Formal Specification of MPI 2.0: Case Study in Specifying a Practical Concurrent Programming API

Guodong Li, Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby

UUCS-09-003

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

May 28, 2009

Abstract

We describe the first formal specification of a non-trivial subset of MPI, the dominant communication API in high performance computing. Engineering a formal specification for a non-trivial concurrency API requires the right combination of rigor, executability, and traceability, while also serving as a smooth elaboration of a pre-existing informal specification. It also requires the modularization of reusable specification components to keep the length of the specification in check. Long-lived APIs such as MPI are not usually 'textbook minimalistic' because they support a diverse array of applications, a diverse community of users, and have efficient implementations over decades of computing hardware. We choose the TLA+ notation to write our specifications, and describe how we organized the specification of 150 of the 300 MPI 2.0 functions. We detail a handful of these functions in this paper, and assess our specification with respect to the aforesaid requirements. We close with a description of possible approaches that may help render the act of writing, understanding, and validating specifications much more productive.

1 Introduction

The Message Passing Interface (MPI, [32]) library has become a *de facto* standard in HPC, and is being actively developed and supported through several implementations [9, 31, 7]. However, it is well known that even experienced programmers misunderstand MPI APIs partially because they are described in natural languages. The behavior of APIs observed through ad hoc experiments on actual platforms is not a conclusive or comprehensive description of the standard. A formalization of the MPI standard will help users avoid misunderstanding the semantics of MPI functions. However, formal specifications, as currently written and distributed, are inaccessible to most practitioners.

In our previous work [22], we presented the formal specification of around 30% of the 128 MPI-1.0 functions (mainly for point-to-point communication) in a specification language TLA+ [33]. TLA+ enjoys wide usage in industry by engineers (e.g. in Microsoft [34] and Intel). The TLA+ language is easy to learn. A new user is able to understand our specification and start practicing it after a half-an-hour tutorial. Additionally, in order to help practitioners access our specification, we built a C front-end in the Microsoft Visual Studio (VS) parallel debugger environment, through which users can submit and run short (perhaps tricky) MPI programs with embedded assertions (called litmus tests). A short litmus test may exhibit a high degree of interleaving and its running will reveal the nuances of the semantics of the MPI functions involved. Such tests are turned into TLA+ code and run through the TLC model checker [33], which searches all the reachable states to check properties such as deadlocks and user-defined invariants. This permits practitioners to play with (and find holes in) the semantics in a formal setting.

While we have demonstrated the merits of our previous work ([22]), this paper, the journal version of our poster paper [15], handles far more details including those pertaining to data transfers. In this work, we have covered much of MPI-2.0 (has over 300 API functions, as opposed to 128 for MPI-1.0). In addition, this new work provides a rich collection of tests that help validate our specifications. It also modularizes the specification, permitting reuse.

Model Validation. In order to make our specification be faithful to the English description, we (i) organize the specification for *easy traceability*: many clauses in our specification are cross-linked with [32] to particular page/line numbers; (ii) provide comprehensive unit tests for MPI functions and a rich set of litmus tests for tricky scenarios; (iii) relate aspects of MPI to each other and verify the self-consistency of the specification (see Section 4.11); and (iv) provide a programming and debugging environment based on TLC, Phoenix, and Visual Studio to help engage expert MPI users (who may not be formal methods experts) into experimenting with our semantic definitions.

The structure of this paper is as follows. We first discuss the related work on formal specifications of large standards and systems; other work on applying formal methods to verify MPI programs is also discussed. Then we give a motivating example and introduce the specification language TLA+. This example illustrates that vendor MPI implementations do not capture the nuances of the semantics of an MPI function. As the main part of this paper, the formal specification is given in Section 4, where the operational semantics of representative MPI functions are presented in a mathematical language abstracted from TLA+. In Section 5 we describe a C MPI front-end that translates MPI programs written in C into TLA+ code, plus the verification framework that helps users execute the semantics. Finally we give the concluding remarks. In the appendix we give an example to show how the formal semantics may help the rigid analysis of MPI programs — we prove formally the definition of a precedence relation is correct, which is the base of a dynamic partial order reduction algorithm.

2 Related Work

The idea of writing formal specifications of standards and building executable environments is a vast area.

The IEEE Floating Point standard [12] was initially conceived as a standard that helped minimize the danger of non-portable floating point implementations, and now has incarnations in various higher order logic specifications (e.g., [10]), finding routine applications in *formal proofs* of modern microprocessor floating point hardware circuits. Formal specifications using TLA+ include Lamport's Win32 Threads API specification [34] and the RPC Memory Problem specified in TLA+ and formally verified in the Isabelle theorem prover by Lamport, Abadi, and Merz [1]. In [13], Jackson presents a lightweight object modeling notation called Alloy, which has tool support [14] in terms of formal analysis and testing based on Boolean satisfiability methods.

Bishop et al [3, 4] formalized in the HOL theorem prover [20] three widely-deployed implementations of the TCP protocol: FreeBSD 4.6-RELEASE, Linux 2.4.20-8, and Windows XP Professional SP1. Analogous to our work, the specification of the interactions between objects are modeled as transition rules. The fact that implementations other than the standard itself are specified requires repeating the same work for different implementations.

In order to validate the specification, they perform a vast number of conformance tests: test programs in a concrete implementation are instrumented and executed to generate execution trances, each of which is then symbolically executed with respect to the formal operational semantics. Constraint solving is used to handle non-determinism in picking rules or determining possible values in a rule. Compared with their work, we also rely on testing for validation check. However, since it is the standard that we formalize, we need to design and write all the test cases by hand.

Norrish [19] formalized in HOL [20] a structural operational semantics and a type system of the majority of the C language, covering the dynamic behavior of C programs. Semantics of expressions, statements and declarations are modeled as transition relations. The soundness of the semantics and the type system is proved formally. Furthermore, in order to verify properties of programs, a set of Hoare rules are derived from the operational semantics. In contrast, the notion of type system does not appear in our specification because TLA+ is an untyped language.

Each of the formal specification frameworks mentioned above solves modeling and analysis issues specific to the object being described. In our case, we were initially not sure how to handle the daunting complexity of MPI nor how to handle its modeling, given that there has only been very limited effort in terms of formal characterization of MPI.

Georgelin and Pierre [8] specify some of the MPI functions in LOTOS [6]. Siegel and Avrunin [29] describe a finite state model of a limited number of MPI point-to-point operations. This finite state model is embedded in the SPIN model checker [11]. They [30] also support a limited partial-order reduction method – one that handles wild-card communications in a restricted manner, as detailed in [24]. Siegel [28] models additional 'non-blocking' MPI primitives in Promela. Our own past efforts in this area are described in [2, 21, 25, 23]. None of these efforts: (i) approach the number of MPI functions we handle, (ii) have the same style of high level specifications (TLA+ is much closer to mathematical logic than finite-state Promela or LOTOS models), (iii) have a model extraction framework starting from C/MPI programs, and (iv) have a practical way of displaying error traces in the user's C code.

3 Motivation

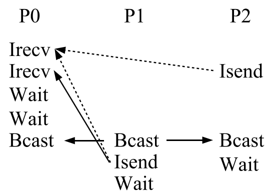
MPI is a standardlized and portable message-passing system defining a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran, C or C++. Versions 1.0 and 2.0 were released in 1994 and 1997 respectively.

Currently more than a dozen implementations exist, on a wide variety of platforms. All segments of the parallel computing communicty including vendors, library writers and application scientists will benefit from a formal specification of this standard.

3.1 Motivating Example

MPI is a portable standard and has a variety of implementations [9, 31, 7]. MPI programs are often manually or automatically (e.g., [5]) re-tuned when ported to another hardware platform, for example by changing its basic functions (e.g., `MPI_Send`) to specialized versions (e.g., `MPI_Isend`). In this context, it is crucial that the designers performing code tuning are aware of the very fine details of the MPI semantics. Unfortunately, such details are far from obvious. For illustration, consider the following MPI pseudo-code involving three processes:

$P0$ $\texttt{MPI\_Irecv}(rcvbuf1, *, req1);$
$\texttt{MPI\_Irecv}(rcvbuf2, \text{from } 1, req2);$
$\texttt{MPI\_Wait}(req1);$
$\texttt{MPI\_Wait}(req2);$
$\texttt{MPI\_Bcast}(revbuf3, root = 1);$
$P1$ $sendbuf1 = 10;$
$\texttt{MPI\_Bcast}(sendbuf1, root = 1);$
$\texttt{MPI\_Isend}(sendbuf2, \text{to } 0, req);$
$\texttt{MPI\_Wait}(req);$
$P2$ $sendbuf2 = 20;$
$\texttt{MPI\_Isend}(sendbuf2, \text{to } 0, req);$
$\texttt{MPI\_Bcast}(recvbuf2, root = 1);$
$\texttt{MPI\_Wait}(req);$



Process 1 and 2 are designed to issue *immediate mode* sends to process 0, while Process 0 is designed to post two immediate-mode receives. The first receive is a wildcard receive that may match the send from P1 or P2. These processes also participate in a broadcast communication with P1 as the root. Consider some simple questions pertaining to the execution of this program:

1. *Is there a case where a deadlock is incurred?* If the broadcast is synchronizing such that the call at each process is blocking, then the answer is 'yes', since P0 cannot complete the broadcast before it receives the messages from P1 and P2, while P1 will not isend the message until the broadcast is complete. On the other hand, this deadlock will not occur if the broadcast is non-synchronizing. As in an actual MPI implementation `MPI_Bcast` may be implemented as synchronizing or non-synchronizing, this deadlock may not be observed through ad hoc experiments on a vendor MPI library. Our specification takes both bases into consideration and always gives reliable answers.

2. *Suppose the broadcast is non-synchronizing, is it possible that a deadlock occurs?* The answer is 'yes', since P0 may first receive a message from P1, then get stuck waiting for another message from P1. Unfortunately, if we run this program in a vendor MPI implementation, P1 may receive messages first from P2 and then from P1. In this case no deadlock occurs. Thus it is possible that we will not encounter this deadlock even we run the program for 1,000 times. In contrast, the TLC model checker enumerates all execution possibilities and is guaranteed to detect this deadlock.

3. *Suppose there is no deadlock, is it guaranteed that rcvbuf1 in P0 will eventually contain the message sent from P2?* The answer is 'no', since P1's incoming messages may arrive out of order. However, running experiments on a vendor implementation may indicate that the answer is yes, especially when the message delivery delay from P1 to P0 is greater than that from P2 to P0. In our framework, we can add in P0 an assertion `rcvbuf1` == 20 right before the broadcast call. If it is possible under the semantics for other values to be assigned to these two variables, then the model checker will find the violation.

4. *Suppose there is no deadlock, when can the buffers be accessed?* Since all sends and receives use the immediate mode, the handles that these calls return have to be tested for completion using an explicit `MPI_Test` or `MPI_Wait` before the associated buffers are allowed to be accessed. Vendor implementations may not give reliable answer for this question. In contrast, we can move the assertions mentioned in the response to the previous question to any other point before the corresponding `MPI_waits`. The model checker then finds violations—meaning that the data cannot be accessed on the receiver until after the `wait`.

5. *Will the first receive always complete before the second at P0?* No such guarantee exists, as these are *immediate mode* receives which are guaranteed only to be *initiated* in program order. Again, the result obtained by observing the running of this program in a vendor implementation may not be accurate. In order to answer this question, we can reverse the order of the `MPI_Wait` commands. If the model checker does not find a deadlock then it is possible for the operations to complete in either order.

The MPI reference standard [32] is a non machine-readable document that offers English descriptions of the individual behaviors of MPI functions. It does not support any executable facility that helps answer the above kinds of simple questions in any tractable and reliable way. Running test programs, using actual MPI libraries, to reveal answers to the above kinds of questions is also futile, given that (i) various MPI implementations exploit the liberties of the standard by specializing the semantics in various ways, and (ii) it is possible that some executions of a test program are not explored in these actual implementations.

Thus we are motivated to write a formal, high-level, and executable standard specification for MPI 2.0. The availability of a formal specification allows formal analysis of MPI programs. For example, we have based on this formalization to create an efficient dynamic partial order reduction algorithm [26]. Moreover, the TLC model checker incorporated in our framework enables users to execute the formal semantic definitions and verify (simple) MPI programs.

3.2 TLA+ and TLC

The specification is written in TLA+ [33], a formal specification notation widely used in industry. It is a formal specification language based on (untyped) ZF set theory. Basically it combines the expressiveness of first order logic with temporal logic operators. TLA+ is particularly suitable for specifying and reasoning about concurrent and reactive systems.

TLC, a model checker for TLA+, explores all reachable states in the model defined by the system. TLC looks for a state (i.e. an assignment of values to variables) where (a) an invariant is not satisfied, (b) there are no exits (deadlocks), (c) the type invariant is violated, or (d) a user-defined TLA+ assertion is violated. When TLC detects an error, a minimal-length trace that leads to the bad state is reported (in our framework this trace turns into a Visual Studio debugger replay of the C source).

It is possible to port our TLA+ specification to other specification languages such as Alloy [13] and SAL [27]. We are working on a formalization of a small subset of MPI functions in SAL, which comes with state-of-the-art symbolic model checkers and automated test generators.

4 Specification

TLA+ provides basic modules for set, function, record, string and sequence. We first extend the TLA+ library by adding the definitions of advanced data structures including array, map, and ordered set (`oset`), which are used to model a variety of MPI objects. For instance, MPI groups and I/O files are represented as ordered sets.

The approximate sizes (without including comments and blank lines) of the major parts in the current TLA+ specification are shown in Table 1, where #funcs and #lines give the number of MPI functions and code lines respectively. We do not model functions

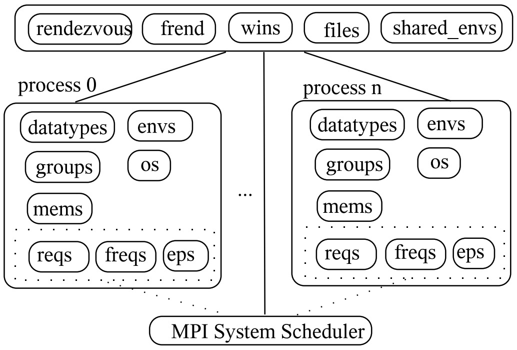


Figure 1: MPI objects and their interaction

whose behavior depends on the underlying operating system. For deprecated items (e.g., MPI_KEYVAL_CREATE), we only model their replacement (MPI_COMM_CREATE_KEYVAL).

Main Module	**#funcs(#lines)**
Point to Point Communication	35(800)
Userdefined Datatype	27(500)
Group and Communicator Management	34(650)
Intra Collective Communication	16(500)
Topology	18(250)
Environment Management in MPI 1.1	10(200)
Process Management	10(250)
One sided Communication	15(550)
Inter Collective Communication	14(350)
I/O	50(1100)
Interface and Environment in MPI 2.0	35(800)

Table 1. Size of the Specification (excluding comments and blank lines)

4.1 Data Structures

The data structures modeling explicit and opaque MPI objects are shown in Figure 1. Each process contains a set of local objects such as the local memory object `mems`. Multiple processes coordinate with each other through shared objects `rendezvous`, `wins`, and so on. The message passing procedure is simulated by the *MPI system scheduler (MSS)*, whose task includes matching requests at origins and destinations and performing message passing. MPI calls and the MSS are able to make transitions non-deterministically.

Request object `reqs` is used in point-to-point communications to initiate and complete messages. A message contains the source, destination, tag, data type, count and communicator handle. It carries the data from the origin to the target. Note that noncontiguous data is represented as (user-defined) datatypes. A similar file request object `freqs` is for parallel I/O communications.

A group is used within a communicator to describe the participants in a communication "universe". Communicators `comms` are divided into two kinds: intra-communicators each of which has a single group of processes, and inter-communicators each of which has two groups of processes. A communicator also includes virtual topology and other attributes.

A rendezvous is a place shared by the processes participating in a collective communication. A process stores its data to the rendezvous on the entry of the communication and fetches the data from the rendezvous on the exit. A similar `frend` object is for (shared) file operations.

For one-sided communications, epoches `epos` are used to control remote memory accesses; each epoch is associated with a "window", modeled by `wins`, which is made accessible to accesses by remote accesses. Similarly, a "file" supporting I/O accesses is shared by a group of processes.

Other MPI objects are represented as components in a shared environment `shared_envs` and local environments `envs`. The underlying operating system is abstracted as `os` in a limited sense, which includes those objects (such as physical files on the disk) visible to the MPI system. Since the physical memory at each process is an important object, we extract it from `os` and define a separate object `mems` for it.

4.2 Notations

We present our specification using notations extended and abstracted from TLA+.

4.2.1 TLA+

The basic concept in TLA+ is functions. A set of functions is expressed by $[domain \rightarrow range]$. Notation $f[e]$ represents the application of function f on e; and $[x \in S \mapsto e]$ defines the function f such that $f[x] = e$ for $x \in S$. For example, the function f_{double} that doubles the input natural number is given by $[x \in \mathbb{N} \mapsto 2 \times x]$ or $[1 \mapsto 2, 2 \mapsto 4, \ldots]$; and $f_{double}[4] = 8$.

For a n-tuple (or n-array) $\langle e_1, \cdots, e_n \rangle$, $e[i]$ returns its i^{th} component. It is actually a function mapping i to $e[i]$ for $1 \leq i \leq n$. Thus function f_{double} is equivalent to the tuple $\langle 2, 4, 6, 8, \cdots \rangle$. An ordered set consisting of n distinct elements is actually a n-tuple.

Notation $[f \texttt{ EXCEPT } ![e_1] = e_2]$ defines a function f' such that $f' = f$ except $f'[e_1] = e_2$. A @ appeared in e_2 represents the old value of $f[e_1]$. For example, $[f_{double} \texttt{ EXCEPT } ![3] = @ + 10]$ is the same as f_{double} except that it returns 16 when the input is 3. Similarly, $[r \texttt{ EXCEPT } !.h = e]$ represents a record r' such that $r' = r$ except $r'.h = e$, where $r.h$ returns the h-field of record r.

The basic temporal logic operator used to define transition relations is the next state operator, denoted using ' or *prime*. For example, $s' = [s \texttt{ EXCEPT } ![x] = e]$ indicates that the next state s' is equal to the original state s except that x's value is changed to e.

For illustration, consider a stop watch that displays hour and minute. A typical behavior of the clock is the sequence $(hr = 0, mnt = 0) \rightarrow (hr = 0, mnt = 1), \rightarrow, \cdots, \rightarrow, (hr = 0, mnt = 59), (hr = 1, mnt = 0), \rightarrow, \cdots$, where $(hr = 0, mnt = 1)$ is a state in which the hour and minute have the value 0 and 1 respectively.

The next-state relation is a formula expressing the relation between the values of hr and mnt in the old (first) state $time$ and new (second) state $time'$ of a step. It assert that mnt equals mnt + 1 except if mnt equals 59, in which case mnt is reset to 0 and hr is increased by 1.

$$\begin{array}{rl} time' = & \texttt{let } c = time[mnt] \neq 59 \texttt{ in} \\ & [time \texttt{ EXCEPT } ![mnt] = \texttt{if } c \text{ then } @ + 1 \texttt{ else } 0, \\ & \qquad\qquad ![hr] = \texttt{if } \neg c \texttt{ then } @ + 1 \texttt{ else } @] \end{array}$$

Additionally, we introduce some commonly used notations when defining the semantics of MPI functions.

$\Gamma_1 \diamond \Gamma_2$	the concatenation of queue Γ_1 and Γ_2
$\Gamma_1 \diamond x_k \diamond \Gamma_2$	the queue with x being the k^{th} element
ϵ	null value
α	an arbitrary value
$\top$ and $\bot$	boolean value $ture$ and $false$
$\Gamma_1 \sqsubseteq \Gamma_2$	Γ_1 is a sub-array (sub-queue) of Γ_2
$\overrightarrow{v}$	v is an array
$f \uplus \langle x, v \rangle$	a new function (map) f_1 such that $f_1[x] = v$ and $\forall y \neq x.\ f_1[y] = f[y]$
$f\|_x$	the index of element x in function f, $i.e.$ $f[f\|_x] = x$
$c \ ?\ e_1 \ :\ e_2$	`if` c `then` x `else` y
$\texttt{size}(f)$ or $\|f\|$	the number of elements in function f
$\texttt{remove}(f, k)$	remove from f the item at index k
$\texttt{unused\_index}(f)$	return an i such that $i \notin \texttt{DOM}(f)$

TLA+ allows us to specify operations in a declarative style. For illustration we show below a helper function used to implement the `MPI_COMM_SPLIT` primitive, where `DOM`, `RNG`, `CARD` return the domain, range and cardinality of a set respectively. This code directly formalizes the English description (see page 147 in [32]): "This function partitions the group into disjoint subgroups, one for each value of `color`. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by `key`, with ties broken according to their rank in the old group. When the process supply the color value `MPI_UNDEFINED`, a null communicator is returned." In contrast, such declarative specification cannot be done in the C language.

$$
\begin{array}{rl}
 & \texttt{Comm\_split}(group, \overrightarrow{colors}, \overrightarrow{keys}, proc) \doteq \\
1: & \texttt{let}\ rank = group|_{proc}\ \texttt{in} \\
2: & \texttt{if}\ \overrightarrow{colors}[rank] = \texttt{MPI\_UNDEFINED}\ \texttt{then}\ \texttt{MPI\_GROUP\_NULL} \\
3: & \texttt{else} \\
4: & \quad \texttt{let}\ s = \{k \in \texttt{DOM}(group) : \overrightarrow{colors}[k] = \overrightarrow{colors}[rank]\}\ \texttt{in} \\
5: & \quad \texttt{let}\ s_1 = \\
6: & \quad\ \ \texttt{choose}\ g \in [0 \mathrel{..} \texttt{CARD}(s) - 1 \rightarrow \texttt{DOM}(group)] : \\
7: & \quad\ \ \wedge\ \texttt{RNG}(g) = s \\
8: & \quad\ \ \wedge\ \forall i, j \in s : \\
9: & \qquad\qquad g|_i < g|_j \Rightarrow (\overrightarrow{keys}[i] < \overrightarrow{keys}[j]\ \vee\ \overrightarrow{keys}[i] = \overrightarrow{keys}[j]\ \wedge\ i < j) \\
12: & \quad\ \texttt{in}\ [i \in \texttt{DOM}(s_1) \mapsto group[s_1[i]]]
\end{array}
$$

After collecting the color and key information from all other processes, a process $proc$ calls this function to create the group of a new communicator. Line 1 calculates the rank of this process in the group; line 4 obtains a set of processes of the same color as $proc$'s; lines 5-11 sort this set in the ascending order of keys, with ties broken according to the ranks. For example, suppose $group = \langle 2, 5, 1 \rangle$, $\overrightarrow{colors} = 1, 0, 0$ and $\overrightarrow{keys} = \langle 0, 2, 1 \rangle$, then the call of this function at process 5 creates a new group $\langle 1, 5 \rangle$.

