is as follows:

\begin{align*}
\forall &\land pc[\texttt{proc}] = \text{“in”} \\
\land &pc = [pc \text{ EXCEPT } ![\texttt{proc}] = \text{“intermediate”}] \\
\land &\texttt{MPI_Ibsend}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{dest}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\
\forall &\land pc[\texttt{proc}] = \text{“intermediate”} \\
\land &pc’ = [pc \text{ EXCEPT } ![\texttt{proc}] = \text{“out”}] \\
\land &\texttt{MPI_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc})
\end{align*}

The restrictions on attaching buffers and managing the buffer space are identical.

6.2.3 MPI\texttt{SSEND}

Applying MPI\texttt{SSEND} is as follows:

\begin{align*}
\forall &\land pc[\texttt{proc}] = \text{“in”} \\
\land &pc = [pc \text{ EXCEPT } ![\texttt{proc}] = \text{“intermediate”}] \\
\land &\texttt{MPI_Isend}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{dest}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\
\forall &\land pc[\texttt{proc}] = \text{“intermediate”} \\
\land &pc’ = [pc \text{ EXCEPT } ![\texttt{proc}] = \text{“out”}] \\
\land &\texttt{MPI_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc})
\end{align*}
6.2.4 MPI RSEND

Applying MPI RSEND is as follows:

\[ \begin{align*}
\forall \ & \land \ pc[proc] = "in" \\
\land \ & \ pc = [pc \ EXCEPT ![proc] = "intermediate"] \\
\land \ & \ MPI\_Irsend(buf, count, datatype, dest, tag, com, req, proc) \\
\forall \ & \land \ pc[proc] = "intermediate" \\
\land \ & \ pc' = [pc \ EXCEPT ![proc] = "out"] \\
\land \ & \ MPI\_Wait(req, stat, proc)
\end{align*} \]

6.2.5 MPI RECV

Applying MPI RECV is as follows:

\[ \begin{align*}
\forall \ & \land \ pc[proc] = "in" \\
\land \ & \ pc = [pc \ EXCEPT ![proc] = "intermediate"] \\
\land \ & \ MPI\_Irecv(buf, count, datatype, source, tag, com, req, proc) \\
\forall \ & \land \ pc[proc] = "intermediate" \\
\land \ & \ pc' = [pc \ EXCEPT ![proc] = "out"] \\
\land \ & \ MPI\_Wait(req, stat, proc)
\end{align*} \]
6.2.6 MPI SENDRECV

Overloading \texttt{req} and \texttt{stat} to be arrays of records appropriately, MPI SENDRECV could be implemented as follows:

\begin{verbatim}
∨ ∧ pc[proc] = "in"
∧ pc' = [pc EXCEPT ![proc] = "intermediate,ecv"
 ∧ MPI_Isend(sendbuf, sendcount, sendtype, dest, sendtag, com, req1, proc)
∨ ∧ pc[proc] = "intermediate,ecv"
∧ pc' = [pc EXCEPT ![proc] = "wait"
 ∧ MPI_Irecv(recvbuf, recvcount, recvtype, source, recvtag, com, req2, proc)
∨ ∧ pc[proc] = "in"
∧ pc' = [pc EXCEPT ![proc] = "intermediate,send"
 ∧ MPI_Irecv(recvbuf, recvcount, recvtype, source, recvtag, com, req2, proc)
∨ ∧ pc[proc] = "intermediate,send"
∧ pc' = [pc EXCEPT ![proc] = "wait"
 ∧ MPI_Isend(sendbuf, sendcount, sendtype, dest, sendtag, com, req1, proc)
∨ ∧ pc = "wait"
∧ pc' = [pc EXCEPT ![proc] = "out"
 ∧ MPI_Waitall(2, [req EXCEPT ![proc] = [0 ↦ req1[proc], 1 ↦ req2[proc]]], stat, proc)
\end{verbatim}

6.2.7 MPI_SENDRECV_REPLACE

In addition to overloading \texttt{req} and \texttt{stat} to be arrays of records appropriately we add a temporary variable for receiving the results. MPI_SENDRECV_REPLACE
could be implemented as follows:

\[
\begin{align*}
\lor & \quad \text{pc}[\text{proc}] = \text{“in”} \\
& \quad \lor \quad \text{pc} = \text{pc EXCEPT } ![\text{proc}] = \text{“intermediate,ecv”} \\
& \quad \lor \quad \text{MPI,Isend(buf, sendcount, sendtype, dest, sendtag, com, req1, proc)} \\
\lor & \quad \text{pc}[\text{proc}] = \text{“intermediate,ecv”} \\
& \quad \lor \quad \text{pc} = \text{pc EXCEPT } ![\text{proc}] = \text{“wait”} \\
& \quad \lor \quad \text{MPI,irecv(tempbuf, recvcount, recvtype, source, recvtag, com, req2, proc)} \\
\lor & \quad \text{pc}[\text{proc}] = \text{“in”} \\
& \quad \lor \quad \text{pc} = \text{pc EXCEPT } ![\text{proc}] = \text{“intermediate,end”} \\
& \quad \lor \quad \text{MPI,irecv(tempbuf, recvcount, recvtype, source, recvtag, com, req2, proc)} \\
\lor & \quad \text{pc}[\text{proc}] = \text{“intermediate,end”} \\
& \quad \lor \quad \text{pc} = \text{pc EXCEPT } ![\text{proc}] = \text{“wait”} \\
& \quad \lor \quad \text{MPI,Isend(sendbuf, sendcount, sendtype, dest, sendtag, com, req1, proc)} \\
\lor & \quad \text{pc} = \text{“wait”} \\
& \quad \lor \quad \text{pc} = \text{pc EXCEPT } ![\text{proc}] = \text{“copy”} \\
& \quad \lor \quad \text{MPI,Waitall(2, [req EXCEPT ![proc] = [0 \mapsto \text{req1[proc]}, 1 \mapsto \text{req2[proc]}], stat, proc)} \\
\lor & \quad \text{pc} = \text{“copy”} \\
& \quad \lor \quad \text{pc} = \text{pc EXCEPT } ![\text{proc}] = \text{“out”} \\
& \quad \lor \quad \text{sendbuf} = [buf EXCEPT ![proc] = temp[proc]] \\
\end{align*}
\]

**6.3 An example**

An example program is included in Appendix A. This program exercises the immediate mode synchronous send, along with the immediate mode receive. Processes are conceptually placed in a ring. Even ranked processes send to the neighbor with higher rank (mod ring size), synchronize on the barrier, and then receive from the neighbor having lower rank (again mod ring size). Odd ranked processes receive from the neighbor with lower rank, synchronize on the barrier and then send to the neighbor with higher rank.

The program is represented as a disjunct of conjuncts similar in style to Section 6.2. This operator has one parameter which is the process id of the process that is currently executing—therein we model the SPMD style where every process executes the same program image.

The next state relation for the entire system is the initial state of the model \textit{Init} and henceforth () the \textit{Next} relation that performs either a \textit{Pair}, \textit{Transmit}, \textit{Buffer}, or \textit{Proc} move for some pid at any step.
7 Conclusions

The TLA model of MPI in connection with this paper describes the reactive behavior of all 35 point to point communication operations from chapter 3 of the MPI 1.1 standard.

We have closed the model for model checking single threaded programs that communicate via MPI point to point operations. We have provided the additional MPI operations necessary to initialize, determine the rank of a process, the size of a communicator’s group, and exit according to the MPI standard.

References


A The Full Specification
The formal MPI library specification.
Robert Palmer
The University of Utah
School of Computing

Some notes:
- Need to split the buffer rule into rules - one for user specified buffering and one for system provided buffering. – don’t really know how
- Need to add deallocation of requests to the model as in mpi_wait.
- Need to add more semantics.
- Need to cause buffers to be freed appropriately when a message is sent.
- Need to add a return code to indicate success or error and error handling.
- Need to fix the buffering of standard mode sends such that they might block forever.

EXTENDS Naturals, TLC, Sequences, FiniteSets

Constants are given values in the configuration file that accompanies this document: mpi_base.cfg

CONSTANTS

\( N \),
The number of processes in the computation.

\( MAX_{\text{COMM}} \),
The highest allowed handle value for a communicator. This is not in the standard but makes our model finite.

\( MAX_{\text{GROUP}} \),
The highest allowed handle value for a group.

\( TYPES \),
The set of user defined types.

\( TAGS \),
The set of user defined tags.

\( SEND_{\text{IS BUFFERED}} \),
A flag to indicate whether sends are to be buffered.

\( RANK_{\text{ORDERINGS SIGNIFICANT}} \),
a flag to indicate whether all possible ranking orders should be considered in verification

\( MPI_{\text{COMM WORLD}} \),
The handle for \( MPI_{\text{COMM WORLD}} \).

\( MPI_{\text{ANY SOURCE}} \),
The wildcard source rank.

\( MPI_{\text{ANY TAG}} \),
The wildcard tag value.

\( MPI_{\text{PROC NULL}} \),
Section 3.11 Null Processes

\( MPI_{\text{REQUEST NULL}} \),
A special handle value for requests.
Set this to 0 in the configuration file and make the initial values of the requests occupied to avoid an array out-of-bounds error.

**MPI_SUCCESS**, The return value of a successful call to an MPI procedure.

**MPI_IDENT**, 5.4: Two communicator handles refer to the same communicator.

**MPI_CONGRUENT**, The communicator handles are different; communicators differ only in context.

**MPI_SIMILAR**, The communicator handles are different; communicators have the same group, however both context and ranking differ.

**MPI_UNEQUAL**, The communicator handles are different; communicators have different groups, contexts, and rankings.

**MPI_UNDEFINED**, A special rank returned to a process that is not a member of the queried communicator.

**MPI_INT**, **MPI_FLOAT**, MPI defined datatype for integers, MPI defined datatype for floating point numbers

**UB** The upper bound on the tag range 19.27 – 19.31

**MPI_GROUP_EMPTY** The empty group

Variables represent the state of the MPI system at any given time. None of these state elements are specified by the standard. However they are useful to describe what is specified. In particular mention is made of handles that reference opaque objects. The communicator and requests arrays are such opaque objects that are referenced by integer handles that in our model are unique across both space and time (i.e., the same value is used for MPI_COMM_WORLD on all processes for the entire execution etc.).

**VARIABLES**

**communicator**, An array of communication universe objects.

**bufsize**, The size of the user attached `message_buffer`.

**message_buffer**, The user attached buffer.

**requests**, A array of message requests lists, one per process. Although we do model the allocation of request objects by adding a structure to a list of requests, we are not modeling the freeing of requests more than setting the associated handle to MPI_REQUEST_NULL.

**initialized**, An array of flags that indicate whether MPI_Init
has been called by a given process.

The collective contexts for all communicators

The array of groups

A model of memory for individual processes.

Type invariant

Memory is considered a program

\[
\text{mpi-vars} \triangleq \langle \text{group, communicator, bufsize, message_buffer, requests, initialized, collective} \rangle
\]

\[
\text{Messages} \triangleq [\begin{align*}
\text{src} & : (0 \ldots (N - 1)) \cup \{\text{MPI\_ANY\_SOURCE}\}, \\
\text{dest} & : (0 \ldots (N - 1)), \\
\text{msgtag} & : 0 \ldots \text{UB} \cup \{\text{MPI\_ANY\_TAG}\}, \\
\text{dtype} & : \text{TYPES} \cup \{\text{MPI\_FLOAT, MPI\_INT}\}, \\
\text{namelements} & : \text{Nat}, \\
\text{universe} & : (\text{MPI\_COMM\_WORLD} \ldots (\text{MPI\_COMM\_WORLD} + \text{MAX\_COMM})), \\
\text{state} & : \{\text{"send", "recv"}\}, \\
\text{addr} & : \text{Nat}\end{align*}]
\]

\[
\text{Message\_types} \triangleq \{\text{"send", "bsend", "ssend", "rsend", "recv"}\}
\]

\[
\text{Collective\_types} \triangleq \{\text{"barrier"}\}
\]

\[
\text{Collective\_states} \triangleq \{\text{"in", "out", "vacant"}\}
\]

\[
\text{Request} \triangleq [\begin{align*}
\text{error} & : \text{Nat}, \\
\text{active} & : \text{BOOLEAN}, \\
\text{transmitted} & : \text{BOOLEAN}, \\
\text{buffered} & : \text{BOOLEAN}, \\
\text{started} & : \text{BOOLEAN}, \\
\text{cancelled} & : \text{BOOLEAN}, \\
\text{deallocated} & : \text{BOOLEAN}, \\
\text{ctype} & : \text{Message\_types}, \\
\text{persist} & : \text{BOOLEAN}, \\
\text{match} & : \text{Seq}\{\text{Nat}\}, \\
\text{message} & : \text{Messages}\end{align*}]
\]

\[
\text{Requests} \triangleq [(0 \ldots (N - 1)) \rightarrow \text{Seq(\text{Request})}]
\]

\[
\text{Statuses} \triangleq [\begin{align*}
\text{state} & : \{\text{"defined", "undefined", "empty"}\}, \\
\text{MPI\_SOURCE} & : (0 \ldots (N - 1)) \cup \{\text{MPI\_PROC\_NULL, MPI\_ANY\_SOURCE}\}, \\
\text{MPI\_TAG} & : \text{TAGS} \cup \{\text{MPI\_ANY\_TAG}\}, \\
\text{MPI\_ERROR} & : \text{Nat}\end{align*}]
\]
Refactored into Memory to allow for a uniform treatment of the model of memory and to facilitate modelling using pointer arithmetic for member accesses.

