Science of Computer Programming 76 (2011) 65–81

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API

Guodong Li [a,*], Robert Palmer [b], Michael DeLisi [a], Ganesh Gopalakrishnan [a], Robert M. Kirby [a]

[a] *School of Computing, University of Utah, Salt Lake City, UT 84112, USA*

[b] *Microsoft Corporation, USA*

ARTICLE INFO

Article history:
Received 31 May 2009
Accepted 9 March 2010
Available online 15 June 2010

Keywords:
MPI
Formal specification
TLA+
Model checking

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 around 200 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 aforementioned requirements. We close with a description of possible approaches that may help render the act of writing, understanding, and validating the specifications of concurrency APIs much more productive.

1. Introduction

Application Programming Interfaces (API) (also known as libraries) are an important part of modern programming — especially concurrent programming. APIs allow significant new functionality (*e.g.*, communication and synchronization) to be provided to programmers without changing the underlying programming language. APIs such as the Message Passing Interface (MPI, [1]) have been in existence for nearly two decades, adapting to the growing needs of programmers for new programming primitives, and growing in the number of primitives supported. The immense popularity of MPI is attributable to the balance it tends to achieve in terms of portability, performance, simplicity, symmetry, modularity, composability, and completeness [2]. While MPI itself has evolved, its basic concepts have essentially remained the same. This has allowed the creation of important long-lived codes — such as weather simulation codes [3]. Despite these successes, *MPI does not have a formal specification*. This is a drastic shortcoming from the point of view of advances in the Science of Programming. In this paper, we present the first formal specification for a significant subset of MPI 2.0.

MPI [4] has become a *de facto* standard in High Performance Computing (HPC) and is being actively developed and supported through several implementations [5–7]. However, for several reasons, even experienced programmers sometimes misunderstand MPI calls. First, MPI calls are traditionally described in natural languages. Such descriptions are prone to being misinterpreted. Another common approach among programmers is to discover MPI's "intended behavior" by conducting ad hoc experiments using MPI implementations. Such experiments cannot reveal all intended behaviors of an MPI call, and

* Corresponding author. Tel.: +1 801 585 3866; fax: +1 801 581 5843.
E-mail addresses: ligd@cs.utah.edu (G. Li), Robert.Palmer@microsoft.com (R. Palmer), delisi@gmail.com (M. DeLisi), ganesh@cs.utah.edu (G. Gopalakrishnan), kirby@cs.utah.edu (R.M. Kirby).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.03.007

may even be misleading. A formalization of the MPI standard can potentially help avoid these misunderstandings, and also *help define what is an acceptable MPI implementation*.

Engineering a formal specification for a non-trivial concurrency API requires the right combination of rigor, executability, and traceability. A formal specification must also be written as an elaboration of a well-written informal specification. It must also be as direct and declarative in nature, *i.e.*, it must not be described in terms of what a specific scheduler might do or rely upon detailed data structures that suggest an actual implementation. Our formal semantics for MPI is written with these goals in mind. At first glance, it may seem that creating a formal specification for MPI which has over 300 fairly complex functions is almost an impossible task. However, as explained in [2], the large size of MPI is somewhat misleading. The primitive concepts involved in MPI are, relatively speaking, quite parsimonious. Our formal specification attempts to take advantage of this situation by first defining a collection of primitives, and then defining MPI calls in terms of these primitives.

Besides contributing directly to MPI, we hope that our work will address the growing need to properly specify and validate future concurrency APIs. In a modern context, APIs allow programmers to harness the rapidly growing power and functionality of computing hardware through new message transfer protocols such as one-sided communication [1] and new implementations of MPI over modern interconnects [8]. Given the explosive growth in concurrency and multi-core computing, one can witness a commensurate growth in the number of concurrency APIs being proposed. Among the more recently proposed APIs are various Transactional Memories [9], OpenMP [10], Ct [11], Thread Building Blocks [12], and Task Parallel Library [13]. There is also a high degree of interest in light weight APIs such as the Multicore Communications API (MCAPI) [14] intended to support core-to-core communication in a systems-on-chip multi-core setting. One could perhaps draw lessons from exercises such as ours and ensure that for these emerging APIs, the community would create formal specifications contemporaneously with informal specifications. As opposed to this, a formal specification for MPI has been late by nearly two decades in arriving on the scene, because none of the prior work described in Section 2 meets our goals for a rigorous specification for MPI.

Besides developing formal specifications, we must also constantly improve the mechanisms that help derive value from formal specifications. For instance, formal specifications can help minimize the effort to *understand an API*. Concurrency APIs possess many non-intuitive but legal behaviors: how can formal specifications help tutor users of the API as to what these are? Second, it is quite easy to end up with an incorrect or incomplete formal specification. How do we best identify the mistakes or omissions in a formal specification? Third, it is crucial that formal specifications offer assistance in validating or verifying API implementations, especially given that these implementations tend to change much more rapidly than the API semantics themselves change. While we only provide preliminary answers to these issues in this paper, our hope is that the availability of a formal specification is the very first step in being able to approach these more formidable problems. Last but not least, many scientists believe that the growth in complexity of APIs can have undesirable or unexpected consequences with respect to the more tightly controlled growth of programming language semantics; see [15] for related discussions. We strongly believe that these discussions point to an even stronger need for formal specifications of concurrency APIs, and as a next step to our work suggest examining how API formal specifications interact with language and compiler semantics.

Background. In our previous work [16], we presented the formal specification of around 30% of the 128 MPI-1.0 functions (mainly for point-to-point communication) in the specification language TLA+ [17]. TLA+ enjoys wide usage in industry by engineers (e.g. in Microsoft [18] and Intel [19]), and is relatively easy to learn. Additionally, in order to help practitioners access our specification, we built a C front-end in the Microsoft Visual Studio (VS) environment, through which users can submit and run short MPI programs with embedded assertions (called litmus tests). Such tests are turned into TLA+ code and run through the TLC model checker [17], 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. In [16], we show that this rather simple approach is surprisingly effective for querying a standard and obtaining all possible execution outcomes (some of which are entirely unexpected), as computed by the underlying TLC model checker. In comparison, a programmer experimenting with an actual MPI implementation will not have the benefit of search that a model checker provides, and be able to check assertions only on executions that materialize in a given MPI implementation along with its (fixed) scheduler.

In order to make our specification faithful to the English description, we (i) organize the specification for *easy traceability*: many clauses in our specification are cross-linked with [4] 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 4.6); 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.

In this work, we expand on the work reported in [16,20], and in addition cover considerably more ground. In particular, we now have a formal specification for nearly 200 MPI functions, including point to point calls, MPI data types, collective communication, communicators, process management, one-sided communication, and IO.

Space restrictions prevent us from elaborating on all these aspects: this paper covers the first three aspects, and a companion technical report [21] covers the rest. (Note: We have not extended the C front-end described in [16] to cover these additional MPI functions.) We have extensively tested our formal specification, as discussed in Section 4.6. Using our formal specification, we have justified a tailored Dynamic Partial Order Reduction algorithm (see [21]).

Organization of the paper. The structure of this paper is as follows. Section 2 discusses related work. Then we give a motivating example to illustrate 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. Next we describe a front-end that translates MPI programs written in C into TLA+ code, plus a verification framework enabling the execution of the semantics. Finally we give the concluding remarks.

2. Related work

The IEEE Floating Point standard [22] 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., [23]), finding routine applications in *formal proofs* of modern microprocessor floating point hardware circuits. Formal specifications using TLA+ include Lamport's Win32 Threads API specification [18] and the RPC Memory Problem specified in TLA+ and formally verified in the Isabelle theorem prover by Lamport et al. [24]. In [25], Jackson presents a lightweight object modeling notation called Alloy, which has tool support [26] in terms of formal analysis and testing based on Boolean satisfiability methods. The approach taken in Alloy is extremely complementary to what we have set out to achieve through our formal specifications. In particular, their specification of the Java Memory Model is indicative of the expressiveness of Alloy. Abstract State Machines (ASMs) [27] have been used for writing formal specifications of concurrent systems, for instance [28].

Bishop et al. [29,30] formalized in the HOL theorem prover [31] 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. They perform a vast number of conformance tests to validate the specification. We also rely on testing for validation check. As it is the standard that we formalize, we need to write all the test cases by hand.

Norrish [32] formalized in HOL [31] 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. In addition, a set of Hoare rules are derived from the operational semantics to assist property verification. In contrast, our specification defines the semantics in a more declarative style and does not encode the correctness requirement into a type system.

Two other related works in terms of writing executable specifications are the Symbolic Analysis Laboratory (SAL) approach [33] and the use of the Maude rewrite technology [34]. The use of these frameworks may allow us to employ alternative reasoning techniques: using decision procedures (in case of SAL), and using term rewriting (in case of Maude). These will be considered during our future work.