4.2.2 Operational Semantics

The formal semantics of an MPI function is modeled by a state transition. A system state consists of explicit and opaque objects mentioned above. We write $\texttt{obj}_p$ for the object `obj` at process p. For example, $\texttt{reqs}_p$ refers to the request object (for point-to-point communications) at process p.

We use notation $\mathrel{\overset{\circ}{=}}$ to define the semantics of an MPI primitive, and $\doteq$ to introduce an auxiliary function. The pre-condition $cond$ of a primitive, if exists, is specifies by "`requires` $\{cond\}$". An error occurs if this pre-condition is violated. In general a transition is expressed as a rule of format $\frac{guard}{action}$, where $guard$ specifies the requirement for the transistion to be triggered, and $action$ defines how the MPI objects are updated after the transition. When the guard is satisfied, the action is enabled and may be performed by the system. A null guard will be omitted, meaning that the transition is always enabled.

For instance, the semantics of `MPI_Buffer_detach` is shown below. The pre-condition says that buffer at process p must exist; the guard indicates that the call will block until all messages in the buffer have been transmitted (*i.e.* the buffer is empty); the action is to write the buffer address and the buffer size into the p's local memory, and deallocate the space taken by the buffer. The `buffer` locates in the `envs` object. A variable such as $buff$ is actually a reference to a location in the memory; in many cases we simply write $buff$ for $\mathtt{mems}_p[buff]$ for brevity.

$$\mathtt{MPI\_Buffer\_detach}(buff, size, p) \triangleq$$
$$\mathtt{requires}\ \{\mathtt{buffer}_p \neq \epsilon\}$$
$$\frac{\mathtt{buffer}_p.capacity = \mathtt{buffer}.max\_capacity}{\mathtt{mems}'_p[buff] = \mathtt{buffer}_p.buff\ \wedge\ \mathtt{mems}'_p[size] = \mathtt{buffer}_p.size\ \wedge\ \mathtt{buffer}'_p = \epsilon}$$

In the following we describe briefly the specification of a set of representative MPI functions. The semantics presented here are abstracted from the actual TLA+ code for succinctness and readability, which has been tested thoroughly using the TLC model checker. The entire specification including tests and examples and the verification framework are available online [17].

4.3 Quick Guide

In this section we use a simple example to illustrate how MPI programs and MPI functions are modeled. Consider the following MPI program involving two processes:

$$\begin{array}{ll} P0: & \text{MPI\_Send}(buf_s, 2, \text{MPI\_INT}, 1, 10, \text{MPI\_COMM\_WORLD}) \\ & \text{MPI\_Bcast}(buf_b, 1, \text{MPI\_FLOAT}, 0, \text{MPI\_COMM\_WORLD}) \\ P1: & \text{MPI\_Bcast}(buf_b, 1, \text{MPI\_FLOAT}, 0, \text{MPI\_COMM\_WORLD}) \\ & \text{MPI\_Recv}(buf_r, 2, \text{MPI\_INT}, 0, \text{MPI\_ANY\_TAG}, \text{MPI\_COMM\_WORLD}) \end{array}$$

This program is converted by the compiler into the following TLA+ code (*i.e.* the model of this program). An extra parameter is added to an MPI function to indicate the process this primitive belongs to. In essence, a model is a transition system consisting of transition rules. When the guard of a rule is satisfied, this rule is enabled and ready for execution. Multiple enabled rules are executed in a non-deterministic manner, leading to multiple executions. The control flow of a program at a process is represented by the pc values: $pc[0]$ and $pc[1]$ store the current values of the program pointers at process 0 and 1 respectively. In our framework, a blocking call is modeled by its non-blocking version followed by a wait operation, *e.g.* `MPI_Send = MPI_Isend + MPI_Wait`. Note that new variables such as $request_0$ and $status_0$ are introduced during the compilation, each of which is assigned an integer address. For example, suppose $request_0 = 5$ at process 0, then this variable's value is given by $mems[0][request_0]$ (*i.e.* $mems[0][5]$). To modify its value to v in a transition rule, we use $mems'[0][request_0] = v$ (or $request'_0 = v$ for brevity purpose).

p0's transition rules

$$
\begin{array}{ll}
\vee & \wedge\ pc[0] = L_1\ \wedge\ pc' = [pc \texttt{ EXCEPT } ![0] = L_2] \\
 & \wedge\ \text{MPI\_Isend}(buf_s, 2, \text{MPI\_INT}, 1, 10, \text{MPI\_COMM\_WORLD}, request_0, 0) \\
\vee & \wedge\ pc[0] = L_2\ \wedge\ pc' = pc' = [pc \texttt{ EXCEPT } ![0] = L_3] \\
 & \wedge\ \text{MPI\_Wait}(request_0, status_0, 0) \\
\vee & \wedge\ pc[0] = L_3\ \wedge\ pc' = [pc \texttt{ EXCEPT } ![pid] = L_4] \\
 & \wedge\ \text{MPI\_Bcast}_{init}(buf_b, 1, \text{MPI\_FLOAT}, 0, \text{MPI\_COMM\_WORLD}, 0) \\
\vee & \wedge\ pc[pid] = L_4\ \wedge\ pc' = [pc \texttt{ EXCEPT } ![pid] = L_5] \\
 & \wedge\ \text{MPI\_Bcast}_{wait}(buf_b, 1, \text{MPI\_FLOAT}, 0, \text{MPI\_COMM\_WORLD}, 0)
\end{array}
$$

p1's transition rules

$$
\begin{array}{ll}
\vee & \wedge\ pc[1] = L_1\ \wedge\ pc' = [pc \texttt{ EXCEPT } ![1] = L_2] \\
 & \wedge\ \text{MPI\_Bcast}_{init}(buf_b, 1, \text{MPI\_FLOAT}, 0, \text{MPI\_COMM\_WORLD}, 1) \\
\vee & \wedge\ pc[1] = L_2\ \wedge\ pc' = [pc \texttt{ EXCEPT } ![1] = L_3] \\
 & \wedge\ \text{MPI\_Bcast}_{wait}(buf_b, 1, \text{MPI\_FLOAT}, 0, \text{MPI\_COMM\_WORLD}, 1) \\
\vee & \wedge\ pc[1] = L_3\ \wedge\ pc' = [pc \texttt{ EXCEPT } ![1] = L_4] \\
 & \wedge\ \text{MPI\_Irecv}(buf_s, 2, \text{MPI\_INT}, 0, \text{MPI\_ANY\_TAG}, \text{MPI\_COMM\_WORLD}, request_1, 0) \\
\vee & \wedge\ pc[1] = L_4\ \wedge\ pc' = [pc \texttt{ EXCEPT } ![1] = L_5] \\
 & \wedge\ \text{MPI\_Wait}(request_1, status_1, 0)
\end{array}
$$

A enabled rule may be executed at any time. Suppose the program pointer of process $p0$ is L_1, then the `MPI_Isend` rule may be executed, modifying the program pointer to L_2. This rule creates a new send request req of format $\langle destination, communicator\ id, tag, value\rangle_{request\ id}$, and appends req to $p0$'s request queue $\texttt{reqs}_0$. Here function $read\_data$ reads an array of data from the memory according to the count and datatype information.

$$
\begin{array}{l}
\texttt{let } v = read\_data(\text{mems}_0, buf_s, 2, \text{MPI\_INT}) \texttt{ in} \\
\texttt{reqs}'_0 = \texttt{reqs}_0 \diamond \langle 1, \text{comms}_0[\text{MPI\_COMM\_WORLD}].cid, 10, v\rangle_{request_0}
\end{array}
$$

Similarly, when the `MPI_Irecv` rule at process $p1$ is executed, a new receive request of format $\langle buffer, source, communicator\ id, tag, \_\rangle_{request\ id}$ is appended to $\texttt{reqs}_1$, where $\_$ indicates that the data is yet to be received.

$$
\texttt{reqs}'_1 = \texttt{reqs}_1 \diamond \langle buf_r, 0, \text{comms}_1[\text{MPI\_COMM\_WORLD}].cid, \text{MPI\_ANY\_TAG}, \_\rangle_{request_1}
$$

As indicated below, the MPI System Scheduler will match the send request and the receive request, and transfers the data v from process $p0$ to process $p1$. Then the send request $request_0$ becomes $\langle 1, cid, 10, \_\rangle$, and the receive request $request_1$ becomes $\langle 1, cid, 10, v\rangle$, where cid is the context id of communicator `MPI_COMM_WORLD`.

$$
\frac{is\_match(\langle 0, request_0\rangle, \langle 1, request_1\rangle)}{\begin{array}{c}\texttt{reqs}'_0[request_0] = [@ \texttt{ EXCEPT } !.value = \_] \\ \texttt{reqs}'_1[request_1] = [@ \texttt{ EXCEPT } !.value = v]\end{array}}
$$

Suppose the send is not buffered at $p0$, then the `MPI_Wait` rule shown below will be blocked until the data in the send request is sent. When the value is sent, the send request will be removed from $p0$'s request queue. We use notation Γ to denote all the requests excluding the one pointed by $request_0$ in $p0$'s request queue, and $\texttt{reqs}_0 = Gamma \diamond \langle \ldots\rangle_{request_0}$ is a predicate for pattern matching.

$$
\frac{\texttt{reqs}_0 = \Gamma \diamond \langle 1, cid, 10, \_\rangle_{request_0}}{\texttt{reqs}'_0 = \Gamma}
$$

Analogously, the $\mathtt{MPI\_Wait}$ rule at process $p1$ is blocked until the receive request receives the incoming value. Then this request is removed from $p1$'s request queue, and the incoming value v is written into $p1$'s local memory.

$$\frac{\mathtt{reqs}_1 = \Gamma \diamond \langle buf_r, 0, cid, \text{MPI\_ANY\_TAG}, v\rangle_{request_1}}{\mathtt{reqs}'_1 = \Gamma \ \wedge \ \mathtt{mems}'_1[buf_r] = v}$$

In our formalization, each process divides a collective call into two phases: an "init" phase that initializes the call, and a "wait" phase that synchronizes the communications with other processes. In these two phases processes synchronize with each other through the $\mathtt{rendezvous}$ (or $\mathtt{rend}$ for short) object which records the information including the status of the communication and the data sent by the processes. For a communicator with context ID cid there exists a separate rendezvous object $\mathtt{rend}[cid]$. In the "init" phase, process p is blocked if the status of the current communication is not 'v' ('vacant'); otherwise p updates the status to be 'e' ('entered') and store its data in the rendezvous. Recall that notation $\Psi \uplus \langle p, `v' \rangle$ represents the function Ψ with the item at p updated to 'v', and $[i \mapsto v_1, j \mapsto v_2]$ is a function that maps i and j to v_1 and v_2 respectively. In the given example, the rendezvous object pertaining to communicator $\mathtt{MPI\_COMM\_WORLD}$ becomes $\langle [0 \mapsto `v', 1 \mapsto `v'], [0 \mapsto v] \rangle$, where $v = \mathit{read\_data}(\mathtt{mems}_0, buf_b, 1, \mathtt{MPI\_FLOAT})$, after the "init" phases of the broadcast at process 0 and 1 are over.

$$\mathtt{syn}_{\mathtt{init}}(cid, v, p) \doteq \boxed{\text{process } p \text{ joins the communication and stores data } v \text{ in } \mathtt{rend}}$$
$$\frac{\mathtt{rend}[cid] = \langle \Psi \uplus \langle p, `v' \rangle, S_v \rangle}{\mathtt{rend}'[cid] = \langle \Psi \uplus \langle p, `e' \rangle,\ S_v \uplus \langle p, v \rangle \rangle}$$

In the "wait" phase, if the communication is synchronizing, then process p has to wait until all other processes in the same communication have finished their "init" phases. If p is the last process that leaves, then the entire collective communication is over and the object will be deleted; otherwise p just updates its status to be l ('left').

$$\mathtt{syn}_{\mathtt{wait}}(cid, p) \doteq \boxed{\text{process } p \text{ leaves the synchronizaing communication}}$$
$$\frac{\begin{array}{l}\mathtt{rend}[cid] = \langle \Psi \uplus \langle p, `e' \rangle, S_v \rangle \wedge \\ \forall k \in \mathtt{comms}_p[cid].group : \Psi[k] \in \{`e', `l'\}\end{array}}{\begin{array}{ll}\mathtt{rend}'[cid] = & \mathtt{if}\ \forall k \in \mathtt{comms}_p[cid].group : \Psi[k] = `l'\ \mathtt{then}\ \epsilon \\ & \mathtt{else}\ \langle \Psi \uplus \langle p, `l' \rangle,\ S_v \rangle\end{array}}$$

These simplified rules illustrate how MPI point-to-point and collective communications are modeled. The standard rules for these communications are given in Section 4.4 and 4.6.

4.4 Point-to-point Communication

In our formalization, a blocking primitive is implemented as an asynchronous operation followed immediately by a wait operation, *e.g.* $\mathtt{MPI\_Ssend} = \mathtt{MPI\_Issend} + \mathtt{MPI\_Wait}$ and $\mathtt{MPI\_Sendrecv} = \mathtt{MPI\_Isend} + \mathtt{MPI\_Wait} + \mathtt{MPI\_Irecv} + \mathtt{MPI\_Wait}$. The semantics of core point to point communication functions are shown in figures 3, 4, 5 and 6; and an example illstruating how a MPI program is "executed" according to these semantics is in figure 2. The reader is supposed to refer to these semantics when reading through this section.

A process p appends its send or receive request containing the message to its request queue $\texttt{reqs}_p$. A send request contains information about the destination process (dst), the context ID of the communincator (cid), the tag to be matched (tag), the data value to be send ($value$), and the status (omitted here) of the message. This request also includes boolean flags indicating whether the request is persistent, active, live, canceled and deallocated or not. For brevity we do not show the last three flags when presenting the content of a request in the queue. In addition, in order to model a ready send, we include in a send request a field $prematch$ of format $\langle destination\ process, request\ index\rangle$ which refers to the receive request that matches this send request. A receive request has the similar format, except that it includes the buffer address and a field to store the incoming data. Initially the data is missing (represented by the "_" in the data field). Later on an incoming message from a sender will replace the "_" with the data it carries. Notation $v\_$ indicates that the data may be missing or contain a value. For example, $\langle buf, 0, 10, *, \_, \top, \top, \langle 0, 5\rangle\rangle_2^{recv}$ is a receive request such that: (i) the source process is process 0; (ii) the context id and the tag are 10 and `MPI_ANY_TAG` respectively; (iii) the incoming data is still missing; (iv) it is a persistent request that is still active; (v) it has been prematched with the send request with index 5 at process 0; and (vi) the index of this receive request in the request queue is 2.

MPI offers four send modes. A standard send may or may not buffer the outgoing message. If buffer space is available, then it behaves the same as a send in the buffered mode; otherwise it acts as a send in the synchronous mode. A buffered mode send will buffer the outgoing message and may complete before a matching receive is posted; while a synchronous send will complete successfully only if a matching receive is posted. A ready mode send may be started only if the matching receive is already posted.

As an illustration, we show below the specification of `MPI_IBsend`. Since $dtype$ and $comm$ are the references (pointers) to a datatype and a communicator object respectively, their values are obtained by $\texttt{datatypes}_p[dtype]$ and $\texttt{comms}_p[comm]$ respectively. The value to be send is read from the local memory of process p through the $read\_data$ function. It is the auxiliary function `ibsend` that creates a new send request and appends it to p's request queue. This function also modifies the send buffer object at process p (*i.e.* $\texttt{buffer}_p$), to accomodated the data. Moreover, the request handle is set to point to the

p_0	p_1	p_2
$\texttt{Issend}(v_1, dst = 1, cid = 5,$ $tag = 0, req = 0)$	$\texttt{Irecv}(b, src = 0, cid = 5,$ $tag = *, req = 0)$	$\texttt{Irecv}(b, src = *, cid = 5,$ $tag = *, req = 0)$
$\texttt{Irsend}(v_2, dst = 2, cid = 5,$ $tag = 0, req = 1)$	$\texttt{Wait}(req = 0)$	$\texttt{Wait}(req = 0)$
$\texttt{Wait}(req = 0)$		
$\texttt{Wait}(req = 1)$		

step	*event*	$\texttt{reqs}_0$	$\texttt{reqs}_1$	$\texttt{reqs}_2$
1	$\texttt{issend}(v, 1, 5, 0, 0)$	$\langle 1, 5, 0, v, \bot, \top, \epsilon\rangle_0^{ss}$		
2	$\texttt{irecv}(b, 0, 5, *, 1)$	$\langle 1, 5, 0, v, \bot, \top, \epsilon\rangle_0^{ss}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon\rangle$	
3	$\texttt{irecv}(b, *, 5, *, 2)$	$\langle 1, 5, 0, v, \bot, \top, \epsilon\rangle_0^{ss}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \epsilon\rangle_0^{rc}$
4	$\texttt{irsend}(v, 2, 5, 0, 0)$	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon\rangle_0^{ss} \diamond$ $\langle 1, 5, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
5	$\texttt{transfer}(0, 1)$	$\langle 1, 5, 0, \_, \bot, \top, \epsilon\rangle_0^{ss} \diamond$ $\langle 1, 5, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
6	$\texttt{wait}(0, 0)$	$\langle 1, 5, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
7	$\texttt{wait}(1, 0)$	$\langle 1, 5, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
8	$\texttt{transfer}(0, 2)$		$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, v_2, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
9	$\texttt{wait}(0, 2)$		$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	
10	$\texttt{wait}(0, 1)$			

Figure 2: A point-to-point communication program and one of its possible executions. Process p_0 sends messages to p_1 and p_2 in synchronous send mode and ready send mode respectively. The scheduler first forwards the message to p_1, then to p_2. A request is deallocated after the wait call on it. Superscripts ss, rs and rc represent $ssend$, $rsend$ and $recv$ respectively. The execution follows from the semantics shown in Figures 3, 4 and 5.

new request, which is the last request in the queue.

$\texttt{ibsend}(v, dst, cid, tag, p) \doteq$ buffer send
$\texttt{requires}\ \{\texttt{size}(v) \leq \texttt{buffer}_p.vacancy\}$
append a new send request (which is active and non-persistent) into the queue
$\texttt{reqs}'_p = reqs_p \diamond \langle dst, cid, tag, v, \bot, \top, \epsilon\rangle^{bsend} \wedge$
reduce the capacity of the send buffer by the size of v
$\texttt{buffer}'_p.vacancy = \texttt{buffer}_p.vacancy - \texttt{size}(v)$

$\texttt{MPI\_IBsend}(buf, count, dtype, dest, tag, comm, request, p) \doteq$ top level definition
$\texttt{let}\ cm = \texttt{comms}_p[comm]\ \texttt{in}$ the communicator
$\wedge\ \texttt{ibsend}(read\_data(\texttt{mems}_p, buf, count, \texttt{datatypes}_p[dtype]), cm.group[dest], cm.cid, tag, p)$
$\wedge\ \texttt{mems}'_p[request] = \texttt{len}(\texttt{reqs}_p)$ set the request handle

The $\texttt{MPI\_Recv}$ is modeled in a similar way. If a send request and a receive request match, then the MPI System Sceduler can transfer the value from the send request to the receive request. Relation $\asymp$ defines the meaning of "matching". There are two cases needed to be considered:

- The send is in ready mode. Recall that when a send request req_s is added into the queue, it is prematched to a receive request req_r such that the $prematch$ field (abbreviated as ω) in req_s stores the tuple $\langle$*destination process, destination request index* $\rangle$, and in req_r stores the tuple $\langle$*source process, source request index*$\rangle$. The MSS knows that req_s and req_r match if these two tuples match.

