Collective Contracts for Message-Passing Parallel Programs

Anonymous Author(s)

ABSTRACT

Procedure contracts are a well-known approach for specifying programs in a modular way. We investigate a new contract theory for collective procedures in parallel message-passing programs. As in the sequential setting, one can verify that a procedure f conforms to its contract using only the contracts, and not the implementations, of the collective procedures invoked by f. We apply this approach to C programs that use the Message Passing Interface (MPI), introducing a new contract language that extends the ANSI/ISO C Specification Language (ACSL). We present contracts for the standard MPI collective functions, as well as many user-defined collective functions. A prototype verification system has been implemented using the CIVL symbolic execution and model checking framework for checking contract satisfaction within small bounds on the number of processes.

ACM Reference Format:

Anonymous Author(s). 2023. Collective Contracts for Message-Passing Parallel Programs. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Divide and conquer is one of the fundamental principles of software engineering. Most approaches to the specification, verification, and construction of large programs benefit from decomposing tasks into manageable pieces which can be worked on independently.

Procedure contracts [36, 59, 60] are an example of a divide and conquer approach for specification. A typical contract specifies a precondition and postcondition. These encode a claim: *if the precondition holds when this procedure is called then the postcondition will hold when the procedure returns.* In a procedural programming language, such as C, a complete specification of the program entails a specification for each procedure.

Contracts have many uses. They serve as precise documentation of intended behavior. When implementing a procedure f, one needs to know only the contracts for the procedures used by f—not their implementations. For debugging, the preconditions and postconditions can be treated as runtime assertions. To verify that a program is correct, one need only verify that each procedure f satisfies its contract under the assumption that the procedures used by f satisfy their contracts.

Contract languages have been developed for many programming languages. These include the *Java Model Language* (JML) [51] for Java and the *ANSI C Specification Language* (ACSL) [11] for C. A number of tools have been developed which (partially) automate the process of verifying that a procedure satisfies its contract; an example for C is Frama-C [24] with the WP plugin [10].

The contract approach has also been extended to concurrent programs. These programs are notoriously difficult to get right, because of the combinatorial explosion in the number of thread interleavings. The VCC verifier [17, 18, 61] takes a contract approach, based on object invariants in addition to pre- and postconditions, to shared-memory concurrent C programs. VCC has been used to verify hundreds of functions comprising tens of thousands of lines of code in Microsoft's Hyper-C hypervisor [17].

In this paper, we explore a procedure contract system for message-passing parallel programs. Message-passing and shared-memory concurrency have very different characteristics, and require different specification and verification techniques. Programming languages and APIs based on message-passing include CSP [42], Ada's rendezvous system, Erlang [4], the Message-Passing Interface (MPI) [58], MCAPI [43], and Go [72].

While some of the ideas proposed here apply to message-passing generally, the focus is on MPI, a standard library used with C, C++, or Fortran. MPI is a dominant means of expressing parallelism in high performance and scientific computing [14]. Numerous applications of great importance to society, from climate models to simulations of galaxies and atomic nuclei, use MPI and run on platforms ranging from desktops with a few cores to the most powerful supercomputers.

In the scientific computing community, there is growing concern with correctness and with the enormous effort expended on testing and debugging. Previous studies on the prevalence of defects in scientific software (e.g., [37–39, 57]), and recent reports on the correctness challenge [34, 35] and workshops devoted to it [20–23] testify to this concern. Much of this work deals with MPI. An effective contract system for MPI could have a serious impact on this community.

Our approach is motivated by the observation that most procedures involving communication in actual MPI programs are *collective*, i.e., they are called by all processes in some group, and the group members communicate and/or synchronize within the call in order to accomplish a coherent change in the global state. Examples include all of the standard blocking MPI collective functions [58, Chapter 5] and a procedure to exchange ghost cells in a standard stencil computation. Typical programs include the Monte Carlo particle transport code OpenMC [65], which consists of over 24K lines of C++/MPI code, and module `parcsr_ls` in the algebraic multigrid solver AMG [75], which is over 35K lines of C/MPI code. Through manual inspection, we confirmed that every function in these codes that involves MPI communication is collective.

Our approach applies to MPI programs with non-collective procedures, it just cannot decompose the specification/verification task for such procedures. If a collective procedure f calls some non-collective procedure g, the user writes a contract for f (but not for

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

g) and the verification of f's contract requires the implementations of both f and g.

The advantage of focusing on collective procedures is that they can be specified almost like sequential procedures. Nevertheless, a number of new issues arise. A collective contract should be able to specify minimum synchronization guarantees—e.g., the root process in a "gather" operation will not return from the call until all non-root processes have entered it. A precondition or postcondition may relate variables on different processes, but since each process may reach the call (or return) at different times, this requires evaluating these conditions on a "composite state" [69]. This also raises the question of exactly when a precondition or postcondition becomes violated. Collective procedures also have certain implicit requirements, such as they are called by all group members in the same order, and there should be no messages sent before the call that could be received inside the call. The issues are surprisingly subtle and are best explored formally using a toy message-passing language that abstracts away much of the accidental complexity of C and MPI. This is the goal of Section 2.

In Section 3, we describe a collective contract language for C/MPI. Rather than design a language from scratch, we start with ACSL and describe changes to incorporate concepts from MPI and collective behavior. The language is expressive enough to fully specify all of the standard MPI collectives as well as many user-defined collectives. We have implemented a prototype verification system using the CIVL symbolic execution and model checking tool, and use it to verify certain procedures conform to their contracts for small bounds on the number of processes. This work is summarized in Section 4. Related work is discussed in Section 5. In Section 6, we wrap up with a discussion of the advantages and limitations of our system, and work that remains.

2 A THEORY OF COLLECTIVE CONTRACTS

2.1 Language

We first describe the simple message-passing language MiniMP. The syntax is given in Figure 1. It does not support "wildcard" receives, multiple communicators, nonblocking communication, or many other MPI features. There is one datatype: integers; 0 is interpreted as *false* and any non-zero integer as *true*. A program consists of global variable declarations followed by (mutually recursive) procedure definitions. Global variables may start with arbitrary values. Each procedure takes a sequence of formal parameters. The procedure body consists of local variable declarations followed by a sequence of statements. Local variables are initially 0. Assignment, branch, loop, call, and compound statements have the usual semantics. Operations have the usual meaning and always return some value—even if the second argument of the division or modulus operator is 0. The operators with '\' are used only in the optional contract and are described below.