Status variables are explicitly allocated by the user. Therefore they are present in the Memory of individual processes. We will use a simple offset mechanism to return the individual member addresses within Memory.

\[
\begin{align*}
\text{count} & \quad : \quad \text{Nat}, \\
\text{cancelled} & \quad : \quad \text{Boolean}
\end{align*}
\]

Status variables are explicitly allocated by the user. Therefore they are present in the Memory of individual processes. We will use a simple offset mechanism to return the individual member addresses within Memory.

\[
\begin{align*}
\text{Status\_Cancelled}(\text{base}) & \triangleq \text{base} \\
\text{Status\_Count}(\text{base}) & \triangleq \text{base} + 1 \\
\text{Status\_Source}(\text{base}) & \triangleq \text{base} + 2 \\
\text{Status\_Tag}(\text{base}) & \triangleq \text{base} + 3 \\
\text{Status\_Err}(\text{base}) & \triangleq \text{base} + 4
\end{align*}
\]

\[
\begin{align*}
\text{Initialized} & \triangleq [0 \ldots (N - 1) \rightarrow \{\text{"initialized", "uninitialized", "finalized"}\}]
\end{align*}
\]

\[
\begin{align*}
\text{MessageBuffers} & \triangleq [0 \ldots (N - 1) \rightarrow \text{Nat}] \\
\text{BufferSizes} & \triangleq [0 \ldots (N - 1) \rightarrow \text{Nat}]
\end{align*}
\]

Groups can be different on different processes.

\[
\begin{align*}
\text{Group} & \triangleq [0 \ldots (N - 1) \rightarrow [\text{MPI\_COMM\_WORLD} \ldots (\text{MPI\_COMM\_WORLD} + \text{MAX\_GROUP}) \rightarrow \\
& \quad \text{members} : \text{SUBSET} \,(0 \ldots (N - 1)), \\
& \quad \text{size} : 0 \ldots N, \\
& \quad \text{ranking} : [0 \ldots (N - 1) \rightarrow 0 \ldots (N - 1)], \\
& \quad \text{inranking} : [0 \ldots (N - 1) \rightarrow 0 \ldots (N - 1)]]]
\end{align*}
\]

\[
\begin{align*}
\text{Communicator} & \triangleq [0 \ldots (N - 1) \rightarrow [\text{MPI\_COMM\_WORLD} \ldots (\text{MPI\_COMM\_WORLD} + \text{MAX\_COMM}) \rightarrow \\
& \quad \text{group} : \text{MPI\_COMM\_WORLD} \ldots (\text{MPI\_COMM\_WORLD} + \text{MAX\_GROUP}), \\
& \quad \text{collective} : \text{MPI\_COMM\_WORLD} \ldots (\text{MPI\_COMM\_WORLD} + \text{MAX\_COMM})]]
\end{align*}
\]

\[
\begin{align*}
\text{Collective} & \triangleq [(\text{MPI\_COMM\_WORLD} \ldots (\text{MPI\_COMM\_WORLD} + \text{MAX\_COMM})) \rightarrow \\
& \quad \text{participants} : \text{SUBSET} \,(0 \ldots (N - 1)), \\
& \quad \text{root} : 0 \ldots (N - 1), \\
& \quad \text{type} : \text{Collective\_types}, \\
& \quad \text{state} : \text{Collective\_states}]
\end{align*}
\]

\[
\begin{align*}
\text{Comm\_inv} & \triangleq \text{communicator} \in \text{Communicator} \\
\text{Buff\_inv} & \triangleq \text{bufsize} \in \text{BufferSizes} \\
\text{Msg\_buf\_inv} & \triangleq \text{message\_buffer} \in \text{MessageBuffers} \\
\text{Initialized\_inv} & \triangleq \text{initialized} \in \text{Initialized} \\
\text{Request\_inv} & \triangleq \text{requests} \in \text{Requests}
\end{align*}
\]
\[ \text{Col}_{\text{inv}} \triangleq \text{collective} \in \text{Collective} \]

\[ \text{group}_{\text{inv}} \triangleq \text{group} \in \text{Group} \]

\[ \text{MPI}_\text{Type}_{\text{Invariant}} \triangleq \]
\[ \land \text{communicator} \in \text{Communicator} \]
\[ \land \text{bufsize} \in \text{BufferSizes} \]
\[ \land \text{message} \_ \text{buffer} \in \text{MessageBuffers} \]
\[ \land \text{initialized} \in \text{Initialized} \]
\[ \land \text{requests} \in \text{Requests} \]
\[ \land \text{collective} \in \text{Collective} \]

\[ \text{Make}_{\text{request}} \text{ is a rule to simplify the expressions that create a new request object. Section } 3.7.1 \]

\[ \text{Make}_{\text{request}}(\text{err}, \text{act}, \text{com}, \text{sta}, \text{sta}, \text{buf}, \text{cty}, \text{per}, \text{mat}, \text{can}, \text{mes}) \triangleq \]
\[ \begin{align*}
\text{error} & \mapsto \text{err}, & \text{The error code associated with this request} \\
\text{active} & \mapsto \text{act}, & \text{The message was initiated} \\
\text{transmitted} & \mapsto \text{com}, & \text{Data was transmitted by this message} \\
\text{started} & \mapsto \text{sta}, & \text{Start this request} \\
\text{buffered} & \mapsto \text{buf}, & \text{The data was copied from the input address} \\
\text{cancelled} & \mapsto \text{can}, & \text{Whether the request was cancelled} \\
\text{allocated} & \mapsto \text{FALSE}, & \text{A new request is created in an allocated state} \\
\text{ctype} & \mapsto \text{cty}, & \text{The type of message (\text{send}, \text{bsend}, \text{rsend}, or \text{ssend})} \\
\text{persist} & \mapsto \text{per}, & \text{Whether the request is a persistent communication} \\
\text{match} & \mapsto \text{mat}, & \text{The matching \langle \text{process,handle} \rangle} \\
\text{message} & \mapsto \text{mes} & \text{The message envelope associated with this request} 
\end{align*} \]

The initial values for the \text{MPI} specification state variables. These are not specified by the standard, however these initial values make the TLA+ representation complete such that it can be verified using TLC.

\[ \text{MPI}_{\text{Specification}}_{\text{Init}} \triangleq \]
\[ \land \text{requests} = [i \in \{0 \ldots (N - 1)\} \mapsto \text{Create an instance of \text{MPI}_\text{REQUEST}_\text{NULL}} \]
\[ \langle \text{Make}_{\text{request}}(0, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{TRUE}, \text{"send"}, \text{FALSE}, \langle\rangle, \text{TRUE}, \]
\[ [\text{src} \mapsto 0, \]
\[ \text{dest} \mapsto 0, \]
\[ \text{msgtag} \mapsto \text{MPI}_\text{ANY}_\text{TAG}, \]
\[ \text{dtype} \mapsto 0, \]
\[ \text{numelements} \mapsto 0, \]
\[ \text{universe} \mapsto \text{MPI}_\text{COMM}_\text{WORLD}, \]
\[ \text{state} \mapsto \text{"send"}, \]
\[ \text{addr} \mapsto 0] \rangle) \]
\[ \land \text{bufsize} = [i \in \{0 \ldots (N - 1)\} \mapsto 0 \text{ Each process starts with no user attached buffer.} \]
\[ \land \text{message} \_ \text{buffer} = [i \in \{0 \ldots (N - 1)\} \mapsto 0 \text{ Each process starts with no messages buffered.} \]
\[ \land \text{initialized} = [i \in \{0 \ldots (N - 1)\} \mapsto \text{"uninitialized"}] \text{ Each process starts uninitialized.} \]
∧ \text{communicator} = [a \in \{0, \ldots, (N - 1)\} \mapsto [i \in \text{MPI}_C\text{OM}_WORLD \cup (\text{MPI}_C\text{OM}_WORLD + \text{MAX}_C\text{OM})] \mapsto
\begin{align*}
\text{participants} & \mapsto \{\}, \\
\text{root} & \mapsto 0, \\
\text{type} & \mapsto "\text{barrier}”, \\
\text{state} & \mapsto "\text{vacant}”]
\end{align*}
∧ \neg \text{RANK}_C\text{ORDERINGS}_C\text{SIGNIFICANT}\ \& \ \ast \ \text{In this case, choose an arbitrary ordering}
∧ \text{choose } f \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \ \ast \ 12.41 - 12.42 \text{ order is not specified.}
\text{choose } \text{finv} \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \ \ast \ \text{The inverse of } f
\forall k \in \text{domain } f : 
\exists n \in 0 \ldots (N - 1) : 
\land f[k] = n 
\land \text{finv}[n] = k
\land \forall m \in \text{domain } f : 
\land f[k] = f[m] \Rightarrow k = m
\land \text{group} = [a \in 0 \ldots (N - 1) \mapsto [i \in \text{MPI}_C\text{OM}_WORLD \cup (\text{MPI}_C\text{OM}_WORLD + \text{MAX}_C\text{GROUP})]] \mapsto
\text{if } i = \text{MPI}_C\text{OM}_WORLD
\text{ then }
\begin{align*}
\text{members} & \mapsto \{x \in 0 \ldots (N - 1) : \text{TRUE}\}, \\
\text{size} & \mapsto N, \\
\text{ranking} & \mapsto f, \\
\text{finv}\text{ranking} & \mapsto \text{finv}
\end{align*}
\text{else }
\begin{align*}
\text{members} & \mapsto \{\}, \\
\text{size} & \mapsto 0, \\
\text{ranking} & \mapsto [j \in 0 \ldots (N - 1) \mapsto 0]], \\
\text{finv}\text{ranking} & \mapsto [j \in 0 \ldots (N - 1) \mapsto 0]]
\end{align*}
\lor \land \text{RANK}_C\text{ORDERINGS}_C\text{SIGNIFICANT}\ \& \ \ast \ \text{in this case, try all orderings}
∧ \exists f \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \ \ast \ 12.41 - 12.42 \text{ order is not specified.}
\exists \text{finv} \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \ \ast \ \text{The inverse of } f
\forall k \in \text{domain } f : 
\exists n \in 0 \ldots (N - 1) : 
\land f[k] = n 
\land \text{finv}[n] = k
\land \forall m \in \text{domain } f : 
\land f[k] = f[m] \Rightarrow k = m
\land \text{group} = [a \in 0 \ldots (N - 1) \mapsto [i \in \text{MPI}_C\text{OM}_WORLD \cup (\text{MPI}_C\text{OM}_WORLD + \text{MAX}_C\text{GROUP})]] \mapsto
\text{if } i = \text{MPI}_C\text{OM}_WORLD
\text{ then }
\begin{align*}
\text{members} & \mapsto \{x \in 0 \ldots (N - 1) : \text{TRUE}\}, \\
\text{size} & \mapsto N, \\
\text{ranking} & \mapsto f, \\
\text{finv}\text{ranking} & \mapsto \text{finv}
\end{align*}
\text{else }
\begin{align*}
\text{members} & \mapsto \{\}, \\
\text{size} & \mapsto 0,
A correct MPI program is one in which all messages that are posted are eventually transmitted or cancelled. A message that is posted but never transmitted is in error. It seems that a message that is transmitted but never completed locally may also be in error... I should check on this.

Conventions on parameters.

1. Parameters that are set (i.e., OUT or INOUT) are all arrays from $0\ldots(N-1)$ with one instance of each object for each process in the model.
2. All other parameters (i.e., IN) are the single instance of the variable value being passed, or are constant.

These rules perform the communication or “matching” of messages that is necessary to complete the MPI communication infrastructure. They are in no way specified in the standard, except that messages are spoken of as being transmitted from one process to another and matching.
Match\((a, b)\) \(\triangleq\)
\[\land \text{Assert}\((a.\text{state} = \text{"recv"} \land b.\text{state} = \text{"send"}) \lor (a.\text{state} = \text{"send"} \land b.\text{state} = \text{"recv"}),
\text{"Error: Match attempted with two send or receives."})\]
\[\land (a.\text{src} = b.\text{src} \lor a.\text{src} = \text{MPI\_ANY\_SOURCE} \lor b.\text{src} = \text{MPI\_ANY\_SOURCE})\]
\[\land (a.\text{dest} = b.\text{dest})\]
\[\land (a.\text{dtype} = b.\text{dtype} \lor a.\text{dtype} = \text{MPI\_ANY\_SOURCE} \lor b.\text{dtype} = \text{MPI\_ANY\_SOURCE})\]
\[\land (a.\text{universe} = b.\text{universe})\]
\[\land \neg (a.\text{src} = \text{MPI\_PROC\_NULL}) \lor \neg (b.\text{src} = \text{MPI\_PROC\_NULL})\]
\[\land \neg (a.\text{dest} = \text{MPI\_PROC\_NULL}) \lor \neg (b.\text{dest} = \text{MPI\_PROC\_NULL})\]
\[21.13 - 21.14\text{ count need not be matched in point to point messages.}\]

Messages match in program order pairwise between processes, however they may complete in a nondeterministic order on both the sender and receiver. This tends to imply that Communicate should in fact be two rules. And it also seems to imply that completion of a message can happen on one side and then on the other also in a non-deterministic way. Therefore Transmit should complete only one side of the communication.

Pairs messages together such that they result in a communication eventually.