Georgelin and Pierre [35] specify some of the MPI functions in LOTOS [36]. Siegel and Avrunin [37] 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 [38]. They [39] also support a limited partial-order reduction method — one that handles wild-card communications in a restricted manner, as detailed in [40]. Siegel [41] models additional 'non-blocking' MPI primitives in Promela. Our own past efforts in this area are described in [42–45]. 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. Motivating example

MPI is a portable standard and has a variety of implementations [5–7]. MPI programs are often manually or automatically (e.g., [46]) 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  MPI_Irecv($rcvbuf1, *, req1$);
    MPI_Irecv($rcvbuf2$, from 1, $req2$);
    MPI_Wait($req1$);
    MPI_Wait($req2$);
    MPI_Bcast($revbuf3, root = 1$);
P1  $sendbuf1 = 10$;
    MPI_Bcast($sendbuf1, root = 1$);
    MPI_Isend($sendbuf2$, to 0, $req$);
    MPI_Wait($req$);
P2  $sendbuf2 = 20$;
    MPI_Isend($sendbuf2$, to 0, $req$);
    MPI_Bcast($recvbuf2, root = 1$);
    MPI_Wait($req$);
```

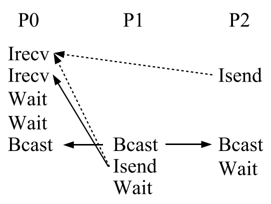


Table 1
Size of the specification (excluding comments and blank lines).

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)

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, which incurs no deadlock. Thus it is possible that we will not encounter this deadlock even we run the program for 1000 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 the incoming messages may arrive out of order such that $rcvbuf$ 1 gets the message from P1. However, vendor implementation may give the wrong answer when the message delivery delay from P1 to P0 is greater than that from P2 to P0. To check this in our framework, we can add in P0 an assertion `rcvbuf1` $== 2\theta$ right before the broadcast call.
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`. While vendor implementations may not give reliable answer for this question, we can move the above assertion to any other point before the corresponding `MPI_Waits` and have the model checker find violations, which means that the data cannot be accessed 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. To check this, we can reverse the order of the `MPI_Wait` commands. If the model checker does not find a deadlock then the operations may complete in either order.

The MPI reference standard [4] 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 entire specification including tests and examples and the verification framework are available online [47].

4. Specification

TLA+ provides built-in support for sets, functions, records, strings and sequences. To model MPI objects, we extend the TLA+ library by defining advanced data structures including maps and ordered sets (`oset`). For instance, MPI groups and I/O files are represented by ordered sets.

The approximate sizes (excluding comments and blank lines) of the major parts in the current specification are shown in Table 1, where #funcs and #lines give the number of MPI primitives and code lines respectively. We do not model functions whose behavior depends on the underlying operating system. For deprecated items, we only model their replacement.

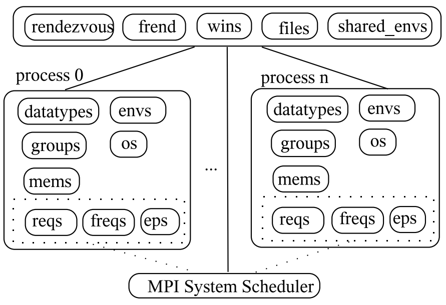


Fig. 1. MPI objects and their interaction.

4.1. Data structures

The data structures modeling explicit and opaque MPI objects are shown in Fig. 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)*, which matches requests at origins and destinations and performs message passing. MPI primitive calls at different processes make transitions non-deterministically.

Request object `reqs` is used in point-to-point communications. A similar file request object `freqs` is for parallel I/O communications. Objects `groups` and `comms` model the groups and (intra- or inter-) communicators respectively. In addition to the group, a communicator also includes virtual topology and other attributes. Objects `rendezvous` and `frend` objects are for collective communications and shared file operations respectively. Objects `epos` and `wins` are used in one-sided communications.

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 the objects visible to the MPI system such as physical files on the disk. We define a separate object `mems` for the physical memory at processes.

4.2. Notations

Our presentation uses notations extended and abstracted from TLA+. The basic concept in TLA+ is functions. We write $f[v]$ for the value of function f applied to v; this value is specified only if v is in f's domain $\mathtt{DOM}\,f$. Notation $[S \rightarrow T]$ specifies the set of all functions f such that $\mathtt{DOM}\,f = S$ and $f[v] \in T$ for all $v \in S$. For example $[\mathtt{int} \rightarrow \mathtt{nat}]$ denotes all functions from integers to natural numbers. This notation is usually used to specify the type of a function.

Functions may be described explicitly with the construct $[x \in S \mapsto e]$ such that $f[x] = e$ for $x \in S$. For example, the function f_{double} that doubles input natural numbers can be specified as $[x \in \mathtt{nat} \mapsto 2x]$. Obviously $f_{double}[1] = 2$ and $f_{double}[4] = 8$. Notation $[f\ \mathtt{EXCEPT}\ ![e_1] = e_2]$ defines a function f' such that f' is the same as f except $f'[e_1] = e_2$. An @ appearing in e_2 represents the old value of $f[e_1]$. For example, $[f_{double}\ \mathtt{EXCEPT}\ ![3] = @ + 10]$ is the same as f_{double} except that it returns 16 for input 3.

Tuples, arrays, records, sequences and ordered sets are special functions with finite domains. They differ mainly in the operators defined over these data structures. An n-tuple is written as $\langle e_1, \ldots, e_n\rangle$, which defines a function f with domain $\{1, \ldots, n\}$ such that $f[i] = e_i$ for $1 \leq i \leq n$. Its ith component is given by $\langle e_1, \ldots, e_n\rangle[i]$. An array resembles a tuple except that its index starts from 0 rather than 1 so as to conform to the convention of the C language. Records can be written explicitly as $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$, which is actually a function mapping field h_i to value e_i. For instance tuple $\langle 1, 4, 9\rangle$, record $[1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9]$, and function $[x \in \{1, 2, 3\} \mapsto x^2]$ are equivalent. Similar to function update, $[r\ \mathtt{EXCEPT}\ !.h = e]$ represents a record r' such that r' is the same as r except $r'.h = e$, where $r.h$ returns the h-field of record r.

A (finite) sequence is represented as a tuple. Operators are provided to obtain the head or tail elements, append elements, concatenate two sequences, and so on. An ordered set is analogous to a usual set except it consists of distinct elements. It may be interpreted as a function too: its domain is $[0, n - 1]$ where n is the number of elements (*i.e.* the cardinality), and its range contains all the elements.

The basic temporal logic operator used to define transition relations is the next state operator, denoted using $'$ or *prime*. For example, $s' = [s\ \mathtt{EXCEPT}\ ![x] = e]$ 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 a sequence $[hr \mapsto 0, mnt \mapsto 0]$, $[hr \mapsto 0, mnt \mapsto 1]$, ..., $[hr \mapsto 0, mnt \mapsto 59]$, $[hr \mapsto 1, mnt \mapsto 0]$, ..., where $[hr \mapsto i, mnt \mapsto j]$ is a state with hour i and minute j. Its next-state relation is a formula expressing the relation between the values of hr and mnt. It asserts that mnt equals $mnt + 1$ if $mnt \neq 59$. When mnt is 59, mnt is reset to 0, and hr increased by 1.

$$
\begin{aligned}
time' = \ & \mathtt{let}\ c = (time[mnt] \neq 59)\ \mathtt{in} \\
& [time\ \mathtt{EXCEPT}\ ![mnt] = \mathtt{if}\ c\ \mathtt{then}\ @ + 1\ \mathtt{else}\ 0,\ ![hr] = \mathtt{if}\ \neg c\ \mathtt{then}\ @ + 1\ \mathtt{else}\ @]
\end{aligned}
$$

To make the specification succinct, we introduce some other commonly used notations. Note that $\top$ and $\bot$ denote boolean value *true* and *false* respectively; and ϵ and α denote the null value and an arbitrary value respectively. Notation $\Gamma_1 \diamond x_k \diamond \Gamma_2$ specifies a queue where x is the kth element, Γ_1 contains the elements before x, and Γ_2 contains the elements after x. When it appears in the precondition of a transition rule it should be interpreted in a *pattern-matching* manner such that Γ_1 returns the first $k-1$ elements, x is the kth element and Γ_2 returns the remaining elements.

$\Gamma_1 \diamond \Gamma_2$	the concatenation of queue Γ_1 and Γ_2
$\Gamma_1 \diamond x_k \diamond \Gamma_2$	the queue with x being the kth element
$\Gamma_1 \sqsubseteq \Gamma_2$	Γ_1 is a sub-queue (sub-array) of Γ_2
$\top, \bot, \epsilon$ and α	true, false, null value and arbitrary value
$f_1 = f \uplus (x, v)$	$\mathtt{DOM}(f_1) = \mathtt{DOM}(f) \cup \{x\} \wedge x \notin \mathtt{DOM}(f) \wedge f_1[x] = v$ $\wedge \forall y \in \mathtt{DOM}(f) : f_1[y] = f[y]$
$f\vert_x$	the index of element x in function f, *i.e.* $f[f\vert_x] = x$
$c\ ?\ e_1 : e_2$	An abbreviation for `if` c `then` e_1 `else` e_2
`size`(f) or $\vert f\vert$	the number of elements in function f

Similar to the separating operator $*$ in separation logic [48], operator $\uplus$ divides a function into two parts with disjoint domains. For example, function $[x \in \{1, 2, 3\} \mapsto x^2]$ can be written as $[x \in \{1, 2\} \mapsto x^2] \uplus (3, 9)$ or $[x \in \{1, 3\} \mapsto x^2] \uplus (2, 4)$. This operator is especially useful when representing the content of a function.

TLA+ allows the specification of MPI primitives in a declarative style. For illustration we show below a helper (auxiliary) function used to implement the `MPI_COMM_SPLIT` primitive, where *group* is an ordered set of processes, *colors* and *keys* are arrays. Here `DOM`, `RNG`, `CARD` return the domain, range and cardinality of an ordered set respectively. This code directly formalizes the English description (see page 147 in [4]): "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 supplies the color value `MPI_UNDEFINED`, a null communicator is returned". In contrast, it is impossible to write such a declarative specification in the C language.