- The send is in other modes. The send request and receive request are matched if their source, destination, context ID and tag information match. Note that the source and tag in the receive request may be `MPI_ANY_SOURCE` and `MPI_ANY_TAG` respectively.

$$
\begin{array}{l}
(\langle p, dst, tag_p, \omega_p, k_p\rangle = \langle src, q, tag_q, \omega_q, k_q\rangle) \doteq \\
\texttt{if}\ \omega_q = \epsilon\ \wedge\ \omega_q = \epsilon\ \texttt{then} \\
\text{the two requests contain no pre-matched information} \\
\quad \wedge \quad tag_q \in \{tag_p, \texttt{ANY\_TAG}\}\ \text{the tags match} \\
\quad \wedge \quad q = dst\ \ q \text{ is the destination} \\
\quad \wedge \quad src \in \{p, \texttt{ANY\_SOURCE}\}\ \text{the source is } p \text{ or any process} \\
\texttt{else}\ \text{the two requests should have been pre-matched} \\
\quad \omega_p = \langle q, k_q\rangle\ \wedge\ \omega_q = \langle p, k_p\rangle
\end{array}
$$

It is the rule `transfer` that models the message passing mechanism: if a send message in process p's queue matches a receive request in q's queue, then the data is transferred. Note that messages from the same source to the same destination should be matched in a FIFO order. Suppose in process p's request queue there exists an active send request $req_i = \langle dst, cid, tag_p, v, pr_p, \top, \omega_p\rangle_i^{send}$, which contains a data value v to be sent; and process q's request queue contains an active receive request $req_i = \langle buf, src, cid, tag_q, \_, pr_q, \top, \omega_q\rangle_j^{recv}$, whose data is yet to be received. If req_p (req_p) is the first request in its queue that matches req_q (req_p), then the value in req_p can be transferred to req_q. The following predicate guarantees this FIFO requirement:

$$
\begin{array}{l}
\nexists \langle dst, cid, tag_1, v, pr_1, \top, \omega_1\rangle_m^{send} \in \Gamma_1^p : \nexists \langle buf, src_2, cid, tag_2, \_, pr_2, \top, \omega_2\rangle_n^{recv} \in \Gamma_1^q : \\
\quad \vee\ \langle p, dst, tag_1, \omega_1, m\rangle = \langle src, q, tag_q, \omega_q, j\rangle\ \vee\ \langle p, dst, tag_p, \omega_p, i\rangle = \langle src_2, q, tag_2, \omega_2, n\rangle \\
\quad \vee\ \langle p, dst, tag_1, \omega_1, m\rangle = \langle src_2, q, tag_2, \omega_2, n\rangle
\end{array}
$$

As shown in this rule, when the transfer is done, the value field in the receive request req_j is filled with the incoming value v, and the value field in the send request req_i is set to _, indicating that the value has been sent out. If the request is not persistent and not live (*i.e.* the corresponding `MPI_Wait` has been called), then it will be removed from the request queue. In addition, if the receive request at process q is not live, then the incoming value will be written to q's local memory.

The `MPI_Wait` call returns when the operation identified by the request $request$ is complete. If $request$ is a null handle, then an empty status (where the tag and source are `MPI_ANY_TAG` and `MPI_ANY_SOURCE` respectively, and count equals to 0) is returned; otherwise the assitant function `wait_one` is invoked, which picks the appropriate wait func-

`Data Structures`

$\mathit{send\ request}$: important fields + less important fields

$\langle \mathit{dst} : \texttt{int},\ \mathit{cid} : \texttt{int},\ \mathit{tag} : \texttt{int},\ \mathit{value},\ \mathit{pr} : \texttt{bool},\ \mathit{active} : \texttt{bool},\ \mathit{prematch} \rangle^{\mathit{mode}} +$
$\langle \mathit{cancelled} : \texttt{bool},\ \mathit{dealloc} : \texttt{bool},\ \mathit{live} : \texttt{bool} \rangle$

$\mathit{recv\ request}$: important fields + less important fields

$\langle \mathit{buf} : \texttt{int},\ \mathit{src} : \texttt{int},\ \mathit{cid} : \texttt{int},\ \mathit{tag} : \texttt{int},\ \mathit{value},\ \mathit{pr} : \texttt{bool},\ \mathit{active} : \texttt{bool},\ \mathit{prematch} \rangle^{\mathit{recv}} +$
$\langle \mathit{cancelled} : \texttt{bool},\ \mathit{dealloc} : \texttt{bool},\ \mathit{live} : \texttt{bool} \rangle$

$\texttt{ibsend}(v, \mathit{dst}, \mathit{cid}, \mathit{tag}, p) \circeq$ buffer send
$\texttt{requires}\ \{\texttt{size}(v) \leq \texttt{buffer}_p.\mathit{vacancy}\}$ check buffer availability
$\texttt{reqs}'_p = \mathit{reqs}_p \diamond \langle \mathit{dst}, \mathit{cid}, \mathit{tag}, v, \bot, \top, \epsilon \rangle^{\mathit{bsend}} \ \wedge$ append a new send request
$\texttt{buffer}'_p.\mathit{vacancy} = \texttt{buffer}_p.\mathit{vacancy} - \texttt{size}(v)$ allocate buffer space

$\texttt{issend}(v, \mathit{dst}, \mathit{cid}, \mathit{tag}, p) \circeq$ synchronous send
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle \mathit{dst}, \mathit{cid}, \mathit{tag}, v, \bot, \top, \epsilon \rangle^{\mathit{ssend}}$

$(\langle p, \mathit{dst}, \mathit{tag}_p, \omega_p, k_p \rangle \eqcirc \langle \mathit{src}, q, \mathit{tag}_q, \omega_q, k_q \rangle) \doteq$ match a send request and a receive request
$\texttt{if}\ \omega_q = \epsilon \ \wedge\ \omega_q = \epsilon\ \texttt{then}$
$\quad \mathit{tag}_q \in \{\mathit{tag}_p, \texttt{ANY\_TAG}\} \ \wedge\ q = \mathit{dst} \ \wedge\ \mathit{src} \in \{p, \texttt{ANY\_SOURCE}\}$
$\texttt{else}\ \omega_p = \langle q, k_q \rangle \ \wedge\ \omega_q = \langle p, k_p \rangle$ prematched requests

$\texttt{irsend}(v, \mathit{dst}, \mathit{cid}, \mathit{tag}, p) \circeq$ ready send
$\texttt{requires}\ \left\{ \begin{array}{l} \exists q : \exists \langle \mathit{src}, \mathit{cid}, \mathit{tag}_1, \_, \mathit{pr}_1, \top, \epsilon \rangle_k^{\mathit{recv}} \in \texttt{reqs}_q : \\ \quad \langle p, \mathit{dst}, \mathit{tag}, \epsilon, \texttt{len}(\texttt{reqs}_p) \rangle \eqcirc \langle \mathit{src}, q, \mathit{tag}_1, \epsilon, k \rangle \end{array} \right\}$ a matching receive exists?
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle \mathit{dst}, \mathit{cid}, \mathit{tag}, v, \bot, \top, \langle q, k \rangle \rangle^{\mathit{rsend}} \ \wedge\ \texttt{reqs}'_q.\omega = \langle p, \texttt{len}(\texttt{reqs}_p) \rangle$

$\texttt{isend} \circeq \texttt{if}\ \mathit{use\_buffer}\ \texttt{then ibsend else issend}$ standard mode send

$\texttt{irecv}(\mathit{buf}, \mathit{src}, \mathit{cid}, \mathit{tag}, p) \circeq$ receive
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle \mathit{buf}, \mathit{src}, \mathit{cid}, \mathit{tag}, \_, \bot, \top, \epsilon \rangle^{\mathit{recv}}$

$\texttt{MPI\_Isend}(\mathit{buf}, \mathit{count}, \mathit{dtype}, \mathit{dest}, \mathit{tag}, \mathit{comm}, \mathit{request}, p) \circeq$ standard immediate send
$\texttt{let}\ \mathit{cm} = \texttt{comms}_p[\mathit{comm}]\ \texttt{in}$ the communicator
$\wedge\ \texttt{isend}(\mathit{read\_data}(\texttt{mems}_p, \mathit{buf}, \mathit{count}, \mathit{dtype}), \mathit{cm}.\mathit{group}[\mathit{dest}], \mathit{cm}.\mathit{cid}, \mathit{tag}, p)$
$\wedge\ \texttt{mems}'_p[\mathit{request}] = \texttt{len}(\texttt{reqs}_p)$ set the request handle

$\texttt{MPI\_Irecv}(\mathit{buf}, \mathit{count}, \mathit{dtype}, \mathit{source}, \mathit{tag}, \mathit{comm}, \mathit{request}, p) \circeq$ immediate receive
$\texttt{let}\ \mathit{cm} = \texttt{comms}_p[\mathit{comm}]\ \texttt{in}$ the communicator
$\wedge\ \texttt{irecv}(\mathit{buf}, \mathit{cm}.\mathit{group}[\mathit{dest}], \mathit{cm}.\mathit{cid}, \mathit{tag}, p)$
$\wedge\ \texttt{mems}'_p[\mathit{request}] = \texttt{len}(\texttt{reqs}_p)$

$\texttt{wait\_one}(\mathit{request}, \mathit{status}, p) \doteq$ wait for one request to complete
$\texttt{if}\ \texttt{reqs}_p[\texttt{mems}_p[\mathit{request}]].\mathit{mode} = \mathit{recv}$
$\texttt{then recv\_wait}(\mathit{request})$ for receive request
$\texttt{else send\_wait}(\mathit{request})$ for send request

$\texttt{MPI\_Wait}(\mathit{request}, \mathit{status}, p) \circeq$ the top level wait function
$\texttt{if}\ \texttt{mems}_p[\mathit{request}] = \texttt{REQUEST\_NULL}\ \texttt{then}$
$\quad \texttt{mems}'_p[\mathit{status}] = \mathit{empty\_status}$ the handle is null, return an empty status
$\texttt{else wait\_one}(\mathit{request}, \mathit{status}, p)$

$\mathtt{transfer}(p, q) \doteq$ message transferring from process p to process q

$\wedge\ \mathtt{reqs}_p = \Gamma_1^p \diamond \langle dst, cid, tag_p, v, pr_p, \top, \omega_p \rangle_i^{send} \diamond \Gamma_2^p$

$\wedge\ \mathtt{reqs}_q = \Gamma_1^q \diamond \langle buf, src, cid, tag_q, \_, pr_q, \top, \omega_q \rangle_j^{recv} \diamond \Gamma_2^q \wedge$

$\wedge$ match the requests in a FIFO manner

$$
\begin{array}{l}
\langle p, dst, tag_p, \omega_p, i \rangle \doteq \langle src, q, tag_q, \omega_q, j \rangle \wedge \\
\nexists \langle dst, cid, tag_1, v, pr_1, \top, \omega_1 \rangle_m^{send} \in \Gamma_1^p : \\
\quad \nexists \langle buf, src_2, cid, tag_2, \_, pr_2, \top, \omega_2 \rangle_n^{recv} \in \Gamma_1^q : \\
\quad\quad \vee\ \langle p, dst, tag_1, \omega_1, m \rangle \doteq \langle src, q, tag_q, \omega_q, j \rangle \\
\quad\quad \vee\ \langle p, dst, tag_p, \omega_p, i \rangle \doteq \langle src_2, q, tag_2, \omega_2, n \rangle \\
\quad\quad \vee\ \langle p, dst, tag_1, \omega_1, m \rangle \doteq \langle src_2, q, tag_2, \omega_2, n \rangle
\end{array}
$$

$\wedge\ \mathtt{reqs}'_p =$ send the data

$$
\begin{array}{l}
\quad \mathtt{let}\ b = \mathtt{reqs}_p[i].live\ \mathtt{in} \\
\quad\quad \mathtt{if}\ \neg b\ \wedge\ \neg \mathtt{reqs}_p[i].pr\ \mathtt{then}\ \Gamma_1^p \diamond \Gamma_2^p \\
\quad\quad \mathtt{else}\ \Gamma_1^p \diamond \langle dst, cid, tag_p, \_, pr_p, b, \omega_p \rangle^{send} \diamond \Gamma_2^p
\end{array}
$$

$\wedge\ \mathtt{reqs}'_q =$ receive the data

$$
\begin{array}{l}
\quad \mathtt{let}\ b = \mathtt{reqs}_q[j].live\ \mathtt{in} \\
\quad\quad \mathtt{if}\ \neg b\ \wedge\ \neg \mathtt{reqs}_q[j].pr\ \mathtt{then}\ \Gamma_1^q \diamond \Gamma_2^q \\
\quad\quad \mathtt{else}\ \Gamma_1^q \diamond \langle buf, p, cid, tag_q, v, pr_q, b, \omega_q \rangle^{recv} \diamond \Gamma_2^q
\end{array}
$$

$\wedge\ \neg \mathtt{reqs}_q[j].live \Rightarrow \mathtt{mems}'_q[buf] = v$ write the data into memory

$\mathtt{recv\_wait}(request, status, p) \doteq$ wait for a receive request to complete

$\mathtt{let}\ req\_index = \mathtt{mems}_p[request]\ \mathtt{in}$

$\wedge\ \mathtt{reqs}'_p[req\_index].live = \bot$ indicate the wait has been called

$\wedge$

$\quad \vee\ \neg \mathtt{reqs}_p[req\_index].active \Rightarrow \mathtt{mems}_p[status] = empty\_status$

$\quad \vee$ the request is still active

$$
\begin{array}{l}
\quad\quad \mathtt{let}\ \Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \top, \omega \rangle_{req\_index}^{recv} \diamond \Gamma_2 = \mathtt{reqs}_q\ \mathtt{in} \\
\quad\quad \mathtt{let}\ b = pr\ \wedge\ \neg \mathtt{reqs}_p[req\_index].dealloc\ \mathtt{in} \\
\quad\quad \mathtt{let}\ new\_reqs = \\
\quad\quad\quad \mathtt{if}\ b\ \mathtt{then} \\
\quad\quad\quad\quad \Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \bot, \omega \rangle^{recv} \diamond \Gamma_2 \quad \text{set the request to be inactive} \\
\quad\quad\quad \mathtt{else}\ \Gamma_1 \diamond \Gamma_2 \quad \text{remove the request} \\
\quad\quad \mathtt{in} \\
\quad\quad \mathtt{let}\ new\_req\_index = \quad \text{update the request handle} \\
\quad\quad\quad \mathtt{if}\ b\ \mathtt{then}\ req\_index\ \mathtt{else}\ \mathtt{REQUEST\_NULL}\ \mathtt{in} \\
\quad\quad \mathtt{if}\ \mathtt{reqs}_q[req\_index].cancelled\ \mathtt{then} \\
\quad\quad\quad \mathtt{mems}'_p[status] = get\_status(\mathtt{reqs}_p[req\_index]) \wedge \\
\quad\quad\quad \mathtt{reqs}'_p = new\_reqs\ \wedge\ \mathtt{mems}'_p[request] = new\_req\_index \\
\quad\quad \mathtt{else\ if}\ src = \mathtt{PROC\_NULL}\ \mathtt{then} \\
\quad\quad\quad \mathtt{mems}'_p[status] = null\_status \wedge \\
\quad\quad\quad \mathtt{reqs}'_p = new\_reqs\ \wedge\ \mathtt{mems}'_p[request] = new\_req\_index \\
\quad\quad \mathtt{else} \\
\quad\quad\quad \text{wait until the data arrive, then write it to the memory}
\end{array}
$$

$$
\frac{v\_ \neq \_}{\begin{array}{l}
\mathtt{mems}'_p[status] = get\_status(\mathtt{reqs}_p[req\_index]) \wedge \\
\mathtt{mems}'_p[buf] = v\_ \wedge \\
\mathtt{reqs}'_p = new\_reqs\ \wedge\ \mathtt{mems}'_p[request] = new\_req\_index
\end{array}}
$$

Figure 4: Modeling point-to-point communications (II)

$\texttt{send\_wait}(request, status, p) \triangleq$ wait for a receive request to complete

$\texttt{let}\ req\_index = \texttt{mems}_p[request]\ \texttt{in}$

$\land\ \texttt{reqs}'_p[req\_index].live = \bot$ indicate the wait has been called

$\land$

$\quad \lor\ \neg\texttt{reqs}_p[req\_index].active \Rightarrow \texttt{mems}_p[status] = empty\_status$

$\quad \lor$ the request is still active

$\quad \texttt{let}\ \Gamma_1 \diamond \langle dst, cid, tag, v\_, pr, \top, \omega\rangle^{mode}_{req\_index} \diamond \Gamma_2 = \texttt{reqs}_q\ \texttt{in}$

$\quad \texttt{let}\ b = pr\ \land\ \neg\texttt{reqs}_p[req\_index].dealloc\ \lor\ v\_ \neq \_\ \texttt{in}$

$\quad \texttt{let}\ new\_reqs =$

$\quad\quad \texttt{if}\ b\ \texttt{then}$

$\quad\quad\quad \Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \bot, \omega\rangle^{recv} \diamond \Gamma_2$ set the request to be inactive

$\quad\quad \texttt{else}\ \Gamma_1 \diamond \Gamma_2$ remove the request

$\quad \texttt{in}$

$\quad \texttt{let}\ new\_req\_index =$ update the request handle

$\quad\quad \texttt{if}\ b\ \texttt{then}\ req\_index\ \texttt{else}\ \texttt{REQUEST\_NULL}\ \texttt{in}$

$\quad \texttt{let}\ action =$

$\quad\quad$ update the request queue, the status and the request handle

$\quad\quad \land\ \texttt{mems}'_p[status] = get\_status(\texttt{reqs}_p[req\_index])$

$\quad\quad \land\ \texttt{reqs}'_p = new\_reqs\ \land\ \texttt{mems}'_p[request] = new\_req\_index$

$\quad \texttt{in}$

$\quad\quad \texttt{if}\ \texttt{reqs}_q[req\_index].cancelled\ \texttt{then}\ action$

$\quad\quad \texttt{else if}\ dst = \texttt{PROC\_NULL}\ \texttt{then}$

$\quad\quad\quad \texttt{mems}'_p[status] = null\_status$

$\quad\quad\quad \texttt{reqs}'_p = new\_reqs\ \land\ \texttt{mems}'_p[request] = new\_req\_index$

$\quad\quad \texttt{else if}\ mode = ssend\ \texttt{then}$ synchronous send

$\quad\quad\quad$ can complete only a matching receive has been started

$$\frac{\exists q : \exists \langle src_1, cid, tag_1, \_, pr_1, \top, \omega_1\rangle^{recv}_k \in \Gamma_1 : \langle dst, p, tag, \omega, req\rangle = \langle src_1, q, tag_1, \omega_1, k\rangle}{action}$$

$\quad\quad \texttt{else if}\ mode = bsend\ \texttt{then}$

$\quad\quad\quad action\ \land\ \texttt{buffer}'.capaticy = \texttt{buffer}.capaticy - \texttt{size}(v\_)$

$\quad\quad \texttt{else}$ if no buffer is used then wait until the value is sent

$$\frac{\neg use\_buffer \Rightarrow (v\_ = \_)}{action}$$

$\texttt{issend\_init}(\texttt{v}, \texttt{dst}, \texttt{cid}, \texttt{tag}, \texttt{p}) \triangleq$ persistent (inactive) request for synchronous send

$\quad \texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle dst, cid, tag, v, \top, \bot, \epsilon\rangle^{ssend}$

$\texttt{irecv\_init}(buf, src, cid, tag, p) \triangleq$ persistent (inactive) receive request

$\quad \texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle buf, src, cid, tag, \_, \top, \bot, \epsilon\rangle^{recv}$

$\texttt{start}(req\_index, p) \triangleq$ start (activate) a persistent request

$\texttt{requires}\ \{\texttt{reqs}_p[req\_index].pr\ \land\ \neg\texttt{reqs}_p[req\_index].active\}$

$\quad \texttt{reqs}'_p[req\_index] = [\texttt{reqs}_p[req\_index]\ \texttt{EXCEPT}\ !.active = \top]$

Figure 5: Modeling point-to-point communications (III)

$\texttt{cancel}(req\_index, p) \doteq$ cancel a request

$\texttt{if reqs}_p[req\_index].active\ \texttt{then reqs}'_p[req\_index].cancelled = \top$ mark for cancellation
$\texttt{else reqs}'_p = \texttt{remove}(\texttt{reqs}_p, req\_index)$

$\texttt{free\_request}(request, p) \doteq$ free a request

$\texttt{let}\ req\_index = \texttt{mems}_p[request]\ \texttt{in}$
$\texttt{let}\ \Gamma_1 \diamond \langle dst, tag, v\_, pr, act, \epsilon\rangle^{mode}_{req\_index} \diamond \Gamma_2 = \texttt{reqs}_q\ \texttt{in}$
$\texttt{if}\ act\ \texttt{then reqs}'_p[req\_index].dealloc = \top$ mark for deallocation
$\texttt{else reqs}'_p = \Gamma_1 \diamond \Gamma_2\ \wedge\ \texttt{mems}'_p[request] = \texttt{REQUEST\_NULL}$ remove the request

$has\_completed(req\_index, p) \doteq$ whether a request has completed

$\vee\ \exists\langle buf, src, cid, tag, v, pr, \top, \omega\rangle^{recv} = \texttt{reqs}_q[req\_index]$ the data v have arrived
$\vee\ \exists\langle dst, cid, tag, v\_, pr, \top, \omega\rangle^{mode} = \texttt{reqs}_q[req\_index]$:
$\quad\vee\ mode = bsend$ the data are buffered
$\quad\vee\ mode = rsend\ \wedge\ (use\_buffer\ \vee\ (v\_ = \_))$ the data have been sent or buffered
$\quad\vee\ mode = ssend\ \wedge$ there must exist a matching receive
$\quad\quad\exists q : \exists\langle buf_1, src_1, cid, tag_1, \_, pr_1, \top, \omega_1\rangle^{recv}_k \in \texttt{reqs}_q :$
$\quad\quad\quad\langle dst, p, tag, \omega, req\rangle \asymp \langle src_1, q, tag_1, \omega_1, k\rangle$

$\texttt{wait\_any}(count, \overrightarrow{req_{array}}, index, status, p) \doteq$ wait for any request in $\overrightarrow{req_{array}}$ to complete

$\texttt{if}\ \forall i \in 0..count-1 : \overrightarrow{req_{array}}[i] = \texttt{REQUEST\_NULL}\ \vee\ \neg\texttt{reqs}_p[\overrightarrow{req_{array}}[i]].active$
$\texttt{then mems}'_p[index] = \texttt{UNDEFINED}\ \wedge\ \texttt{mems}_p[status] = empty\_status$

$$\texttt{else}\ \frac{\texttt{choose}\ i : has\_completed(\overrightarrow{req_{array}}[i], q)}{\begin{array}{l}\texttt{mems}'_p[index] = i\ \wedge \\ \texttt{mems}'_p[status] = get\_status(\texttt{reqs}_p[\overrightarrow{req_{array}}[i]])\end{array}}$$

$\texttt{wait\_all}(count, \overrightarrow{req\_array}, \overrightarrow{status\_array}, p) \doteq$ wait for all requests in $\overrightarrow{req_{array}}$ to complete

$\forall i \in 0\,..\,count-1 : wait\_one(\overrightarrow{req_{array}}[i], \overrightarrow{status\_array}[i], p)$

wait for all enabled requests in $\overrightarrow{req_{array}}$ to complete, abstracting away the statuses

$\texttt{wait\_some}(incount, \overrightarrow{req_{array}}, outcount, \overrightarrow{indice_{array}}, p) \doteq$
$\texttt{if}\ \forall i \in 0\,..\,count-1 : \overrightarrow{req_{array}}[i] = \texttt{REQUEST\_NULL}\ \vee\ \neg\texttt{reqs}_p[\overrightarrow{req_{array}}[i]].active$
$\texttt{then mems}'_p[index] = \texttt{UNDEFINED}$
$\texttt{else}$
$\quad\texttt{let}\ (\overrightarrow{index}, count) =$ pick all the completed requests
$\quad\quad\texttt{choose}\ (\overrightarrow{A} \sqsubseteq \overrightarrow{req_{array}},\ \texttt{max}\ k \in 1\,..\,incount-1) : \forall l \in 0\,..\,k-1 : has\_completed(\overrightarrow{A}[l], p)$
$\quad\texttt{in}$

$$\frac{\texttt{wait\_all}(count, \overrightarrow{index}, p)}{outcount' = count\ \wedge\ \overrightarrow{indice_{array}}' = \overrightarrow{index}}$$

Figure 6: Modeling point-to-point communications (IV)

tion according to the type of the request.