A procedure is executed by specifying a positive integer n, the number of processes. Each process executes its own "copy" of the code; there is no shared memory. Each process has a unique ID number in PID $= \{0, \ldots, n-1\}$. A process can obtain its ID using the primitive `pid`; it can obtain n using `nprocs`.

The command "`send` *data* `to` *dest*" sends a message consisting of the value of *data* to the process with ID *dest*; *dest* can be any expression that evaluates to an integer in PID. There is one unbounded FIFO message buffer for each ordered pair of processes $p \rightarrow q$ and the effect of `send` is to append the message to the end of the buffer for which p is the ID of the sender and q is *dest*. As the buffers are unbounded, `send` never blocks. The command "`recv` *buf* `from` *source*" removes the oldest buffered message originating from *source* and stores it in variable *buf*; this command blocks until a message becomes available. A *dest* or *source* not in PID results in a no-op.

A procedure f with a contract is a *collective procedure*. The contract encodes a claim about executions of f: if f is called collectively (by all processes), in such a way that the precondition (specified in the `requires` clause) holds, then all of the following hold for each process p: p will eventually return; p's postcondition (specified in the `ensures` clause) will hold at the post-state; all variables not listed in p's `assigns` clause will have their pre-state values at the post-state; and if q is in p's `waitsfor` set then p will not return before q enters the call. These notions will be made precise below.

Global variables and the formal parameters of the procedure are the only variables that may occur free in a contract; only globals may occur in the *assigns* clause. A postcondition may use `\old(`e`)` to refer to the value of expression e in the pre-state; `\old` may not occur in this e. Pre- and postconditions can use `\on(`e,i`)` to refer to the value of e on process i. These constructs allow contracts to relate the state of different processes, and the state before and after the call.

Example 2.1. The program of Figure 2 has two procedures, both of which are collective. Procedure `g` accepts an argument `k` and sends its value for global variable `x` to its right neighbor, in a cyclic ordering. It then receives into local variable `y` from its left neighbor q, adds `k` to the received value, and stores the result in `x`. The contract for `g` states that when p exits (returns), the value of `x` on p will be the sum of k and the original value of `x` on q. It also declares that p cannot exit until q has entered.

Procedure `f` invokes `g` `nprocs` times. Its contract requires that all processes call `f` with the same value for `k`. It ensures that upon return, the value of `x` will be the sum of its original value and the product of `nprocs` and `k`. It also declares that no process can exit until every process has entered.

2.2 Semantics

Semantics for procedural programs are well-known (e.g., [2]), so we will only summarize the standard aspects of the MiniMP semantics. Fix a program P and an integer $n \geq 1$ for the remainder of this section. Each procedure in P may be represented as a *program graph*, which is a directed graph in which nodes correspond to locations in the procedure body. Each program graph has a designated start node. An edge is labeled by either an expression ϕ (a *guard*) or one of the following kinds of statements: *assignment*, *call*, *return*, *send* or *receive*. An edge labeled *return* is added to the end of each program graph, and leads to the terminal node, which has no outgoing edges.

A *process state* comprises an assignment of values to global variables and a call stack. Each entry in the stack specifies a procedure f, the values of the local variables (including formal parameters) for f, and the program counter, which is a node in the program graph of f. A *state* specifies a process state for each process, as well

program	::=	(`int` *x* ;)∗ *procdef*+
procdef	::=	*contract*? `void` *f* ((`int` *x* (, `int` *x*)∗)?) { (`int` *x* ;)∗ *s*∗ }
s ∈ *stmt*	::=	*x* = *e* ; | *f* ((*e* (, *e*)∗)?) ; | `if` (*e*) *s* (`else` *s*)? | `while` (*e*) *s* | { *s*∗ } | `send` *e* `to` *e* ; | `recv` *x* `from` *e* ;
e ∈ *expr*	::=	`c` | *x* | `nprocs` | `pid` | ⊖*e* | *e* ⊙ *e* | `\on(`*e*,*e*`)` | `\old(`*e*`)`
contract	::=	`/*@ requires` *e*; `ensures` *e*; `assigns` (*x*(,*x*)∗)?; `waitsfor` { *e* | `int` *x* ; *e* }; `*/`

$c \in \mathbb{Z}$ $\quad x, f \in ID$ $\quad \ominus \in \{ \texttt{-}, \texttt{!} \}$ $\quad \odot \in \{ \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{==}, \texttt{<}, \texttt{<=}, \texttt{\&\&}, \texttt{||} \}$

Figure 1: MiniMP syntax

```
int x;
/*@ requires 1;
    ensures x == \on(\old(x), (pid + nprocs - 1) % nprocs) + k;
     assigns x;
    waitsfor { j | int j; j == (pid + nprocs - 1) % nprocs }; */
void g(int k) {
  int y;
  send x to (pid + 1) % nprocs;
  recv y from (pid + nprocs - 1) % nprocs;
  x = y + k;
}
/*@ requires k == \on(k,0);
     ensures x == \old(x) + nprocs * k;
     assigns x;
     waitsfor { j | int j; 0 <= j && j < nprocs }; */
void f(int k) {
  int i; i = 0;
  while (i < nprocs) { g(k); i = i+1; }
}
```

Figure 2: cyc: a MiniMP program

as the state of channel $p \to q$ for all $p, q \in$ PID. The channel state is a finite sequence of integers, the buffered messages sent from p to q.

An *action* is a pair $t = \langle e, p \rangle$, where e is an edge $u \xrightarrow{\alpha} v$ in a program graph and $p \in$ PID. Action t is *enabled* at state s if the program counter of the top entry of p's call stack in s is u and one of the following holds: α is a guard ϕ and ϕ evaluates to *true* in s; α is an assignment, call, return, or send; or α is a receive with source q and channel $q \to p$ is nonempty in s. The execution of an enabled action from s results in a new state s' in the natural way. In particular, execution of a call pushes a new entry onto the stack of the calling process; execution of a return pops the stack and, if the resulting stack is not empty, moves the caller to the location just after the call. The triple $s \xrightarrow{t} s'$ is a *transition*.