```
   Comm_split(group, colors, keys, proc) $\doteq$
1 : let $rank = group|_{proc}$ in
2 : if $(colors[rank] = \mathtt{MPI\_UNDEFINED})$ then MPI_GROUP_NULL
3 : else
4 :   let $same\_colors = \{k \in \mathtt{DOM}(group) : colors[k] = colors[rank]\}$ in
5 :   let $sorted\_same\_colors =$
6 :     choose $g \in [\mathtt{DOM}(same\_colors) \to \mathtt{RNG}(same\_colors)]$ :
7 :     $\wedge\ \mathtt{RNG}(g) = same\_colors$
8 :     $\wedge\ \forall i, j \in same\_colors : g|_i < g|_j \Rightarrow (keys[i] < keys[j] \vee (keys[i] = keys[j] \wedge i < j))$
9 :   in $[i \in \mathtt{DOM}(sorted\_same\_colors) \mapsto group[sorted\_same\_colors[i]]]$
```

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 *proc*'s rank in the group; line 4 obtains an ordered set of the ranks of all the processes with the same color as *proc*; lines 5–8 sort this rank set in the ascending order of keys, with ties broken according to the ranks. Specifically, lines 6–7 pick an ordered set g with the same domain and range as *same_colors*; line 8 indicates that, in g, rank i shall appear before rank j (*i.e.* $g_{|i} < g_{|j}$) if the key at i is less than that at j. This specification may be a little tricky as we need to map a process to its rank before accessing its color and key. This merits our formalization which explicitly describes all the details. For illustration, suppose $group = \langle 2, 5, 1\rangle$, $colors = \langle 1, 0, 0\rangle$ and $keys = \langle 0, 2, 1\rangle$, then the call of this function at process 5 creates a new group $\langle 1, 5\rangle$.

4.2.1. Operational semantics

The formal semantics of an MPI primitive is modeled by a state transition. A system state consists of explicit and opaque objects mentioned in 4.1. An object may involve multiple processes; we write $\mathtt{obj}_p$ for the object `obj` at process p. For example, $\mathtt{reqs}_p$ refers to the request object (for point-to-point communications) at process p.

We use notation $\mathrel{\hat{=}}$ to define the semantics of an MPI primitive, and $\doteq$ to introduce a helper function. The precondition *cond* of a transition, if it exists, is specified by "`requires` $\{cond\}$". An error is reported if this precondition is violated. The body of a transition is expressed by a rule of format $\frac{guard}{action}$, where *guard* specifies the requirement for the transition 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. Otherwise the rule is blocked and the action will be delayed. A true guard will be omitted, meaning that the transition is always enabled.

For instance, the semantics of `MPI_Buffer_detach` is shown below. A buffer object contains several fields: *buff* and *size* record the start address in the memory and the size respectively; *capacity* and *max_capacity* record the available space and maximum space respectively. The values of these fields are set when the buffer is created. The precondition of the `MPI_Buffer_detach` rule enforces that process p's buffer must exist; the guard indicates that the transition will block until all messages in the buffer have been transmitted (*i.e.* the entire space is available); the action is to write the buffer address and the buffer size into *p*'s local memory, and deallocate the space occupied by the buffer.

$$\texttt{MPI\_Buffer\_detach}(\mathit{buff}, \mathit{size}, p) \doteq$$
$$\texttt{requires}\ \{\texttt{buffer}_p \neq \epsilon\}$$
$$\frac{\texttt{buffer}_p.\mathit{capacity} = \texttt{buffer}_p.\mathit{max\_capacity}}{\texttt{mems}'_p = [\texttt{mems}_p\ \text{EXCEPT}\ ![\mathit{buff}] = \texttt{buffer}_p.\mathit{buff},\ ![\mathit{size}] = \texttt{buffer}_p.\mathit{size}]\ \wedge\ \texttt{buffer}'_p = \epsilon}$$

It may be desirable to specify only the objects and components that are affected by the transition such that those not appeared in the action are assumed to be unchanged. Thus the action of the above rule can be written as follows. We will use this lighter notation throughout the rest of Section 4.

$$\texttt{mems}'_p[\mathit{buff}] = \texttt{buffer}_p.\mathit{buff}\ \wedge\ \texttt{mems}'_p[\mathit{size}] = \texttt{buffer}_p.\mathit{size}\ \wedge\ \texttt{buffer}'_p = \epsilon$$

4.3. Quick overview of the methodology

We first give a simple example to illustrate how MPI programs and MPI primitives are modeled. Consider the following program involving two processes:

$P0$: MPI_Send(buf_s, 2, MPI_INT, 1, 10, MPI_COMM_WORLD)
MPI_Bcast(buf_b, 1, MPI_FLOAT, 0, MPI_COMM_WORLD)
$P1$: MPI_Recv(buf_r, 2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD)
MPI_Bcast(buf_b, 1, MPI_FLOAT, 0, MPI_COMM_WORLD)

This program is converted by our compiler into the following TLA+ code (*i.e.* the model of this program), where the TLA+ code of MPI primitives will be presented in subsequent sections. An extra parameter is added to an MPI primitive to specify the process it belongs to. In essence, a program 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. The control flow of a program at process *p* is represented by the *pc* values: *pc*[*p*] stores the current values of the program pointer. The *pc* values are integer-value labels such as L_1, L_2, and so forth. A blocking call is modeled by its non-blocking version followed by a wait operation, *e.g.* MPI_Send $\doteq$ (MPI_Isend; MPI_Wait). The compiler treats $request_0$ and $status_0$ as references to memory locations. For example, suppose reference $request_0$ has address 5, then the value it points to is $\texttt{mems}_p[request_0]$ (*i.e.* $\texttt{mems}_p[5]$). As all variables in the source C program are mapped to memory locations.

p0's transition rules

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

p1's transition rules

$\vee\ \ \wedge\ pc[1] = L_1\ \wedge\ pc'[1] = L_2$
$\quad\ \ \wedge\ \texttt{MPI\_Irecv}(buf_r, 2, \texttt{MPI\_INT}, 0, \texttt{MPI\_ANY\_TAG}, \texttt{MPI\_COMM\_WORLD}, request_1, 1)$
$\vee\ \ \wedge\ pc[1] = L_2\ \wedge\ pc'[1] = L_3\ \wedge\ \texttt{MPI\_Wait}(request_1, status_1, 1)$
$\vee\ \ \wedge\ pc[1] = L_3\ \wedge\ pc'[1] = L_4\ \wedge\ \texttt{MPI\_Bcast}_{init}(buf_b, 1, \texttt{MPI\_FLOAT}, 0, \texttt{MPI\_COMM\_WORLD}, 1)$
$\vee\ \ \wedge\ pc[1] = L_4\ \wedge\ pc'[1] = L_5\ \wedge\ \texttt{MPI\_Bcast}_{wait}(buf_b, 1, \texttt{MPI\_FLOAT}, 0, \texttt{MPI\_COMM\_WORLD}, 1)$

An enabled rule may be executed at any time. Suppose the program pointer of process *p*0 is L_1, then the MPI_Isend rule may be executed, modifying the program pointer to L_2. As indicated below, it creates a new send request of format $\langle destination, communicator\_id, tag, value\rangle_{request\_id}$, and appends it to *p*0's request queue $\texttt{reqs}_0$. Value v is obtained by *read_data*($\texttt{mems}_0$, buf_s, 2, MPI_INT), which reads from the memory two consecutive integers starting from address buf_s.

	process *p*0	process *p*1
reqs	$\ldots \diamond \langle 1, cid, 10, v\rangle_{request_0}$	$\ldots$

Similarly, when the MPI_Irecv rule at process *p*1 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 value is yet to be received.

	p0	p1
reqs	$\ldots \diamond \langle 1, cid, 10, v\rangle_{request_0}$	$\ldots \diamond \langle buf_r, 0, cid, \text{ANY\_TAG}, \_\rangle_{request_1}$

The MPI System Scheduler matches the send request and the receive request, and transfers the data value v from *p*0 to *p*1. After the transferring, the value fields in the send and receive requests become _ and v respectively.

	p0	p1
reqs	$\ldots \diamond \langle 1, cid, 10, \_\rangle_{request_0}$	$\ldots \diamond \langle buf_r, 0, cid, \text{ANY\_TAG}, v\rangle_{request_1}$

If the send is not buffered at $p0$, then the `MPI_Wait` call will be blocked until the data v is sent. After that the send request is removed from the queue. Analogously, the `MPI_Wait` rule at $p1$ is blocked until the incoming value arrives. Then v is written into $p1$'s local memory and this request is removed.

	p0	p1
`reqs`	...	...

In our formalization, each process divides a collective primitive call into two phases: an "init" phase that initializes the call, and a "wait" phase that synchronizes the communication with other processes. Processes synchronize with each other through the `rendezvous` (or `rend` for short) object which records the status of the communication (denoted by Ψ) and the data sent by the processes (denoted by S_v). For a communicator with context ID *cid* there exists an individual rendezvous object `rend`[*cid*]. In the "init" phase, process p_i is able to proceed only if it is not in the domain of the status component (*i.e.* p_i is not participating the communication). It updates its status to "e" ("entered") and store its data in the rendezvous. In the given example, after the "init" phases of the broadcast at process 0 and 1 are over, the rendezvous pertaining to communicator `MPI_COMM_WORLD` becomes $\langle[0 \mapsto \text{“e”}, 1 \mapsto \text{“e”}], [0 \mapsto \text{val}]\rangle$, where $val = read\_data(\texttt{mems}_0, buf_b, 1, \texttt{MPI\_FLOAT})$.

$$\texttt{syn}_{\texttt{init}}(cid, val, p_i) \doteq \boxed{\text{process } p_i \text{ joins the communication and stores data } v \text{ in } \texttt{rend}}$$
$$\frac{p_i \notin \texttt{DOM}(\Psi)}{\texttt{rend}'[cid] = \langle \Psi \uplus (p_i, \text{“e”}),\ S_v \uplus (p_i, val)\rangle}$$

In the "wait" phase, if the communication is synchronizing, then process p_i has to wait until all other processes finish their "init" phases. If p_i is the last process that leaves the communication, then the `rend` object will be deleted; otherwise p_i just updates its status to "l" ("left").

	before wait								after wait
	Ψ								Ψ
`rend`[*cid*]	"l"	...	"l"	"e"	"l"	...	"l"		
	p_1	...	p_{i-1}	p_i	p_{i+1}	...	p_n		

$$\texttt{syn}_{\texttt{wait}}(cid, p_i) \doteq \boxed{\text{process } p \text{ leaves the synchronizaing communication}}$$
$$\frac{\texttt{rend}[cid] = \langle \Psi \uplus (p_i, \text{“e”}), S_v\rangle \ \wedge\ \forall k \in \texttt{comms}_{p_i}[cid].group : k \in \texttt{DOM}(\Psi)}{\texttt{rend}'[cid] = \texttt{if}\ \forall k \in \texttt{comms}_{p_i}[cid].group : \Psi[k] = \text{“l”}\ \texttt{then}\ \epsilon\ \texttt{else}\ \langle \Psi \uplus (p_i, \text{“l”}),\ S_v\rangle}$$

These simplified rules illustrate how MPI point-to-point and collective communications are modeled. The standard rules are given in Sections 4.4 and 4.5.

4.4. Point-to-point communication

The semantics of core point-to-point communication primitives are shown in Figs. 3–5. Readers should refer to the semantics when reading through this section. An example illustrating the "execution" of an MPI program according to the semantics is shown in Fig. 2.

New send and receive requests are appended to the request queues. A send request contains information about the destination process (*dst*), the context ID of the communicator (*cid*), the tag to be matched (*tag*), the data value to be sent (*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 the ready send, we include in the send request a field *prematch* of format $\langle destination, request\_index\rangle$ which points to the receive request matching this send request. A receive request contains similar fields plus the buffer address and a field to store the incoming data. Initially the data value is missing (represented by the "_" in the data field); an incoming message from a sender will replace the "_" with the data it carries. Notation $v\_$ denotes either data value v arrives or the data is still missing. 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) the request is persistent and active; (v) the request 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 (represented by a global flag *use_buffer*). If buffer space is available, then it behaves the same as a send in the buffered mode; otherwise it acts as a synchronous send. We show below the specification of `MPI_IBsend`. As *dtype* and *comm* are the references (pointers) to datatype and communicator objects, their values are obtained by $\texttt{datatypes}_p[dtype]$ and $\texttt{comms}_p[comm]$. Helper function `ibsend` creates a new send request, appends it to p's request queue, and puts the data in p's send buffer $\texttt{buffer}_p$. The request handle points to the last request in the queue.

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

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	$\texttt{reqs}_0$	$\texttt{reqs}_1$	$\texttt{reqs}_2$
1	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon\rangle_0^{ss}$		
2	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon\rangle_0^{ss}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon\rangle$	
3	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon\rangle_0^{ss}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \epsilon\rangle_0^{rc}$
4	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon\rangle_0^{ss} \diamond$ $\langle 2, 5, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
5	$\langle 1, 5, 0, \_, \bot, \top, \epsilon\rangle_0^{ss} \diamond$ $\langle 2, 5, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
6	$\langle 2, 5, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
7	$\langle 2, 5, 0, \_, \bot, \top, \langle 2, 0\rangle\rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, v_2, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
8		$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	$\langle b, *, 5, *, v_2, \bot, \top, \langle 0, 1\rangle\rangle_0^{rc}$
9		$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon\rangle_0^{rc}$	
10			

$1 : \texttt{issend}(v_1, 1, 5, 0, p_0)$ $\quad 2 : \texttt{irecv}(b, 0, 5, *, p_1)$ $\quad 3 : \texttt{irecv}(b, *, 5, *, p_2)$ $\quad 4 : \texttt{irsend}(v_2, 2, 5, 0, p_0)$
$5 : \texttt{transfer}(p_0, p_1)$ $\quad 6 : \texttt{wait}(0, p_0)$ $\quad 7 : \texttt{transfer}(p_0, p_2)$ $\quad 8 : \texttt{wait}(1, p_0)$ $\quad 9 : \texttt{wait}(0, p_2)$ $\quad 10 : \texttt{wait}(0, p_1)$

Fig. 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 Figs. 3–5.

`MPI_Recv` is specified in a similar way. The MPI System Scheduler transfers values from a send request to its matching receive request. Relation $\doteq$ defines the meaning of "matching". Two cases are considered:

- The send is in ready mode. 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) of req_s stores the tuple $\langle$*destination process, destination request index*$\rangle$, and req_r's *prematch* field stores the tuple $\langle$*source process, source request index*$\rangle$. req_s and req_r match iff these two tuples match.
- The send is in other modes. The send request and receive request are matched if relevant information (*e.g.* source, destination, context ID and tag) matches. The source and tag in the receive request may be `MPI_ANY_SOURCE` and `MPI_ANY_TAG` respectively.

It is the `transfer` rule (see Fig. 4) that models message passing. Messages from the same source to the same destination must be matched in a FIFO order: only the first send request in the send queue and the first matching receive request in the receive queue will participate in the transferring. The FIFO requirement is enforced by the following predicate which indicates that there exist no prior send requests and prior receive requests that match.

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

When the transfer is done, the value field in the receive request req_j is filled with the incoming value v, and that in the send request req_i becomes _ to indicate 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.

The `MPI_Wait` call returns when the operation associated with request *request* is complete. If *request* is a null handle, then an empty status is returned; otherwise the helper function `wait_one` is invoked to pick the appropriate wait function according to the request's type. Let us look closer at the definition of `recv_wait` (see Fig. 4). First of all, after the call the request is not "live" any more, thus the *live* flag becomes false. When the call is made with an inactive request, it returns immediately with an empty status. If the request is persistent and 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`.

If the request has been marked for cancellation, then the call completes without writing the data into memory. If the source process is a null process, then the call returns immediately with a null status where 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 completion of a request is modeled by the *has_completed* predicate. A receive request completes when the value has been received. A send request in the buffer mode completes when the value has been buffered or transferred. This function is used to implement multiple communication primitives. For instance, `MPI_Waitany` blocks until one of the requests completes.