Pair \(\triangleq\)
\[\land \exists i \in 0..(N - 1) :\]
\[\exists j \in 0..(N - 1) :\]
\[\exists m \in 1..\text{Len}\(\text{requests}[i])\):\]
\[\exists n \in 1..\text{Len}\(\text{requests}[j])\):\]
\[\text{LET } a \triangleq \text{requests}[i][m]\text{IN}\]
\[\text{LET } b \triangleq \text{requests}[j][n]\text{IN}\]
\[\land a.\text{started}\]
\[\land b.\text{started}\]
\[\land \neg a.\text{cancelled}\]
\[\land \neg b.\text{cancelled}\]
\[\land \neg a.\text{transmitted}\]
\[\land \neg b.\text{transmitted}\]
\[\land \lor \land a.\text{message.state} = \text{"send"}\]
\[\land b.\text{message.state} = \text{"recv"}\]
\[\land \lor \land a.\text{message.state} = \text{"recv"}\]
\[\land b.\text{message.state} = \text{"send"}\]
\[\land a.\text{match} = \emptyset\]
\[\land b.\text{match} = \emptyset\]
\[\land \text{Match}(a.\text{message}, b.\text{message})\]
\[\land \forall r \in 1..\text{Len}\(\text{requests}[i])\):\]
\[\forall s \in 1..\text{Len}\(\text{requests}[j])\):\]
\[\text{LET } c \triangleq \text{requests}[i][r]\text{IN}\]
\[\text{LET } d \triangleq \text{requests}[j][s]\text{IN}\]

This conjunct enforces the fifo.
\( \land \lor \land c.\text{message}\cdot\text{state} = \text{“send”} \)
\( \land d.\text{message}\cdot\text{state} = \text{“recv”} \)
\( \lor \land c.\text{message}\cdot\text{state} = \text{“recv”} \)
\( \land d.\text{message}\cdot\text{state} = \text{“send”} \)
\( \land \text{Match}(c.\text{message}, d.\text{message}) \)
\( \land a.\text{started} \)
\( \land b.\text{started} \)
\( \land \neg c.\text{cancelled} \)
\( \land \neg d.\text{transmitted} \)
\( \land c.\text{match} = \langle \rangle \)
\( \land d.\text{match} = \langle \rangle \)
\( \Rightarrow \land m \leq r \) \text{Section 3.7.4}
\( \land n \leq s \)
\( \land \text{requests}' = [\text{requests} \setminus ![i] =
[@ \text{EXCEPT} ![m] =
[@ \text{EXCEPT} !.\text{match} = (j, n)],
![j] =
[@ \text{EXCEPT} ![n] =
[@ \text{EXCEPT} !.\text{match} = (i, m)]] \)
\( \land \text{UNCHANGED} \langle \text{group, communicator, bsize, message\_buffer, initialized, collective} \rangle \)

Causes the communication that is already paired to complete.
Need to move arrays of data too.

**Transmit** \( \triangleq \)
\( \land \exists i \in 0 \ldots (N - 1) : 
\exists j \in 1 \ldots \text{Len(} \text{requests}[i] \text{)} : 
\text{LET } m \triangleq \text{requests}[i][j] \text{IN} 
\land m.\text{started} 
\land \neg m.\text{cancelled} 
\land \neg m.\text{transmitted} 
\land m.\text{match} \neq \langle \rangle 
\land \text{requests}' = [\text{requests} \setminus ![i] =
[@ \text{EXCEPT} ![j] =
[@ \text{EXCEPT} !.\text{transmitted} = \text{TRUE}]] \)
\( \land \text{IF } \neg \text{requests}[m.\text{match}[1]][m.\text{match}[2]], \text{transmitted} \) THEN
\( \text{IF } m.\text{message}\cdot\text{state} = \text{“recv”} \)
THEN \( \text{Memory}' = [\text{Memory} \setminus ![i] = [@ \text{EXCEPT} ![m.\text{message}\cdot\text{addr}] = \text{Memory}[m.\text{match}[1]][m.\text{match}[2]]] \)
ELSE \( \text{Memory}' = [\text{Memory} \setminus ![m.\text{match}[1]] = [@ \text{EXCEPT} ![\text{requests}[m.\text{match}[1]]][m.\text{match}[2]]] \)
ELSE
\( \text{UNCHANGED} \langle \text{Memory} \rangle \)
\[ \land \text{IF } m.\text{ctype} = \text{"bsend"} \]
\[ \text{THEN} \]
\[ \text{message\_buffer'} = [\text{message\_buffer EXCEPT } ![i] = @ - 1] \]
\[ \text{ELSE} \]
\[ \text{UNCHANGED } \langle \text{message\_buffer} \rangle \]
\[ \land \text{UNCHANGED } \langle \text{group, communicator, bufsize, initialized, collective} \rangle \]

The specification indicates that messages are buffered in an asynchronous manner. The rule Buffer is not part of the standard but necessary to allow buffering to complete asynchronously.

Buffer \[\overset{\Delta}{=}\]
\[ \lor \land \exists i \in (0..(N-1)) : \]
\[ \exists m \in 1..\text{Len}(\text{requests}[i]) : \]
\[ \land \text{requests}[i][m].\text{started} \]
\[ \land \text{requests}[i][m].\text{active} \]
\[ \land \lnot \text{requests}[i][m].\text{buffered} \]
\[ \land \lnot \text{requests}[i][m].\text{cancelled} \]
\[ \land \lnot \text{requests}[i][m].\text{transmitted} \]
\[ \land \lor \land \text{requests}[i][m].\text{ctype} = \text{"bsend"} \]
\[ \land \text{requests'} = \]
\[ [\text{requests EXCEPT } ![i] = \]
\[ [@[EXCEPT ![m] = \]
\[ [@[EXCEPT !.buffered = \text{TRUE}]]] \]
\[ \lor \land \text{requests}[i][m].\text{ctype} = \text{"send"} \]
\[ \land \lor \text{requests'} = \]
\[ [\text{requests EXCEPT } ![i] = \]
\[ [@[EXCEPT ![m] = \]
\[ [@[EXCEPT !.buffered = \text{TRUE}]]] \]
\[ \lor \text{UNCHANGED } \text{requests} \]
\[ \land \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer, initialized, collective} \rangle \]

General Comments:
1.19.23 – 19.24 The message source is provided in the envelope implicitly. Operators in our model must be passed this information as a parameter. As such we extend the argument list to include proc, being the unique identity of the applying process.

Section 3.2 Blocking Send and Receive Operations

Section 3.2.1 Blocking send

Can these really be done in a single transition? I am thinking that it is not possible under an interleaving semantics. In particular, either the send must be two transitions or the receive must be two transitions, it cannot be the case that they are both only one transition.
MPI_Send(buf, count, datatype, dest, tag, comm, proc) \triangleq
MPI_Isend; MPI_Wait

Section 3.2.4 Blocking receive If receive is modeled using only one transition, it is just a combination of the MPI_Irecv and Communicate rules.

MPI_Recv(buf, count, datatype, source, tag, comm, status) \triangleq
MPI_Irecv; MPI_Wait

Section 3.2.5 Return status

Returns in count the number of data elements in the message represented by status.

MPI_Get_count(status, datatype, count, return, proc) \triangleq
\begin{align*}
22.24 & - 22.37 \\
& \land \text{Assert} (\text{Memory}[\text{proc}][\text{Status_Cancelled}(\text{status})] = \text{FALSE}, 54.47 \\
& \quad \text{"Error: count is undefined on a status from a cancelled message."} ) \\
& \land \text{Assert} (\text{initialized}[\text{proc}] = \text{"initialized"}, 200.10 - 200.12 \\
& \quad \text{"Error: MPI_Get_count called before process was initialized."} ) \\
& \land \text{Memory}' = [\text{Memory} \text{EXCEPT } ![\text{proc}] = [@ \text{EXCEPT } ![\text{count}] = \text{Memory}[\text{proc}][\text{Status_Count}(\text{status})]]) \\
& \land \text{UNCHANGED } \langle \text{group}, \text{communicator}, \text{bufsize}, \text{message_buffer}, \text{requests}, \text{initialized}, \text{collective} \rangle
\end{align*}

Section 3.4 Communication Modes

Notes: These, like the above blocking communications really should be modeled using two transitions. In this way, the interleaving semantics is able to schedule another process to complete the communications.

MPI_Bsend(buf, count, datatype, dest, tag, comm, proc) \triangleq
MPI_Ssend(buf, count, datatype, dest, tag, comm, proc) \triangleq
MPI_Rsend(buf, count, datatype, dest, tag, comm, proc) \triangleq

Section 3.6 Buffer allocation and usage

We ignore the buffer argument as data is abstracted away in our model. Buffering is modeled as a counting semaphore, keeping track of the resources available but not exactly which resources are used or what is done with those resources.

Return value is unspecified:

MPI_Buffer_attach(buffer, size, return, proc) \triangleq
\begin{align*}
& <34.17 - 34.33> \\
& \land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, 200.10 - 200.12 \\
& \quad \text{"Error: MPI_Buffer_attach called with proc not in initialized state."} ) \\
& \land \text{Assert}(\text{bufsize}[\text{proc}] = 0, 34.32 \\
& \quad \text{"Error: MPI_Buffer_attach called when processes buffer is non-zero."} )
\end{align*}
∧ bufsize' = [bufsize EXCEPT ![proc] = size[proc]] <34.17−34.33>
∧ UNCHANGED {group, communicator, message_buffer, requests, initialized, collective}
∧ UNCHANGED Memory

Again we ignore the buffer_addr argument as we are abstracting data.

The standard does not indicate what the result is when there is no buffer currently attached.

\[\text{MPI Buffer detach}(\text{buffer_addr}, \text{size}, \text{return}, \text{proc}) \triangleq \]
∧ Assert(initialized[proc] = "initialized", 200.10 − 200.12
  "Error: MPI Buffer detach called with proc not in initialized state.")
∧ Assert(bufsize[proc] ≠ 0,
  "Error: MPI Buffer detach called when no buffer is currently associated with this process.")
∧ ∀ j ∈ 1..Len(requests[proc]) : 34.47
  requests[proc][j].ctype = "bsend" ⇒ requests[proc][j].transmitted
<34.36−35.2>
∧ Memory = [Memory EXCEPT ![proc] = [0 EXCEPT ![size] = bufsize[proc]]] 34.47
∧ UNCHANGED {group, communicator, message_buffer, requests, initialized, collective}

Section 3.7.2 Communication initiation

Notes: I am not sure how to model this construct. The main problem lies in the nondeterministic buffering scheme that the standard refers to. For a correct program one must expect no buffering, however is it possible to write a program in such a way as to require synchronous handshakes?

Start a non-blocking standard send. 38.17 − 38.35, 58.13 − 58.18
\[\text{MPI Isend}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm}, \text{request}, \text{return}, \text{proc}) \triangleq \]
∧ Assert(initialized[proc] = "initialized", 200.10 − 200.12
  "Error: MPI Isend called with proc not in initialized state.")
∧ Assert(proc ∈ group[proc][communicator[proc][comm].group].members,
  "Error: MPI Isend called on a communicator which this process is not a member of.")
∧ LET msg \triangleq
  [addr \mapsto \text{buf},
   src \mapsto \text{group[proc][communicator[proc][comm].group].ranking[proc]},
   dest \mapsto \text{dest},
   msgtag \mapsto \text{tag},
   dtype \mapsto \text{datatype},
   numelements \mapsto \text{count},
   universe \mapsto \text{comm},
   state \mapsto "send"
  ]
IN
request's' = [requests EXCEPT ![proc] = \@ \(\text{Make_request}(0, \text{true}, \text{false}, \text{true}, \text{true}, "send", \text{false}, \text{false}, \text{msg})\)]
∧ Memory' = [Memory EXCEPT ![proc] = [@ EXCEPT ![request] = Len(requests[proc]) + 1]] 40.41

12
∧ UNCHANGED \{group, communicator, bufsize, message_buffer, initialized, collective\}

Set up a non-blocking buffered send. 39.1 – 39.19, 58.13 – 58.18

\[\text{MPI\_Ibsend}(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq\]

∧ Assert\[\text{initialized}[\text{proc}] = "initialized", 200.10 – 200.12\]

“Error: MPI\_Ibsend called with proc not in initialized state.”

∧ Assert\[\text{message\_buffer}[\text{proc}] < \text{bufsize}[\text{proc}], 28.6, 35.34 – 35.35\]

“Error: MPI\_Ibsend called when insufficient buffering was available.”

∧ Assert\[\text{proc} \in \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{members},

“Error: MPI\_Ibsend called on a communicator which this process is not a member of.”

∧ LET \text{msg} \triangleq

\[
\begin{align*}
\text{addr} & \mapsto buf, \\
\text{src} & \mapsto \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{ranking}[\text{proc}], \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{numelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto "send" \]
\end{align*}
\]

IN \[\text{requests}' = [\text{requests} \text{EXCEPT} ![\text{proc}] = 40.40 \]

@ o \{\text{Make\_request}(0, \text{TRUE, TRUE, TRUE, TRUE, "bsend", FALSE, \{\}, FALSE, m})\}\]

∧ Memory' = [Memory \text{EXCEPT} ![\text{proc}] = [0 \text{ EXCEPT} ![\text{request}] = \text{Len}(\text{requests}[\text{proc}]) + 1] 40.41

∧ message\_buffer' = [message\_buffer \text{EXCEPT} ![\text{proc}] = [0 + 1] 28.6 Consume necessary buffer space

∧ UNCHANGED \{message\_buffer \text{EXCEPT} ![\text{proc}] = @ + 1\}

Tested

Set up a non-blocking synchronous send. 39.21 – 39.39, 58.13 – 58.18

\[\text{MPI\_Issend}(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq\]

∧ Assert\[\text{initialized}[\text{proc}] = "initialized", 200.10 – 200.12\]

“Error: MPI\_Issend called with proc not in initialized state.”