Let f be a procedure and s_0 a state with empty channels, and in which each process has one entry on its stack, the program counter of which is the start location for f. An n-process *execution* ζ of f is a finite or infinite chain of transitions $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots$. The *length* of ζ, denoted len(ζ), is the number of transitions in ζ. An execution must be *fair*: if a process p becomes enabled at some point in an infinite execution, then eventually p will execute. Note that, once p becomes enabled, it will remain enabled until it executes, as no process other than p can remove a buffered message with destination p.

A process p *terminates* in ζ if for some i, the stack for p is empty in s_i. We say ζ *terminates* if p terminates in ζ for all $p \in$ PID. The execution *deadlocks* if it is finite, does not terminate, and ends in a state with no enabled action.

It is often convenient to add a "driver" to P when reasoning about executions of a collective procedure f. Then a call of f will be the first action in each process. Say f takes m formal parameters. Form a new program P^f by adding fresh global variables $x_1, \ldots, x_m$ to P, and adding a procedure

$$\texttt{void main() \{ } f(x_1, \ldots, x_m)\texttt{; \}.}$$

By "execution of P^f," we mean an execution of `main` in this new program.

2.3 Collective Correctness

The goal of this section is to define what it means for a procedure to conform to its contract. This notion comprises several conditions on the invocation of collective procedures and their interaction with communication operations. Some of the conditions are generic and others are specified by the contract clauses.

Fix a program P and integer $n \geq 1$. Let C be the set of names of collective procedures of P. Let ζ be an execution $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots$ of a procedure in P. For $i \in 1..\text{len}(\zeta)$, let ζ^i denote the prefix of ζ of length i, i.e., the execution $s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_i} s_i$.

The first correctness condition for ζ is *collective consistency*. To define this concept, consider strings over the alphabet consisting of symbols of the form e^f and x^f, for $f \in C$. Given an action t and $p \in$ PID, define string $T_p(t)$ as follows:

- if t is a call by p to some $f \in C$, $T_p(t) = \mathsf{e}^f$ (t is called an *enter* action)
- if t is a return by p from some $f \in C$, $T_p(t) = \mathsf{x}^f$ (t is called an *exit* action)
- otherwise, $T_p(t)$ is the empty string.

Now let $T_p(\zeta)$ be the concatenation $T_p(t_1)T_p(t_2)\cdots$. Hence $T_p(\zeta)$ records the sequence of collective actions—enter or exit actions—taken by p.

Definition 2.2. An execution ζ is *collective consistent* if there is some $p \in$ PID such that for all $q \in$ PID, $T_q(\zeta)$ equals or is a prefix of $T_p(\zeta)$. We say ζ *commits a consistency violation at step i* if ζ^{i-1} is collective consistent but ζ^i is not.

For the remainder of this section, we assume ζ is collective consistent.

The sequence of actions performed by p in ζ is divided into segments whose boundaries are the collective actions of p. More precisely, given $i \in 0..\text{len}(\zeta)$ and $p \in$ PID, define $k = \text{seg}_p(\zeta, i)$ to be the number of collective actions of p in $t_1, \ldots, t_i$. We say p *is in segment* k *at state* i.

Example 2.3. In program cyc of Figure 2, there is a 3-process execution ζ of P^f illustrated in Figure 3. The execution is collective consistent: $T_p(\zeta)$ is a prefix of $T_1(\zeta) = \mathrm{e}^f \mathrm{e}^g \mathrm{x}^g \mathrm{e}^g \mathrm{x}^g \mathrm{e}^g \mathrm{x}^g \mathrm{x}^f$ for all $p \in \{0, 1, 2\}$. A process is in segment 0 at any point before it executes e^f; it is in segment 1 after executing e^f but before executing its first e^g; and so on. At a given state in the execution, processes can be in different segments; e.g., when process 2 is in segment 1, process 1 is in segment 3 and process 0 is in segment 2.

We now turn to the issue of evaluation of pre- and postconditions. Let f be a collective procedure in P with precondition $\mathrm{pre}(f)$ and postcondition $\mathrm{post}(f)$. Let V_f be the union of the set of formal parameters of f and the global variables of P. As noted above, these are the only variables that may occur free in $\mathrm{pre}(f)$ and $\mathrm{post}(f)$. An f*-valuation* is a function $\alpha\colon \mathsf{PID} \to (V_f \to \mathbb{Z})$. For each process, α specifies a value for each free variable that may occur in $\mathrm{pre}(f)$ or $\mathrm{post}(f)$.

For any expression e that may occur as a sub-expression of $\mathrm{pre}(f)$, and $p \in \mathsf{PID}$, define $[\![e]\!]_{\alpha,p} \in \mathbb{Z}$ as follows:

$$
\begin{aligned}
[\![c]\!]_{\alpha,p} &= c \\
[\![x]\!]_{\alpha,p} &= \alpha(p)(x) \\
[\![\mathtt{nprocs}]\!]_{\alpha,p} &= n \\
[\![\mathtt{pid}]\!]_{\alpha,p} &= p \\
[\![\ominus e]\!]_{\alpha,p} &= \ominus[\![e]\!]_{\alpha,p} \\
[\![e_1 \odot e_2]\!]_{\alpha,p} &= [\![e_1]\!]_{\alpha,p} \odot [\![e_2]\!]_{\alpha,p} \\
[\![\texttt{\textbackslash on}(e_1, e_2)]\!]_{\alpha,p} &= [\![e_1]\!]_{\alpha,q}, \text{ where } q = [\![e_2]\!]_{\alpha,p}.
\end{aligned}
$$

This is the result of evaluating e in process p. Note how `\on` shifts the evaluation context from process p to the process specified by e_2, allowing the precondition to refer to the value of an expression on another process.