```
Data Structures
  send request : important fields + less important fields
    $\langle dst : \mathtt{int},\ cid : \mathtt{int},\ tag : \mathtt{int},\ value,\ pr : \mathtt{bool},\ active : \mathtt{bool},\ prematch\rangle^{mode}$ +
    $\langle cancelled : \mathtt{bool},\ dealloc : \mathtt{bool},\ live : \mathtt{bool}\rangle$
  recv request : important fields + less important fields
    $\langle buf : \mathtt{int},\ src : \mathtt{int},\ cid : \mathtt{int},\ tag : \mathtt{int},\ value,\ pr : \mathtt{bool},\ active : \mathtt{bool},\ prematch\rangle^{recv}$
    $+ \langle cancelled : \mathtt{bool},\ dealloc : \mathtt{bool},\ live : \mathtt{bool}\rangle$
  $\mathtt{ibsend}(v, dst, cid, tag, p) \doteq$   buffer send
  $\mathtt{requires}\ \{\mathtt{size}(v) \le \mathtt{buffer}_p.vacancy\}$   check buffer availability
    $\mathtt{reqs}'_p = \mathtt{reqs}_p \diamond \langle dst, cid, tag, v, \bot, \top, \epsilon\rangle^{bsend}\ \wedge$   append a new send request
    $\mathtt{buffer}'_p.vacancy = \mathtt{buffer}_p.vacancy - \mathtt{size}(v)$   allocate buffer space

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

  $(\langle p, dst, tag_p, \omega_p, k_p\rangle = \langle src, q, tag_q, \omega_q, k_q\rangle) \doteq$   match send and receive requests
  $\mathtt{if}\ \omega_p = \epsilon\ \wedge\ \omega_q = \epsilon\ \mathtt{then}\ tag_q \in \{tag_p, \mathtt{ANY\_TAG}\}\ \wedge\ q = dst\ \wedge\ src \in \{p, \mathtt{ANY\_SOURCE}\}$
  $\mathtt{else}\ \omega_p = \langle q, k_q\rangle\ \wedge\ \omega_q = \langle p, k_p\rangle$   prematched requests

  $\mathtt{irsend}(v, dst, cid, tag, p) \doteq$   ready send
  $\mathtt{requires}\ \{\ \exists q : \exists\langle src, cid, tag_1, \_, pr_1, \top, \epsilon\rangle_k^{recv} \in \mathtt{reqs}_q :$
      $\langle p, dst, tag, \epsilon, \mathtt{size}(\mathtt{reqs}_p)\rangle = \langle src, q, tag_1, \epsilon, k\rangle\ \}$   a matching recv exists?
  $\mathtt{reqs}'_p = \mathtt{reqs}_p \diamond \langle dst, cid, tag, v, \bot, \top, \langle q, k\rangle\rangle^{rsend}\ \wedge\ \mathtt{reqs}'_q.\omega = \langle p, \mathtt{size}(\mathtt{reqs}_p)\rangle$

  $\mathtt{isend} \doteq \mathtt{if}\ use\_buffer\ \mathtt{then\ ibsend\ else\ issend}$   standard mode send

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

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

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

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

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

Fig. 3. Modeling point-to-point communications (I).