$\texttt{wait\_one}(req, status, p) \doteq$ wait for one request to complete
$\texttt{if reqs}_p[req].mode = recv$
  $\texttt{then recv\_wait}(req)$ for receive request
  $\texttt{else send\_wait}(req)$ for send request

$\texttt{MPI\_Wait}(request, status, p) \stackrel{\circ}{=}$
$\texttt{let } req\_index = mems_p[request] \texttt{ in}$
$\texttt{if } req\_index = \texttt{REQUEST\_NULL then}$
  $\texttt{mems}'_p[status] = empty\_status$ the handle is null, return an empty status
$\texttt{else wait\_one}(req\_index, status, p)$

Let us look closer at the definition of `recv_wait` (see figure 4). First of all, after this wait call the request is not "live" any more, thus the $live$ flag is set to false. When the call is made with an inactive request, it returns immediately with an empty status. If the request is persistent and is not marked for deallocation, then the request becomes inactive after the call; otherwise it is removed from the request queue and the corresponding request handle is set to `MPI_REQUEST_NULL`.

Then, if the request has been marked for cancellation, then the call completes without writing the data into the memory. If the source process is a null process, then the call returns immediately with a null status with source = `MPI_PROC_NULL`, tag = `MPI_ANY_TAG`, and count = 0. Finally, if the value has been received (*i.e.* $v\_ \neq \_$), then the value v is written to process p's local memory and the status object is updated accordingly.

The semantics of a wait call on a send request is defined similarly, especially when the call is made with a null or inactive or cancelled request, or the target process is null. The main difference is that the wait on a receive request can complete only after the incoming data have arrived, while the wait on a send request may complete before the data are sent out. Thus we cannot delete the send request when its data haven't been sent, this requires the condition b to be $pr \ \wedge \ \neg\texttt{reqs}_p[req\_index].dealloc \ \vee \ v\_ \neq \_$. After the call, the status object, request queue and request handle are updated. In particular, if the request has sent the data, and it is not persistent or has been marked for deallocation, then the request handle is set to `MPI_REQUEST_NULL`. On the other hand, if the data have not been sent (*i.e.* $v\_ \neq \_$), then the request handle will be intact.

$$\begin{array}{l} \texttt{mems}'_p[status] = get\_status(\texttt{reqs}_p[req\_index]) \\ \texttt{reqs}'_p = new\_reqs \ \wedge \ \texttt{mems}'_p[request] = new\_req\_index \end{array}$$

Depending on the send mode, the wait call may or may not complete before the data are sent. A send in a synchronous mode will complete only if a matching receive is already posted.

$$\begin{array}{l} \exists q : \exists \langle src_1, cid_1, tag_1, \_, pr_1, \top, \omega_1 \rangle_k^{recv} \in \Gamma_1 : \\ \quad \langle dst, p, cid, tag, \omega, req \rangle \asymp \langle src_1, q, cid_1, tag_1, \omega_1, k \rangle \end{array}$$

A buffered mode send will complete immediately since the data is buffered. If no buffer is used, a ready mode send will be blocked until the data is transferred; otherwise it returns intermediately.

When a persistent communication request is created, we set its $presistent$ flag. A communication using a persistent request is initiated by the `start` function. When this function is called, the request should be inactive. The request becomes active after the call. A pending, nonblocking communication can be canceled by a `cancel` call, which marks the request for cancellation. A `free_request` call marks the request object for deallocation and set the request handle to `MPI_REQUEST_NULL`. An ongoing communication will be allowed to complete and the request will be deallocated only after its completion.

In our implementation, the requirement for a request to be complete is modeled by the $has\_completed$ function. A receive request is complete when the data have been received. A send request in the buffer mode is complete when the data have been buffered or transferred. This function is used to implement communication operations of multiple completions. For example, `MPI_Waitany` blocks until one of the communication associated with requests in the array has completed. It returns in $index$ the array location of the completed request. `MPI_Waitall` blocks until all communications complete, and returns the statuses of all requests. `MPI_Waitsome` waits until at least one of the communications completes and returns the completed requests.

4.5 Datatype

A general datatype is an opague object that specifies a sequence of basic datatypes and integer displacements. The `extend` of a datatype is the span from the first byte to the last byte in this datatype. A datatype can be derived from simpler datatypes through datatype constructors. The simplest datatype constructor, modeled by `contiguous_copy`, allows replication of a datatype into contiguous locations. For example, $contiguous\_copy(2, \langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle\rangle)$ results in $\langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle, \langle \texttt{double}, 16\rangle, \langle \texttt{char}, 24\rangle\rangle$.

Constructor `type_vector` constructs a type consisting of the replication of a datatype into locations that consist of equally spaced blocks; each block is obtained by concatenating the same number of copies of the old datatype. `type_indexed` allows one to specify a noncontiguous data layout where displacements between blocks need not be equal. `type_struct` is the most general type constructor; it allows each block to consist of replications of different datatypes. These constructors are defined with the `contiguous_copy` constructor and the `set_offset` function (which increases the displacements of the items in the type by a certain offset). Other constructors are defined similarly. For instance,

$$\begin{array}{l}
\texttt{type_vector}(2, 2, 3, \langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle\rangle) = \\
\quad \langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle, \langle \texttt{double}, 16\rangle, \langle \texttt{char}, 24\rangle, \\
\quad \langle \texttt{double}, 48\rangle, \langle \texttt{char}, 56\rangle, \langle \texttt{double}, 64\rangle, \langle \texttt{char}, 72\rangle\rangle \\
\texttt{type_indexed}(2, \langle 3, 1\rangle, \langle 4, 0\rangle, \langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle\rangle) = \\
\quad \langle\langle \texttt{double}, 64\rangle, \langle \texttt{char}, 72\rangle, \langle \texttt{double}, 80\rangle, \langle \texttt{char}, 88\rangle, \\
\quad \langle \texttt{double}, 96\rangle, \langle \texttt{char}, 104\rangle, \langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle\rangle \\
\texttt{type_struct}(3, \langle 2, 1, 3\rangle, \langle 0, 16, 26\rangle, \langle \texttt{float}, \langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle\rangle, \texttt{char}\rangle) = \\
\quad \langle\langle \texttt{float}, 0\rangle, \langle \texttt{float}, 4\rangle, \langle \texttt{double}, 16\rangle, \langle \texttt{char}, 24\rangle, \langle \texttt{char}, 26\rangle, \langle \texttt{char}, 27\rangle, \langle \texttt{char}, 28\rangle\rangle
\end{array}$$

When creating a new type at process p, we store the type in an unused place in the $\texttt{datatypes}_p$ object, and have the output reference $datatype$ point to this place. When deleting a datatype at process p, we remove it from the $\texttt{datatypes}_p$ object and set the reference to MPI_DATATYPE_NULL. Derived datatypes support the specification of noncontiguous communication buffers. We show in Figure 7 how to read data from such buffers: noncontiguous data are "packed" into contiguous data which may be "unpacked" later in accordance to other datatypes.

Datatype operations are local function — no interprocess communication is needed when such an operation is executed. In the transition relations, only the datatypes object at the calling process is modified. For example, the transition implementing MPI_Type_index is as follows. Note that arguement $blocklengths$ is actually the start address of the block length array in the memory; auguments $oldtype$ and $newtype$ store the references to datatypes in the datatypes objects.

$$\begin{array}{l}
\texttt{MPI_Type_index}(count, blocklengths, displacements, oldtype, newtype, p) \triangleq \\
\texttt{let}\ \overrightarrow{lengths} = [i \in 0\,..\,count \mapsto \mathrm{mems}_p[blocklengths + i]]\ \texttt{in}\ \boxed{\text{length array}} \\
\texttt{let}\ \overrightarrow{displacements} = [i \in 0\,..\,count \mapsto \mathrm{mems}_p[displacements + i]]\ \texttt{in} \\
\texttt{let}\ type_index = \texttt{unused_index}(datatypes_p)\ \texttt{in}\ \boxed{\text{new datatype index}} \\
\texttt{let}\ dtype = \texttt{datatypes}_p[oldtype]\ \texttt{in} \\
\land\ datatypes_p[type_index] = \texttt{type_indexed}(count, \overrightarrow{blocklengths}, \overrightarrow{displacements}, dtype) \\
\land\ \mathrm{mems}_p[newtype]' = type_index\ \boxed{\text{update the reference to the new datatype}}
\end{array}$$

4.6 Collective Communication

All processes participating in a collective communication coordinate with each other through the shared rend object. There is a rend object corresponding to each communicator; and $\texttt{rend}[cid]$ refers to the rendezvous used by the communicator with context id cid. A rend object consists of a sequence of communication slots. In each slot, the $status$ field records the status of each process: 'e' ('entered'), 'l' ('left') or 'v' ('vacant', which is the initial value); the $shared\_data$ field stores the data shared among all processes; and $data$ stores

Data Structures

$\quad typemap : \langle type, disp : \mathtt{int} \rangle$ array

$contiguous\_copy(count, dtype) \doteq$ replicate a datatype into contiguous locations

let $F(i) =$

$\quad$ if $i = 1$ then $dtype$

$\quad$ else $F(i-1) \ \diamond [k \in \mathtt{DOM}(dtype) \mapsto$

$\qquad\qquad \langle dtype[k].type, dtype[k].disp + (i-1) * \mathtt{extend}(dtype)\rangle]$

in $F(count)$

$set\_offset(dtype, offset) \doteq$ adjust displacements

$\quad [k \in \mathtt{DOM}(dtype) \mapsto \langle dtype[k].type, dtype[k] + offset\rangle]$

replicate a datatype into equally spaced blocks

$\mathtt{type\_vector}(count, blocklength, stride, dtype) \stackrel{\circ}{=}$

let $F(i) =$

$\quad$ if $i = count$ then $\langle\rangle$

$\quad$ else let $offset = set\_offset(dtype, \mathtt{extend}(dtype) * stride * i)$ in

$\qquad contiguous\_copy(blocklength, offset) \diamond F(i+1)$

in $F(0)$

replicate a datatype into a sequence of blocks

$\mathtt{type\_indexed}(count, \overrightarrow{blocklengths}, \overrightarrow{displacements}, dtype) \stackrel{\circ}{=}$

let $F(i) =$

$\quad$ if $i = 0$ then $\langle\rangle$

$\quad$ else $F(i-1) \diamond$

$\qquad contiguous\_copy(\overrightarrow{blocklengths}[i-1],$

$\qquad\qquad set\_offset(dtype, \overrightarrow{displacements}[i-1] * \mathtt{extend}(dtype)))$

in $F(count)$

replicate a datatype to blocks that may consist of different datatypes

$\mathtt{type\_struct}(count, \overrightarrow{blocklengths}, \overrightarrow{displacements}, \overrightarrow{dtypes}) \stackrel{\circ}{=}$

let $F(i) =$

$\quad$ if $i = 0$ then $\langle\rangle$

$\quad$ else $F(i-1) \diamond contiguous\_copy(\overrightarrow{blocklengths}[i-1],$

$\qquad\qquad set\_offset(\overrightarrow{dtypes}[i-1], \overrightarrow{displacements}[i-1])$

in $F(count)$

$create\_datatype(datatype, dtype, p) \doteq$ create a new datatype

let $index = \mathtt{unused\_index}(\mathtt{datatypes}_p)$ in

$\quad \mathtt{datatypes}'_p[index] = dtype \ \wedge \ \mathtt{mems}'_p[datatype] = index$

$\mathtt{type\_free}(datatype, p) \doteq$ free a datatype

$\mathtt{datatypes}'_p = \mathtt{datatypes}_p \setminus \{\mathtt{datatypes}_p[datatype]\} \ \wedge \ datatype' = \mathtt{DATATYPE\_NULL}$

$read\_data(mem, buf, count, dtype) \doteq$ read (non-contiguous) data from the memory

let $read\_one(buf) =$

$\quad$ let $F_1(i) =$ if $i = 0$ then $\langle\rangle$ else $F_1(i-1) \diamond mem[buf + dtype[i-1].disp]$

$\quad$ in $F_1(\mathtt{size}(dtype))$

in let $F_2(i \in 0\,..\,count) =$

$\quad$ if $i = 0$ then $\langle\rangle$ else $F_2(i-1) \diamond read\_one(buf + (i-1) * \mathtt{extend}(dtype))$

in $F_2(count)$

the data sent by each process to the rendezvous. We use the notation Ψ to represent the content in the $status$.

Most collective communications are synchronizing, while the rest (like `MPI_Bcast`) can either be synchronizing or non-synchronizing. A collective primitive is implemented by a loose synchronization protocol: in the first "init" phase, process p checks whether there exists a slot such that p has not participant in. A negative answer means that p is initializing a new collective communication, thus p creates a new slot, sets its status to be 'entered' and stores its value v in this slot. If there are slots indicating that p has not joined the associated communications (*i.e.* p's status is 'v'), then p registers itself in the first of such slots by updating its status and value in the slot. This phase is the same for both synchronizing and non-synchronizing communications. Rule $\texttt{syn}_{init}$ and $\texttt{syn}_{write}$ are the simplified cases of $\texttt{syn}_{put}$.

After the "init" phase, process p proceeds to its next "wait" phase. Among all the slots p locates the first one indicating that it has entered but not left the associated communication. If the communication is synchronizing, then it has to wait until all other processes in the same communication have finished their "init" phases; otherwise it does not have to wait. If p is the last process that leaves, then the entire collective communication is over and the communication slot can be removed from the queue; otherwise p just updates its status to be 'left'.

These protocols are used to specify collective communication primitives. For example, `MPI_Bcast` is implemented as two transitions: $\texttt{MPI\_Bcast}_{init}$ and $\texttt{MPI\_Bcast}_{wait}$. The root first sends its data to the rendezvous in $\texttt{MPI\_Bcast}_{init}$, then by using the $\texttt{asyn}_{wait}$ rule it can return immediately without waiting for the completion of other processes. On the other hand, if the call is synchronizing then it will use the $\texttt{syn}_{wait}$ rule. In contrast, a non-root process p needs to call the $\texttt{syn}_{wait}$ because it must wait for the data from the root to "reach" the rendezvous.

In the `MPI_Gather` call, each process including the root sends data to the root; and the root stores all data in rank order. Expression $[i \in \texttt{DOM}(gr) \rightarrow \texttt{rend}_p[comm.cid].data[gr[i]]]$ returns the concatenation of the data of all processes in rank order. Function $write\_data$ writes an array of data into the memory. `MPI_Scatter` is the inverse operation to `MPI_Gather`. In `MPI_Alltoall`, each process sends distinct data to each of the receivers. The j^{th} block sent from process i is received by process j and is placed in the i^{th} block of the receive buffer. Additionally, data from all processes in a group can be combined using a reduction operation op. The call of `MPI_Scan` at a process with rank i returns in the receive buffer the reduction of the values from processes with ranks $0, \cdots, i$ (inclusive).

`Data Structures`

$rendezvous\ for\ a\ communication:$

$\langle status : [p : \texttt{int} \to \{`l', `e', `v'\}],\ sdata,\ data : [p : \texttt{int} \to value]\rangle$ `array`

process p joins the communication and stores the shared data v_s and its own data v in the rendevous

$\texttt{syn}_{\texttt{put}}(cid, v_s, v, p) \doteq$

$\texttt{if}\ cid \notin \texttt{DOM}(\texttt{rend})\ \texttt{then}\ \texttt{rend}'[cid] = \langle [p \mapsto `e`], v_s, [p \mapsto v]\rangle$

$\texttt{else if}\ \forall slot \in \texttt{rend}[cid] : slot.status[p] \in \{`e', `l'\}\ \texttt{then}$

$\quad \texttt{rend}'[cid] = \texttt{rend}[cid] \diamond \langle [p \mapsto `e`], v_s, [p \mapsto v]\rangle$

$\texttt{else}$

$$\frac{\texttt{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus \langle p, `v'\rangle, v_s, S_v\rangle \diamond \Gamma_2 \wedge \quad \forall slot \in \Gamma_1 : slot.status[p] \neq `v'}{\texttt{rend}'[cid] = \Gamma_1 \diamond \langle \Psi \uplus \langle p, `e'\rangle,\ v_s,\ S_v \uplus \langle p, v\rangle\rangle \diamond \Gamma_2}$$

$\texttt{syn}_{\texttt{init}}(cid, p) \doteq \texttt{syn\_write}(cid, \epsilon, \epsilon, p)$ no data are stored

$\texttt{syn}_{\texttt{write}}(cid, v, p) \doteq \texttt{syn\_write}(cid, \epsilon, v, p)$ no shared data are stored

$\texttt{syn}_{\texttt{wait}}(cid, p) \doteq$ process p leaves the synchronizaing communication

$$\frac{\texttt{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus \langle p, `e'\rangle, v_s, S_v\rangle \diamond \Gamma_2 \wedge \quad \forall k \in \texttt{comms}_p[cid].group : \Psi[k] \in \{`e', `l'\} \wedge \quad \forall slot \in \Gamma_1 : slot.status[p] \neq `e'}{\texttt{rend}'[cid] = \texttt{if}\ \forall k \in \texttt{comms}_p[cid].group : \Psi[k] = `l'\ \texttt{then}\ \Gamma_1 \diamond \Gamma_2\ \texttt{else}\ \Gamma_1 \diamond \langle \Psi \uplus \langle p, `l'\rangle,\ v_s,\ S_v\rangle \diamond \Gamma_2}$$

$\texttt{asyn}_{\texttt{wait}}(cid, p) \doteq$ process p leaves the non-synchronizaing communication

$$\frac{\texttt{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus \langle p, `e'\rangle, v_s, S_v\rangle \diamond \Gamma_2 \wedge \quad \forall slot \in \Gamma_1 : slot.status[p] \neq `e'}{\texttt{rend}'[cid] = \texttt{if}\ \forall k \in \texttt{comms}_p[cid].group : \Psi[k] = `l'\ \texttt{then}\ \Gamma_1 \diamond \Gamma_2\ \texttt{else}\ \Gamma_1 \diamond \langle \Psi \uplus \langle p, `l'\rangle,\ v_s,\ S_v\rangle \diamond \Gamma_2}$$

Figure 8: The basic protocol for collective communications

$$
\begin{array}{lll}
p_0 & p_1 & p_2 \\
\texttt{syn}_{\texttt{put}}(cid = 0, sdata = v_s, data = v_0) & \texttt{syn}_{\texttt{init}}(cid = 0) & \texttt{syn}_{\texttt{write}}(cid = 0, data = v_2) \\
\texttt{asyn}_{\texttt{wait}}(cid = 0) & \texttt{syn}_{\texttt{wait}}(cid = 0) & \texttt{syn}_{\texttt{wait}}(cid = 0) \\
\texttt{syn}_{\texttt{init}}(cid = 0) & &
\end{array}
$$

step	*event*	$\texttt{rend}[0]$
1	$\texttt{syn}_{\texttt{put}}(0, v_s, v_0, 0)$	$\langle [0 \mapsto `e'], v_s, [0 \mapsto v_0] \rangle$
2	$\texttt{syn}_{\texttt{init}}(0, 1)$	$\langle [0 \mapsto `e', 1 \mapsto `e'], v_s, [0 \mapsto v_0] \rangle$
3	$\texttt{syn}_{\texttt{wait}}(0, 0)$	$\langle [0 \mapsto `l', 1 \mapsto `e'], v_s, [0 \mapsto v_0] \rangle$
4	$\texttt{syn}_{\texttt{init}}(0, 0)$	$\langle [0 \mapsto `l', 1 \mapsto `e'], v_s, [0 \mapsto v_0] \rangle \diamond \langle [0 \mapsto `e'], \epsilon, \epsilon \rangle$
5	$\texttt{syn}_{\texttt{write}}(0, v_2, 2)$	$\langle [0 \mapsto `l', 1 \mapsto `e', 2 \mapsto `e'], v_s, [0 \mapsto v_0, 2 \mapsto v_2] \rangle \diamond \langle [0 \mapsto `e'], \epsilon, \epsilon \rangle$
6	$\texttt{syn}_{\texttt{wait}}(0, 2)$	$\langle [0 \mapsto `l', 1 \mapsto `e', 2 \mapsto `l'], v_s, [0 \mapsto v_0, 2 \mapsto v_2] \rangle \diamond \langle [0 \mapsto `e'], \epsilon, \epsilon \rangle$
7	$\texttt{syn}_{\texttt{wait}}(0, 1)$	$\langle [0 \mapsto `e'], \epsilon, \epsilon \rangle$

Figure 9: An example using the collective protocol. Three processes participate in collective communications via a communicator with context ID = 0. Process p_0's asynchronous wait returns even before p_2 joins the synchronization; it also initializes a new synchronization after it returns. Process p_2, the last one joining the synchronization, deallocates the slot. The execution follows from the semantics shown in figure 8.