Evaluation of an expression involving `\old`, which may occur only in $\mathrm{post}(f)$, requires a second f-valuation β specifying values in the pre-state. The definition of $[\![\cdot]\!]_{\alpha,\beta,p}$ repeats the rules above, replacing each subscript "α" with "α, β", and adds one rule:

$$[\![\texttt{\textbackslash old}(e)]\!]_{\alpha,\beta,p} = [\![e]\!]_{\beta,p}.$$

Say $1 \le i \le \mathrm{len}(\zeta)$ and t_i is an e^f action in process p. Let $r = \mathrm{seg}_p(\zeta, i)$ and

$$Q = \{q \in \mathsf{PID} \mid \mathrm{seg}_q(\zeta, i) \ge r\}, \qquad \alpha'\colon Q \to (V_f \to \mathbb{Z}),$$

where $\alpha'(q)(v)$ is the value of v on process q in state $s_{j(q)}$, and $j(q)$ is the unique integer in $1..i$ such that $t_{j(q)}$ is the r-th collective action of q in ζ. (As ζ is collective consistent, $t_{j(q)}$ is also an e^f action.) In other words, α' uses the values of process q's variables just after q entered the call. Now, α' is not an f-valuation unless $Q = \mathsf{PID}$. Nevertheless, we can ask whether α' can be extended to an f-valuation α such that $[\![\mathrm{pre}(f)]\!]_{\alpha,q}$ holds for all $q \in \mathsf{PID}$. If no such α exists, we say a *precondition violation* occurs at step i.

Example 2.4. Consider program cyc of Figure 2. Suppose process 1 calls $f(1)$ and process 2 calls $f(2)$. Then a precondition violation of f occurs with the second call, because there is no value that can be assigned to k on process 0 for which $1 = \texttt{\textbackslash on}(k, 0)$ and $2 = \texttt{\textbackslash on}(k, 0)$ both hold.

If t_i is an x^f action, define Q and $j(q)$ as above; for any $q \in Q$, $t_{j(q)}$ is also an x^f action. Let $\alpha'(q)(v)$ be the value of v in q at state $s_{j(q)-1}$, i.e., just before q exits. Define $k(q) \in 1..j(q) - 1$ so that $t_{k(q)}$ is the e^f action in q corresponding to $t_{j(q)}$, i.e., $t_{k(q)}$ is the call that led to the return $t_{j(q)}$. Define $\beta'\colon Q \to (V_f \to \mathbb{Z})$ so that $\beta'(q)(v)$ is the value of v on q in state $s_{k(q)}$, i.e., in the pre-state. A *postcondition violation* occurs if it is not the case that there are extensions of α' and β' to f-valuations α and β such that $[\![\mathrm{post}(f)]\!]_{\alpha,\beta,q}$ holds for all $q \in \mathsf{PID}$.

We now explain the *waitsfor* contract clause. Assume again that t_i is an x^f action in process p, and that k is the index of the corresponding e^f action in p. The expession in the `waitsfor` clause is evaluated at the pre-state s_k to yield a set $W \subseteq \mathsf{PID}$. A *waitsfor violation* occurs at step i if there is some $q \in W$ such that $\mathrm{seg}_q(\zeta, i) < \mathrm{seg}_p(\zeta, k)$, i.e., p exits a collective call before q has entered it.

We can now encapsulate all the ways something may go wrong with collective procedures and their contracts:

Definition 2.5. Let P be a program, $\zeta = s_0 \xrightarrow{t_1} s_1 \cdots$ an execution of a procedure in P, and $i \in 1..\mathrm{len}(\zeta)$. Let p be the process of t_i and $r = \mathrm{seg}_p(\zeta, i)$. We say ζ commits a *collective error* at step i if any of the following occur at step i:

(1) a consistency, precondition, postcondition, or *waitsfor* violation,
(2) an *assigns violation*: t_i is an exit action and the value of a variable not in p's *assigns* set differs from its pre-state value,
(3) a *segment boundary violation*: t_i is a receive of a message sent from a process q at t_j ($j < i$) and $\mathrm{seg}_q(\zeta, j) > r$; or t_i is a send to q and $\mathrm{seg}_q(\zeta, i) > r$, or
(4) an *unreceived message violation*: t_i is a collective action and there is an unreceived message sent to p from q at t_j ($j < i$), and $\mathrm{seg}_q(\zeta, j) = r - 1$.

The last two conditions imply that a message that crosses segment boundaries is erroneous. In particular, if an execution terminates without collective errors, every message sent within a segment is received within that same segment.

We can now define what it means for a procedure to conform to its contract. Let f be a collective procedure in P. By a *pre*(f)*-state*, we mean a state of P^f with empty call stacks, empty channels, and an assignment to the global variables satisfying the precondition of f for all processes.

Definition 2.6. A collective procedure f *conforms* (to its contract) if all executions of P^f from *pre*(f)-states are finite and free of deadlocks and collective errors.

Note that any maximal non-deadlocking finite execution terminates. So a conforming procedure will always terminate if invoked from a *pre*(f)-state, i.e., ours is a "total" (not "partial") notion of correctness in the Hoare logic sense.

2.4 Simulation

In the sequential theory, one may verify properties of a procedure f using only the contracts of the procedures called by f. We now generalize that approach for collective procedures. We will assume

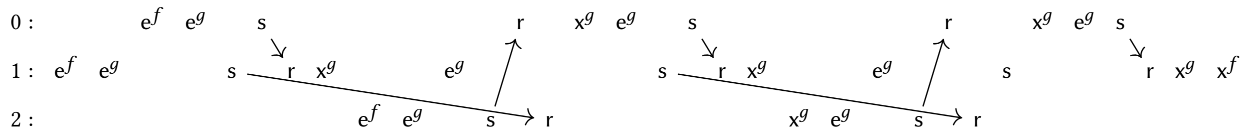

Figure 3: Representation of a 3-process execution of cyc^f of Figure 2. e^f = enter (call) f; x^f = exit (return from) f; s = send; r = receive. The execution has no collective errors and ends in a state with one buffered message sent from process 1 to process 2.

from now on that P has no "collective recursion." That is, in the call graph for P—the graph with nodes the procedures of P and an edge from f to g if the body of f contains a call to g—there is no cycle that includes a collective procedure. This simplifies reasoning about termination.

If $f, g \in C$, we say f *uses* g if there is a path of positive length in the call graph from f to g such that any node in the path other than the first or last is not collective.