4.5. *Collective communication*

Processes participating in a collective communication coordinate with each other through rendezvous objects. Each communicator with context id *cid* is associated with object $\mathtt{rend}[cid]$, which consists of a sequence of communication slots. In each slot, the *status* field records the status of each process: "e" ("entered") or "l" ("left"); the *shared_data* field stores the data shared among all processes; and *data* stores the data sent by each process. We use notation Ψ to represent *status*'s content.

Many collective communications are synchronizing, while the rest (such as `MPI_Bcast`) can be either synchronizing or non-synchronizing. A collective primitive is implemented by a loose synchronization protocol: in the first "init" phase $\mathtt{syn}_{put}$, process p checks whether there exists a slot in which p has not participated. A negative answer means that p is initializing a new communication, thus p creates a new slot, sets its status to be "e", and stores its value v in this slot. If there are multiple slots that p has not joined into (*i.e.* p is not in the domains of these slots), then p registers itself in the first one. This phase is the same for both synchronizing and non-synchronizing communications. Rules $\mathtt{syn}_{init}$ and $\mathtt{syn}_{write}$ are the simplified cases of $\mathtt{syn}_{put}$.

```
transfer$(p, q) \doteq$ message transferring from process $p$ to process $q$
    $\wedge$ reqs$_p = \Gamma_1^p \diamond \langle dst, cid, tag_p, v, pr_p, \top, \omega_p\rangle_i^{send} \diamond \Gamma_2^p$
    $\wedge$ 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
      $\langle p, dst, tag_p, \omega_p, i\rangle \Rrightarrow \langle src, q, tag_q, \omega_q, j\rangle \wedge$
      $\nexists\langle dst, cid, tag_1, v, pr_1, \top, \omega_1\rangle_m^{send} \in \Gamma_1^p$ :
        $\nexists\langle buf, src_2, cid, tag_2, \_, pr_2, \top, \omega_2\rangle_n^{recv} \in \Gamma_1^q$ :
          $\vee \langle p, dst, tag_1, \omega_1, m\rangle \Rrightarrow \langle src, q, tag_q, \omega_q, j\rangle$
          $\vee \langle p, dst, tag_p, \omega_p, i\rangle \Rrightarrow \langle src_2, q, tag_2, \omega_2, n\rangle$
          $\vee \langle p, dst, tag_1, \omega_1, m\rangle \Rrightarrow \langle src_2, q, tag_2, \omega_2, n\rangle$
--------------------------------------------------------------------------
  $\wedge$ reqs$'_p$ = send the data
      let $b$ = reqs$_p[i].live$ in
        if $\neg b \wedge \neg$reqs$_p[i].pr$ then $\Gamma_1^p \diamond \Gamma_2^p$
        else $\Gamma_1^p \diamond \langle dst, cid, tag_p, \_, pr_p, b, \omega_p\rangle^{send} \diamond \Gamma_2^p$
  $\wedge$ reqs$'_q$ = receive the data
      let $b$ = reqs$_q[j].live$ in
        if $\neg b \wedge \neg$reqs$_q[j].pr$ then $\Gamma_1^q \diamond \Gamma_2^q$
        else $\Gamma_1^q \diamond \langle buf, p, cid, tag_q, v, pr_q, b, \omega_q\rangle^{recv} \diamond \Gamma_2^q$
  $\wedge \neg$reqs$_q[j].live \Rightarrow$ mems$'_q[buf] = v$ write the data into memory

recv_wait$(request, status, p) \doteq$ wait for a receive request to complete
let $req\_index$ = mems$_p[request]$ in
$\wedge$ reqs$'_p[req\_index].live = \bot$ indicate the wait has been called
$\wedge$
  $\vee$ ($\neg$reqs$_p[req\_index].active \Rightarrow$ mems$'_p[status] = empty\_status$)
  $\vee$ the request is still active
    let $\Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \top, \omega\rangle_{req\_index}^{recv} \diamond \Gamma_2$ = reqs$_q$ in
    let $b = pr \wedge \neg$reqs$_p[req\_index].dealloc$ in
    let $new\_reqs$ =
      if $b$ then $\Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \bot, \omega\rangle^{recv} \diamond \Gamma_2$ deactivate the request
      else $\Gamma_1 \diamond \Gamma_2$ remove the request
    in
    let $new\_req\_index$ = if $b$ then $req\_index$ else REQUEST_NULL in update the handle
    if reqs$_q[req\_index].cancelled$ then
      mems$'_p[status] = get\_status($reqs$_p[req\_index]) \wedge$
      reqs$'_p = new\_reqs \wedge$ mems$'_p[request] = new\_req\_index$
    else if $src$ = PROC_NULL then
      mems$'_p[status] = null\_status \wedge$ reqs$'_p = new\_reqs \wedge$ mems$'_p[request] = new\_req\_index$
    else
      wait until the data arrive, then write it to the memory
                    $v\_ \neq \_$
      --------------------------------------------------------------------
       mems$'_p[status] = get\_status($reqs$_p[req\_index]) \wedge$ mems$'_p[buf] = v\_ \wedge$
      reqs$'_p = new\_reqs \wedge$ mems$'_p[request] = new\_req\_index$
```

Fig. 4. Modeling point-to-point communications (II).

After the "init" phase, process p proceeds to its "wait" phase. Among all the slots, p locates the first one it has entered but not left. If the communication is synchronizing, then p has to wait until all other processes finish their "init" phases; otherwise it proceeds. 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 "left".

These protocols are used to specify collective communication primitives (Fig. 7). For example, MPI_Bcast is implemented by two transitions: MPI_Bcast$_{init}$ and MPI_Bcast$_{wait}$. The root first sends its data to the rendezvous in MPI_Bcast$_{init}$, then it calls either the asyn$_{wait}$ rule or the syn$_{wait}$ rule depending on whether the primitive is synchronizing. In the synchronizing case the wait returns immediately without waiting for the completion of other processes. On the other hand, a non-root process always calls the syn$_{wait}$ rule because it must wait for the data from the root to "reach" the rendezvous.

```
bcast$_{\texttt{init}}(buf, v, root, comm, p) \doteq$ the root broadcasts data to processes
  $(comm.group[root] = p)$ ? syn$_{\texttt{put}}(comm.cid, v, \epsilon, p)$ : syn$_{\texttt{init}}(comm.cid, p)$
bcast$_{\texttt{wait}}(buf, v, root, comm, p) \doteq$
  if $comm.group[root] = p$ then $need\_syn$ is a global flag whose value is set by the user
    $need\_syn$ ? syn$_{\texttt{wait}}(comm.cid, p)$ : asyn$_{\texttt{wait}}(comm.cid, p)$
  else syn$_{\texttt{wait}}(comm.cid, p) \wedge$ mems$'_p[buf]$ = rend$_p[comm.cid].sdata$
```

```
$\mathtt{send\_wait}(request, status, p) \doteq$ 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
$\wedge$
  $\vee\ (\neg\mathtt{reqs}_p[req\_index].active \Rightarrow \mathtt{mems}'_p[status] = empty\_status)$
  $\vee$ the request is still active
  $\mathtt{let}\ \Gamma_1 \diamond \langle dst, cid, tag, v\_, pr, \top, \omega\rangle^{mode}_{req\_index} \diamond \Gamma_2 = \mathtt{reqs}_q\ \mathtt{in}$
  $\mathtt{let}\ b = pr\ \wedge\ \neg\mathtt{reqs}_p[req\_index].dealloc\ \vee\ v\_ \neq \_\ \mathtt{in}$
  $\mathtt{let}\ new\_reqs =$
    $\mathtt{if}\ \neg b\ \mathtt{then}\ \Gamma_1 \diamond \Gamma_2$ remove the request
    $\mathtt{else}\ \Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \bot, \omega\rangle^{recv} \diamond \Gamma_2$ deactive the request
  $\mathtt{in}$
  $\mathtt{let}\ new\_req\_index = \mathtt{if}\ b\ \mathtt{then}\ req\_index\ \mathtt{else}\ \mathtt{REQUEST\_NULL}\ \mathtt{in}$
  $\mathtt{let}\ action =$ update the queue, the status and the request handle
    $\wedge\ \mathtt{mems}'_p[status] = get\_status(\mathtt{reqs}_p[req\_index])$
    $\wedge\ \mathtt{reqs}'_p = new\_reqs\ \wedge\ \mathtt{mems}'_p[request] = new\_req\_index$
  $\mathtt{in}$
    $\mathtt{if}\ \mathtt{reqs}_q[req\_index].cancelled\ \mathtt{then}\ action$
    $\mathtt{else\ if}\ dst = \mathtt{PROC\_NULL}\ \mathtt{then}$
      $\mathtt{mems}'_p[status] = null\_status\ \wedge\ \mathtt{reqs}'_p = new\_reqs\ \wedge\ \mathtt{mems}'_p[request] = new\_req\_index$
    $\mathtt{else\ if}\ mode = ssend\ \mathtt{then}$ synchronous send, the guard requires a matching receive
        $\dfrac{\exists q : \exists\langle src_1, cid, tag_1, \_, pr_1, \top, \omega_1\rangle^{recv}_k \in \Gamma_1 : \langle dst, p, tag, \omega, req\rangle \asymp \langle src_1, q, tag_1, \omega_1, k\rangle}{action}$
    $\mathtt{else\ if}\ mode = bsend\ \mathtt{then}$
      $action\ \wedge\ \mathtt{buffer}'.capacity = \mathtt{buffer}.capacity - \mathtt{size}(v\_)$
    $\mathtt{else}$ if no buffer is used then wait until the value is sent
      $\dfrac{\neg use\_buffer \Rightarrow (v\_ = \_)}{action}$

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

$\mathtt{wait\_any}(count, req_{array}, index, status, p) \doteq$ wait for any request in $req_{array}$
$\mathtt{if}\ \forall i \in 0..count - 1 : req_{array}[i] = \mathtt{REQUEST\_NULL}\ \vee\ \neg\mathtt{reqs}_p[req_{array}[i]].active$
$\mathtt{then}\ \mathtt{mems}'_p[index] = \mathtt{UNDEFINED}\ \wedge\ \mathtt{mems}'_p[status] = empty\_status$
$\mathtt{else}\ \dfrac{\exists\ i : has\_completed(req_{array}[i], p)}{\mathtt{mems}'_p[index] = \mathtt{choose}\ i : has\_completed(req_{array}[i], p)\ \wedge\ \mathtt{mems}'_p[status] = get\_status(\mathtt{reqs}_p[req_{array}[i]])}$

$\mathtt{wait\_all}(count, req\_array, status\_array, p) \doteq$ wait for all requests to complete
$\forall i \in 0..count - 1 : \mathtt{wait\_one}(req_{array}[i], status\_array[i], p)$
```