MPI-2 introduces extensions of many of MPI-1 collective routines to intercommunicators, each of which contain a local group and a remote group. In this case, we just need to replace $\texttt{comms}_p[cid].group$ with $\texttt{comms}_p[cid].group \cup \texttt{comms}_p[cid].remote\_group$ in the rules shown in figure 8. In our TLA+ specification we take both cases into account when designing the collective protocol.

For example, if the *comm* in `MPI_Bcast` is an intercommunicator, then the call involves all processes in the intercommunicator, broadcasting from the root in one group (group A) to all processes in the other group (group B). All processes in group B pass the same value in argument *root*, which is the rank of the root in group A. The root passes the value `MPI_ROOT` in *root*, and other processes in group A pass the value `MPI_PROC_NULL` in *root*.

4.7 Communicator

Message passing in MPI is via communicators, each of which specifies a set (group) of processes that participate in the communication. Communicators can be created and destroyed dynamically by coordinating processes. Information about topology and other attributes of a communicator can be updated too. An intercommunicator is used for communication between two disjoint groups of processes. No topology is associated with an intercommuthe root broadcasts data to procosess

$$
\begin{aligned}
&\texttt{bcast}_{\texttt{init}}(buf, v, root, comm, p) \mathrel{\hat{=}} \\
&\quad (comm.group[root] = p)\ ?\ \texttt{syn}_{\texttt{put}}(comm.cid, v, \epsilon, p)\ :\ \texttt{syn}_{\texttt{init}}(comm.cid, p) \\
&\texttt{bcast}_{\texttt{wait}}(buf, v, root, comm, p) \mathrel{\hat{=}} \\
&\quad \texttt{if } comm.group[root] = p \texttt{ then} \\
&\qquad need\_syn\ ?\ \texttt{syn}_{\texttt{wait}}(comm.cid, p)\ :\ \texttt{asyn}_{\texttt{wait}}(comm.cid, p) \\
&\quad \texttt{else } \texttt{syn}_{\texttt{wait}}(comm.cid, p)\ \wedge\ \texttt{mems}'_p[buf] = \texttt{rend}_p[comm.cid].sdata
\end{aligned}
$$

the root gather data from procosess

$$
\begin{aligned}
&\texttt{gather}_{\texttt{init}}(buf, v, root, comm, p) \mathrel{\hat{=}} \texttt{syn}_{\texttt{write}}(comm.cid, v, p) \\
&\texttt{gather}_{\texttt{wait}}(buf, v, root, comm, p) \mathrel{\hat{=}} \\
&\quad \texttt{if } comm.group[root] \neq p \texttt{ then} \\
&\qquad need\_syn\ ?\ \texttt{syn}_{\texttt{wait}}(comm.cid, p)\ :\ \texttt{asyn}_{\texttt{wait}}(comm.cid, p) \\
&\quad \texttt{else} \\
&\qquad \wedge\ \texttt{syn}_{\texttt{wait}}(comm.cid, p) \\
&\qquad \wedge\ \texttt{let } data = [i \in \texttt{DOM}(comm.group) \rightarrow \texttt{rend}_p[comm.cid].data[comm.group[i]]] \\
&\qquad\quad \texttt{in } \texttt{mems}'_p = write\_data(\texttt{mems}_p, buf, data)
\end{aligned}
$$

the root scatters data to procosess

$$
\begin{aligned}
&\texttt{scatter}_{\texttt{init}}(buf, \overrightarrow{v}, root, comm, p) \mathrel{\hat{=}} \\
&\quad (comm.group[root] = p)\ ?\ \texttt{syn}_{\texttt{put}}(comm.cid, \overrightarrow{v}, \epsilon, p)\ :\ \texttt{syn}_{\texttt{init}}(comm.cid, p) \\
&\texttt{scatter}_{\texttt{wait}}(buf, \overrightarrow{v}, root, comm, p) \mathrel{\hat{=}} \\
&\quad \texttt{if } comm.group[root] = p\ \wedge\ \neg need\_syn \texttt{ then } \texttt{asyn}_{\texttt{wait}}(comm.cid, p) \\
&\quad \texttt{else } \texttt{syn}_{\texttt{wait}}(comm.cid, p)\ \wedge\ \texttt{mems}'_p[buf] = \texttt{rend}_p.sdata[comm.group|_p]
\end{aligned}
$$

all procosess send and receive data

$$
\begin{aligned}
&\texttt{alltoall}_{\texttt{init}}(buf, \overrightarrow{v}, comm, p) \mathrel{\hat{=}} \texttt{syn\_write}(comm.cid, \overrightarrow{v}, p) \\
&\texttt{alltoall}_{\texttt{wait}}(buf, \overrightarrow{v}, comm, p) \mathrel{\hat{=}} \\
&\quad \wedge\ \texttt{syn}_{\texttt{wait}}(comm.cid, p) \\
&\quad \wedge\ \texttt{let } gr = comm.group \texttt{ in} \\
&\qquad \texttt{let } data = [i \in \texttt{DOM}\ gr \rightarrow \texttt{rend}[comm.cid].data[gr[i]][gr|_p]] \texttt{ in} \\
&\qquad \texttt{mems}'_p = write\_data(\texttt{mems}_p, buf, data)
\end{aligned}
$$

$$
\begin{aligned}
&reduce\_range(\texttt{op}, \overrightarrow{data}, start, end) \doteq \quad \text{reduce the data according to the range} \\
&\quad \texttt{let } F(i) = \texttt{if } i = start \texttt{ then } \overrightarrow{data}[i] \texttt{ else } \texttt{op}(F(i-1), \overrightarrow{data}[i]) \texttt{ in } F(end) \\
&reduce(\texttt{op}, \overrightarrow{data}) \doteq reduce\_range(\texttt{op}, \overrightarrow{data}, 0, \texttt{size}(\overrightarrow{data})) \quad \text{reduce an array of values}
\end{aligned}
$$

prefix reduction on the data distributed across the group

$$
\begin{aligned}
&\texttt{scan}_{\texttt{init}}(buf, v, op, comm, p) \mathrel{\hat{=}} \texttt{syn}_{\texttt{write}}(comm.cid, v, p) \\
&\texttt{scan}_{\texttt{wait}}(buf, v, op, comm, p) \mathrel{\hat{=}} \\
&\quad \wedge\ \texttt{syn}_{\texttt{wait}}(comm.cid, p) \\
&\quad \wedge\ \texttt{let } gr = comm.group \texttt{ in} \\
&\qquad \texttt{let } data = [i \in 0 \mathinner{..} gr|_p \mapsto \texttt{rend}_p[comm.cid].data[gr[i]]] \\
&\qquad \texttt{in } \texttt{mems}'_p[buf] = reduce\_range(\texttt{op}, data, 0, gr|_p)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{inter\_bcast}_{\texttt{init}}(buf, v, root, comm, p) \mathrel{\hat{=}} \quad \text{broadcast in an inter-communicator} \\
&\quad (comm.group[root] = \texttt{ROOT})\ ?\ \texttt{syn\_put}(comm.cid, v, \epsilon, p)\ :\ \texttt{syn}_{\texttt{init}}(comm.cid, p) \\
&\texttt{inter\_bcast}_{\texttt{wait}}(buf, v, root, comm, p) \mathrel{\hat{=}} \\
&\quad \texttt{if } root \in \{\texttt{PROC\_NULL}, \texttt{ROOT}\}\ \wedge\ \neg need\_syn \texttt{ then } \texttt{asyn}_{\texttt{wait}}(comm.cid, p) \\
&\quad \texttt{else } \texttt{syn}_{\texttt{wait}}(comm.cid, p)\ \wedge\ \texttt{mems}'_p[buf] = \texttt{rend}_p[comm.cid].sdata
\end{aligned}
$$

Figure 10: Modeling collective communications

nicator.

4.7.1 Group

A group defines the participants in the communication of a communicator. It is actually an ordered collection of processes, each with a rank. An ordered set containing n elements ranging from 0 to N can be modeled as a function:

$$[i \in 0\,..\,n-1 \rightarrow 0\,..\,N]$$

Given a group gr modeled as an ordered set, the rank of a process p in this group is given by $gr|_p$, and the process with rank i is by $gr[i]$.

The distinct concatenation of two ordered sets s_1 and s_2 is obtained by appending the elements in $s_2 \setminus s_1$ to s_1:

$$s_1 \boxplus s_2 \doteq [i \in 0\,..\,(|s_1|+|s_2|-1) \mapsto i<|s_1|\ ?\ s_1[i]\ :\ s_2[i-\mathtt{size}(s_1)]].$$

The difference, intersection and union of two ordered sets are given by

$$\begin{array}{l} s_1 \ominus s_2 \doteq \text{ ordered set difference}\\ \quad \mathtt{let}\ F(i\in 0..|s_1|) = \\ \qquad (i=0)\ ?\ \langle\rangle\ :\ (s_1[i-1]\notin s_2)\ ?\ F(i-1) \boxplus \langle s_1[i-1]\rangle\ :\ F(i-1)\\ \quad \mathtt{in}\ F[|s_1|]\end{array}$$

$$\begin{array}{l} s_1 \odot s_2 \doteq \text{ ordered set intersection}\\ \quad \mathtt{let}\ F(i\in 0..|s_1|) = \\ \qquad (i=0)\ ?\ \langle\rangle\ :\ (s_1[i-1]\in s_2)\ ?\ F(i-1) \boxplus \langle s_1[i-1]\rangle\ :\ F(i-1)\\ \quad \mathtt{in}\ F[|s_1|]\end{array}$$

$$s_1 \oplus s_2 \doteq s_1 \boxplus (s_2 \ominus s_1) \text{ ordered set union}$$

Function $\mathtt{incl}(s, n, ranks)$ creates an ordered set that consists of the n elements in s with ranks $ranks[0], \ldots, ranks[n-1]$; `excl` creates an ordered set that is obtained by deleting from s those elements with ranks $ranks[0], \ldots, ranks[n-1]$; `range_incl` (`range_excl`) accepts a $ranges$ argument of form *(first rank,last rank,stride)* indicating ranks in s to be included (excluded) in the new ordered set.

$$
\begin{array}{l}
incl(s, n, ranks) \doteq [i \in 0 \mathinner{..} n-1 \mapsto s[ranks[i]]] \\
\\
excl(s, n, ranks) \doteq s \ominus (incl(s, n, ranks)) \\
\\
range\_incl(s, n, ranges) \doteq \\
\texttt{let } flatten(first, last, stride) = \boxed{\text{process one range}} \\
\quad \texttt{if } last < first \texttt{ then } \langle\rangle \\
\quad \texttt{else } first \diamond flatten(first + stride, last, stride) \\
\texttt{in} \\
\texttt{let } F(i) = \boxed{\text{process all the ranges}} \\
\quad \texttt{if } i = 0 \texttt{ then } \langle\rangle \\
\quad \texttt{else let } ranks = flatten(ranges[i-1]) \\
\qquad\quad \texttt{in } F(i-1) \diamond incl(s, \texttt{size}(ranks), ranks) \\
\texttt{in } F(n) \\
\\
range\_excl(s, n, ranges) \doteq s \ominus (range\_incl(s, n, ranges))
\end{array}
$$

For example, suppose $s_1 = \langle a, b, c, d \rangle$ and $s_2 = \langle d, a, e \rangle$, then $s_1 \oplus s_2 = \langle a, b, c, d, e \rangle$, $s_1 \odot s_2 = \langle a, d \rangle$, and $s_1 \ominus s_2 = \langle b, c \rangle$. Suppose $s = \langle a, b, c, d, e, f, g, h, i, j \rangle$ and $ranges = \langle \langle 6, 7, 1 \rangle, \langle 1, 6, 2 \rangle, \langle 0, 9, 4 \rangle \rangle$, then $range\_incl(s, 3, ranges) = \langle g, h, b, d, f, a, e, i \rangle$ and $range\_excl(s, 3, ranges) = \langle c, j \rangle$.

Since most group operations are local and their execution do not require interprocess communication, in the transition relations corresponding to such operations, only the `groups` object at the calling process is modified. For example, the transition implementing the union of two groups is as follows.

$$
\begin{array}{l}
\texttt{MPI\_Group\_union}(group_1, group_2, group_{new}, p) \stackrel{\circ}{=} \\
\texttt{let } gid = \texttt{unused\_item}(\texttt{groups}_p) \texttt{ in} \\
\texttt{groups}'_p = \texttt{groups}_p \uplus \langle gid,\ \texttt{groups}_p[group_1] \oplus \texttt{groups}_p[group_2] \rangle \wedge \\
\texttt{mems}'_p[group_{new}] = gid
\end{array}
$$

4.7.2 Communicator Operations

Communicator constructors and destructors are collective functions that are invoked by all processes in the involved group. When a new communicator is created, each participanting process first invokes the "synchronization initialization" primitive (mentioned in the Section 4.6) to express its willing to join the creation; then it calls the "synchronization wait" primitive to wait for the joining of all other processes; finally it creates the local version of the new communicator and store it in its `comms` object.

Communicators may be attached with arbitrary pieces of information (called attributes). When a attribute key is allocated (*e.g.* by calling the `MPI_Comm_create_keyval`) and stored in the `keyvals` object, it is attached with a copy callback function, a delete callback function and an extra state for callback functions. When a communicator is created using functions like `MPI_Comm_dup`, all callback copy functions for attributes are invoked (in arbitrary order). When the copy function returns $flag = \perp$, then the attribute is deleted in the created communicator; otherwise the new attribute value is set to the value returned in $attribute\_val\_out$.

The `MPI_Comm_dup` code shown in Figure 11 creates a new intracommunicator with the same group and topology as the input intracommunicator. The association of cached attributes is controlled by the copy callback functions. As the new communicator must have a unique context id, the the process with rank 0 picks an unused context id, write it to the shared area of the rendezvous, and registers it in the system. In the "synchronization wait" phase each process fetches the unique context id, finds a place for the new communicator in its `comms` object, and updates the reference to this place.

Intercommunicator operations are a little more complicated. For example, `Intercomm_merge` creates an intracommunicator from the union of the two groups of a intercommunicator. All processes should provide the same $high$ value within each of the two groups. The group providing the value $high = \top$ should be ordered before the one providing $high = \perp$; and the order is arbitrary if all processes provide the same $high$ argument.

The TLA+ specification of communicator operations is more detailed, where we need to: (i) check whether all processes propose the same `group` and the group is a subset of the group associated with the old communicator; (ii) have the function returns `MPI_COMM_NULL` to processes that are not in the `group`; (iii) call the error callback functions when errors occur.

4.7.3 Topology

A topology can provide a convenient naming mechanism for the processes within a communicator, and additionally, may assist the runtime system in mapping the processes onto hardware. A topology can be represented by a graph, with nodes and edges standing for processes and communication links respectively. In some cases it is desirable to use Cartesian topologies (of arbitrary dimensions).

The primitive `Cart_create` builds a new communicator with Cartesian topology information. Arguments $ndims$ and $dims$ give the number of dimensions and an inte-

Data Structures

$communicator : cid : \texttt{int},\ group : \texttt{oset},\ remote\_group : \texttt{oset},\ topology,\ attributes : \texttt{map}$

$create\_comm(comm, keyvals) \doteq$ create a new communicator

$\texttt{let}\ copy\_attr(comm, attr, keyvals) =$ call the copy function

$\quad \texttt{let}\ keyval = keyvals[attr.key]\ \texttt{in}$

$\quad \texttt{let}\ y = keyval.copy\_attr\_fn(comm, attr.key, keyval.extra\_state, attr.value)\ \texttt{in}$

$\quad\ [comm\ \texttt{EXCEPT}\ !.attributes =$

$\qquad \texttt{if}\ y.flag = \bot\ \texttt{then}\ \texttt{remove}(@, attr.key)\ \texttt{else}\ @ \uplus \langle attr.key, y.attribute\_val\_out\rangle$

$\quad\ ]\ \texttt{in}$

$\texttt{let}\ traverse(T) =$ call the copy functions of all attributes

$\quad \texttt{if}\ T = \{\}\ \texttt{then}\ comm$

$\quad \texttt{else choose}\ attr \in T : copy\_attr(traverse(T \setminus \{attr\}), attr, keyvals)\ \texttt{in}$

$\texttt{if}\ attributes \notin \texttt{DOM}\ comm\ \texttt{then}\ comm\ \texttt{else}\ traverse(comm.attributes)$

$\texttt{comm\_dup}_{\texttt{init}}(comm, newcomm, p) \stackrel{\circ}{=}$ duplicate a communicator

$\texttt{let}\ cid = \texttt{next\_comm\_cid}\ \texttt{in}$ obtain an unused context id

$\texttt{if}\ comm.gr|_p = 0\ \texttt{then}\ \texttt{syn\_put}(comm, cid, \epsilon, p)\ \wedge\ \texttt{register\_cid}(cid)$

$\texttt{else}\ \texttt{syn\_init}(\texttt{comm}, \texttt{p})$

$\texttt{comm\_dup}_{\texttt{wait}}(comm, newcomm, p) \stackrel{\circ}{=}$

$\texttt{syn\_wait}(comm, p) \wedge$

$\texttt{let}\ slot \diamond \Gamma = \texttt{rend}[comm.cid]\ \texttt{in}$

$\texttt{let}\ cid = slot.sdata\ \texttt{in let}\ new\_index = \texttt{unused\_index}(\texttt{comms}_p)\ \texttt{in}$

$\texttt{comms}'_p = \texttt{comms}_p \uplus \langle new\_index,\ [create\_comm(comm, keyvals_p)\ \texttt{EXCEPT}\ !.cid = cid]\rangle \wedge$

$newcomm' = new\_index$

create a new intracommunicator by merging the two groups of the inter-communicator

$\texttt{intercomm\_merge}_{\texttt{init}}(intercomm, high, intracomm_{new}, p) \stackrel{\circ}{=}$

$\texttt{let}\ cid = \texttt{next\_comm\_cid}\ \texttt{in}$

$\texttt{if}\ comm.gr|_p = 0\ \texttt{then}\ \texttt{syn\_put}(intercomm, cid, high, p)\ \wedge\ \texttt{register\_cid}(cid)$

$\texttt{else}\ \texttt{syn\_write}(\texttt{intercomm}, \texttt{high}, \texttt{p})$

$\texttt{intercomm\_merge}_{\texttt{wait}}(intercomm, high, intracomm_{new}, p) \stackrel{\circ}{=}$

$\texttt{syn\_wait}(intercomm, p) \wedge$

$\texttt{let}\ slot \diamond \Gamma = \texttt{rend}[intercomm.cid]\ \texttt{in}$

$\texttt{let}\ cid = slot.sdata\ \texttt{in let}\ new\_index = \texttt{unused\_index}(\texttt{comms}_p)\ \texttt{in}$

$\texttt{let}\ lr = intercomm.group \oplus intercomm.remote\_group\ \texttt{in}$

$\texttt{let}\ rl = intercomm.remote\_group \oplus intercomm.group\ \texttt{in}$

$\texttt{let}\ group =$

$\quad \texttt{if}\ \forall i, j \in intercomm.group \cup intercomm.remote\_group :$

$\qquad \texttt{rend}[intercomm.cid].data[i] = \texttt{rend}[intercomm.cid].data[j]$

$\quad \texttt{then choose}\ gr \in \{lr, rl\}$ processes propose the same $high$ value

$\quad \texttt{else}\ high\ ?\ lr\ :\ rl\ \texttt{in}$ order the two groups according to the $high$ value

$\texttt{comms}'_p = \texttt{comms}_p \uplus (new\_index,$

$\qquad [create\_comm(@, keyvals_p)\ \texttt{EXCEPT}$

$\qquad\quad !.cid = cid,\ !.group = group,\ !.remote\_group = \epsilon]$

$\qquad) \wedge$

$intercomm'_{new} = new\_index$

Figure 11: Modeling communicator operations

ger array specifying the number of processes in each dimension respectively. $periods$ specifies whether the grid is periodic or not in each dimension; and $reorder$ specifies whether ranks may be reordered or not. If the total size of the grid is smaller than the size of the group of $comm$, then those processes not fitting into the grid are returned MPI_COMM_NULL. Here the helper function $range\_product(ndims, dims, i, j)$ computes the value of $dims[i] \times \cdots \times dims[j]$.