Given $f \in C$, we construct a program $\overline{P^f}$ which abstracts away the implementation details of each collective procedure g used by f, replacing the body of g with a stub that simulates g's contract. The stub consists of two new statements. The first may be represented with pseudocode

```
havoc(assigns(g)); assume(post(g));
```

This nondeterministic statement assigns arbitrary values to the variables specified in the *assigns* clause of g's contract, as long as those values do not commit a postcondition violation for g. The second statement may be represented

```
wait(\old(waitsfor(g)));
```

and blocks the calling process p until all processes in p's wait set (evaluated in p's pre-state) reach this statement. This ensures the stub will obey g's *waitsfor* contract clause. Now $\overline{P^f}$ is a program with the same set of collective procedure names, and same contracts, as P^f. A *simulation* of f is an execution of $\overline{P^f}$.

Theorem 1. *Let P be a program with no collective recursion. Let f be a collective procedure in P and assume all collective procedures used by f conform. If all simulations of f from a pre(f)-state are finite and free of deadlock and collective errors, then f conforms.*

Theorem 1 is the basis for the contract-checking tool described in Section 4.2. The tool consumes a C/MPI program annotated with procedure contracts. The user specifies a single procedure f and the tool constructs a CIVL-C program that simulates f by replacing the collective procedures called by f with stubs derived from their contracts. It then uses symbolic execution and model checking techniques to verify that all simulations of f behave correctly. By Theorem 1, one can conclude that f conforms.

A detailed proof of Theorem 1 is given in [3]. Here we summarize the main ideas of the proof.

Two actions from different processes commute as long as the second does not receive a message sent by the first. Two executions are *equivalent* if one can be obtained from the other by a finite number of transpositions of commuting adjacent transitions. The first observation is that equivalence preserves most kinds of violations:

Lemma 1. *Let P be a program and f a procedure in P. Suppose ζ and η are equivalent executions of f. Then all of the following hold:*

1. *ζ commits a consistency, precondition, postcondition, assigns, segment boundary, or unreceived message violation iff η commits a violation of one of those kinds.*
2. *ζ deadlocks iff η deadlocks.*
3. *ζ is finite iff η is finite.*

We say an execution ζ of a collective function f is *correct* if it is finite, does not deadlock, and has no collective errors.

We say ζ commits an *observable* collective error if it commits a collective error when control is not inside a collective call made by f, i.e., when f is the only collective function on the call stack. We say ζ is *observably correct* if it is finite, does not deadlock, and is free of observable collective errors. We are interested in observable errors because those are the kind that will be visible in a simulation, i.e., when each collective function g called by f is replaced with a stub that mimics g's contract.

When ζ has no observable collective error, it can be shown that a collective call to g made within ζ can be *extracted* to yield an execution of g. The idea behind the proof is to transpose adjacent transitions in ζ until all of the actions inside the call to g form a contiguous subsequence of ζ. The resulting execution ξ is equivalent to ζ. Using Lemma 1, it can be shown that ξ is also observably correct and the segment involving the call to g can be excised to yield an execution of g.

The next step is to show that extraction preserves internal errors:

Lemma 2. *Assume ζ is an observably correct execution of collective procedure f in a collective-recursion-free program P. Let $g_1, g_2, \ldots$ be the sequence of collective procedures called from f. If a transition in region r of ζ commits an internal collective error then the execution of P^{g_r} extracted from region r of ζ is incorrect.*

A corollary of Lemma 2 may be summarized as "conforming + observably correct = correct". More precisely,

Lemma 3. *Let P be a collective-recursion-free program and f a collective procedure of P. Assume all collective procedures used by f conform. Let ζ be an execution of P^f. Then ζ is correct if and only if ζ is observably correct.*

To see this, suppose ζ is observably correct but commits an internal collective error. Let r be the region of the transition committing the first internal collective error of ζ. Let g be the associated collective procedure used by f, and χ the execution of P^g extracted from region r of ζ. By Lemma 2, χ is incorrect, contradicting the assumption that g conforms.

The next step is to show that observable errors will be picked up by some simulation. The following is proved using extraction and Lemma 3:

Lemma 4. *Suppose P is a collective-recursion-free program, f is a collective procedure of P, all collective procedures used by f conform, and ζ is an execution of P^f. If ζ has an observable collective error or ends in deadlock then there exists an incorrect simulation of f.*

Since infinite executions are also considered erroneous, we must ensure they are detected by simulation:

Lemma 5. *Suppose P is a collective-recursion-free program, f is a collective procedure of P, and all collective procedures used by f conform. If ζ is an infinite execution of P^f with no observable collective error then there exists an incorrect simulation of f.*

Finally, we can prove Theorem 1. Let ζ be an execution of P^f from a pre(f)-state. Suppose ζ is incorrect. By Lemma 3, ζ is not observably correct. If ζ is finite, Lemma 4 implies an incorrect simulation exists. If ζ is infinite, Lemma 5 implies an incorrect simulation exists.

3 COLLECTIVE CONTRACTS FOR C/MPI

In Section 3.1, we summarize the salient aspects of C/MPI needed for a contract system. Section 3.2 describes the overall grammar of MPI contracts, and Section 3.3 summarizes the syntax and semantics of each new contract primitive.

3.1 Background from MPI

In the toy language of Section 2, every collective procedure was invoked by all processes. In MPI, a collective procedure is invoked by all processes in a *communicator*, an abstraction representing an ordered set of processes and an isolated communication universe.[1] Programs may use multiple communicators. The *size* of a communicator is the number of processes. Each process has a unique *rank* in the communicator, an ID number in $0..size-1$.

In Section 2, a receive always selects the oldest message in a channel. In MPI, a point-to-point send operation specifies a *tag*, an integer attached to the “message envelope.” A receive can specify a tag, in which case the oldest message in the channel with that tag is removed, or the receive can use `MPI_ANY_TAG`, in which case the oldest message is. MPI collective functions do not use tags.