∧ Assert\[\text{proc} \in \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{members},

“Error: MPI\_Issend called on a communicator which this process is not a member of.”

∧ LET \text{msg} \triangleq

\[
\begin{align*}
\text{addr} & \mapsto buf, \\
\text{src} & \mapsto \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{ranking}[\text{proc}], \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{numelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto "send" \]
\end{align*}
\]

IN \[\text{requests}' = [\text{requests} \text{EXCEPT} ![\text{proc}] = 40.40 \]

@ o \{\text{Make\_request}(0, \text{TRUE, TRUE, TRUE, TRUE, "ssend", FALSE, \{\}, FALSE, m})\}\]

13
\[\text{Memory}' = [\text{Memory EXCEPT } ![\text{proc}] = @! \text{EXCEPT } ![\text{request}] = \text{Len(requests[proc])} + 1] \]
\[\text{UNCHANGED } \langle\text{group, communicator, message_buffer, bufsize, initialized, collective}\rangle\]

Set up a non-blocking ready send. 40.1 – 40.19, 58.13 – 58.18
\[\text{MPI} \text{Irsend(buffer, count, datatype, dest, tag, comm, request, return, proc)} \triangleq \]
\[\wedge \text{Assert(} \text{initialized}[\text{proc}] = \text{"initialized"}, 200.10 – 200.12\]
\[\text{"Error: MPI} \text{Irsend called with proc not in initialized state."}\]
\[\wedge \text{Assert(} \exists k \in (1 \ldots \text{Len(requests[dest]})) : 37.6 – 37.8\]
\[\wedge \text{requests[dest][k].active} \land \text{~requests[dest][k].transmitted} \land \text{~requests[dest][k].cancelled}\]
\[\wedge \text{Match(requests[proc][request].message, requests[dest][k].message),}\]
\[\text{"Error: MPI} \text{Start tried to start a rsend request when no matching message exists."}\]
\[\wedge \text{requests}' = [\text{requests EXCEPT } ![\text{proc}] = \]
\[\text{LET } \text{msg} \triangleq \]
\[\langle \text{addr } \rightarrow \text{buf}, \]
\[\text{src } \rightarrow \text{group[proc][communicator[proc][comm].group].ranking[proc]}, \]
\[\text{dest } \rightarrow \text{dest}, \]
\[\text{msgtag } \rightarrow \text{tag}, \]
\[\text{dtype } \rightarrow \text{datatype}, \]
\[\text{numelements } \rightarrow \text{count}, \]
\[\text{universe } \rightarrow \text{comm}, \]
\[\text{state } \rightarrow \text{"send"}\rangle\]
\[@ \circ (\text{Make_request}(0, \text{TRUE, FALSE, TRUE, FALSE, "rsend", FALSE, }\langle\rangle, \text{FALSE, msg}))\]
\[\wedge \text{Memory}' = [\text{Memory EXCEPT } ![\text{proc}] = @! \text{EXCEPT } ![\text{request}] = \text{Len(requests[proc])} + 1] \]
\[\wedge \text{UNCHANGED } \langle\text{group, communicator, bufsize, message_buffer, requests, initialized, collective}\rangle\]

Set up a non-blocking receive. 40.21 – 40.39, 58.13 – 58.18
\[\text{MPI} \text{Irecv(buffer, count, datatype, source, tag, comm, request, return, proc)} \triangleq \]
\[\wedge \text{Assert(} \text{initialized}[\text{proc}] = \text{"initialized"}, 200.10 – 200.12\]
\[\text{"Error: MPI} \text{Irecv called with proc not in initialized state."}\]
\[\wedge \text{Assert(} \exists k \in (1 \ldots \text{Len(requests[proc]})) : 37.6 – 37.8\]
\[\wedge \text{requests[proc][request].message, requests[dest][k].message,}\]
\[\text{"Error: MPI} \text{Start tried to start a rsend request when no matching message exists."}\]
\[\wedge \text{LET } \text{msg} \triangleq \]
\[\langle \text{addr } \rightarrow \text{buf}, \]
\[\text{src } \rightarrow \text{source}, \]
\[\text{dest } \rightarrow \text{group[proc][communicator[proc][comm].group].ranking[proc]}, \]
\[\text{msgtag } \rightarrow \text{tag}, \]
\[\text{dtype } \rightarrow \text{datatype}, \]
\[\text{numelements } \rightarrow \text{count}, \]
\[\text{universe } \rightarrow \text{comm}, \]
\[ \text{requests}' = [\text{requests} \text{ EXCEPT } ![\text{proc}] = 40.40 \]
\[ \odot \langle \text{Make_request}(0, \text{true, false, true, false, "recv"}, \text{false, }\langle\rangle, \text{false, }\langle\rangle) \rangle \]
\[ \wedge \text{Memory}' = [\text{Memory} \text{ EXCEPT } ![\text{proc}] = [@ \text{EXCEPT } ![\text{request}] = \text{Len}(\text{requests}[\text{proc}]) + 1]\]
\[ \wedge \text{UNCHANGED } \langle\text{group, communicator, message_buffer, bufsize, initialized, collective}\rangle \]

Section 3.7.3 Communication Completion

Would if... then... else be a better, more readable form here? Maybe not because we need to block.

Wait for request to complete. Return information about the message in status. 41.23 – 42.6

No specification on what the status value is when a send is posted with \texttt{MPI_PROC_NULL}.

Specifies next state for status and request

\[ \text{MPI}_\text{Wait}(\text{request}, \text{status}, \text{return}, \text{proc}) \triangleq \]
\[ \text{LET } r \triangleq \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{request}]] \]
\[ \wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, 200.10 – 200.12) \]

\[ \text{"Error: MPI}_\text{Wait called with proc not in initialized state."} ) \]
\[ \wedge \vee \wedge \text{Memory}[\text{proc}][\text{request}] \neq \text{MPI}_\text{REQUEST}_\text{NULL} \]

The request handle is not the null handle.

\[ \wedge r.\text{active} \]

The request is active.

\[ \wedge \vee \wedge r.\text{message.src} \neq \text{MPI}_\text{PROC}_\text{NULL} \]

The message source is not null.

\[ \wedge r.\text{message.dest} \neq \text{MPI}_\text{PROC}_\text{NULL} \]

The message destination is not null.

41.32 – Blocks until complete

\[ \wedge \vee r.\text{transmitted} \]

The communication actually happened or

\[ \vee r.\text{cancelled} \]

the communication got cancelled by the user program or

\[ \vee r.\text{buffered} \]

the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).

\[ \text{A status object for a completed communication.} \]

\[ \wedge \text{Memory}' = [\text{Memory} \text{ EXCEPT } ![\text{proc}] = 41.36 \]
\[ \odot \langle \text{Status}_\text{Cancelled}(\text{status}) = r.\text{cancelled} \wedge \neg r.\text{transmitted}, 54.46 \]
\[ \langle \text{Status}_\text{Count}(\text{status}) = r.\text{message.numelements}, \rangle \]
\[ \langle \text{Status}_\text{Source}(\text{status}) = r.\text{message.src}, \rangle \]
\[ \langle \text{Status}_\text{Tag}(\text{status}) = r.\text{message.msgtag}, \rangle \]
\[ \langle \text{Status}_\text{Err}(\text{status}) = r.\text{error}, \rangle \]
\[ ![\text{request}] = \text{if } r.\text{persist} \text{ THEN } @ \text{ELSE } \text{MPI}_\text{REQUEST}_\text{NULL} \rangle \]

41.32 – 41.35, 15
Status_Err(status) = 0, request = if r.persist then MPI_REQUEST_NULL
\[ \text{requests}' = [\text{requests} \setminus ![\text{proc}] = \text{MPI_REQUEST_NULL}] \]
\[ \text{if r.persist} \]
\[ \text{THEN} \]
\[ \text{if active = FALSE] \]
\[ \text{ELSE} \]
\[ \text{if active = FALSE, deallocated = TRUE]} \]
\[ \text{The request is not active} \]
\[ \text{or the request handle is null} \]
\[ \text{The message source is not null} \]
\[ \text{The message destination is not null} \]
\[ \text{The communication actually happened or} \]
\[ \text{the communication got cancelled by the user program or} \]
\[ \text{the communication got buffered either into explicit user provided} \]
\[ \text{buffer space or into system provided buffer space (if regular send is used).} \]
\[ \text{THEN} \]
\[ \text{Memory'} = [\text{Memory} \setminus ![\text{proc}] = \text{MPI_REQUEST_NULL}] \]
\[ \text{if cancelled \&\& transmitted} \]
\[ \text{The request handle is not the null handle.} \]
\[ \text{The request is active.} \]
\[ \text{The message source is not null} \]
\[ \text{The message destination is not null} \]
\[ \text{The communication actually happened or} \]
\[ \text{the communication got cancelled by the user program or} \]
\[ \text{the communication got buffered either into explicit user provided} \]
\[ \text{buffer space or into system provided buffer space (if regular send is used).} \]
\[ \text{THEN} \]
\[ \text{Memory'} = [\text{Memory} \setminus ![\text{proc}] = \text{MPI_REQUEST_NULL}] \]
\[ \text{if cancelled \&\& transmitted,} \]
\[ \text{The request handle is not the null handle.} \]
\[ \text{The request is active.} \]
\[ \text{The message source is not null} \]
\[ \text{The message destination is not null} \]
\[ \text{The communication actually happened or} \]
\[ \text{the communication got cancelled by the user program or} \]
\[ \text{the communication got buffered either into explicit user provided} \]
\[ \text{buffer space or into system provided buffer space (if regular send is used).} \]
\[ \text{THEN} \]
\[ \text{Memory'} = [\text{Memory} \setminus ![\text{proc}] = \text{MPI_REQUEST_NULL}] \]
\[ \text{if cancelled \&\& transmitted,} \]
\[ \text{The request handle is not the null handle.} \]
\[ \text{The request is active.} \]
\[ \text{The message source is not null} \]
\[ \text{The message destination is not null} \]
\[ \text{The communication actually happened or} \]
\[ \text{the communication got cancelled by the user program or} \]
\[ \text{the communication got buffered either into explicit user provided} \]
\[ \text{buffer space or into system provided buffer space (if regular send is used).} \]
\[ \text{THEN} \]
![flag] = TRUE,
![request] = if r.persist then @ else MPI_REQUEST_NULL]
∧ requests' =
   [requests EXCEPT ![proc] =
    [@ EXCEPT ![Memory][proc][request]] =
    ![active = FALSE]]
ELSE
∧ Memory' = [Memory EXCEPT ![proc] =
    [@ EXCEPT ![flag] = FALSE]]
∧ UNCHANGED ⟨requests⟩
∀ r.message.src = MPI_PROC_NULL The source or destination were actually
∧ r.message.dest = MPI_PROC_NULL the null process 42.29 – 42.31
∧ Memory' = [Memory EXCEPT ![proc] =
    [@ EXCEPT ![flag] = TRUE, ![request] = if r.persist then @ else MPI_REQUEST_NULL]]
∧ requests' =
   [requests EXCEPT ![proc] =
    [@ EXCEPT ![active = FALSE]]] 42.22 – 42.23, 58.34 Not modeling deallocation
∧ UNCHANGED ⟨requests⟩
∀ r.active The request is not active or the request
∧ Memory[proc][request] = MPI_REQUEST_NULL handle is null 42.29 – 42.31
∧ Memory' = [Memory EXCEPT ![proc] =
    [@ EXCEPT ![Status_Cancelled(status)] = FALSE,
    ![Status_Count(status)] = 0,
    ![Status_Source(status)] = MPI_PROC_NULL,
    ![Status_Tag(status)] = MPI_ANY_TAG,
    ![Status_Err(status)] = 0,
    ![flag] = TRUE, ![request] = if r.persist then @ else MPI_REQUEST_NULL]]
∧ requests' =
   [requests EXCEPT ![proc] =
    [@ EXCEPT ![active = FALSE]]] 42.22 – 42.23, 58.34 Not modeling deallocation
∀ ¬r.active The request is not active or the request
∧ Memory[proc][request] = MPI_REQUEST_NULL handle is null 42.29 – 42.31
∧ Memory' = [Memory EXCEPT ![proc] =
    [@ EXCEPT ![Status_Cancelled(status)] = FALSE,
    ![Status_Count(status)] = 0,
    ![Status_Source(status)] = MPI_ANY_SOURCE,
    ![Status_Tag(status)] = MPI_ANY_TAG,
    ![Status_Err(status)] = 0,
    ![flag] = TRUE]]
∧ UNCHANGED ⟨requests⟩
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, initialized, collective⟩

Frees the request specified.
Modifies request.

MPI_Request_free(request, return, proc) ≡
  "MPI_Request_free called with proc not in initialized state." )
∧ Assert(¬requests[proc][Memory][proc][request][active], 43.37 – 43.39
  "MPI_Request_free called with an inactive request.")
∧ Memory' = [Memory EXCEPT ![proc] = ![request] = MPI_REQUEST_NULL]] 43.20 Not a
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, requests, initialized, collective⟩
Section 3.7.5 Multiple Completions

Wait for one of the requests referenced in array_of_requests to complete.