Fig. 5. Modeling point-to-point communications (III).

MPI-2 extends many MPI-1 collective primitives to intercommunicators. An intercommunicator contains a local group and a remote group. To model this, we replace $\mathtt{comms}_p[cid].group$ with $\mathtt{comms}_p[cid].group \cup \mathtt{comms}_p[cid].remote\_group$ in the rules shown in Fig. 6.

4.6. Evaluation and discussion

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 model checked using TLC. Although typically test cases are of only dozens of lines of code, they are able to expose most of the formalization errors.

Another set of test cases are built to verify the *self-consistency* of the specification modeled after [49] where self-consistency rules are used as performance guidelines. It is possible to relate aspects of MPI to each other, *e.g.* explain certain MPI primitives in terms of other MPI primitives.

```
Data Structures
  rendezvous for a communication :
    ⟨status : [int → {"e", "l"}], sdata, data : [int → value]⟩ array
```

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

$\mathtt{syn}_{\mathtt{put}}(cid, v_s, v, p) \doteq$
$\mathtt{if}\ cid \notin \mathtt{DOM\ rend\ then\ rend}'[cid] = \langle [p \mapsto \text{“e”}], v_s, [p \mapsto v] \rangle$
$\mathtt{else\ if}\ \forall slot \in \mathtt{rend}[cid] : p \in \mathtt{DOM}\ (slot.status)\ \mathtt{then}$
$\quad \mathtt{rend}'[cid] = \mathtt{rend}[cid] \diamond \langle [p \mapsto \text{“e”}], v_s, [p \mapsto v] \rangle$
$\mathtt{else}$

$$\frac{\mathtt{rend}[cid] = \Gamma_1 \diamond \langle \Psi, \alpha, S_v \rangle \diamond \Gamma_2 \ \wedge\ p \notin \mathtt{DOM}\ \Psi \ \wedge\ \forall slot \in \Gamma_1 : p \in \mathtt{DOM}\ (slot.status)}{\mathtt{rend}'[cid] = \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“e”}),\ v_s,\ S_v \uplus (p, v) \rangle \diamond \Gamma_2}$$

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

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

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

$$\frac{\begin{array}{l}\mathtt{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“e”}), v_s, S_v \rangle \diamond \Gamma_2 \ \wedge \\ \forall k \in \mathtt{comms}_p[cid].group : k \in \mathtt{DOM}\ \Psi \ \wedge\ \forall slot \in \Gamma_1 : slot.status[p] \neq \text{“e”}\end{array}}{\begin{array}{ll}\mathtt{rend}'[cid] = & \mathtt{if}\ \forall k \in \mathtt{comms}_p[cid].group : k \in \mathtt{DOM}\ \Psi \wedge \Psi[k] = \text{“l”}\ \mathtt{then}\ \Gamma_1 \diamond \Gamma_2 \\ & \mathtt{else}\ \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“l”}),\ v_s,\ S_v \rangle \diamond \Gamma_2\end{array}}$$

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

$$\frac{\mathtt{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“e”}), v_s, S_v \rangle \diamond \Gamma_2 \ \wedge\ \forall slot \in \Gamma_1 : slot.status[p] \neq \text{“e”}}{\begin{array}{ll}\mathtt{rend}'[cid] = & \mathtt{if}\ \forall k \in \mathtt{comms}_p[cid].group : k \in \mathtt{DOM}\ \Psi \wedge \Psi[k] = \text{“l”}\ \mathtt{then}\ \Gamma_1 \diamond \Gamma_2 \\ & \mathtt{else}\ \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“l”}),\ v_s,\ S_v \rangle \diamond \Gamma_2\end{array}}$$

Fig. 6. The basic protocol for collective communications.

p_0	p_1	p_2
$\mathtt{syn}_{\mathtt{put}}(cid = 0, sdata = v_s, data = v_0)$	$\mathtt{syn}_{\mathtt{init}}(cid = 0)$	$\mathtt{syn}_{\mathtt{write}}(cid = 0, data = v_2)$
$\mathtt{asyn}_{\mathtt{wait}}(cid = 0)$	$\mathtt{syn}_{\mathtt{wait}}(cid = 0)$	$\mathtt{syn}_{\mathtt{wait}}(cid = 0)$
$\mathtt{syn}_{\mathtt{init}}(cid = 0)$		

step	*event*	rend[0]
1	$\mathtt{syn}_{\mathtt{put}}(0, v_s, v_0, p_0)$	$\langle [0 \mapsto \text{“e”}], v_s, [0 \mapsto v_0] \rangle$
2	$\mathtt{syn}_{\mathtt{init}}(0, p_1)$	$\langle [0 \mapsto \text{“e”}, 1 \mapsto \text{“e”}], v_s, [0 \mapsto v_0] \rangle$
3	$\mathtt{asyn}_{\mathtt{wait}}(0, p_0)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}], v_s, [0 \mapsto v_0] \rangle$
4	$\mathtt{syn}_{\mathtt{init}}(0, p_0)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}], v_s, [0 \mapsto v_0] \rangle \diamond \langle [0 \mapsto \text{“e”}], \epsilon, \epsilon \rangle$
5	$\mathtt{syn}_{\mathtt{write}}(0, v_2, p_2)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}, 2 \mapsto \text{“e”}], v_s, [0 \mapsto v_0, 2 \mapsto v_2] \rangle \diamond \langle [0 \mapsto \text{“e”}], \epsilon, \epsilon \rangle$
6	$\mathtt{syn}_{\mathtt{wait}}(0, p_2)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}, 2 \mapsto \text{“l”}], v_s, [0 \mapsto v_0, 2 \mapsto v_2] \rangle \diamond \langle [0 \mapsto \text{“e”}], \epsilon, \epsilon \rangle$
7	$\mathtt{syn}_{\mathtt{wait}}(0, p_1)$	$\langle [0 \mapsto \text{“e”}], \epsilon, \epsilon \rangle$

Fig. 7. An example using the collective protocol. Three processes participate in collective communications via a communicator with $cid = 0$. 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 Fig. 6.