MPI communication operations use *communication buffers*. A buffer b is specified by a void pointer p, *datatype* d (an object of type `MPI_Datatype`), and nonnegative integer *count*. There are constants of type `MPI_Datatype` corresponding to the C basic types: `MPI_INT` for `int`, `MPI_DOUBLE` for `double`, etc. MPI provides a number of functions to build aggregate datatypes. Each datatype specifies a *type map*: a sequence of ordered pairs (t, m) where t is a basic type and m is an integer representing a displacement in bytes. A type map is *nonoverlapping* if the memory regions specified by distinct entries in the type map do not intersect. A receive operation requires a nonoverlapping type map; no such requirement applies to sends. For example, the type map $\{(\texttt{int}, 0), (\texttt{double}, 8)\}$, together with p, specifies an `int` at p and a `double` at `(char*)p+8`. As long as `sizeof(int)` ≤ 8, this type map is nonoverlapping.

The *extent* of d is the distance from its lowest to its highest byte, including possible padding bytes at the end needed for alignment; the precise definition is given in the MPI Standard. The type map of b is defined to be the concatenation of $T_0, \ldots, T_{count-1}$, where T_i is the type map obtained by adding $i * extent(d)$ to the displacements of the entries in the type map of d. For example, if *count* is 2, `sizeof(double) = 8` and `int`s and `double`s are aligned at multiples of 8 bytes, the buffer type map in the example above is

$$\{(\texttt{int}, 0), (\texttt{double}, 8), (\texttt{int}, 16), (\texttt{double}, 24)\}.$$

A message is created by reading memory specified by the send buffer, yielding a sequence of basic values. The message has a *type signature*—the sequence of basic types obtained by projecting the type map onto the first component. The receive operation consumes a message and writes the values into memory according to the receive buffer’s type map. Behavior is undefined if the send and receive buffers do not have the same type signature.

3.2 High-level contract structure

We now describe the syntax and semantics for C/MPI function contracts. A contract may specify either an MPI collective function, or a user-defined collective function. A user function may be implemented using one or more communicators, point-to-point operations, and MPI collectives.

The top level grammar is given in Figure 4. A function contract begins with a sequence of distinct behaviors, each with an assumption that specifies when that behavior is active. Clauses in the global contract scope preceding the first named behavior are thought of as comprising a single behavior with a unique name and assumption *true*. The behaviors may be followed by `disjoint behaviors` and `complete behaviors` clauses, which encode claims that the assumptions are pairwise disjoint, and their disjunction is equivalent to *true*, respectively. All of this is standard ACSL, and we refer to it as the *sequential part* of the contract.

A new kind of clause, the *comm-clause*, may occur in the sequential part. A comm-clause begins “`mpi uses`” and is followed by a list of terms of type `MPI_Comm`. Such a clause specifies a guarantee that no communication will take place on a communicator *not* in the list. When multiple comm-clauses occur within a behavior, it is as if the lists were appended into one.

Collective contracts appear after the sequential part. A collective contract begins “`mpi collective`” and names a communicator c which provides the context for the contract; c must occur in a comm-clause from the sequential part. A collective contract on c encodes the claim that the function conforms to its contract (Definition 2.6) with the adjustment that all of the collective errors defined in Definition 2.5 are interpreted with respect to c only.

A collective contract may comprise multiple behaviors. As with the sequential part, clauses occurring in the collective contract before the first named behavior are considered to comprise a behavior with a unique name and assumption *true*.

3.3 New contract primitives for MPI

Type signatures. The new logic type `mpi_sig_t` represents MPI type signatures. Its domain consists of all finite sequences of basic C types. As with all ACSL types, equality is defined and `==` and `!=` can be used on two such values in a logic specification. If `t` is a term of integer type and `s` is a term of type `mpi_sig_t`, then `t*s` is a term of type `mpi_sig_t`. If the value of `t` is n and $n \geq 0$, then `t*s` denotes the result of concatenating the sequence of `s` n times.

[1] We consider only *intra-communicators* in this paper.

function-contract	::=	*requires-clause*∗ *terminates-clause*∗ *decreases-clause*? *simple-clause*∗ *comm-clause*∗ *named-behavior*∗ *completeness-clause*∗ *collective-contract*∗
simple-clause	::=	*assigns-clause* | *ensures-clause* | *allocation-clause* | *abrupt-clause*
named-behavior	::=	`behavior` *id* : *assumes-clause*∗ *requires-clause*∗ *simple-clause*∗ *comm-clause*∗
comm-clause	::=	`mpi uses` *term* (, *term*)∗ ;
collective-contract	::=	`mpi collective(`*term*`)`: *requires-clause*∗ *simple-clause*∗ *waitsfor-clause*∗ *mpi-named-behavior*∗ *completeness-clause*∗
mpi-named-behavior	::=	`behavior` *id* : *assumes-clause*∗ *requires-clause*∗ *simple-clause*∗ *waitsfor-clause*∗

Figure 4: Grammar for ACSL function contracts, extended for MPI.

Operations on datatypes. Two logic functions and one predicate are defined:

```
int \mpi_extent(MPI_Datatype datatype);
mpi_sig_t \mpi_sig(MPI_Datatype datatype);
\mpi_nonoverlapping(MPI_Datatype datatype);
```

The first returns the extent (in bytes) of a datatype. The second returns the type signature of the datatype. The predicate holds iff the type map of the datatype is nonoverlapping, a requirement for any communication buffer that receives data.

Value sequences. The domain of type `mpi_seq_t` consists of all finite sequences of pairs (t, v), where t is a basic C type and v is a value of type t. Such a sequence represents the values stored in a communication buffer or message. Similar to the case with type signatures, we define multiplication of an integer with a value of type `mpi_seq_t` to be repeated concatenation.

Communication buffers. The type `mpi_buf_t` is a structure with fields `base` (of type `void*`), `count` (`int`), and `datatype` (`MPI_Datatype`). A value of this type specifies an MPI communication buffer and is created with the logic function

```
mpi_buf_t
\mpi_buf(void * base, int count, MPI_Datatype datatype);
```

The ACSL predicate `\valid` is extended to accept arguments of type `mpi_buf_t` and indicates that the entire extent of the buffer is allocated memory; `\valid_read` is extended similarly.