Specifies next state for index and status

\[\text{MPI}_-\text{Waitany}(\text{count}, \text{array}_-\text{of}_-\text{requests}, \text{index}, \text{status}, \text{return}, \text{proc}) \triangleq\]

\[\text{LET } r(v) \triangleq \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{array}_-\text{of}_-\text{requests} + v]]\]

\[\wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, 
\quad \text{200.10 – 200.12})\]

"Error: \text{MPI}_-\text{Waitany} \text{called with proc not in initialized state.}"

\[\wedge \exists i \in (0..(\text{count} - 1)) : \]

\[\wedge \text{Memory}[\text{proc}][\text{array}_-\text{of}_-\text{requests} + i] \neq \text{MPI}_-\text{REQUEST}_-\text{NULL} \] 

The request handle is not the null handle.

\[\wedge r(i).\text{active} \] 

The request is active.

\[\wedge \forall r(i).\text{message}.\text{src} \neq \text{MPI}_-\text{PROC}_-\text{NULL} \] 

The message source is not null.

\[\wedge r(i).\text{message}.\text{dest} \neq \text{MPI}_-\text{PROC}_-\text{NULL} \] 

The message destination is not null.

\[\wedge \forall r(i).\text{transmitted}\] 

The communication actually happened or

\[\forall r(i).\text{cancelled} \]

the communication got cancelled by the user program or

\[\forall r(i).\text{buffered}\]

the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).

\[\wedge \text{Memory'} = [\text{Memory \text{EXCEPT} ![\text{proc}]} = \text{45.46 – 45.47}]\]

[\@ \text{EXCEPT}\]

\[![[\text{Status}_-\text{Source}(\text{status})] = r(i).\text{message}.\text{src}], \text{45.47 – 45.48}\]

\[![[\text{Status}_-\text{Tag}(\text{status})] = r(i).\text{message}.\text{msgtag}],\]

\[![[\text{Status}_-\text{Err}(\text{status})] = r(i).\text{error}],\]

\[![[\text{Status}_-\text{Count}(\text{status})] = r(i).\text{message}.\text{nulelements}],\]

\[![[\text{Status}_-\text{Cancelled}(\text{status})] = r(i).\text{cancelled} \land \neg r(i).\text{transmitted}], \text{54.46}\]

\[![[\text{array}_-\text{of}_-\text{requests} + i] = \text{IF } r(i).\text{persist} \text{ THEN } @ \text{ELSE } \text{MPI}_-\text{REQUEST}_-\text{NULL}, \text{46.1 – 46.2}]\]

\[![[\text{index}] = i]] \text{45.46}\]

\[\forall \wedge \forall r(i).\text{message}.\text{src} = \text{MPI}_-\text{PROC}_-\text{NULL} \] 

The source or destination was actually the null process.

\[\wedge \text{Memory'} = [\text{Memory \text{EXCEPT} ![\text{proc}]} = @ \text{EXCEPT}\]

\[![[\text{Status}_-\text{Source}(\text{status})] = \text{MPI}_-\text{PROC}_-\text{NULL}],\]

\[![[\text{Status}_-\text{Tag}(\text{status})] = \text{MPI}_-\text{ANY}_-\text{TAG}],\]

\[![[\text{Status}_-\text{Err}(\text{status})] = 0],\]

\[![[\text{Status}_-\text{Count}(\text{status})] = 0],\]

\[![[\text{Status}_-\text{Cancelled}(\text{status})] = r(i).\text{cancelled}],\]

\[![[\text{array}_-\text{of}_-\text{requests} + i] = \text{IF } r(i).\text{persist} \text{ THEN } @ \text{ELSE } \text{MPI}_-\text{REQUEST}_-\text{NULL}, \text{46.2}]\]

\[![[\text{index}] = i]] \text{45.46}\]

\[\forall \forall i \in (0..(\text{count} - 1)) : \]

\[\wedge \neg r(i).\text{active}\] 

The request is not active or the request

\[\forall \text{Memory}[\text{proc}][\text{array}_-\text{of}_-\text{requests} + i] = \text{MPI}_-\text{REQUEST}_-\text{NULL}\] 

handle is null.

\[\wedge \text{Memory'} = [\text{Memory \text{EXCEPT} ![\text{proc}]} = \text{46.5}]\]
The request is active.

The message source is not null

The message destination is not null

The request handle is not the null handle.

The source or destination were actually unchanged

The communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).

The communication actually happened or

the communication got cancelled by the user program or

the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).

The communication was actually buffered.

The source or destination were actually unchanged.
\[ \forall r(i).message.dest = MPI\_PROC\_NULL \quad \text{the null process 61.3 - 61.4} \]

\[ \wedge \text{Memory}' = [\text{Memory except } ![\text{proc}]] = [\@ \text{EXCEPT} \]

\[
\begin{align*}
&[@] = \text{TRUE}, \quad 46.29 \\
&[@] = MPI\_PROC\_NULL, \\
&[@] = MPI\_ANY\_SOURCE, \quad 46.36 \\
&[@] = MPI\_ANY\_TAG, \\
&[@] = 0, \\
&[@] = 0, \\
&[@] = r(i).\text{cancelled}, \\
&[@] = i, \quad 46.29 \\
&[@] = \text{IF } r(i).\text{persist} \quad \text{THEN } @ \text{ELSE } MPI\_REQUEST\_NULL] \quad 46.31 - \\
\wedge \text{requests}' = \\
\begin{align*}
&[@] = 46.31 - 46.32, 58.34 \\
&[@] = [\@ \text{EXCEPT} ![\text{Memory}[\text{proc}][\text{array_of_requests} + i]] = \\
&[@] = [\@ \text{EXCEPT} !.\text{active} = \text{FALSE}]] \\
&\forall i \in (0 \ldots (\text{count} - 1)) : \quad 46.35 - 46.37 \\
&\wedge \forall r(i).\text{active} \quad \text{The request is not active or the request} \\
\checkmark \text{array_of_requests}[\text{proc}][i] = MPI\_REQUEST\_NULL \quad \text{handle is null} \\
\wedge \text{Memory}' = [\text{Memory except } ![\text{proc}]] = [\@ \text{EXCEPT} \]

\[
\begin{align*}
&[@] = \text{TRUE}, \quad 46.36 \\
&[@] = MPI\_ANY\_SOURCE, \quad 46.36 \\
&[@] = MPI\_ANY\_TAG, \\
&[@] = 0, \\
&[@] = 0, \\
&[@] = \text{false}, \\
&[@] = \text{MPL\_UNDEFINED}] \quad 46.36 \\
\wedge \text{UNCHANGED } (\text{requests}) \\
\wedge \text{UNCHANGED } (\text{group, communicator, bufsize, message_buffer, initialized, collective}) \\
\end{align*}
\]

A long version of MPI\_Waitall – includes the line by line reference. 

Specifies the next state for array\_of\_requests and array\_of\_statuses.

\text{MPI\_Waitall(count, array\_of\_requests, array\_of\_statuses, return, proc) } \triangleq \\
\text{LET } r(v) \triangleq \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{array_of_requests} + v]] \text{IN} \\
\wedge \text{Assert(initialized)[proc] = “initialized”,} \quad 200.10 - 200.12 \\
\text{“Error: MPI\_Waitall called with proc not in initialized state.”)}

\[ \wedge \forall i \in (0 \ldots (\text{count} - 1)) : \quad 47.18 \\
\wedge \text{Memory}[\text{proc}][\text{array_of_requests} + i] \neq MPI\_REQUEST\_NULL \quad \text{The request handle is not the null} \\
\wedge r(i).\text{active} \quad \text{The request is active.} \\
\wedge \forall r(i).\text{message.src} \neq MPI\_PROC\_NULL \quad \text{The message source is not null} \\
\wedge r(i).\text{message.dest} \neq MPI\_PROC\_NULL \quad \text{The message destination is not null} \\
\wedge \forall r(i).\text{transmitted} \quad \text{The communication actually happened or} \\
\quad \forall r(i).\text{cancelled} \quad \text{the communication got cancelled by the user program or} \\
\quad \forall r(i).\text{buffered} \quad \text{the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).} \\
\wedge \forall r(i).\text{message.src} = MPI\_PROC\_NULL \quad \text{The source or destination was actually} \\
\]
\( \forall r(i).message.dest = MPI\_PROC\_NULL \) the null process

\( \wedge array\_of\_requests' = [array\_of\_requests \text{ EXCEPT } ![proc] = 47.22 - 47.23 \)

\[ j \in 0 .. (\text{count} - 1) \] \( \rightarrow \)

\( \text{IF } r(j).\text{persist} \)

\( \text{THEN } \)

\( array\_of\_requests[proc][j] \)

\( \text{ELSE } \)

\( MPI\_REQUEST\_NULL] \)

\( \wedge array\_of\_statuses' = [array\_of\_statuses \text{ EXCEPT } ![proc] = 47.18 - 47.21 \)

\[ j \in (0 .. \text{count} - 1) \] \( \rightarrow \)

\( \text{IF } \forall r(j).message.src = MPI\_PROC\_NULL \)

\( \text{\lor } r(j).message.dest = MPI\_PROC\_NULL \)

\( \text{THEN } \)

\( 61.3 - 61.4 \)

\( \text{ state } \rightarrow \text{"defined"}, \)

\( \text{MPI\_SOURCE } \rightarrow \text{MPI\_PROC\_NULL}, \)

\( \text{MPI\_TAG } \rightarrow \text{MPI\_ANY\_TAG}, \)

\( \text{MPI\_ERROR } \rightarrow 0, \)

\( \text{count } \rightarrow 0, \)

\( \text{cancelled } \rightarrow r(j).\text{cancelled} \)

\( \text{ELSE } \)

\( \text{ state } \rightarrow \text{"defined"}, \)

\( \text{MPI\_SOURCE } \rightarrow r(j).message.src, \)

\( \text{MPI\_TAG } \rightarrow r(j).message.msgtag, \)

\( \text{MPI\_ERROR } \rightarrow r(j).error, \)

\( \text{count } \rightarrow r(j).message.numelements, \)

\( \text{cancelled } \rightarrow r(j).\text{cancelled } \land \neg r(i).\text{transmitted}] 84.46 \)

\( \wedge requests' = [requests \text{ EXCEPT } ![proc] = 47.22, 58.34 \text{ Not modeling deallocation} \)

\[ j \in 1 .. \text{Len}(\emptyset) \] \( \rightarrow \)

\( \text{IF } \exists k \in 0 .. (\text{count} - 1) : j = array\_of\_requests[proc][k] \)

\( \text{THEN } \)

\( \text{[requests[proc][j] \text{ EXCEPT } ![active = FALSE]} \)

\( \text{ELSE } \)

\( \text{requests[proc][j]]} \)

\( \lor \forall i \in 0 .. (\text{count} - 1) : \quad 47.23 - 47.24 \)

\( \lor Memory[proc][array\_of\_requests + i] = MPI\_REQUEST\_NULL \) The request handle is null or

\( \lor \neg r(i).\text{active } \)

\( \lor Memory' = [Memory \text{ EXCEPT } ![proc] = \)

\[ j \in 1 .. \text{Len}(Memory[proc]) \] \( \rightarrow \)

\( \text{IF } j \in array\_of\_statuses .. (array\_of\_statuses + ((\text{count} + 5) - 1)) \)

\( \text{THEN } \)

\( \text{IF } (j - array\_of\_statuses)\%5 = 0 \)

\( \text{THEN FALSE } \)

\( \text{ELSE } \)

\( \text{IF } (j - array\_of\_statuses)\%5 = 1 \)
then 0
else if (j - array_of_statuses)%5 = 2
then MPI_ANY_SOURCE
else if (j - array_of_statuses)%5 = 3
then MPI_ANY_TAG
else if (j - array_of_statuses)%5 = 4
then 0
else Assert(FALSE, "Internal Error: Cannot have any other cases.")
else Memory[proc][j]
∧ unchanged ⟨array_of_requests, requests⟩
∧ unchanged ⟨group, communicator, bufsize, message_buffer, requests, initialized, collective⟩

Test whether all requests referenced in array_of_requests have completed.
MPI_Testall(count, array_of_requests, flag, array_of_statuses, return, proc) ≡
LET r(v) ≡ requests[proc][array_of_requests][proc][v]IN
"Error: MPI_Testall called with proc not in initialized state.")
∧ IF ∀ i ∈ (0 .. (count –)):
  ∨ ∧ array_of_requests[proc][i] ≠ MPI_REQUEST_NULL The request handle is not the null handle.
  ∧ r(i).active The request is active.
  ∧ ∨ ∧ r(i).message.src ≠ MPI_PROC_NULL The message source is not null.
  ∧ r(i).message.dest ≠ MPI_PROC_NULL The message destination is not null.
  ∧ ∨ r(i).transmitted The communication actually happened or
  ∨ r(i).cancelled the communication got cancelled by the user program or
  ∨ r(i).buffered the communication got buffered either into explicit user provided
   buffer space or into system provided buffer space (if regular send is used).
  ∨ ∧ ∨ r(i).message.src = MPI_PROC_NULL The source or destination were actually
  ∨ r(i).message.dest = MPI_PROC_NULL the null process.
  ∨ ∀ i ∈ (0 .. (count –)):
    ∨ array_of_requests[proc][i] = MPI_REQUEST_NULL
  ∨ ¬r(i).active
THEN
∧ array_of_statuses’ = [array_of_statuses EXCEPT ![proc] =
[i ∈ (0 .. (count – 1))]
IF ∨ r(i).message.src = MPI_PROC_NULL
  ∨ r(i).message.dest = MPI_PROC_NULL
THEN
  [state = “defined”,
  MPI_SOURCE ⇒ MPI_PROC_NULL,
  MPI_TAG ⇒ MPI_ANY_TAG,
  MPI_ERROR ⇒ 0,
  61.3 – 61.4
A status object for a communication
with a null process.
}
count \mapsto 0,
cancelled \mapsto \text{FALSE}