Function $coord\_2\_rank$ translates the logical process coordinates to process ranks; function $rank\_2\_coord$ is the rank-to-coordinates translator. They are used to implemented the MPI_Cart_rank and MPI_Cart_coords primitives.

For further illustration we give the code of MPI_Cart_shift. When a MPI_Sendrecv operation is called along a coordinate direction to perform a shift of data, the rank of a source process for the receive and the rank of a destination process for the send can be calculated by this MPI_Cart_shift function. The dir argument indicates the dimension of the shift. In the case of an end-off shift, out-of-range processes will be returned the value MPI_PROC_NULL. Clearly MPI_Cart_shift is not a collective function.

4.8 Process Management

The MPI-2 process model allows for the creation and cooperative termination of processes after an MPI application has started. Since the runtime environment involving process creation and termination is not modeled, we do not specify MPI_Comm_spawn, which starts multiple copies of an MPI program specification, MPI_Comm_spawn_multiple, which starts multiple executable specifications, and MPI_Comm_get_parent, which is related to the “spawn” primitives.

Some functions are provided to establish communication between two groups of MPI processes that do not share a communicator. One group of processes (the $server$) indicates its willingness to accept connections from other groups of processes; the other group (the $client$) connects to the server. In order to the client to locate the server, the server provides a *port_name* that encodes a low-level network address. In our specification it consists of a process id and a port number. A server can publish a port_name with MPI_Publish_name and clients can retrieve the port name from the service name.

A server first calls MPI_Open_port to establish a port at which it may be contacted; then it calls MPI_Comm_accept to accept connections from clients. This port name may be reused after it is freed with MPI_Close_port. All published names must be unpublished

Data Structures
  Cartesian topology :
    $ndims$: int, $dims$: int array, $periods$: bool array, $coordinate$: int array

$range\_product(ndims, dims, i, j) \doteq$ compute $dims[i] \times \cdots \times dims[j]$
let $F(k) = k > j \ ? \ 1 \ : \ dims[k] * F(k+1)$ in $F(i)$

create a communicator with Cartesian topology
cart_create_init$(comm, ndims, dims, periods, reorder, comm_cart, p) \mathrel{\hat{=}}$
let cid = next_comm_cid in
if $comm.gr|_p = 0$ then syn_put$(comm, cid, \epsilon, p) \ \wedge$ register_cid(cid)
else syn_init(comm, p)

cart_create_wait$(comm, ndims, dims, periods, reorder, comm_cart, p) \mathrel{\hat{=}}$
syn_wait$(comm, p) \wedge$
let $slot \diamond \Gamma$ = rend$[comm.cid]$ in
let $cid = slot.sdata$ in let new_index = unused_item$(\mathtt{comms}_p)$ in
let $comm_{new}$ =
  if $proc \leq range\_product(ndims, dims, 0, ndims - 1)$ then COMM_NULL
  else
    $[create\_comm(comm_{old}, keyvals_p)$ EXCEPT
      $!.cid = cid,$
      $!.group = reorder \ ?$ permute(@) : @
    $] \uplus \langle topology, \ [ndims \mapsto ndims, \ dims \mapsto dims, \ periods \mapsto periods]\rangle$
in $\mathtt{comms}'_p = \mathtt{comms}_p \uplus \langle new\_index, comm_{new}\rangle \wedge$
  $comm\_cart' = new\_index$

$coord\_2\_rank(coord, ndims, dims) \doteq$ convert a coordinate to the rank
let $F(n)$ = if $n = size(coord)$ then 0
  else $range\_product(ndims, dims, n+1, ndims-1) \times coord[n] + F(n+1)$
in $F(0)$

$rank\_2\_coord(rank, ndims, dims) \doteq$ convert a rank to the coordinate
let $F(x, n)$ = if $n = 0$ then $\langle x \rangle$ else $F(x \div dims[n], n-1) \diamond (x \ \% \ dims[n])$
in $F(rank, ndims - 1)$

cart_shift$(comm, dir, disp, p) \doteq$ Cartesian shift coordinates
let $tp = comm.topology$ in
let $\langle dims, ndims\rangle = \langle tp.dims, tp.ndims\rangle$ in
let $rank = comm.group|_p$ in let $coord = rank\_2\_coord(rank, ndims, dims)$ in
let $f(i) =$ compute the rank of a node in a direction
  if $\neg tp.periods[rank] \ \wedge \ (i \leq dims[dir] \ \vee \ i < 0)$ then PROC_NULL
  else $coord\_2\_rank([coord$ EXCEPT $![dir] = i], ndims, dims)$
in $[rank_{source} \mapsto f((@ - disp) \ \% \ dims[dir]),$
    $rank_{dest} \mapsto f((@ + disp) \ \% \ dims[dir])]$

Figure 12: Modeling topology operations

before the corresponding port is closed.

Call `MPI_Comm_accept` is collective over the calling communicator. It returns an intercommunicator that allows communication with the client. In the "init" phase, the root process sets the port's client group to be its group. In the "wait" phase, each process creates a new intercommunicator with the local (remote) group being the server (client) group of the port. Furthermore, the root process sets the port's status to be 'waiting' so that new connection requests from clients can be accepted.

Call `MPI_Comm_connect` establishes communication with a server specified by a port name. It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an `MPI_Comm_accept`. We do not model the time-out mechanism; instead, we assume the time out period is infinitely long (thus will lead to deadlock if there is no matching `MPI_Comm_accept`). As shown in the code, the root process picks a new context id in its "init" phase. In the "wait" phase, each process creates a new intercommunicator; and the root process updates the port so that the server can proceed to create intercommunicators.

4.9 One-sided Communication

Remote Memory Access (RMA) allows one process to specify all communication parameters, both for the sending side and for the receiving side. This mechanism separates the communication of data from the synchronizations.

A process exposes a "window" of its memory accessible by remote processes. The *wins* object represents the group of processes that own and access the set of windows they expose. The management of this object, *e.g.* the creation and destroying of a window, is similar to that of the communicator object *comms* except that window operations are synchronizing.

RMA communication calls associated with a window occur at a process only within an epoch for this window. Such an epoch starts with a RMA synchronization call, proceeds with some RMA communication calls (`MPI_Put`, `MPI_Get` and `MPI_Accumulate`), and completes with another synchronization call. RMA communications fall in two categories: *active target* communication, where both the origin and target processes involve in the communication, and *passive target* communication, where only the origin process involves in the communication. We model active (passive) target communication with the `eps` (`locks`) object.

Data Structures

$port : \langle name : \langle proc : \texttt{int}, port : \texttt{int}\rangle,\ cid : \texttt{int},\ status : \{\text{‘}connected', \text{‘}waiting'\},$
$\quad server\_group : \texttt{oset},\ client\_group : \texttt{oset}\rangle$

$\texttt{open\_port}(port\_name, p) \doteq$ establish a network address
$\texttt{let}\ new\_port\_id = \texttt{unused\_item}(\texttt{ports}_p)\ \texttt{in}$
$\texttt{let}\ new\_port = [name \mapsto \langle p, new\_port\_id\rangle,\ status \mapsto \text{‘}waiting']\ \texttt{in}$
$\texttt{ports}'_p = \texttt{ports}_p \uplus [new\_port\_id \mapsto new\_port] \wedge$
$port\_name' = new\_port.name$

$\texttt{close\_port}(port\_name, p) \doteq$ release a network address
$\texttt{requires}\{port\_name \notin \texttt{service\_names}\}$
$\texttt{ports}'_p = \texttt{remove}(\texttt{ports}_p, port\_name.port)$

the server attempts to establish communication with a client
$\texttt{comm\_accept}_{\texttt{init}}(port\_name, root, comm, newcomm, p) \mathrel{\hat{=}}$
$\texttt{let}\ port\_no = port\_name.port\ \texttt{in}$
$\texttt{if}\ comm.gr|_p = root\ \texttt{then}$
$$\frac{\texttt{ports}_p[port\_no].status = \text{‘}waiting' \ \wedge\ \texttt{syn}_{\texttt{put}}(comm.cid, port\_no, \epsilon, p)}{\texttt{ports}'_p[port\_no] = [\texttt{ports}_p[port\_no]\ \texttt{EXCEPT}\ !.server\_group = comm.group]}$$
$\texttt{else syn}_{\texttt{init}}(\texttt{comm.cid}, \texttt{p})$

$\texttt{comm\_accept}_{\texttt{wait}}(port\_name, root, comm, newcomm, p) \mathrel{\hat{=}}$
$\texttt{let}\ port\_no = \texttt{rend}_p[cid].sdata\ \texttt{in}$
$\texttt{let}\ port = \texttt{ports}_{comm.group[root]}[port\_no]\ \texttt{in}$
$$\frac{\texttt{syn}_{wait}(comm, p)\ \wedge\ port[port\_no].status = \text{‘}connected'}{\begin{array}{l}\texttt{comms}'_p[newcomm] = [\ cid \mapsto port.cid,\ group \mapsto port.server\_group, \\ \qquad remote\_group \mapsto port.client\_group] \wedge \\ (p = comm.group[root]) \Rightarrow \texttt{ports}'_p[port\_no].status = \text{‘}waiting'\end{array}}$$

the client attempts to establish communication with a server
$\texttt{comm\_connect}_{\texttt{init}}(port\_name, root, comm, newcomm, p) \mathrel{\hat{=}}$
$\texttt{let}\ port = \texttt{ports}_{port\_name.proc}[port\_name.port]\ \texttt{in}$
$\texttt{let}\ cid = \texttt{next\_comm\_cidin}$
$\texttt{if}\ comm.gr|_p = root\ \texttt{then}$
$$\frac{port.status = \text{‘}waiting'\ \wedge\ \texttt{syn}_{\texttt{put}}(comm.cid, cid, \epsilon, p)}{\texttt{register\_cid}(cid)}$$
$\texttt{else syn}_{\texttt{init}}(\texttt{comm.cid}, \texttt{p})$

$\texttt{comm\_connect}_{\texttt{wait}}(port\_name, root, comm, newcomm, p) \mathrel{\hat{=}}$
$$\frac{\texttt{syn}_{wait}(comm.cid, p)}{\begin{array}{l}\texttt{let}\ cid = \texttt{rend}_p[comm.cid].sdata\ \texttt{in} \\ \texttt{let}\ port = \texttt{ports}_{comm.group[root]}[port\_no]\ \texttt{in} \\ \texttt{let}\ \langle host, port\_no\rangle = \langle port\_name.proc, port\_name.port\rangle\ \texttt{in} \\ \texttt{comms}'_p[newcomm] = \\ \quad [cid \mapsto cid,\ group \mapsto comm.group, \\ \quad remote\_group \mapsto \texttt{ports}_{host}[port\_no].server\_group] \wedge \\ (p = comm.group[root]) \Rightarrow \\ \quad \texttt{ports}'_p[port\_no].status = \text{‘}connected' \wedge \\ \quad \texttt{ports}'_p[port\_no].client\_group = comm.group \wedge \\ \quad \texttt{ports}'_p[port\_no].cid = cid\end{array}}$$

Figure 13: Modeling client-server communications

MPI_Win_start and MPI_Win_complete start and complete an access epoch (with $mode = ac$) respectively; while MPI_Win_post and MPI_Win_wait start and complete an exposure epoch (with $mode = ex$) respectively. There is one-to-one matching between access epoches at origin processes and exposure epoches on target processes. Distinct access epoches for a window at the same process must be disjoint; so must distinct exposure epoches. In a typical communication, the target process first calls MPI_Win_post to start an exposure epoch, then the origin process calls MPI_Win_start to starts an access epoch, and then after some RMA communications it calls MPI_Win_complete to complete this access epoch, finally the target process calls MPI_Win_wait to complete the exposure epoch. This MPI_Win_post call will block until all matching class to MPI_Win_complete have occured. Both MPI_Win_complete and MPI_Win_wait enforce completion of all preceding RMA calls. If MPI_Win_start is blocking, then the corresponding MPI_Win_post must have executed. However, these calls may be non-blocking and complete ahead of the completion of others.

A process p maintains in $\mathtt{eps}_p$ a queue of epoches. Each epoch contains a sequence of RMA communications yet to be completed. Its $match$ field contains a set of $\langle matching\ process, matching\ epoch\rangle$ tuples, each of which points to a matching epoch at another process. An epoch becomes inactive when it is completed. When a new epoch ep is created and appended to the end of the epoch queue, this matching information is updated by calling the helper function $find\_match$, which locates at a process the first active epoch that has not be matched with ep. Additionally, since MPI_Win_start can be non-blocking such that it may complete before MPI_Win_post is issued, MPI_Win_post needs to update the matching information each time it is called. We do not remove completed epoches because their status may be needed by other processes to perform synchronization.

Designed for passive target communication, MPI_Win_lock and MPI_Win_unlock start and complete an access epoch repsectively. They are similar to those for active target communication, except that no corresponding exposure epoches are needed. Accesses that are protected by an exclusive lock will not be concurrent with other accesses to the same window. We maintain these epoches in a different object locks, which resides in the envs object in our specification.

RMA communication call MPI_Put transfers data from the caller memory to the target memory; MPI_Put transfers data from the target memory to the caller memory; and MPI_Accumulate updates locations in the target memory. When each of these calls is issued, it is appended to the current active access epoch which may be in the eps or locks object. Note that there is at most one active access epoch for a window at each process. The calls in an epoch is performed in a FIFO manner. When a call completes, it is removed from the queue.

The `active_transfer` rule performs data transferring: when the corresponding exposure epoch exists, the first RMA communication call in the current active epoch is carried out and the value v will be written (or reduced) to the memory of the destination. The rule for passive target communication is analogous.

p_0	p_1	p_2
$\texttt{win\_start}(group = \langle 1, 2\rangle, win_0)$	$\texttt{win\_post}(group = \langle 0\rangle, win_0)$	$\texttt{win\_post}(group = \langle 0\rangle, win_0)$
$\texttt{put}(origin = 0, target = 1, win_0)$	$\texttt{win\_wait}(win_0)$	$\texttt{win\_wait}(win_0)$
$\texttt{get}(origin = 0, target = 2, win_0)$		
$\texttt{win\_complete}(win_0)$		

step	$\texttt{eps}_0$	$\texttt{eps}_1$	$\texttt{eps}_2$
1		$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\}\rangle_0^{ex}$	
2		$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\}\rangle_0^{ex}$
3	$\langle 0, \langle 1, 2\rangle, \langle\rangle, \top, \{\langle 1, 0\rangle, \langle 2, 0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$
4	$\langle 0, \langle 1, 2\rangle, \langle\langle 0, 1\rangle^{put}\rangle, \top, \{\langle 1, 0\rangle, \langle 2, 0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$
5	$\langle 0, \langle 1, 2\rangle, \langle\langle 0, 1\rangle^{put} \diamond \langle 0, 1\rangle^{get}\rangle,$ $\top, \{\langle 1, 0\rangle, \langle 2, 0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$
6	$\langle 0, \langle 1, 2\rangle, \langle\langle 0, 2\rangle^{get}\rangle, \top, \{\langle 1, 0\rangle, \langle 2, 0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$
7	$\langle 0, \langle 1, 2\rangle, \langle\rangle, \bot, \{\langle 1, 0\rangle, \langle 2, 0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$
8	$\langle 0, \langle 1, 2\rangle, \langle\rangle, \bot, \{\langle 1, 0\rangle, \langle 2, 0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \bot, \{\langle 0, 0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0, 0\rangle\}\rangle_0^{ex}$
9	$\langle 0, \langle 1, 2\rangle, \langle\rangle, \bot, \{\langle 1, 0\rangle, \langle 2, 0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \bot, \{\langle 0, 0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \bot, \{\langle 0, 0\rangle\}\rangle_0^{ex}$

the execution (format: $event_{step}$) :
$\texttt{win\_post}(\langle 0\rangle, win_0, 1)_1, \texttt{win\_post}(\langle 0\rangle, win_0, 2)_2, \texttt{win\_start}(\langle 1, 2\rangle, win_0, 0)_3, \texttt{put}(0, 1, win_0, 0)_4,$
$\texttt{get}(0, 2, win_0, 0)_5, \texttt{active\_transfer}(0)_6, \texttt{win\_complete}(win_0, 0)_7, \texttt{win\_wait}(win_0, 2)_8, \texttt{win\_wait}(win_0, 1)_9$

Figure 14: An active target communication example. The execution shows a case of strong synchronization in the window win_0's with wid 0. Process p_0 creates an access epoch, p_1 and p_2 creates an exposure epoch respectively. An epoch becomes inactive after it completes. For brevity we omit the value in a RMA operation. The execution follows from the semantics shown in Figure 15 and 16.

4.10 I/O

MPI provides routines for transferring data to or from files on an external storage device. An MPI file is an ordered collection of typed data items. It is opened collectively by a group of processes. All subsequent collective I/O operations on the file are collective over this group.

MPI supports blocking and nonblocking I/O routines. As usual, we model a blocking call by a nonblocking one followed by a wait call such as `MPI_Wait`. In addition to normal collective routines (*e.g.* `MPI_File_read_all`), MPI provides *split collective* data access routines each of which is split into a begin routine and an end routine. Thus two rounds of synchronizations are needed for a collective I/O communication to complete. This is analogous to our splitting the collective communications into an "init" phase and a "wait"

Data Structures

$epoch$:

$\langle wid : \texttt{int},\ group : \texttt{oset},\ rma : (RMA\ communication)\ \texttt{array}, active : \texttt{bool},$
$match : \langle \texttt{int}, \texttt{int}\rangle\ \texttt{set}\rangle^{mode:\{ac,ex,fe\}}$

$lock : \langle wid : \texttt{int},\ RMA : (RMA\ communication)\ \texttt{array},\ active : \texttt{bool}\rangle^{type:\{\text{EXCLUSIVE},\text{SHARED}\}}$

$RMA\ communication : \langle src : \texttt{int},\ dst : \texttt{int},\ value\rangle^{op:\{put,get,accumulate\}}$

$$find\_match(mode, group, p) \doteq \boxed{\text{match access epoches and exposure epoches}}$$
$$\{\langle q, \texttt{first}\ k\rangle \mid \ q \in group\ \wedge\ \texttt{eps}_q[k].mode = mode \wedge p \in \texttt{eps}_q[k].group\ \wedge\ \nexists\langle p, \alpha\rangle \in \texttt{eps}_q[k].match\}$$

$\texttt{win\_post}(group, win, p) \mathrel{\hat{=}} \boxed{\text{start an exposure epoch}}$

$\texttt{requires}\ \{\nexists\langle win.wid, \alpha, \alpha, \top, \alpha\rangle^{ex} \in \texttt{eps}_p\}\ \boxed{\text{non-overlapping requirement}}$
$\texttt{let}\ mt = find\_match(ac, group, p)\ \texttt{in}$
$\quad \texttt{eps}'_p = eps_p \diamond \langle win.wid, group, \langle\rangle, \top, mt\rangle^{ex} \wedge$
$\quad \forall q \in group : \exists\langle q, k\rangle \in mt \Rightarrow \texttt{eps}'_q[k].mt = \texttt{eps}_q[k].mt \cup \langle p, \texttt{len}(\texttt{eps}'_p)\rangle$

$\texttt{win\_start}(group, win, p) \mathrel{\hat{=}} \boxed{\text{start an access epoch}}$

$\texttt{requires}\ \{\nexists\langle win.wid, \alpha, \alpha, \top\rangle^{ac} \in \texttt{eps}_p\}\ \boxed{\text{non-overlapping requirement}}$
$\texttt{let}\ mt = find\_match(ex, group, p)\ \texttt{in}$
$\texttt{let}\ action =$
$\quad \texttt{eps}'_p = eps_p \diamond \langle win.wid, group, \langle\rangle, \top, mt\rangle^{ac} \wedge$
$\quad \forall q \in group : \exists\langle q, k\rangle \in mt \Rightarrow \texttt{eps}'_q[k].mt = \texttt{eps}_q[k].mt \cup \langle p, \texttt{len}(\texttt{eps}'_p)\rangle$
$\texttt{in if}\ \neg is\_block\ \texttt{then}\ action$
$\quad \texttt{else}\ \dfrac{\forall q \in group : \exists ep^{ex} \in \texttt{eps}_q : p \in ep.group}{action}$