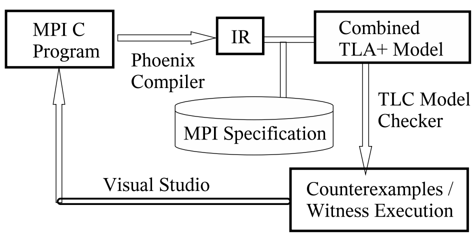
For example, a message of size $k \times n$ can be divided into k sub-messages sent separately; a collective primitive can be replaced by the combination of several point-to-point or one-sided primitives. We introduce relation $\mathtt{MPI\_}A \simeq \mathtt{MPI\_}B$ to indicate that A and B have the same functionality. This relation helps us to design test cases to test the specification of some MPI primitives. To verify these relations we design test cases with concrete inputs and run the TLC to make sure that the same outputs are obtained. We plan to prove them formally in the Isabelle/TLA tool [17].

$$\begin{aligned}
\mathtt{MPI\_A(k \times n)} &\simeq (\mathtt{MPI\_A(n)}_1;\ \ldots;\ \mathtt{MPI\_A(n)}_k) \\
\mathtt{MPI\_A(k \times n)} &\simeq (\mathtt{MPI\_A(k)}_1;\ \ldots;\ \mathtt{MPI\_A(k)}_n) \\
\mathtt{MPI\_Bcast(n)} &\simeq (\mathtt{MPI\_Send(n)};\ \ldots;\ \mathtt{MPI\_Send(n)}) \\
\mathtt{MPI\_Gather(n)} &\simeq (\mathtt{MPI\_Recv(n/p)}_1;\ \ldots;\ \mathtt{MPI\_Recv(n/p)}_p)
\end{aligned}$$

It should be noted that we have not modeled all the details of the MPI standard, which include:

- *Implementation details.* To the greatest extent we have avoided asserting implementation-specific details in our formal semantics. One obvious example is the **info** object is ignored.
- *Physical hardware.* The underlying physical hardware is invisible in our model. Thus we do not model hardware related primitives like `MPI_Cart_map`.
- *Profiling interface.* The MPI profiling interface is to permit the implementation of profiling tools. It is irrelevant to the semantics of MPI primitives.

Issues raised by modeling. While creating the model we became aware of some specific issues that have not been discussed in the standard. For example, `MPI_Probe` on process j becomes enabled when there is a matching request posted on process j;



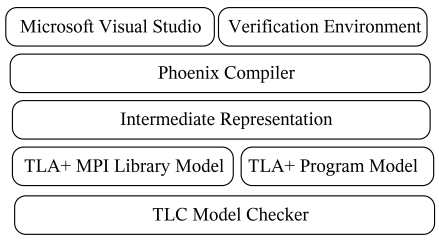


Fig. 8. Architecture of the verification framework. The left (right) one indicates the flow (hierarchical) relation of the components.

`MPI_Cancel` attempts to cancel the corresponding communication. The standard says the message may still complete, and it is up to the user to program appropriately. In this context, we identify some specific issues: (i) There are numerous ways that `MPI_Probe` and `MPI_Cancel` can interact, resulting in an undefined system state. In particular, any time a message is probed successfully, it is not specified whether it is still possible for the message to be canceled or if the message must at that point be delivered. (ii) `MPI_Cancel` also creates an undefined state when used with ready mode send. Consider an execution trace: `MPI_Irecv`; `MPI_Irsend`; `MPI_Cancel`; . . .. If the ready send is successful, can the receive still be canceled? and (iii) Continuing with Cancel, what happens if the null request is canceled?

5. Verification framework

In our previous work [16], we developed a C front-end for supporting direct execution against our semantics. The modeling framework uses the Microsoft Phoenix Compiler [50] as the front-end. 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, assignments are modeled by their effect on the memory. Jumps are modeled by transition rules modifying the values of the program counters. This transition system completely captures the control skeleton of the input MPI program.

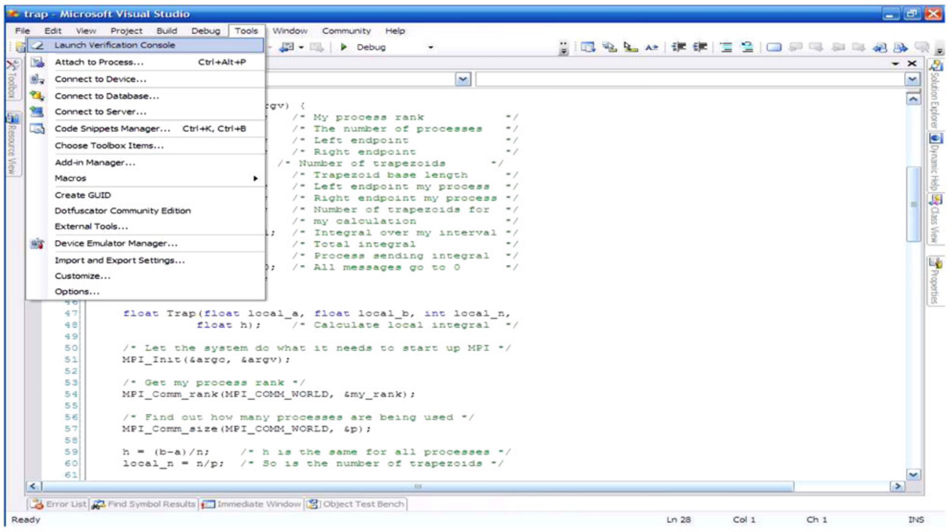
The architecture of the verification framework is shown in Fig. 8. The input program is compiled into an intermediate representation, the Phoenix IR. We read the Phoenix IR to produce TLA+ code. The TLC model checker integrated in our framework enables us to perform verification on the input C programs. 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`; and (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 [51].

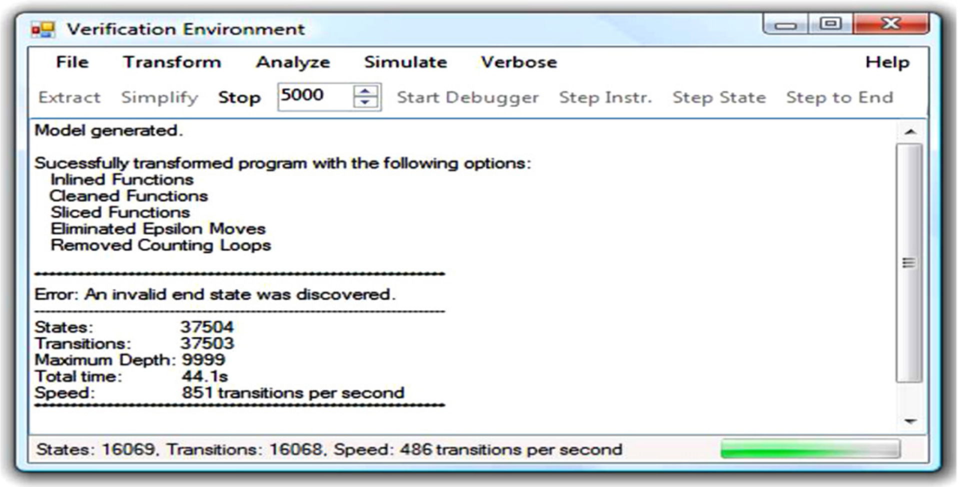
Code generation. During the translation from Phoenix IR to TLA+, we build a record *map* to store all the variables in the intermediate language. The address of a variable x is given by the TLA+ expression $map.x$; and its value at the memory is returned by $\texttt{mems}[map.x]$. Before running TLC, the initial values of all constants and variables are specified 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 $transition_1$, $transition_2$, . . . are produced by translating MPI primitive calls and IR statements.

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

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 primitive 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 Fig. 9 show the debugger interface and the report of an error trace.



```
rgv) {
        /* My process rank               */
        /* The number of processes       */
        /* Left endpoint                 */
        /* Right endpoint                */
      /* Number of trapezoids        */
        /* Trapezoid base length         */
        /* Left endpoint my process      */
        /* Right endpoint my process     */
        /* Number of trapezoids for      */
        /* my calculation                */
    ;   /* Integral over my interval     */
        /* Total integral                */
        /* Process sending integral      */
    );  /* All messages go to 0          */

    float Trap(float local_a, float local_b, int local_n,
               float h);     /* Calculate local integral  */

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;     /* h is the same for all processes */
    local_n = n/p;   /* So is the number of trapezoids */
```


Verification Environment
File Transform Analyze Simulate Verbose Help
Extract Simplify Stop 5000 Start Debugger Step Instr. Step State Step to End

```
Model generated.

Sucessfully transformed program with the following options:
  Inlined Functions
  Cleaned Functions
  Sliced Functions
  Eliminated Epsilon Moves
  Removed Counting Loops

**************************************************
Error: An invalid end state was discovered.
--------------------------------------------------
States:          37504
Transitions:     37503
Maximum Depth:   9999
Total time:      44.1s
Speed:           851 transitions per second
**************************************************

States: 16069, Transitions: 16068, Speed: 486 transitions per second
```

Fig. 9. Two screenshots of the verification framework. The upper one shows the development environment extended from Visual Studio; the lower one displays the result on the "TRAP" example taken from [52].

Examples. A simple C program containing only one statement

if (rank == 0) MPI_Bcast (&b, 1, MPI_INT, 0, comm1);

is translated to the following code. Here the variable appearing in the source is prefixed by a "_".

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

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). Fig. 10 shows a more complicated example.

6. Discussions and conclusions

To help reason about programs that use MPI for communication, we have developed a formal TLA+ semantic definition of MPI 2.0 primitives to augment the existing standard. We described this specification, as well as our framework to extract models from SPMD-style C programs.

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:

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

Fig. 10. An example C program and its corresponding TLA+ code. Notation *changed*(`mems`) specifies that all objects other than `mem` are unchanged. Statement "return 0" is not translated since we do not model procedure calls (functions are inlined during compilation).