\text{ELSE}

\text{IF} \ \forall \text{array\_of\_requests}[\text{proc}][i] = \text{MPI\_REQUEST\_NULL} \ \\ \text{AND} \ \ \neg \exists \text{active} \ \text{THEN}

\begin{align*}
\text{[state \mapsto "empty"}, & \quad \text{The resultant empty status.} \\
\text{MPI\_SOURCE \mapsto MPI\_ANY\_SOURCE}, & \\
\text{MPI\_TAG \mapsto MPI\_ANY\_TAG}, & \\
\text{MPI\_ERROR \mapsto 0}, & \\
\text{count \mapsto 0}, & \\
\text{cancelled \mapsto \text{FALSE}}]
\end{align*}

\text{ELSE} \ \text{48.17} \ \text{AND} \ \text{requests} \ ' = \text{48.18} \ \text{AND} \ \text{requests} \ ^{\prime} = \text{48.19}

\text{Not modeling deallocation}

\text{[requests \ except \ ![proc] =}

\begin{align*}
\text{\[i \in 1 \ldots \text{Len}(0) \mapsto} & \\
\text{\quad \text{IF} \ \exists j \in 0 \ldots (\text{count} - 1) : \text{array\_of\_requests}[\text{proc}][j] = i} & \\
\text{\quad \text{THEN}} & \\
\text{\quad \text{[requests}[\text{proc}][i] \ \text{EXCEPT ![active = FALSE]} & \\
\text{\quad \text{ELSE}} & \\
\text{\quad \text{requests}[\text{proc}][i]]}
\end{align*}

\text{\text{AND} \ \text{array\_of\_requests} \ ' = \text{[array\_of\_requests} \ \text{EXCEPT ![proc] =}

\begin{align*}
\text{\[i \in 0 \ldots (\text{count} - 1) \mapsto} & \\
\text{\quad \text{IF} \ r(i).\text{persist}} & \\
\text{\quad \text{THEN}} & \\
\text{\quad \text{array\_of\_requests}[\text{proc}][i]} & \text{58.34 - 58.35} \\
\text{\quad \text{ELSE}} & \\
\text{\quad \text{MPI\_REQUEST\_NULL]]} & \text{48.19 - 48.21}
\end{align*}

\text{\text{AND} \ \text{flag} \ ' = \text{[flag} \ \text{EXCEPT ![proc] = TRUE]} \ \text{48.15}

\text{\text{ELSE}} \ \text{48.15 - 48.22}

\text{\text{AND} \ \text{array\_of\_statuses} \ ' = \text{[array\_of\_statuses} \ \text{EXCEPT ![proc] =}

\begin{align*}
\text{\[i \in 0 \ldots (\text{count} - 1) \mapsto} & \\
\text{\quad \text{[array\_of\_statuses}[\text{proc}][i] \ \text{EXCEPT ![state = "undefined"]]]]
\end{align*}

\text{\text{AND} \ \text{UNCHANGED} \ \text{[array\_of\_requests, requests]}}

\text{\text{AND} \ \text{UNCHANGED} \ \{\text{group, communicator, bufsize, message\_buffer, initialized, collective}\}}
Wait for some subset of the requests referenced in array_of_requests to complete.
The ordering of array_of_indices or array_of_statuses is not specified.

Not modeling the possibility of arbitrary ordering of the array_of_indices or array_of_statuses.

\[ \text{MPI\_Waitsome}(\text{incount}, \text{array\_of\_requests}, \text{outcount}, \]
\[ \text{array\_of\_indices, array\_of\_statuses, return, proc} \]
\[ \Delta \]
\[ \text{LET } r(v) \triangleq \text{requests}[\text{proc}][\text{array\_of\_requests}[\text{proc}][v]]\]
\[ \text{LET } \text{msgs} \triangleq \]
\[ \{ x \in (0..(\text{incount} - 1)) : \]
\[ \text{array\_of\_requests}[\text{proc}][x] \neq \text{MPI\_REQUEST\_NULL} \]
\[ \land r(x).\text{active} \]
\[ \land \lor r(x).\text{transmitted} \]
\[ \land \lor r(x).\text{cancelled} \]
\[ \land \lor r(x).\text{buffered} \}
\[ \text{The messages that have completed in the array\_of\_requests}
\[ \land \text{The request handle is not the null handle.}
\[ \land \text{The request is active.}
\[ \land \text{The communication actually happened or}
\[ \land \text{the communication got cancelled by the user program or}
\[ \land \text{the communication got buffered either into explicit user provided}
\[ \land \text{buffer space or into system provided buffer space (if regular send is used).}
\[ \text{IN}
\[ \land \text{Assert(initialized[proc] = "initialized",}
\[ \land \lor \land \text{Cardinality(msgs) > 0}}
\[ \land \lor \land \text{outcount}' = [\text{outcount EXCEPT } ![\text{proc} = \text{Cardinality(msgs)}]}
\[ \land \lor \land \exists \text{seq } \in \text{Seq(msgs)} : \text{from FiniteSets.tla module}
\[ \land \lor \land \forall s \in \text{msgs} : \]
\[ \exists i \in 1..\text{Len(seq)} : \]
\[ \land \text{seq}[i] = s \]
\[ \land \forall m \in 1..\text{Len(seq)} : \text{seq}[n] = \text{seq}[m] \Rightarrow m = n \]
\[ \land \text{array\_of\_indices}' = [\text{array\_of\_indices EXCEPT } ![\text{proc} =}
\[ [i \in 0..(\text{incount} - 1) \Rightarrow}
\[ \text{IF } i < \text{Len(seq)} \]
\[ \text{THEN seq}[i + 1] \]
\[ \text{ELSE array\_of\_indices}[\text{proc}][i]]
\[ \land \text{array\_of\_statuses}' = [\text{array\_of\_statuses EXCEPT } ![\text{proc} =}
\[ [i \in 0..(\text{incount} - 1) \Rightarrow}
\[ \text{IF } i < \text{Len(seq)} \]
\[ \text{THEN}
\[ [\text{state } \Rightarrow \text{"defined"}, A status object for a completed communication.}
\[ \text{MPI\_SOURCE } \Rightarrow r(seq[i + 1]).\text{message.src},
\[ \text{MPI\_TAG } \Rightarrow r(seq[i + 1]).\text{message.msgtag},
\[ \text{MPI\_ERROR } \Rightarrow r(seq[i + 1]).\text{error},
\[ \text{count } \Rightarrow r(seq[i + 1]).\text{message.numelements},
\[ \text{cancelled } \Rightarrow r(seq[i + 1]).\text{cancelled} \land \neg r(seq[i + 1]).\text{transmitted}]}
\[ \text{ELSE}
\[ \text{array\_of\_statuses}[\text{proc}[i]]]
\[ \land \text{requests}' = [\text{requests EXCEPT } ![\text{proc} =}
\[ [i \in 1..\text{Len(requests[proc])} \Rightarrow}
\[ \text{IF } \exists m \in \text{msgs} : i = \text{array\_of\_requests}[\text{proc}][m]
\[ \text{24}
\[\text{if } \exists m \in \text{msgs} : i = \text{array\_of\_requests}[\text{proc}][m] \wedge r(i).\text{persist} \]

\[\text{then} \quad \text{array\_of\_requests}[\text{proc}][i] \]

\[\text{else} \quad \text{MPI\_REQUEST\_NULL} \]

\[\lor \quad \forall i \in (0 \ldots (\text{incount} - 1)) : \quad 49.5 - 49.6\]

\[\lor \quad \text{array\_of\_requests}[\text{proc}][i] = \text{MPI\_REQUEST\_NULL} \]

\[\lor \quad \neg \text{requests}[\text{proc}].[\text{array\_of\_requests}[\text{proc}][i]].\text{active} \]

\[\wedge \text{outcount}' = [\text{outcount} \text{ except } ![\text{proc}] = \text{Cardinality}(\text{msgs})] \quad 48.46\]

\[\wedge \exists \text{seq} \in \text{Seq}(\text{msgs}) : \quad \text{from FiniteSets.tla module!} \]

\[\wedge \forall s \in \text{msgs} : \]

\[\exists n \in 1 \ldots \text{Len}(\text{seq}) : \quad 48.47 - 49.2\]

\[\wedge \forall m \in 1 \ldots \text{Len}(\text{seq}) : \quad \text{seq}[n] = \text{seq}[m] \Rightarrow m = n\]

Test for some subset of the requests referenced in the \text{array\_of\_requests} to complete.

Defined in terms of \text{MPI\_Waitsome}.

\[\text{MPI\_Testsome}(\text{incount}, \text{array\_of\_requests}, \text{outcount},\]

\[\quad \text{array\_of\_indices}, \text{array\_of\_statuses, return, proc}) \]

\[\Delta = \text{let } r(v) \triangleq \text{requests}[\text{proc}].[\text{array\_of\_requests}[\text{proc}][v]]\]

\[\text{LET } \text{msgs} \triangleq \]

\[\{ x \in (0 \ldots (\text{incount} - 1)) : \quad \text{The messages that have completed in the array\_of\_requests}\]

\[\wedge \text{array\_of\_requests}[\text{proc}][x] \neq \text{MPI\_REQUEST\_NULL} \quad \text{The request handle is not the null handle.}\]

\[\wedge r(x).\text{active} \quad \text{The request is active.}\]

\[\wedge \forall r(x).\text{transmitted} \quad \text{The communication actually happened or}\]

\[\forall r(x).\text{cancelled} \quad \text{the communication got cancelled by the user program or}\]

\[\forall r(x).\text{buffered} \quad \text{the communication got buffered either into explicit user provided}\]

\[\text{buffer space or into system provided buffer space (if regular send is used).}\]

\[\wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{“initialized"}, \quad 200.10 - 200.12\]

\[\text{“Error: MPI\_Testsome called with proc not in initialized state."}) \]

\[\wedge \forall i \in (0 \ldots (\text{incount} - 1)) : \quad 49.35 - 49.36, 49.5\]

\[\wedge \text{array\_of\_requests}[\text{proc}][i] \neq \text{MPI\_REQUEST\_NULL} \]

\[\wedge r(i).\text{active} \]

\[\wedge \text{if } \text{Cardinality}(\text{msgs}) > 0 \quad \text{number of completed messages} \]

\[\text{then} \quad \]

\[\wedge \text{outcount}' = [\text{outcount} \text{ except } ![\text{proc}] = \text{Cardinality}(\text{msgs})] \quad 48.46\]

\[\wedge \exists \text{seq} \in \text{Seq}(\text{msgs}) : \quad \text{from FiniteSets.tla module!} \]

\[\wedge \forall s \in \text{msgs} : \]

\[\exists n \in 1 \ldots \text{Len}(\text{seq}) : \quad 48.47 - 49.2\]

\[\wedge \forall m \in 1 \ldots \text{Len}(\text{seq}) : \quad \text{seq}[n] = \text{seq}[m] \Rightarrow m = n\]
array_of_indices' = [array_of_indices except ![proc] =
  [j ∈ 0..(incount – 1) ↦
   IF j < Len(seq)
   THEN seq[j + 1]
   ELSE array_of_indices[proc][j]]]
∧ array_of_statuses' = [array_of_statuses except ![proc] =
  [j ∈ 0..(incount – 1) ↦
   THEN
   [state ↦ “defined”,
    MPI_SOURCE ↦ r(seq[j + 1]).message.src,
    MPI_TAG ↦ r(seq[j + 1]).message.msgtag,
    MPI_ERROR ↦ r(seq[j + 1]).error,
    count ↦ r(seq[j + 1]).numelements,
    cancelled ↦ r(seq[j + 1]).cancelled ∧ ¬r(seq[j + 1]).transmitted]
   ELSE
   array_of_statuses[proc][j]]]
∧ requests' = [requests except ![proc] =
  [j ∈ 0..(incount – 1) ↦
   IF ∃ m ∈ msgs : j = array_of_requests[proc][m]
   THEN ![r(j) EXCEPT ![active = FALSE]
   ELSE ![r(j)]]]
∧ array_of_requests' = [array_of_requests except ![proc] =
  [j ∈ 0..(incount – 1) ↦
   IF ∃ m ∈ msgs : j = array_of_requests[proc][m]
   ∧ ![r(j).persist]
   THEN
   array_of_requests[proc][j]
   ELSE ![MPI_REQUEST_NULL]]
∧ outcount' = [outcount except ![proc] = 0]
∧ UNCHANGED {array_of_indices, array_of_statuses, requests, array_of_requests}
∨ ∀ i ∈ 0..(incount – 1) : ![MPI_REQUEST_NULL]
∨ ![requests[proc][array_of_requests[proc][i]].active]
∧ outcount' = [outcount except ![proc] = ![MPI_UNDEFINED]]
∧ UNCHANGED {group, communicator, bufsize, message_buffer, requests, initialized, collective}

Section 3.8 Probe and Cancel

What happens in the following scenario: 1: send 2: probe 1: cancel 2: recv
Probe for a message. Nonblocking; note the leading if

\[ \text{MPI}_I\text{probe}(\text{source}, \text{tag}, \text{comm}, \text{flag}, \text{status}, \text{return}, \text{proc}) \]

\[ \Delta = \wedge \text{Assert}([\text{initialized}] = \text{"initialized"}. \quad 200.10 - 200.12 \]