Buffer arithmetic. An integer and a buffer can be added or multiplied. Both operations are commutative. These are defined by

```
n * \mpi_buf(p, m, dt) == \mpi_buf(p, n * m, dt)
n + \mpi_buf(p, m, dt) ==
\mpi_buf((char*)p + n * \mpi_extent(dt), m, dt)
```

Multiplication corresponds to multiplying the size of a buffer by n. It is meaningful only when both n and m are nonnegative. Addition corresponds to shifting a buffer by n units, where a unit is the extent of the datatype `dt`. It is meaningful for any integer n.

Buffer dereferencing. The dereference operator `*` may take an `mpi_buf_t` b as an argument. The result is the value sequence (of type `mpi_seq_t`) obtained by reading the sequence of values from the buffer specified by b.

The term $*b$ used in an `assigns` clause specifies that any of the memory locations associated to b may be modified; these are the bytes in the range $p + m$ to $p + m + \texttt{sizeof}(t) - 1$, for some entry (t, m) in the type map of b.

The ACSL predicate `\separated` takes a comma-separated list of expressions, each of which denotes a set of memory locations. It holds if those sets are pairwise disjoint. We extend the syntax to allow expressions of type `mpi_buf_t` in the list; these expressions represent sets of memory locations as above.

Terms. The grammar for ACSL *terms* is extended:

term ::= `\mpi_comm_rank` | `\mpi_comm_size` | `\mpi_on(`*term*`,` *term*`)`

The term `\mpi_comm_size` is a constant, the number of processes in the communicator; `\mpi_comm_rank` is the rank of "this" process. In the term `\mpi_on(`t`,`r`)`, r must have integer type and is the rank of a process in the communicator. Term t is evaluated in the state of the process of rank r. For convenience, we define a macro `\mpi_agree(x)` which expands to `x==\mpi_on(x,0)`. This is used to say the value of `x` is the same on all processes.

Reduction. A predicate for reductions is defined:

```
\mpi_reduce(mpi_seq_t out, integer lo, integer hi,
            MPI_Op op, (integer)->mpi_seq_t in);
```

The predicate holds iff the value sequence `out` on this process is a point-wise reduction, using operator `op`, of the `hi` − `lo` value sequences `in(lo)`, `in(lo + 1)`, ..., `in(hi − 1)`. Note `in` is a function from `integer` to `mpi_seq_t`. We say *a* reduction, and not *the* reduction, because `op` may not be strictly commutative and associative (e.g., floating-point addition).

4 EVALUATION

4.1 Collective Contract Examples

The first part of our evaluation involved writing contracts for a variety of collective functions. We started with the 17 MPI blocking collective functions specified in [58, Chapter 5]. These represent the most commonly used message-passing patterns, such as broadcast, scatter, gather, transpose, and reduce (fold). The MPI Standard is a precisely written natural language document, similar to the C Standard. We scrutinized each sentence in the description of each function and checked that it was reflected accurately in the contract.

Figure 5 shows the contract for the MPI collective function `MPI_Allreduce`. This function "combines the elements provided in the input buffer of each process. . . using the operator `op`" and "the result is returned to all processes" [58]. This guarantee is reflected in lines 13–14. "The 'in place' option . . . is specified by passing the value `MPI_IN_PLACE` to the argument `sendbuf` at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data." This option is

```
1  #define SBUF \mpi_buf(sbuf, count, dt)
2  #define RBUF \mpi_buf(rbuf, count, dt)
3  /*@ mpi uses comm; mpi collective(comm):
4    requires \valid(RBUF) && \mpi_nonoverlapping(dt);
5    requires \mpi_agree(count) && \mpi_agree(dt) && \mpi_agree(op) && count >= 0;
6    requires \separated(RBUF, { SBUF | int i; sbuf != MPI_IN_PLACE });
7    assigns *RBUF;
8    ensures \mpi_agree(*RBUF);
9    waitsfor { i | int i; 0 <= i < \mpi_comm_size };
10   behavior not_in_place:
11     assumes sbuf != MPI_IN_PLACE;
12     requires \mpi_agree(sbuf != MPI_IN_PLACE) && \valid_read(SBUF);
13     ensures \mpi_reduce(*RBUF, 0, \mpi_comm_size, op,
14                         \lambda integer t; \mpi_on(*SBUF, t));
15   behavior in_place:
16     assumes sbuf == MPI_IN_PLACE;
17     requires \mpi_agree(sbuf == MPI_IN_PLACE);
18     ensures \mpi_reduce(*RBUF, 0, \mpi_comm_size, op,
19                         \lambda integer t; \mpi_on(\old(*RBUF), t));
20 disjoint behaviors; complete behaviors; */
21 int MPI_Allreduce(const void *sbuf, void *rbuf, int count,
22                   MPI_Datatype dt, MPI_Op op, MPI_Comm comm);
```

Figure 5: The contract of the `MPI_Allreduce` function.

represented using two behaviors. These are just a few examples of the tight mapping between the natural language and the contract.

The only ambiguity we could not resolve concerned synchronization. The Standard is clear that collective operations may or may not impose barriers. It is less clear on whether certain forms of synchronization are implied by the semantics of the operation. For example, many users assume that a non-root process must wait for the root in a broadcast, or that all-reduce necessarily entails a barrier. But these operations could be implemented with no synchronization when *count* is 0. (Similarly, a process executing all-reduce with *logical and* could return immediately if its contribution is *false*.) This issue has also been discussed in the MPI Forum [19]. Our `MPI_Allreduce` contract declares, on line 9, that barrier synchronization will occur, but other choices could also be encoded.

In addition to the MPI collectives, we have selected various user-defined collectives from the literature and written contracts for them, including:

(1) `exchange`: a "ghost cell exchange" function in a 1d-diffusion solver [70]
(2) `diff1dIter`: a function performing one time step in 1d-diffusion [70]
(3) `dotProd`: a parallel dot-product procedure from the Hypre project [30]
(4) `matmat`: matrix multiplication using a block-striped decomposition [64]
(5) `oddEvenIter`: an odd-even parallel sorting algorithm [44, 54].

We also implemented `cyc` of Figure 2 in MPI and our contract language.