It is advantageous to have both declarative and executable specifications for APIs. While we have been relatively happy with TLA+ as a specification language, much of the value we derived from TLA+ is from the accompanying model checker TLC which uses the explicit state enumeration technology to calculate reachable states. Such tools cannot be used to calculate the outcome of general scenarios such as these: "what will happen if we initialize an MPI runtime to a state satisfying a high level predicate and some partially specified symbolic inputs are applied?" It is well known that the technology underlying tools such as Alloy [25] (namely Boolean satisfiability) and SAL [33] (namely decision procedures) would be more appropriate for such calculations. Therefore, it would also be of interest to explore the use of symbolic reasoning capabilities in conjunction with API specifications.

The experience gained in our effort makes us firmly believe that a formal semantic definition and exploration approach as described here should accompany every future effort in creating parallel and distributed programming APIs. Following this belief, we are currently involved in writing a formal specification for MCAPI, an API of growing interest for multi-core communication [14]. This exercise is expected to be much more tractable and timely: (i) MCAPI is in its inception with few (or no) widely used MCAPI library implementations, (ii) MCAPI is much smaller than MPI — a 40-page printout of the specification document, as opposed to a 600-page printout, and (iii) the correctness of MCAPI implementations will be more critical than that of MPI in the sense that some of these implementations may be committed to silicon, where bug-fixing is far more expensive. We also plan to derive additional value from our formalization by generating valuable platform tests for MCAPI implementations, in a manner similar to [53].

In conclusion, our effort with respect to formalizing MPI has been a major learning experience. For practical reasons, we could not influence the MPI community as much as an idealized view would suggest: (i) the MPI specification is already large, and is growing even larger with the new additions to MPI currently under discussion, (ii) our efforts have been fairly late considering when MPI was initially introduced. That said, our effort has helped identify a few omissions and ambiguities in the original MPI reference standard document [16]. While these were brought to the attention of members of the MPI forum, they have been independently addressed by subsequent revisions to the published MPI standard documents. Nevertheless, the genuine interest (and even admiration) expressed by the MPI community encourages us to pursue formalization in a more timely fashion for future efforts.

References

[1] MPI: A Message-Passing Interface Standard Version 2.1, June 23, 2008, with revisions that appeared since then. Up to date specifications are at http://www.mpi-forum.org.

[2] W.D. Gropp, Learning from the success of MPI, in: 8th International Conference High Performance Computing, HiPC 2001, 2001, pp. 81–92.

[3] A. Geist, MPI must evolve or die, invited Talk Given at EuroPVM/MPI 2008, 2008.

[4] The message passing interface forum, MPI: a Message-Passing Interface Standard, http://www.mpi-forum.org/docs/, 1995.

[5] W. Gropp, E.L. Lusk, N.E. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing 22 (6) (1996) 789–828.
[6] J.M. Squyres, A. Lumsdaine, A component architecture for LAM/MPI, in: Proceedings, 10th European PVM/MPI Users' Group Meeting, 2003, pp. 379–387.
[7] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, T.S. Woodall, Open MPI: goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, 2004, pp. 97–104.
[8] D.K.P. Jiuxing Liu, Jiesheng Wu, High performance RDMA-based MPI implementation over infiniband, International Journal of Parallel Programming 32 (3) (2004) 167–198.
[9] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008.
[10] B. Chapman, G. Jost, R.V. Pas, Using OpenMP, MIT Press, 2008.
[11] Ct: C for throughput computing, http://techresearch.intel.com/articles/Tera-Scale/1514.htm.
[12] J. Reinders, Intel thread building blocks, 2008.
[13] D. Leijens, W. Schulte, The design of a task parallel library, http://research.microsoft.com/apps/pubs/default.aspx?id=77368, 2008.
[14] Multicore communications API, http://www.multicore-association.org.
[15] H.-J. Boehm, Threads cannot be implemented as a library, in: Programming Language Design and Implementation, PLDI, 2005, pp. 261–268.
[16] R. Palmer, M. Delisi, G. Gopalakrishnan, R.M. Kirby, An approach to formalization and analysis of message passing libraries, in: Formal Methods for Industry Critical Systems, FMICS, Berlin, 2007, pp. 164–181, best Paper Award.
[17] TLA — The Temporal Logic of Actions, http://research.microsoft.com/users/lamport/tla/tla.html.
[18] Leslie Lamport, The Win32 threads API specification, http://research.microsoft.com/users/lamport/tla/threads/threads.html.
[19] B. Batson, L. Lamport, High-level specifications: lessons from industry, in: Formal Methods for Components and Objects, FMCO, 2002, pp. 242–261.
[20] G. Li, M. DeLisi, G. Gopalakrishnan, R.M. Kirby, Formal specification of the MPI-2.0 standard in TLA+, in: Principles and Practices of Parallel Programming, PPoPP, Poster Session. 2008, pp. 283–284.
[21] G. Li, R. Palmer, M. DeLisi, G. Gopalakrishnan, R.M. Kirby, Formal specification of MPI 2.0: case study in specifying a practical concurrent programming API, Tech. Rep. UUCS-09-003, University of Utah, 2009. http://www.cs.utah.edu/research/techreports/2009/pdf/UUCS-09-003.pdf.
[22] IEEE, IEEE standard for Radix-independent floating-point arithmetic, ANSI/IEEE Std 854-1987, 1987.
[23] J. Harrison, Formal verification of square root algorithms, Formal Methods in System Design 22 (2) (2003) 143–154.
[24] M. Abadi, L. Lamport, S. Merz, A TLA solution to the RPC-memory specification problem, in: Formal Systems Specification, 1994, pp. 21–66.
[25] D. Jackson, Alloy: a new technology for software modeling, in: TACAS'02, in: LNCS, vol. 2280, Springer, 2002, p. 20.
[26] D. Jackson, I. Schechter, H. Shlyahter, Alcoa: the ALLOY constraint analyzer, in: ICSE'00: proceedings of the 22nd International Conference on Software Engineering, 2000, pp. 730–733.
[27] Abstract state machines, http://www.eecs.umich.edu/gasm/.
[28] W. Kuchera, C. Wallace, Toward a programmer-friendly formal specification of the UPC memory model, Tech. Rep. 03-01, Michigan Technological University, 2003.
[29] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, K. Wansbrough, Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets, in: SIGCOMM, 2005, pp. 265–276.
[30] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, K. Wansbrough, Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations, in: Symposium on the Principles of Programming Languages, POPL, 2006, pp. 55–66.
[31] The HOL theorem prover, http://hol.sourceforge.net/.
[32] M. Norrish, C formalised in HOL, Ph.D. thesis, University of Cambridge, 1998. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf.
[33] SAL, http://sal.csl.sri.com/.
[34] The maude system, http://maude.cs.uiuc.edu/.
[35] P. Georgelin, L. Pierre, T. Nguyen, A formal specification of the MPI primitives and communication mechanisms, Tech. Rep., LIM, 1999.
[36] P.V. Eijk, M. Diaz (Eds.), Formal Description Technique Lotos: Results of the Esprit Sedos Project, Elsevier Science Inc., New York, NY, USA, 1989.
[37] S.F. Siegel, G. Avrunin, Analysis of MPI programs, Tech. Rep. UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst, 2003.
[38] G. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[39] S.F. Siegel, G.S. Avrunin, Modeling wildcard-free MPI programs for verification, in: Principles and Practices of Parallel Programming, PPoPP, 2005, pp. 95–106.
[40] R. Palmer, G. Gopalakrishnan, R.M. Kirby, Semantics driven dynamic partial-order reduction of MPI-based parallel programs, in: PADTAD '07, 2007, pp. 43–53.
[41] S.F. Siegel, Model checking nonblocking MPI programs, in: VMCAI 07, 2007, pp. 44–58.
[42] S. Barrus, G. Gopalakrishnan, R.M. Kirby, R. Palmer, Verification of MPI programs using SPIN, Tech. Rep. UUCS-04-008, University of Utah, 2004.
[43] R. Palmer, S. Barrus, Y. Yang, G. Gopalakrishnan, R.M. Kirby, Gauss: a framework for verifying scientific computing software, Electronic Notes in Theoretical Computer Science 144 (3) (2006) 95–106.
[44] S. Pervez, G. Gopalakrishnan, R.M. Kirby, R. Thakur, W. Gropp, Formal verification of programs that use MPI one-sided communication, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI, 2006, pp. 30–39.
[45] R. Palmer, G. Gopalakrishnan, R.M. Kirby, The communication semantics of the message passing interface, Tech. Rep. UUCS-06-012, The University of Utah, October 2006.
[46] A. Danalis, K.-Y. Kim, L. Pollock, M. Swany, Transformations to parallel codes for communication-computation overlap, in: SC'05: ACM/IEEE Conference on Supercomputing, 2005, p. 58.
[47] Formal specification of MPI 2.0 in TLA+, http://www.cs.utah.edu/formal_verification/mpitla/.
[48] J. Reynolds, Separation logic: a logic for shared mutable data structures, in: LICS'02, 2002, pp. 55–74.
[49] J.L. Träff, W. Gropp, R. Thakur, Self-consistent MPI performance requirements, in: PVM/MPI, 2007, pp. 36–45.
[50] Microsoft, Phoenix academic program, http://research.microsoft.com/phoenix, 2007.
[51] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, New York, Inc., Secaucus, NJ, USA, 1999.
[52] P.S. Pacheco, Parallel programming with MPI, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
[53] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L. Nachmanson, Model-based testing of object-oriented reactive systems with Spec Explorer, in: Formal Methods and Testing, in: LNCS, vol. 4949, 2008, pp. 39–76. http://www.springerlink.com/content/x63053756207l2l8/.