"Error: MPI\_Testany called with proc not in initialized state.")

\[ \wedge \text{IF } \exists i \in (0..(N - 1)) : \quad 51.39 - 51.41 \]
\[ \exists j \in (1..\text{Len}([\text{requests}[i]])) : \]
\[ \text{LET } m = \text{requests}[i][j].\text{messageIN} \]
\[ \wedge \forall m.\text{src} = \text{source} \]
\[ \wedge \forall m.\text{msgtag} = \text{tag} \]
\[ \wedge \forall \text{tag} = \text{MPI\_ANY\_TAG} \]
\[ \wedge m.\text{universe} = \text{comm} \quad \text{unique across space/time – not required by standard} \]
\[ \wedge m.\text{state} = \text{"send"} \quad 51.41 - 51.42 \]
\[ \wedge \text{requests}[i][j].\text{active} \quad 51.41 - 51.42 \]
\[ \wedge \neg \text{requests}[i][j].\text{transmitted} \]
\[ \wedge \neg \text{requests}[i][j].\text{cancelled} \]

THEN

\[ \exists i \in (0..(N - 1)) : \quad 51.39 - 51.41 \]
\[ \exists j \in (1..\text{Len}([\text{requests}[i]])) : \]
\[ \text{LET } m = \text{requests}[i][j].\text{messageIN} \]
\[ \wedge \forall m.\text{src} = \text{source} \]
\[ \wedge \forall m.\text{msgtag} = \text{tag} \]
\[ \wedge \forall \text{tag} = \text{MPI\_ANY\_TAG} \]
\[ \wedge m.\text{universe} = \text{comm} \quad \text{unique across space/time – not required by standard} \]
\[ \wedge m.\text{state} = \text{"send"} \quad 51.41 - 51.42 \]
\[ \wedge \text{requests}[i][j].\text{active} \quad 51.41 - 51.42 \]
\[ \wedge \neg \text{requests}[i][j].\text{transmitted} \]
\[ \wedge \neg \text{requests}[i][j].\text{cancelled} \]
\[ \wedge \forall k \in (1..\text{Len}([\text{requests}[i]])) : \quad \text{least match} \]
\[ \wedge \text{requests}[i][k].\text{active} \]
\[ \wedge \neg \text{requests}[i][k].\text{cancelled} \]
\[ \wedge \neg \text{requests}[i][k].\text{transmitted} \]
\[ \Rightarrow j \leq k \]
\[ \wedge \text{Memory}' = [\text{Memory EXCEPT ![proc]} = \]
\[ \quad \forall \text{loc} \in (1..\text{Len}([\text{Memory}[proc]])) \mapsto 51.42 \]
\[ \text{IF } \text{loc} = \text{Status\_Cancelled}(\text{status}) \]
\[ \text{THEN FALSE} \]
\[ \text{ELSE} \]
\[ \text{IF } \text{loc} = \text{Status\_Count}(\text{status}) \]
\[ \text{THEN } m.\text{numelements} \]
\[ \text{ELSE} \]
\[ \text{IF } \text{loc} = \text{Status\_Source}(\text{status}) \]
\[ \text{THEN } m.\text{src} \]
\[ \text{ELSE} \]
\[
\text{if } \text{loc} = \text{Status\_Tag}(\text{status}) \\
\quad \text{then } m.\text{msgtag} \\
\text{else} \\
\quad \text{if } \text{loc} = \text{Status\_Err}(\text{status}) \\
\quad \quad \text{then requests}[i][j].\text{error} \\
\quad \text{else Memory}[\text{proc}][\text{loc}] \\
\text{except } ![\text{flag}] = \text{TRUE}] \\
\text{else} \\
\quad \wedge \text{Memory}' = [\text{Memory \ except \ ![proc]} = [\Theta \ \text{except \ ![flag}] = \text{FALSE}]) \\
\wedge \text{UNCHANGED}\ (\text{group, communicator, buflen, message_buffer, requests, initialized, collective})
\]\n
Wait on a probe for a message. 52.24 – 52.25

\[
\text{MPI\_Probe(source, tag, comm, status, return, proc) } \triangleq \\
\quad \wedge \text{Assert(initialized}[\text{proc}] = \text{"initialized"}, \quad 200.10 – 200.12 \quad \text{"Error: MPI\_Testany called with proc not in initialized state."})
\]

\[
\wedge \exists i \in (0 \ldots (N - 1)) : \\
\exists j \in (1 \ldots \text{Len(requests}[i])] : \\
\quad \text{LET } m \overset{\Delta}{=} \text{requests}[i][j].\text{messageIN} \\
\wedge \forall m.\text{src} = \text{source} \\
\wedge \forall m.\text{msgtag} = \text{tag} \\
\wedge \forall \text{tag} = \text{MPI\_ANY\_TAG} \\
\wedge m.\text{universe} = \text{comm} \quad \text{unique across space/time – not required by standard} \\
\wedge m.\text{state} = \text{"send"} \\
\wedge \text{requests}[i][j].\text{active} \\
\wedge \neg \text{requests}[i][j].\text{transmitted} \\
\wedge \neg \text{requests}[i][j].\text{cancelled} \\
\wedge \forall k \in (1 \ldots \text{Len(requests}[i])] : \\
\quad \text{LET } m \overset{\Delta}{=} \text{requests}[i][k].\text{active} \\
\wedge \neg \text{requests}[i][k].\text{transmitted} \\
\wedge \neg \text{requests}[i][k].\text{cancelled} \\
\Rightarrow j \leq k \\
\wedge \text{Memory}' = [\text{Memory \ except \ ![proc]} = [\Theta \ \text{except \ ![flag}] = \text{FALSE}]) \\
\text{if } \text{loc} \in 1 \ldots \text{Len(Memory}[\text{proc}]) \mapsto 51.42 \\
\text{if } \text{loc} = \text{Status\_Cancelled}(\text{status}) \\
\quad \text{then requests}[i][j].\text{cancelled} \wedge \neg \text{requests}[i][j].\text{transmitted} \\
\text{else} \\
\quad \text{if } \text{loc} = \text{Status\_Count}(\text{status}) \\
\quad \quad \text{then } m.\text{nulelements} \\
\text{else} \\
\quad \text{if } \text{loc} = \text{Status\_Source}(\text{status}) \\
\quad \quad \text{then } m.\text{src} \\
\text{else} \\
\quad \text{if } \text{loc} = \text{Status\_Tag}(\text{status})
then m.msgtag
ELSE
  IF loc = Status_Err(status)
  THEN requests[i][j].error
  ELSE Memory[proc][loc]]
∧ UNCHANGED {group, communicator, bufsize, message_buffer, requests, initialized, collective}

Cancel an active request.

What do you do when the request is MPI_REQUEST_NULL?

MPI_Cancel(request, return, proc) ≡ 54.8 - 54.10
∧ requests' = [requests EXCEPT ![proc] =
  [0 EXCEPT ![Memory[proc][request]] =
  [0 EXCEPT .cancelled = TRUE]]]
∧ UNCHANGED {group, communicator, bufsize, message_buffer, requests, initialized, collective}
∧ UNCHANGED {Memory}

Test whether a request was cancelled successfully.

MPI_Test_cancelled(status, flag, return, proc) ≡ 54.46 - 55.1
∧ Memory' = [Memory EXCEPT ![proc] = [0 EXCEPT ![flag] = Memory[proc][Status_Cancelled(status)]]]
∧ UNCHANGED {group, communicator, bufsize, message_buffer, requests, initialized, collective}
∧ UNCHANGED {Memory}

Section 3.9 Persistent communication requests

Create a persistent standard mode send request.

MPI_Send_init(buf, count, datatype, dest, tag, comm, request, return, proc) ≡
  “Error: MPI_Send_init called with proc not in initialized state.”)
∧ requests' = [requests EXCEPT ![proc] = 56.4 - 56.5
LET msg ≡ [addr ↦→ buf ,
  src ↦→ group[proc][communicator[proc][comm].group].ranking[proc],
  dest ↦→ dest ,
  msgtag ↦→ tag ,
  dtype ↦→ datatype ,
  numelements ↦→ count ,
  universe ↦→ comm ,
  state ↦→ “send”]
IN
  0 o ⟨Make_request(0, FALSE, FALSE, FALSE, FALSE, “send”, TRUE, {}, FALSE, msg)⟩]
∧ Memory' = [Memory EXCEPT ![proc] = [0 EXCEPT ![request] = Len(requests[proc]) + 1]]
∧ UNCHANGED {group, communicator, bufsize, message_buffer, requests, initialized, collective}

Create a persistent buffered mode send request.


\[
\text{MPI\_Bsend\_init}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm}, \text{request}, \text{return}, \text{proc}) \triangleq \wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{“initialized”}, 200.10 - 200.12
\]

\[
\text{“Error: MPI\_Bsend\_init called with proc not in initialized state.”}
\]

\[
\wedge \text{requests}' = [\text{requests} \text{ EXCEPT } ![\text{proc}] = 56.26
\]

\[
\text{LET msg} \triangleq \begin{cases}
\text{addr} & \mapsto \text{buf} \\
\text{src} & \mapsto \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{ranking}[\text{proc}], \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{namelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{“send”}
\end{cases}
\]

\[
\oplus \circ (\text{Make\_request}(0, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{“bsend"}, \text{TRUE}, [], \text{FALSE}, \text{msg}))]
\]

\[
\wedge \text{Memory}' = [\text{Memory EXCEPT } ![\text{proc}] = [\oplus \text{EXCEPT } ![\text{request}] = \text{Len}(\text{requests}[\text{proc}]) + 1]
\]

\[
\wedge \text{UNCHANGED} \langle \text{group, communicator, bufsize, message\_buffer, initialized, collective} \rangle
\]

Create a persistent synchronous mode send request.

\[
\text{MPI\_Ssend\_init}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm}, \text{request}, \text{return}, \text{proc}) \triangleq \wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{“initialized”}, 200.10 - 200.12
\]

\[
\text{“Error: MPI\_Ssend\_init called with proc not in initialized state.”}
\]

\[
\wedge \text{requests}' = [\text{requests} \text{ EXCEPT } ![\text{proc}] = 56.46
\]

\[
\text{LET msg} \triangleq \begin{cases}
\text{addr} & \mapsto \text{buf} \\
\text{src} & \mapsto \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{ranking}[\text{proc}], \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{namelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{“send”}
\end{cases}
\]

\[
\oplus \circ (\text{Make\_request}(0, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{“ssend"}, \text{TRUE}, [], \text{FALSE}, \text{msg}))]
\]

\[
\wedge \text{Memory}' = [\text{Memory EXCEPT } ![\text{proc}] = [\oplus \text{EXCEPT } ![\text{request}] = \text{Len}(\text{requests}[\text{proc}]) + 1]
\]

\[
\wedge \text{UNCHANGED} \langle \text{group, communicator, bufsize, message\_buffer, initialized, collective} \rangle
\]

Create a persistent ready mode send request.

\[
\text{MPI\_Rsend\_init}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm}, \text{request}, \text{return}, \text{proc}) \triangleq \wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{“initialized”}, 200.10 - 200.12
\]

\[
\text{“Error: MPI\_Rsend\_init called with proc not in initialized state.”}
\]

\[
\wedge \text{requests}' = [\text{requests} \text{ EXCEPT } ![\text{proc}] = 57.18
\]

\[
\text{LET msg} \triangleq \begin{cases}
\text{addr} & \mapsto \text{buf} \\
\text{src} & \mapsto \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{ranking}[\text{proc}], \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{namelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{“send”}
\end{cases}
\]
\[
\begin{align*}
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{nulelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{“send”} \\
\end{align*}
\]

\[
\circ \text{ (Make request}(0, \text{false}, \text{false}, \text{false}, \text{false}, \text{false}, \text{“rsend”}, \text{true}, (), \text{false}, \text{msg}))\]

\[
\wedge \text{Memory’} = \left[ \text{Memory EXCEPT } ![\text{proc}] = [\circ \text{ EXCEPT } ![\text{request}] = \text{Len}(\text{requests}[\text{proc}]) + 1]\right] \\
\wedge \text{UNCHANGED } (\text{group, communicator, bufsize, message buffer, initialized, collective})
\]

Create a persistant receive request.
\[
\text{MPI} \text{_Recv} \text{init}(\text{buf}, \text{count, datatype, source, tag, comm, request, return, proc) } \triangleq \\
\wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{“initialized”}, 200.10 – 200.12 \\
\quad \text{“Error: MPI}_\text{Recv} \text{init called with proc not in initialized state.”}) \\
\wedge \text{requests’} = [\text{requests EXCEPT } ![\text{proc}] = 57.39 \\
\text{LET msg } \triangleq [\text{addr } \mapsto \text{buf}, \\
\quad \text{src } \mapsto \text{source}, \\
\quad \text{dest } \mapsto \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{ranking}[\text{proc}], \\
\quad \text{msgtag } \mapsto \text{tag}, \\
\quad \text{dtype } \mapsto \text{datatype}, \\
\quad \text{nulelements } \mapsto \text{count}, \\
\quad \text{universe } \mapsto \text{comm}, \\
\quad \text{state } \mapsto \text{“recv”}] \\
\]

\[
\circ \text{ (Make request}(0, \text{false}, \text{false}, \text{false}, \text{false}, \text{false}, \text{“recv”}, \text{true}, (), \text{false}, \text{msg}))\]

\[
\wedge \text{Memory’} = \left[ \text{Memory EXCEPT } ![\text{proc}] = [\circ \text{ EXCEPT } ![\text{request}] = \text{Len}(\text{requests}[\text{proc}]) + 1]\right] \\
\wedge \text{UNCHANGED } (\text{group, communicator, bufsize, message buffer, initialized, collective})
\]

Start a persistant communication.
\[
\text{MPI} \text{_Start}(\text{request, return, proc) } \triangleq \\
\wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{“initialized”}, 200.10 – 200.12 \\
\quad \text{“Error: MPI}_\text{Start called with proc not in initialized state.”}) \\
\wedge \text{Assert}(\neg \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{request}]].\text{active}, 58.9 \\
\quad \text{“Error: MPI}_\text{Start tried to start a request that is already active.”}) \\
\wedge \text{Assert}(\text{Memory}[\text{proc}][\text{request}] \neq \text{MPI}_\text{REQUEST}_\text{NULL}, \\
\quad \text{“Error: MPI}_\text{Start tried to start a request that is null.”}) \\
\wedge \text{Assert}(\text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{request}]].\text{ctype} = \text{“rsend”} \Rightarrow 58.10 – 58.11 \\
\quad \exists j \in (0 \ldots (N - 1)) : \\
\quad \exists k \in (1 \ldots \text{Len}(\text{requests}[j])) : \\
\]

31
Start a list of persistant communications.

Can you start many bsends with only enough buffering for a subset of the sends? –maybe no

Can you start many rsends with only one matching receive posted? –maybe yes

MPI_Startall(count, array_of_requests, return, proc) ⇐

let m = \{ x ∈ (0 . . (count − 1)) : requests[proc][Memory[proc][array_of_requests + x]].ctype = “bsend” \} in


“Error: MPL_Startall called with proc not in initialized state.”)