$\texttt{win\_complete}(win, p) \mathrel{\hat{=}} \boxed{\text{complete an access epoch}}$

$\texttt{let}\ k = \texttt{first}\ i : \texttt{eps}_p[i].wid = win.wid\ \wedge\ \texttt{eps}_p[i].mode = ac\ \wedge\ \texttt{eps}_p[i].active$
$\texttt{in}$

$$\dfrac{\forall \texttt{eps}_p[k].rma = \langle\rangle}{\begin{array}{l}\texttt{if}\ \neg is\_block\ \texttt{then}\ \texttt{eps}'_p[k].active = \bot \\ \texttt{else}\ \dfrac{\texttt{size}(\texttt{eps}_p[k].match) = \texttt{size}(\texttt{eps}_p[k].group)}{\texttt{eps}'_p[k].active = \bot}\end{array}}$$

$\texttt{win\_wait}(win, p) \mathrel{\hat{=}} \boxed{\text{complete an exposure epoch}}$

$\texttt{let}\ k = \texttt{first}\ i : \texttt{eps}_p[i].wid = win.wid\ \wedge\ \texttt{eps}_p[i].mode = ex\ \wedge\ \texttt{eps}_p[i].active$
$\texttt{in}$

$$\dfrac{\begin{array}{l}\forall call \in \texttt{eps}_p[k] : \neg call.active\ \wedge \\ \forall\langle q, i\rangle \in \texttt{eps}_p[k].match : \neg\texttt{eps}_q[i].active\end{array}}{\texttt{eps}'_p[k].active = \bot}$$

Figure 15: Modeling one-sided communications (I)

post a RMA operation by adding it into the active epoch

$\texttt{RMA\_op}(type, origin, target, disp, v, op, win, p) \triangleq$
 $\texttt{if}\ \exists k : \texttt{locks}_p[i].wid = win.wid\ \wedge\ \texttt{locks}_p[i].active\ \texttt{then}$
  $\texttt{let}\ k = \texttt{first}\ i : \texttt{locks}_p[i].wid = win.wid\ \wedge\ \texttt{locks}_p[i].active$
  $\texttt{in}\ \texttt{locks}'_p[k].rma = \texttt{locks}_p[k].rma \diamond \langle origin, target, disp, v, op\rangle^{type}$
 $\texttt{else}$
  $\texttt{let}\ k = \texttt{first}\ i :$
   $\texttt{eps}_p[i].wid = win.wid\ \wedge\ \texttt{eps}_p[i].mode = ac\ \wedge\ \texttt{eps}_p[i].active$
  $\texttt{in}\ \texttt{eps}'_p[k].rma = \texttt{eps}_p[k].rma \diamond \langle origin, target, disp, v, op\rangle^{type}$

$\texttt{put}(origin, target, addr_{origin}, disp_{target}, win, p) \triangleq$ the "put" operation
$\texttt{RMA\_op}(put, origin, target, disp_{target}, read\_data(\texttt{mems}_p, addr_{addr}), win, p)$

perform active message passing origining at process p

$\texttt{active\_transfer}(p) \triangleq$
$\texttt{let}\ k = \texttt{first}\ i : \texttt{eps}_p[i].mode = ac\ \wedge\ \texttt{eps}_p[i].rma \neq \langle\rangle\ \texttt{in}$
$\texttt{let}\ \langle src, dst, disp, v, op\rangle^{type} \diamond \Gamma = \texttt{eps}_p[k]\ \texttt{in}$
 $\texttt{eps}'_p[k].rma = \Gamma\ \wedge$
 $\texttt{if}\ type = get\ \texttt{then}\ \texttt{mems}'_p = write\_data(\texttt{mems}_p, win.base + disp, v)$
 $\texttt{else if}\ type = put\ \texttt{then}$
  $\texttt{let}\ \langle q, \alpha\rangle = \texttt{eps}_p[k].match\ \texttt{in}$
  $\texttt{mems}'_q = write\_data(\texttt{mems}_q, win.base + disp, v)$
 $\texttt{else}$
  $\texttt{let}\ \langle q, \alpha\rangle = \texttt{eps}_p[k].match\ \texttt{in}$
  $\texttt{mems}'_p = reduce\_data(\texttt{mems}_p, win.base + disp, v, op)$

start an access epoch for passive target communication

$\texttt{win\_lock}(lock\_type, dst, win, p) \triangleq$
 $\texttt{requires}\ \{\nexists \langle win.wid, \alpha, \alpha, \top\rangle^{ex} \in \texttt{eps}_p\}$ non-overlapping requirement
 $\texttt{if}\ lock\_type = \texttt{SHARED}\ \texttt{then}$
  $\texttt{locks}'_p = locks_p \diamond \langle win.wid, dst, \langle\rangle, \top\rangle^{lock\_type}$
 $\texttt{else}$
  $$\frac{\forall q \in win.group : \nexists k : \texttt{locks}_q[k].wid = win.wid\ \wedge\ \texttt{locks}_q[k].active}{\texttt{locks}'_p = locks_p \diamond \langle win.wid, dst, \langle\rangle, \top\rangle^{lock\_type}}$$

complete an access epoch for passive target communication

$\texttt{win\_unlock}(dst, win, p) \triangleq$
 $\texttt{let}\ k = \texttt{first}\ i : \texttt{locks}_p[i].wid = win.wid\ \wedge\ \texttt{eps}_p[i].active$
 $\texttt{in}$
  $$\frac{\texttt{locks}_p[k].rma = \langle\rangle}{\texttt{locks}'_p[k].active = \bot}$$

Figure 16: Modeling one-sided communications (II)

phase.

Since at each process each file handle may have at most one active split collective operation, the `frend` object, which represents the place where processes rendezvous, stores the information of one operation rather than a queue of operations for each file.

With respect to this fact, we design a protocol shown below to implement collective I/O communications: in the first "begin" phase, process p will proceed to its "end" phase provided that it has not participated in the current synchronization (say `syn`) and `syn`'s status is $`entering'$ (or $`e'$). Note that if all expected processes have participated then `syn`'s status will advance to $`leaving'$ (or $l$). In the "end" phase, $p$ is blocked if `syn` is not in leaving status or $p$ has left. The last leaving process will delete the `syn`. Here notation Ψ represents the participants of a synchronization.

`Data Structures`
`frend` for each file :
$\langle status : \{`e', `l'\},\ participants(\psi) : \texttt{int set},$
$[shared\_data], [data : \langle proc : \texttt{int}, data\rangle\ \texttt{set}]\rangle$

$\mathtt{file_{put}(fid, v_s, v, p)} \doteq$ process p joins the synchronization
$\texttt{if}\ fid \notin \texttt{DOM frend then frend}'[fid] = \langle `e`, \{p\}, v_s, \{\langle p, v\rangle\}\rangle$
`else`

$$\frac{\texttt{frend}[fid] = \langle `e', \Psi, v_{s_1}, S_v\rangle \ \wedge\ p \notin \Psi}{\begin{array}{l}\texttt{frend}'[fid] = \langle\ \ (\Psi \cup \{p\} = \texttt{files}_p[fid].group)\ ?\ `l`\ :\ `e`, \\ \qquad\qquad\qquad\ \ \Psi \cup\ \{p\},\ v_s,\ S_v \cup \{\langle p, v\rangle\}\rangle\end{array}}$$

$\mathtt{file_{begin}(fid, p)} \doteq \mathtt{file_{put}(fid}, \epsilon, \epsilon, \mathtt{p)}$
$\mathtt{file_{write}(fid, v, p)} \doteq \mathtt{file_{put}(fid}, \epsilon, \mathtt{v, p)}$

$\mathtt{file_{end}(fid, p)} \doteq$ process p leaves the synchronization

$$\frac{\texttt{frend}[fid] = \langle `l', \Psi \cup \{p\}, v_s, S_v\rangle}{\texttt{frend}'[fid] = \texttt{if}\ \Psi = \{\}\ \texttt{then}\ \epsilon\ \texttt{else}\ \langle `l', \Psi,\ v_s,\ S_v\rangle}$$

We use the `files` object to store the file information, which includes an *individual* file pointer, which is local to a process, and a *shared* file pointer, which is shared by the group of processes that opened the file. These pointers are used to locate the positions in the file relative to the current view. A file is opened by the `MPI_File_open` call, which is collective over all participanting processes.

When a process p wants to access the file in the operating system `os.file`, it appends a read or write request to its request queue $\texttt{freqs}_p$. A request contains information about the offset in the file, the buffer address in the memory, the number of items to be read, and a flag indicating whether this request is active or not. The MPI system schedules the requests in the queue asynchronously, allowing the first active access to take effect at any time. After the access is finished, the request becomes inactive, and a subsequent wait call will return without being blocked. Note that we need to move the file pointers after the access to the file.

Analogous to usual collective communications, a split collective data access call is split into a begin phase and a end phase. For example, in the begin phase a collective read access reads the data from the file and stores the data in the `frend` object; then in the end phase it fetches the data and updates its own memory.

4.11 Evaluation

How to ensure that our formalization is faithful with the English description? To attack this problem we rely heavily on testing in our formal framework. We provide comprehensive unit tests and a rich set of short litmus tests of the specification. Generally it suffices to test local, collective, and asynchronous MPI primitives on one, two and three processes respectively. These test cases, which include many simple examples in the MPI reference, are hand-written directly in TLA+ and modeled checked using TLC. As we have mentioned in Section 3, thanks to the power of the TLC model checker our framework supports thorough testing of MPI programs, thus giving more precise answers than vendor MPI implementations can.

Another set of test cases are built to verify the *self-consistency* of the specification. For a communication (pattern), there may be many ways to express it. Thus it is possible to relate aspects of MPI to each other. Actually, in the MPI definition certain MPI functions are explained in terms of other MPI functions.

We introduce the notation $\mathtt{MPI\_}A \simeq \mathtt{MPI\_}B$ to indicate that A and B have the same functionality with respect to their semantics.

Our specification defines a blocking point-to-point operation by a corresponding nonblocking operation followed immediately by a `MPI_Wait` operation. Thus we have

$$\begin{aligned}
\mathtt{MPI_Send(n)} &\simeq \mathtt{MPI_Isend(n)} + \mathtt{MPI_Wait} \\
\mathtt{MPI_Recv(n)} &\simeq \mathtt{MPI_Irecv(n)} + \mathtt{MPI_Wait} \\
\mathtt{MPI_Sendrecv(n_1, n_2)} &\simeq \mathtt{MPI_Isend(n_1)} + \mathtt{MPI_Irecv(n_2)} + \mathtt{MPI_Wait} + \mathtt{MPI_Wait}
\end{aligned}$$

Typical relationships between the MPI communication routines, together with some examples, include:

$\texttt{Data Structures}$

file information at a process :

$fid : \texttt{int},\ group : \texttt{oset},\ fname : \texttt{string},\ amode :\ mode\ \texttt{set},\ size : \texttt{int},\ view,$
$pts : \langle p_{shared}, \texttt{int},\ p_{ind} : \texttt{int}\rangle$

file access request :

$\langle fh,\ offset : \texttt{int},\ buf : \texttt{int},\ count : \texttt{int},\ active : \texttt{bool}\rangle$

$\texttt{iread}(fh, offset, buf, count, request, p) \doteq$ nonblocking file access
$\quad \texttt{freqs}'_p = \texttt{freqs}_p \diamond \langle fh, offset, buf, count, \top\rangle^{read} \wedge$
$\quad \texttt{mems}'_p[request] = \texttt{size}(\texttt{freqs}_p)$
$\texttt{iwrite}(fh, offset, buf, count, request, p) \doteq$
$\quad \texttt{freqs}'_p = \texttt{freqs}_p \diamond \langle fh, offset, buf, count, \top\rangle^{write} \wedge$
$\quad \texttt{mems}'_p[request] = \texttt{size}(\texttt{freqs}_p)$

$\texttt{file\_access}(p) \doteq$ perform file access asynchronously
$\texttt{let}\ \langle fh, offset, buf, count, \top\rangle^{mode} \diamond \Gamma = \texttt{freqs}_p\ \texttt{in}$
$\wedge\ \texttt{freqs}'_p = \langle fh, offset, buf, count, \bot\rangle^{mode} \diamond \Gamma$
$\wedge\ \texttt{if}\ mode = write\ \texttt{then}$
$\quad\quad \texttt{let}\ v = read\_mem(\texttt{mems}_p, buf, count)\ \texttt{in}$
$\quad\quad\quad \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v) \wedge$
$\quad\quad\quad \texttt{os.file}'_p = write\_file(fh, \texttt{os.file}, v)$
$\quad \texttt{else}$
$\quad\quad \texttt{let}\ v = read\_file(fh, \texttt{os.file}_p, offset, count)\ \texttt{in}$
$\quad\quad\quad \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v) \wedge$
$\quad\quad\quad \texttt{mems}'_p = write\_mem(\texttt{mems}_p, buf, v)$

$\texttt{file\_wait}(req, p) \doteq$
$\texttt{let}\ \Gamma_1 \diamond \langle fh, offset, buf, count, \bot\rangle^{mode}_{req} \diamond \Gamma_2 = \texttt{freqs}_p$
$\texttt{in}\ \texttt{freqs}'_p = \Gamma_1 \diamond \Gamma_2$ remove the request

the begin call of a split collective file read operation
$\texttt{file\_read\_all}_{\texttt{begin}}(fh, offset, buf, count, p) \doteq$
$\quad \texttt{file}_{\texttt{write}}(fh.fid, read\_file(fh, \texttt{os.file}_p, offset, count), p)$

$\texttt{file\_read\_all}_{\texttt{end}}(fh, buf, p) \doteq$ the end call of a split collective file read operation

$$\frac{\texttt{file}_{\texttt{end}}(fh.fid, p)}{\begin{array}{l}\texttt{let}\ v = \texttt{frend}[fh.fid].data[p]\ \texttt{in}\\ \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v) \wedge\\ \texttt{mems}'_p = write\_mem(\texttt{mems}_p, buf, v)\end{array}}$$

the begin call of a split collective file write operation
$\texttt{file\_write\_all}_{\texttt{begin}}(fh, buf, count, p) \doteq$
$\quad \texttt{file}_{\texttt{write}}(fh.fid, read\_mem(\texttt{mems}_p, buf, count), p)$

$\texttt{file\_write\_all}_{\texttt{end}}(fh, buf, p) \doteq$ the begin call of a split collective file write operation

$$\frac{\texttt{file}_{\texttt{end}}(fh.fid, p)}{\begin{array}{l}\texttt{let}\ v = \texttt{frend}[fh.fid].data[p]\ \texttt{in}\\ \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v) \wedge\\ \texttt{os.file}'_p = write\_file(fh, \texttt{os.file}_p, v)\end{array}}$$

Figure 17: Modeling I/O operations

- A message can be divided into multiple sub-messages sent separately.

$$\begin{gathered}\texttt{MPI\_A(k}\times\texttt{n)}\simeq\texttt{MPI\_A(n)}_1+\cdots+\texttt{MPI\_A(n)}_k\\ \texttt{MPI\_A(k}\times\texttt{n)}\simeq\texttt{MPI\_A(k)}_1+\cdots+\texttt{MPI\_A(k)}_n\end{gathered}$$

- A collective routine can be replaced by several point-to-point or one-sided routines.

$$\begin{gathered}\texttt{MPI\_Bcast(n)}\simeq\texttt{MPI\_Send(n)}+\cdots+\texttt{MPI\_Send(n)}\\ \texttt{MPI\_Gather(n)}\simeq\texttt{MPI\_Recv(n/p)}_1+\cdots+\texttt{MPI\_Recv(n/p)}_p\end{gathered}$$

- Communications using `MPI_Send`, `MPI_Recv` can be implemented by one-sided communications.

$$\begin{aligned}&\texttt{MPI\_Win\_fence}+\texttt{MPI\_Get(n)}+\texttt{MPI\_Win\_fence}\simeq\\ &\texttt{MPI\_Barrier}+\texttt{MPI\_Recv(d)}+\texttt{MPI\_Recv(n)}+\texttt{MPI\_Barrier},\\ &\text{where } d \text{ is the address and datatype information}\end{aligned}$$

- Process topologies do not affect the results of message passing. Communications using a communicator that implements a random topology should has the same semantics as the communication with a process topology (like a Cartesian topology).

Our specification is shown to meet the correctness requirements by model checking test cases.

4.12 Discussion

It is important to point out that we have not modeled all the details of the MPI standard. We list below the details that are omitted and the reasons why we do not model them:

- *Implementation details.* To the greatest extent possible we have avoided asserting implementation-specific details in our formal semantics. One obvious example is that the **info** object, which is one arguments of some MPI 2.0 functions, is ignored.
- *Physical Hardware.* The underlying, physical hardware is invisible in our model. Thus we do not model low-level topology functions such as `MPI_Cart_map` and `MPI_Graph_map`.
- *Profiling Interface.* The MPI profiling interface is to permit the implementation of profiling tools. It is irrelevant to the semantics of MPI functions.

- *Runtime Environment.* Since we do not model the operation system to allow for the dynamic process management (*e.g.* process creation and cooperative process termination), MPI routines accessing the runtime environment such as `MPI_Comm_spawn` are not modeled. Functions associated with the thread environment are not specified either.

Often our formal specifications mimic programs written using detailed data structures, *i.e.* they are not as "declarative" as possible. We believe that this is in some sense inevitable when attempting to obtain executable semantics of real world APIs. Even so, TLA+ based "programs" can be considered superior to executable models created in C: (i) the notation has a precise semantics, as opposed to C, (ii) another specification in a programming language can provide complementary details, (iii) in our experience, there are still plenty of short but tricky MPI programs that can be executed fast in our framework.

5 Verification Framework

Our modeling framework uses the Microsoft Phoenix [16] Compiler as a front-end for C programs. Of course other front-end tools such as GCC can also be used. The Phoenix framework allows developers to insert a compilation phase between existing compiler phases in the process of lowering a program from language independent MSIL (Microsoft Intermediate Language) to device specific assembly. We place our phase at the point where the input program has (i) been simplified into a single static assignment (SSA) form, with (ii) a homogenized pointer referencing style that is (iii) still device independent.

From Phoenix intermediate representation (IR) we build a state-transition system by converting the control flow graph into TLA+ relations and mapping MPI primitives to their names in TLA+. Specifically, control locations in the program are represented by states, and program statements are represented using transitions. Assignments are modeled by their effect on the memory. Jumps have standard transition rules modifying the values of the program counters. This transition system will completely capture the control skeleton of the input MPI program.

The architecture of the verification framework is shown in Figure 18. The user may input a program in any language that can be compiled using the Phoenix back-end — we have experimented only with C. The program is compiled into an intermediate representation, the Phoenix IR. We read the Phoenix IR to create a separate intermediate representation, which is used to produce TLA+ code. The TLC model checker integrated in our framework

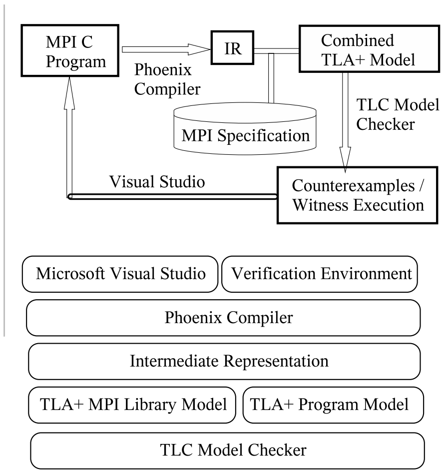


Figure 18: Architecture of the verification framework. The upper (bottom) one indicates the flow (hierarchical) relation of the components.

enables us to perform verification on the input C programs. If an error is found, the error trail is then made available to the verification environment, and can be used by our tool to drive the Visual Studio debugger to replay the trace to the error. In the following we describe the simplification, code generation and replay capabilities of our framework.

Simplification. In order to reduce the complexity of model checking, we perform a sequence of transformations: (i) inline all user defined functions (currently function pointers and recursion are not supported); (ii) remove operations foreign to the model checking framework, e.g. `printf`; (iii) slice the model with respect to communications and user assertions: the cone of influence of variables is computed using a chaotic iteration over the program graph, similar to what is described in [18]; and (iv) eliminate redundant counting loops.

Code Generation. During the translation from Phoenix IR to TLA+, we build a record map to store all the variables in the intermediate language. The address of a variable x is given by the TLA+ expression $map.x$; and its value at the memory is returned by $mems[map.x]$. Before running the TLC, the initial values of all constants and variables are specified (*e.g.* in a configuration file). The format of the main transition relation is shown below, where N is the number of processes, and $predefined\_nxt$ is the "system" transition which performs message passing for point-to-point communications, one-sided communications, and so on. In addition, "program" transitions $transition_1, transition_2, \cdots$ are produced by translating MPI function calls and IR statements. In the examples shown later we only show the program transition part.