∧ Assert(∀ i ∈ (0 . . (count − 1)) : ¬requests[proc][array_of_requests[i]].active,

“Error: MPL_Startall called with some request already active.”)

∧ Assert(∀ i ∈ (0 . . (count − 1)) : array_of_requests[i] ≠ MPI_REQUEST_NULL,

“Error: MPL_Startall called with some request null.”)

∧ Assert(∀ i ∈ (0 . . (count − 1)) :

requests[proc][array_of_requests[i]].ctype = “rsend” ⇒ 58.10 – 58.11

∃ j ∈ (0 . . (N − 1)) :

∃ k ∈ (1 . . Len(requests[j])) :

∧ requests[j][k].active

∧ ¬requests[j][k].transmitted

∧ ¬requests[j][k].cancelled

∧ Match(requests[proc][array_of_requests[i]].message, requests[j][k].message),
“Error: MPI_Start tried to start a rsend request when no matching message exists.”
∧ \text{Assert}(\forall i \in (0 .. (\text{count} - 1)) : 
\text{requests}[\text{proc}][\text{array_of_requests}[i]].\text{ctype} = \text{"bsend"} \Rightarrow 
\text{message_buffer}[\text{proc}] + \text{Cardinality}(m) < \text{bufsize}[\text{proc}].
“Error: MPI_Start tried to start a bsend request when insufficient buffering was available.”
∧ \text{Assert}(\forall i \in (0 .. (\text{count} - 1)) : 
\text{requests}[\text{proc}][\text{array_of_requests}[i]].\text{persist} = \text{57.44} - \text{57.45}, \text{58.8}
“Error: MPI_Start tried to start a non-persistant request.”
∧ \text{requests}' = [\text{requests} \text{EXCEPT} ![\text{proc}] = [i \in (1 .. \text{Len}(\text{requests}[\text{proc}])) \mapsto 
\text{IF} \exists j \in (0 .. (\text{count} - 1)) : \text{array_of_requests}[j] = i 
\text{THEN} \text{58.9} 
\text{requests}[\text{proc}][i] \text{EXCEPT}
\text{!.active} = \text{TRUE},
\text{!.started} = \text{TRUE},
\text{!.transmitted} = \text{FALSE},
\text{!.cancelled} = \text{FALSE}]
\text{ELSE}
\text{requests}[\text{proc}][i]]
∧ \text{message_buffer}' = [\text{message_buffer} \text{EXCEPT} ![\text{proc}] = @ + \text{Cardinality}(m)]
∧ \text{UNCHANGED} \langle \text{group, communicator, bufsize, initialized, collective} \rangle
∧ \text{UNCHANGED} \langle \text{Memory} \rangle

Section 3.10 Send-receive

Can this be done with only one transition? I don’t think so.
\text{MPI_Sendrecv}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{dest},
\text{sendtag}, \text{recvbuf}, \text{recvcount}, \text{recvtype},
\text{source}, \text{recvtag}, \text{comm}, \text{status}) \triangleq

Section 4.3 Barrier

\text{MPI_BARRIER_INIT}(\text{comm}, \text{return}, \text{proc}) \triangleq
∧ \forall \forall \text{collective}\text{|communicator}[\text{proc}][\text{comm}].\text{collective}.\text{state} = \text{"vacant"}
∧ \text{collective}' = [\text{collective} \text{EXCEPT} ![\text{communicator}[\text{proc}][\text{comm}].\text{collective}] = 
[@ \text{EXCEPT}
\text{!.participants} = @ \cup \{\text{proc}\},
\text{!.type} = \text{"barrier"},
\text{!.state} = \text{"in"}]
∧ \text{collective}\text{|communicator}[\text{proc}][\text{comm}].\text{collective}.\text{state} = \text{"in"}
∧ \text{proc} \notin \text{collective}\text{|communicator}[\text{proc}][\text{comm}].\text{collective}.\text{participants}
∧ \text{collective}' = [\text{collective} \text{EXCEPT} ![\text{communicator}[\text{proc}][\text{comm}].\text{collective}] = 
[@ \text{EXCEPT} \text{!.participants} = @ \cup \{\text{proc}\}]]
∧ \text{UNCHANGED} \langle \text{group, communicator, bufsize, message_buffer, requests, initialized} \rangle
∧ UNCHANGED ⟨Memory⟩

\[ MPI_{-}Barrier\_wait(comm, return, proc) ≜ \]
∧ ∨ ∧ collective[communicator[proc][comm], collective, participants = group[proc][communicator[proc][comm], collective, participants]
∧ proc ∈ collective[communicator[proc][comm], collective, participants]
∧ collective[communicator[proc][comm], collective, state = "in"
∧ collective' = [collective EXCEPT ![communicator[proc][comm], collective] =
[@ EXCEPT
  ![participants = @ \ {proc},
  ![state = "out"]]
∨ ∧ proc ∈ collective[communicator[proc][comm], collective, participants]
∧ collective[communicator[proc][comm], collective, state = "out"
∧ IF collective[communicator[proc][comm], collective, participants = {proc}
THEN
  collective' = [collective EXCEPT ![communicator[proc][comm], collective] =
[@ EXCEPT
  ![participants = {},
  ![state = "vacant"]]
ELSE
  collective' = [collective EXCEPT ![communicator[proc][comm], collective] =
[@ EXCEPT ![participants = @ \ {proc}]]]
∧ UNCHANGED ⟨group, communicator, bufsize, message\_buffer, requests, initialized⟩
∧ UNCHANGED ⟨Memory⟩

Section 5.3.1 Group Accessors

No text description.

\[ MPI_{-}Group\_size(gr, size, return, proc) ≜ \]
∧ Assert(initialized[proc] = "initialized", \quad 200.10 – 200.12
  "Error: MPI\_Group\_size called with proc not in initialized state."
) ∧ Memory’ = [Memory EXCEPT ![proc] = ![size] = group[proc][gr].size]
∧ UNCHANGED ⟨group, communicator, bufsize, message\_buffer, requests, initialized, collective⟩

No text description.

\[ MPI_{-}Group\_rank(gr, rank, return, proc) ≜ \]
∧ Assert(initialized[proc] = "initialized", \quad 200.10 – 200.12
  "Error: MPI\_Group\_rank called with proc not in initialized state."
) ∧ Memory’ = [Memory EXCEPT ![proc] = ![rank] =
  IF proc ∈ group[proc][gr].members THEN group-ranking[proc] ELSE MPI\_UNDEFINED]
∧ UNCHANGED ⟨group, communicator, bufsize, message\_buffer, requests, initialized, collective⟩

\[ MPI_{-}Group\_translate\_ranks(group1, n, ranks1, group2, ranks2, return, proc) ≜ \]

34
   “Error: MPI_Group_translate_ranks called before MPI_Init.”)
∧ Assert(group1 ∈ MPI_COMM_WORLD . . . (MPI_COMM_WORLD + MAX_GROUP),
   “Error: MPI_Group_translate_ranks called with invalid handle for group1.”)
∧ Assert(group2 ∈ MPI_COMM_WORLD . . . (MPI_COMM_WORLD + MAX_GROUP),
   “Error: MPI_Group_translate_ranks called with invalid handle for group2.”)
∧ Assert(n = Cardinality(DOMAIN ranks1), 138.3
   “Error: MPI_Group_translate_ranks called with invalid n.”)
∧ Memory′ = [Memory except ![proc] = 
   [i ∈ 1 . . Len(Memory[proc]) ↦
     if i ∈ ranks2 . . . (ranks2 + n)
     THEN group[proc][group2], ranking = group[proc][group1].inveranking[ranks1[i]]
     ELSE Memory[proc][i]]] not quite right as there is no possibility of MPI_UNDEFINED being assigned.
∧ unchanged mpi_vars

MPI_Group_compare(group1, group2, result, return, proc) ≡
   “Error: MPI_Group_compare called before MPI_Init.”)
∧ result′ = [result except ![proc] = 
   if ∀ group1 = group2 138.31
   ∀ group[proc][group1].members = group[proc][group2].members
   ∧ group[proc][group1].ranking = group[proc][group2].ranking
   THEN 
   MPI_IDENT
   ELSE 138.32
   if ∀ group[proc][group1].members = group[proc][group2].members
   ∧ group[proc][group1].ranking ≠ group[proc][group2].ranking
   THEN 
   MPI_SIMILAR
   ELSE
   MPI_UNEQUAL] 138.33
∧ unchanged mpi_vars

Section 5.3.2 Group Constructors

MPI_Comm_group(comm, gr, return, proc) ≡
   “Error: MPI_Comm_group called before MPI_Init.”)
∧ Memory′ = [Memory except ![proc] = [@ except ![gr] = communicator[proc][comm].group]] 139.19
∧ unchanged mpi_vars

MPI_Group_union(group1, group2, newgroup, return, proc) ≡
   “Error: MPI_Group_union called before MPI_Init.”)
∧ ∃ i ∈ 0 . . (MAX_GROUP – 1) :
   let newmembers ≡ group[proc][group1].members ∪
\[
\text{in}
\]
\[
\land \text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\text{\tex
\begin{align*}
\wedge \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer, requests, initialized, collective} \rangle
\end{align*}

\textbf{Section 7.5 Startup}

199.12 – 199.17
Initialize the participation of this process within a distributed computation.

\texttt{MPI\_Init(argc, argv, return, proc) :=} \\
\wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{"uninitialized"}, \text{199.12}) \quad \text{"MPI\_Init called with proc not in uninitialized state."}
\wedge \text{initialized}' = [\text{initialized} \text{EXCEPT ![proc] = \"initialized\}] \quad \text{199.13}
\wedge \text{UNCHANGED } \langle \text{Memory} \rangle
\wedge \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer, requests, collective} \rangle

Finalize the participation of this process within a distributed computation.

Do buffered operations complete when the message is transmitted or buffered?

\texttt{MPI\_Finalize(return, proc) :=} \\
\wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \text{200.10 – 200.12}) \quad \text{"Error: MPI\_Finalize called with proc not in initialized state."}
\wedge \text{assert}((\forall i \in (1 . . \text{Len(requests}[\text{proc}]))) : \text{199.47}) \neg\text{requests}[\text{proc}][i].\text{active}, \quad \text{"Error: MPI\_Finalize called when some message was still active."}
\wedge \text{assert}(\text{bufsize}[\text{proc}] = 0, \text{199.48}) \quad \text{"Error: MPI\_Finalize called before the buffer is detached."}
\wedge \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer, requests, collective} \rangle
\wedge \text{UNCHANGED } \langle \text{Memory} \rangle

Determine whether \texttt{MPI\_Init} has been called.

\texttt{MPI\_Initialized(flag, return, proc) :=} \\
\wedge \text{MEMORY}' = [\text{MEMORY EXCEPT ![proc] = [@ EXCEPT ![flag] = \text{"initialized"}] 200.2} \quad \text{"Error: MPI\_Finalize called before the buffer is detached."}
\wedge \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer, requests, collective} \rangle
\wedge \text{UNCHANGED } \langle \text{Memory} \rangle

"Best effort to clean up"

\texttt{MPI\_Abort(comm, errorcode, return, proc) :=} \\
\forall p \in (0 . . (N - 1)) : \\
\forall m \in (1 . . \text{Len(requests}[p]) : \\
\wedge \text{requests}[p][m].\text{active} \\
\wedge \neg\text{requests}[p][m].\text{transmitted} \\
\Rightarrow \text{requests}[p][m]' = [\text{requests}[p][m] \text{EXCEPT ![\text{cancelled} = \text{true}]}
\wedge \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer, requests, initialized, collective} \rangle