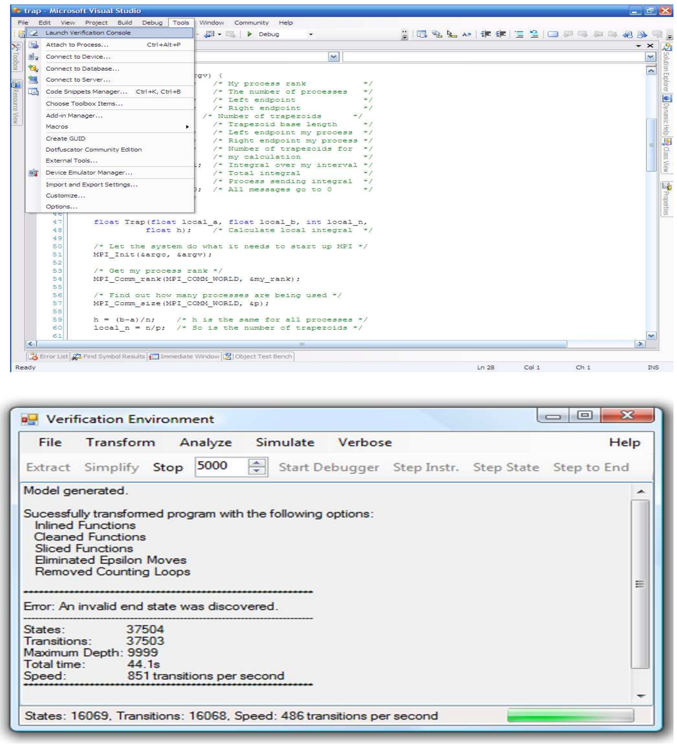


Figure 19: Two screenshots of the verification framework.

$$
\begin{array}{ll}
\lor & \land\ predefined\_nxt \quad \text{transitions performed by the MSS} \\
 & \land\ \texttt{UNCHANGED}\ \langle\langle map \rangle\rangle \\
\lor & \exists pid \in 0..(N-1): \quad \text{execute an enabled transition at a process} \\
 & \lor\ transition_1 \\
 & \lor\ transition_2 \\
 & \lor \cdots \\
\lor & \forall pid \in 0..(N-1): \quad \text{eliminate spurious deadlocks} \\
 & \land\ pc[pid] = last\_label \\
 & \land\ \texttt{UNCHANGED}\ all\_varaibles
\end{array}
$$

Error Trail Generation. In the event that the model contains an error, an error trail is produced by the model checker and returned to the verification environment. To map the error trail back onto the actual program we observe MPI function calls and the changes in the error trail to variable values that appear in the program text. For each change on a variable, we step the Visual Studio debugger until the corresponding value of the variable in the debugger matches. We also observe which process moves at every step in the error trail and context switch between processes in the debugger at corresponding points. When the error trail ends, the debugger is within a few steps of the error with the process that causes the error scheduled. The screenshots in figure 19 show the debugger interface and the report of an error trace.

Examples. A simple C program containing only one statement "if (rank == 0) MPI_Bcast (&b, 1, MPI_INT, 0, comm1) " is translated to:

$$
\begin{array}{ll}
\lor & \land\ pc[pid] = L_1\ \land\ pc' = [pc\ \texttt{EXCEPT}\ ![pid] = L_2] \\
 & \land\ mems' = [mems\ \texttt{EXCEPT}\ ![pid] = [@\ \texttt{EXCEPT}\ ![map.t_1] = (mems[pid][map.\_rank] = 0)]] \\
\lor & \land\ pc[pid] = L_2\ \land\ mems[pid][map.t_1] \\
 & \land\ pc' = [pc\ \texttt{EXCEPT}\ ![pid] = L_3] \\
\lor & \land\ pc[pid] = L_2\ \land\ \neg(mems[pid][map.t_1]) \\
 & \land\ pc' = [pc\ \texttt{EXCEPT}\ ![pid] = L_5] \\
\lor & \land\ pc[pid] = L_3\ \land\ pc' = [pc\ \texttt{EXCEPT}\ ![pid] = L_4] \\
 & \land\ \text{MPI\_Bcast\_init}(map.\_b, 1, \text{MPI\_INT}, 0, map.\_comm1, pid) \\
\lor & \land\ pc[pid] = L_4\ \land\ pc' = [pc\ \texttt{EXCEPT}\ ![pid] = L_5] \\
 & \land\ \text{MPI\_Bcast\_wait}(map.\_b, 1, \text{MPI\_INT}, 0, map.\_comm1, pid)
\end{array}
$$

At label L_1, the value of $rank == 0$ is assigned to a temporary variable t_1, and the pc advances to L_2. In the next step, if the value of t_1 is true, then the pc advances to L_3; otherwise to the exit label L_5. The broadcast is divided into an "init" phase (where pc advances from L_3 to $L-4$) and a "wait" phase (where pc advances from L_4 to $L-5$). In Figure 20 we show a more complicated example.

The source C program:

```
int main(int argc, char* argv[])
{
  int rank;
  int data;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    data = 10;
    MPI_Send(&data,1,MPI_INT,1,0,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(&data,1,MPI_INT,0,0,MPI_COMM_WORLD, &status);
  }
  MPI_Finalize();
  return 0;
}
```

The TLA+ code generated by the compiler:

$$
\begin{array}{ll}
\vee & \wedge\, pc[pid] = \_main \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_1] \\
 & \wedge\, \text{MPI\_Init}(map.\_argc, map.\_argv, pid) \\
\vee & \wedge\, pc[pid] = L_7 \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_9] \\
 & \wedge\, mems' = [mems \texttt{ EXCEPT } ![pid] = Update(@, map.\_data, 10)] \\
 & \wedge\, changed(mems) \\
\vee & \wedge\, pc[pid] = L_6 \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_{14}] \\
 & \wedge\, \text{MPI\_Irecv}(map.\_data, 1, \text{MPI\_INT}, 0, 0, \text{MPI\_COMM\_WORLD}, map.tmprequest_1, pid) \\
\vee & \wedge\, pc[pid] = L_1 \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_2] \\
 & \wedge\, \text{MPI\_Comm\_rank}(\text{MPI\_COMM\_WORLD}, map.\_rank, pid) \\
\vee & \wedge\, pc[pid] = L_2 \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_5] \\
 & \wedge\, mems' = [mems \texttt{ EXCEPT } ![pid] = \\
 & \qquad Update(@, map.t_{277}, mems[pid][map.\_rank] = 0)] \\
 & \wedge\, changed(mems) \\
\vee & \wedge\, pc[pid] = L_5 \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_7] \\
 & \wedge\, mems[pid][map.t_{277}] \\
\vee & \wedge\, pc[pid] = L_5 \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_6] \\
 & \wedge\, \neg(mems[pid][map.t_{277}]) \\
\vee & \wedge\, pc[pid] = L_9 \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_{13}] \\
 & \wedge\, \text{MPI\_Isend}(map._data, 1, \text{MPI\_INT}, 1, 0, \text{MPI\_COMM\_WORLD}, map.tmprequest_0, pid) \\
\vee & \wedge\, pc[pid] = L_{11} \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_{12}] \\
 & \wedge\, \text{MPI\_Finalize}(pid) \\
\vee & \wedge\, pc[pid] = L_{13} \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_{11}] \\
 & \wedge\, \text{MPI\_Wait}(map.tmprequest_0, map.tmpstatus_0, pid) \\
\vee & \wedge\, pc[pid] = L_{14} \;\wedge\; pc' = [pc \texttt{ EXCEPT } ![pid] = L_{11}] \\
 & \wedge\, \text{MPI\_Wait}(map.tmprequest_1, map.\_status, pid)
\end{array}
$$

Figure 20: An example C program and its corresponding TLA+ code.

When we run the TLC to demonstrate the absence of deadlocks for 2 processes, 51 distinct states are visited, and the depth of the complete state graph search is 17. The verification time is less than 0.1 second on a 3GHz processor with 1GB of memory. However, although it suffices in general to perform the test on a small number of processes, increasing the number of processes will increase the verification time exponentially. Thus we are implementing efficient methods such as partial order reduction algorithms [26][36] to reduce the state space.

6 Conclusion

To help reason about programs that use MPI for communication, we have developed a formal TLA+ semantic definition of MPI 2.0 operations to augment the existing standard. We described this formal specification, as well as our framework to extract models from SPMD-style C programs. We discuss how the framework incorporates high level formal specifications, and yet allows designers to experiment with these specifications, using model checking, in a familiar debugging environment. Our effort has helped identify a few omissions and ambiguities in the original MPI reference standard document. The experience gained so far suggests that a formal semantic definition and exploration approach as described here must accompany every future effort in creating parallel and distributed programming libraries.

In future, we hope to write general theorems (inspired by our litmus tests), and establish them using the Isabelle theorem prover that has a tight TLA+ integration.

References

[1] Martín Abadi, Leslie Lamport, and Stephan Merz, *A tla solution to the rpc-memory specification problem.*, Formal Systems Specification, 1994, pp. 21–66.

[2] Steven Barrus, Ganesh Gopalakrishnan, Robert M. Kirby, and Robert Palmer, *Verification of MPI programs using SPIN*, Tech. Report UUCS-04-008, The University of Utah, 2004.

[3] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough, *Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets*, SIGCOMM, 2005, pp. 265–276.

[4] ———, *Engineering with logic: Hol specification and symbolic-evaluation testing for tcp implementations*, Symposium on the Principles of Programming Languages (POPL), 2006, pp. 55–66.

[5] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany, *Transformations to parallel codes for communication-computation overlap*, SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (Washington, DC, USA), IEEE Computer Society, 2005, p. 58.

[6] P. Van Eijk and Michel Diaz (eds.), *Formal description technique lotos: Results of the esprit sedos project*, Elsevier Science Inc., New York, NY, USA, 1989.

[7] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall, *Open MPI: Goals, concept, and design of a next generation MPI implementation*, Proceedings, 11th European PVM/MPI Users' Group Meeting (Budapest, Hungary), September 2004, pp. 97–104.

[8] Philippe Georgelin, Laurence Pierre, and Tin Nguyen, *A formal specification of the MPI primitives and communication mechanisms*, Tech. report, LIM, 1999.

[9] William Gropp, Ewing L. Lusk, Nathan E. Doss, and Anthony Skjellum, *A high-performance, portable implementation of the mpi message passing interface standard*, Parallel Computing **22** (1996), no. 6, 789–828.

[10] John Harrison, *Formal verification of square root algorithms*, Formal Methods in System Design **22** (2003), no. 2, 143–154, Guest Editors: Ganesh Gopalakrishnan and Warren Hunt, Jr.

[11] G. Holzmann, *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), no. 5, 279–295.

[12] IEEE, *IEEE Standard for Radix-independent Floating-point Arithmetic, ANSI/IEEE Std 854-1987*, 1987.

[13] Daniel Jackson, *Alloy: A new technology for software modeling*, TACAS '02, LNCS, vol. 2280, Springer, 2002, p. 20.

[14] Daniel Jackson, Ian Schechter, and Hya Shlyahter, *Alcoa: the alloy constraint analyzer*, ICSE '00: Proceedings of the 22nd international conference on Software engineering (New York, NY, USA), ACM Press, 2000, pp. 730–733.

[15] Guodong Li, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby, *Formal specification of the MPI-2.0 standard in TLA+*, Principles and Practices of Parallel Programming (PPoPP), 2008.

[16] Microsoft, *Phoenix academic program*, http://research.microsoft.com/phoenix, 2007.

[17] http://www.cs.utah.edu/formal_verification/mpitla/.

[18] Flemming Nielson, Hanne R. Nielson, and Chris Hankin, *Principles of program analysis*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[19] Michael Norrish, *C formalised in HOL*, Ph.D. thesis, University of Cambridge, 1998.

[20] Michael Norrish and Konrad Slind, *HOL-4 manuals*, 1998-2006, Available at `http://hol.sourceforge.net/`.

[21] Robert Palmer, Steven Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby, *Gauss: A framework for verifying scientific computing software.*, SoftMC: Workshop on Software Model Checking, ENTCS, no. 953, August 2005.

[22] Robert Palmer, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby, *An approach to formalization and analysis of message passing libraries*, Formal Methods for Industry Critical Systems (FMICS) (Berlin), 2007, Best Paper Award.

[23] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby, *The communication semantics of the message passing interface*, Tech. Report UUCS-06-012, The University of Utah, October 2006.

[24] ———, *Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs*, PADTAD '07, 2007.

[25] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp, *Formal verification of programs that use MPI one-sided communication*, Recent Advances in Parallel Virtual Machine and Message Passing Interface (Berlin/Heidelberg), LNCS, vol. 4192/2006, Springer, 2006, pp. 30–39.

[26] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp, *Formal verification of programs that use MPI one-sided communication*, Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI), LNCS 4192, 2006, Outstanding Paper, pp. 30–39.

[27] http://sal.csl.sri.com/.

[28] Stephen F. Siegel, *Model Checking Nonblocking MPI Programs*, VMCAI 07 (B. Cook and A. Podelski, eds.), LNCS, no. 4349, Springer-Verlag, 2007, pp. 44–58.

[29] Stephen F. Siegel and George Avrunin, *Analysis of mpi programs*, Tech. Report UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst, 2003.

[30] Stephen F. Siegel and George S. Avrunin, *Modeling wildcard-free MPI programs for verification*, ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (Chicago), June 2005, pp. 95–106.

[31] Jeffrey M. Squyres and Andrew Lumsdaine, *A Component Architecture for LAM/MPI*, Proceedings, 10th European PVM/MPI Users' Group Meeting (Venice, Italy), Lecture Notes in Computer Science, no. 2840, Springer-Verlag, September / October 2003, pp. 379–387.

[32] The Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.*, 1995.

[33] Leslie Lamport, research.microsoft.com/users/lamport/tla/tla.html.

[34] Leslie Lamport, The Win32 Threads API Specification. research.microsoft.com/users/lamport/tla/threads/threads.html.

[35] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby, *Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings*, 20th International Conference on Computer Aided Verification (CAV), 2008.

[36] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby, *Isp: A tool for model checking mpi programs*, Principles and Practices of Parallel Programming (PPoPP), 2008.

A Soundness Proof with Formal Semantics

The main problem of model checking MPI programs is the state space explosion problem. This problem may be mitigated by using partial order reduction techniques. A sound partial-order reduction guarantees that if there is a property violation in the full state space, that violation will be discovered by the model checker while enumerating a subset of the state space.

We have developed several partial order reduction (DPOR) algorithms [24, 26, 35] to model check MPI programs. For instance, the ISP checker [35] exploits the out of order completion semantics of MPI by issuing MPI calls according to match-sets which are ample 'big-step' moves. The core of a DPOR algorithm is to base on an dependence analysis to determine when it is safe to execute only a subset of the enabled calls. Such dependence information is computed based on the semantics of MPI calls. In this section we show how to justify the definition of dependence in our DPOR algorithms according to the formal semantics of MPI calls.

Our goal is to prove the soundness of the *complete-before* relation $\prec$ defined in [35]. Relation $\prec$ specifies the order enforced on the completion of MPI calls. An MPI immediate send $S_{i,j}(k, \langle i, j\rangle, \ldots)$, where k is the process targeted, i, j is the request handle used to track the processes of this send, completes when it matches a receive (e.g. by the MPI System Scheduler). An MPI immediate receive $S_{i,j}(k, \langle i, j\rangle, \ldots)$, where k is the process from which the message is sourced ($k = *$ means a 'wildcard receive'), completes when it receives the message. A barrier operation $B_{i,j}$ completes when all participants exit the synchronization. A wait operation $W_{i,j}\langle i, j\rangle$ completes when the corresponding send (receive) operation completes and the data has been sent out (copied into the target process's memory).

The formal definition of the completes-before relation is given below as eight rules.

(Css-kk) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow S_{i,j_1}(k, \ldots)\ \prec S_{i,j_2}(k, \ldots)$
(Crr-kk) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow R_{i,j_1}(k, \ldots)\ \prec R_{i,j_2}(k, \ldots)$
(Crr-*k) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow R_{i,j_1}(*, \ldots)\ \prec R_{i,j_2}(k, \ldots)$
(Crr-**) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow R_{i,j_1}(*, \ldots)\ \prec R_{i,j_2}(*, \ldots)$
(Csw) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow S_{i,j_1}(k, \langle i, j_1\rangle)\ \prec W_{i,j_2}(\langle i, j_1\rangle)$
(Crw) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow R_{i,j_1}(k, \langle i, j_1\rangle)\ \prec W_{i,j_2}(\langle i, j_1\rangle)$
(Cb) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow B_{i,j_1}\ \prec any_{i,j_2}(\ldots)$
(Cw) $\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow W_{i,j_1}(\ldots)\ \prec any_{i,j_2}(\ldots)$

Now we proceed to prove the correctness of these rules with respect to our formal semantics. As in [35], We abstract away such fields as communicator ID, tag, prematch, value and flags. First of all, rule `Csw` and rule `Crw` are valid because a blocking send or receive operation is modeled by a non-blocking operation followed by a wait operation. As indicated in the semantics, a non-blocking operation sets the active flag of the request, and the corresponding wait operation can return only if this flag is set. Hence these two operations cannot execute out of order.

A.0.1 Send and Receive.

Now consider the `Css-kk` rule, which specifies the order of two immediate sends from process i to process k. Assume that the request queue at process i contains two active send requests $S_{i,j_1}(k,\ldots)$ and $S_{i,j_2}(k,\ldots)$:

$$\langle k,\ldots\rangle_{j_1}^{send} \diamond \langle k,\ldots\rangle_{j_2}^{send}$$

Suppose for contradiction that request j_2 may complete before request j_1. In order for j_2 to complete, there must exist a receive request $\langle buf,i,\ldots\rangle_n^{recv}$ at process k that matches this send request, and the following condition specified in the `transfer` rule must hold (note that the first request $\langle k,\ldots\rangle_{j_1}^{send}$ is in Γ_1^i):

$$\nexists\langle k,\ldots\rangle_m^{send} \in \Gamma_1^i : \langle i,k,\ldots,m\rangle \leftrightharpoons \langle i,k,\ldots,n\rangle$$

However, if m equals to j_1, then this condition is false immediately because request j_1 matches the receive request. This contradiction implies the correctness of rule `Css-kk`. Rule `Crr-kk` can be proved in a similar way.

Let us look at rule `Crr-*k` and rule `Crr-**`, where the first receive is a wildcard receive. Assume that the request queue at process i contains two active receive requests $R_{i,j_1}(*,\ldots)$ and $R_{i,j_2}(k_*,\ldots)$. In the second receive either $k_* = k$ (*i.e.* the source is process k) or $k_* = *$ (*i.e.* it is a wildcard receive):

$$\langle buf_1,*,\ldots\rangle_{j_1}^{recv} \diamond \langle buf_2,k_*,\ldots\rangle_{j_2}^{recv}$$

If request j_2 completes before request j_1, then there must exist a send request $\langle i,\ldots\rangle_n^{send}$ at a process p (which may be k) that matches this receive request, and the FIFO condition specified in the `transfer` rule must hold. In other words, we have

$$\begin{array}{l} \langle p,i,\ldots,n\rangle \leftrightharpoons \langle k_*,i,\ldots,j_2\rangle \wedge \\ \nexists\langle buf,q,\ldots\rangle_m^{recv} \in \Gamma_1^i : \langle p,i,\ldots,n\rangle \leftrightharpoons \langle q,i,\ldots,m\rangle \end{array}$$

Let m equal to j_1, then the second condition requires us to prove that $\langle p,i,\ldots,n\rangle \leftrightharpoons \langle *,i,\ldots,j_1\rangle$ is false. Using the definition of $\leftrightharpoons$ (where the prematch fields are empty),

$$\begin{array}{l} (\langle p,dst,\ldots,k_p\rangle \leftrightharpoons \langle src,q,\ldots,k_q\rangle) \doteq \\ \quad q = dst \ \wedge \ src \in \{p,*\} \quad \text{the source and target must match} \end{array}$$

after simplification we have

$$k_* \in \{p,*\} \ \wedge \ \neg(* \in \{p,*\}),$$

which is obviously false. Thus request j_2 cannot complete before j_1, which implies the correctness of these two rules.

On the other hand, the rule $\forall i,j_1,j_2,k : \ j_1 < j_2 \Rightarrow R_{i,j_1}(k,\ldots) \prec R_{i,j_2}(*,\ldots)$ is invalid. If we perform the same contradiction proof as shown above, then finally we will get a predicate not leading to a contradiction: $* \in \{p,*\} \ \wedge \ \neg(k \in \{p,*\})$. This predicate is true when $k \neq p$, *i.e.* a process other than k sends the message.

A.0.2 Barrier.

Rule `Cb` specifies that any MPI call starting after a barrier operation will complete after the barrier. This rule is valid because the barrier function has blocking semantics: the "wait" phase of a barrier operation B_{i,j_1} at process i will be blocked until i leaves the synchronizing communication. Thus only after B_{i,j_1} returns will a subsequent MPI call any_{i,j_2} start and then complete. Similarly, rule `Cw` is valid because `Wait` also has blocking semantics.

On the other hand, the rule $\{\forall i, j_1, j_2, k : \; j_1 < j_2 \Rightarrow any_{i,j_1}(\ldots) \prec B_{i,j_2}\}$ is invalid. This can be explained easily with the formal semantics. Recall that B_{i,j_2} is implemented as B_{i,j_2}_init followed by B_{i,j_2}_wait. Suppose any_{i,j_1} is a send operation, as the barrier and send operate on different MPI objects (*i.e.* `rend` and `reqs` respectively), the B_{i,j_2}_wait needs not to wait for the completion of the send. Hence the following sequence is possible, implying that $send_{i,j_1}(\ldots) \prec B_{i,j_2}$ is false.

$$send_{i,j_1} \text{ starts} < B_{i,j_2}\_init < B_{i,j_2}\_wait < send_{i,j_1} \text{ completes}$$