Figure 6 shows the contract and implementation for `dotProd`. The `hypre_MPI*` routines are simple wrappers for the corresponding MPI functions. The input vectors are block distributed. Each process gets its blocks and computes their inner product. The results are summed across processes with an all-reduce. The contract uses the ACSL `\sum` function to express the local result on a process (line 4) as well as the global result (line 15). Thus the contract is only valid if a real number model of arithmetic is used. This is a

```
1  #define hypre_ParVectorComm(vector) ((vector) -> comm)
2  #define PAR_SIZE x->local_vector->size * x->local_vector->num_vectors
3  #define LOCAL_RESULT \
4    \sum(0, PAR_SIZE-1, \lambda int t; x->local_vector->data[t] *
5                                       y->local_vector->data[t])
6  /*@
7  requires \valid_read(x) && \valid_read(x->local_vector);
8  requires \valid_read(y) && \valid_read(y->local_vector);
9  requires \valid_read(x->local_vector->data + (0..PAR_SIZE-1));
10 requires \valid_read(y->local_vector->data + (0..PAR_SIZE-1));
11 requires x->local_vector->size > 0 && x->local_vector->num_vectors > 0;
12 mpi uses hypre_ParVectorComm(x);
13 mpi collective(hypre_ParVectorComm(x)):
14   assigns \nothing;
15   ensures \result == \sum(0, \mpi_comm_size-1,
16                      \lambda integer k; \mpi_on(LOCAL_RESULT, k));
17   waitsfor {i | int i; 0 <= i < \mpi_comm_size};
18 */
19 HYPRE_Real hypre_ParVectorInnerProd(hypre_ParVector *x, hypre_ParVector *y) {
20   MPI_Comm comm = hypre_ParVectorComm(x);
21   hypre_Vector *my_x = hypre_ParVectorLocalVector(x);
22   hypre_Vector *my_y = hypre_ParVectorLocalVector(y);
23   HYPRE_Real result = 0.0;
24   HYPRE_Real local_result = hypre_SeqVectorInnerProd(my_x,my_y);
25   hypre_MPI_Allreduce(&local_result, &result, 1, hypre_MPI_REAL,
26                       hypre_MPI_SUM, comm);
27   return result;
28 }
```

Figure 6: The parallel `dotProd` function from Hypre [30], with contract.

convenient and commonly-used assumption when specifying numerical code. We could instead use our predicate `\mpi_reduce` for a contract that holds in the floating-point model.

4.2 Bounded verification of collective contracts

For the second part of our evaluation, we developed a prototype tool for verifying that C/MPI collective procedures conform to their contracts. We used CIVL, a symbolic execution and model checking framework [68] written in Java, because it provides a flexible intermediate verification language and it already has strong support for concurrency and MPI [56]. We created a branch of CIVL and modified the Java code in several ways, which we summarize here.

We modified the front-end to accept contracts in our extended version of ACSL. This required expanding the grammar, adding new kinds of AST nodes, and updating the analysis passes. Our prototype can therefore parse and perform basic semantic checks on contracts.

We then added several new primitives to the intermediate language to support the formal concepts described in Section 2. For example, in order to evaluate pre- and postconditions using `\mpi_on` expressions, we added a type for *collective state*, with operations to take a "snapshot" of a process state and merge snapshots into a program state, in order to check collective conditions.

Finally, we implemented a *transformer*, which consumes a C/MPI program annotated with contracts and the name of the function f to be verified. It generates a program similar to $\overline{P^f}$ (Section 2.4). This program has a driver that initializes the global variables and arguments for f to arbitrary values constrained only by f's precondition, using CIVL's `$assume` statement. The body of a collective function g used by f is replaced by code of the form

```
wait(waitsfor(g)); $assert(precondition);
$havoc(assigns(g));
wait(waitsfor(g)); $assume(postcondition);
```

function	states	prover	time(s)
`g (cyc)`	3,562	6	3
`allreduceDR`	7,390	12	5
`oddEvenIter`	14,216	52	8
`dotProd`	4,690	61	8
`f (cyc)`	7,913	11	15
`bcast`	29,256	55	15
`allreduce`	14,174	37	16
`allgather`	14,606	236	30
`reduce`	118,278	41	46
`gather`	125,900	284	69
`diff1dIter`	4,762	99	92
`scatter`	126,436	306	179
`matmat`	8,345	237	190
`reduceScatterNC`	264,215	210	220
`exchange`	896,869	6,285	478
`reduceScatter`	211,541	377	517

Figure 7: Verification performance for nprocs ≤ 5.

where `wait` is implemented using CIVL primitive `$when`, which blocks until a condition holds. When the CIVL verifier is applied to this program, it explores all simulations of f, verifying they terminate and are free of collective errors. By Thm. 1, the verifier can prove, for a bounded number of processes, f conforms.

Our prototype has several limitations. It assumes no wildcard is used in the program. It does not check *assigns violation* for the verifying function. It assumes all communication uses standard mode blocking point-to-point functions and blocking MPI collective functions. Nevertheless, it can successfully verify a number of examples with nontrivial bounds on the number of processes.

For the experiment, we found implementations for several of the MPI collective functions. Some of these are straightforward; e.g., the implementation of `MPI_Allreduce` consists of calls to `MPI_Reduce` followed by a call to `MPI_Bcast`. Two are more advanced: `allreduceDR` implements `MPI_Allreduce` using a double recursive algorithm; `reduceScatterNC` implements `MPI_Reduce_scatter` using an algorithm optimized for non-commutative reduction operations [13].

We applied our prototype to these collective implementations, using the contracts described in Section 4.1. We also applied it to the 5 user-defined collectives listed there. We were able to verify these contracts for up to 5 processes (no other input was bounded), using a Mac Mini with an M1 chip and 16GB memory. Results are given in Figure 7. For each problem, we give the number of states saved by CIVL, the number of calls to an external theorem prover (CVC4 [9] or Z3 [25]), and the verification time in seconds.

The times range from 3 seconds to 9 minutes. In general, time increases with the number of states and prover calls. Exceptions to this pattern occur when prover queries are very complex and the prover times out—two seconds in our case. For example, `matmat`, whose queries involve integer multiplications and uninterpreted functions, times out often. It is slower than most of the test cases despite a smaller state space.

Comparing `reduceScatter` with `reduceScatterNC`, it is noteworthy that verifying the simple implementation costs more than the advanced version. This is because the simple implementation re-uses verified collective functions. Reasoning about the contracts of those functions may involve expensive prover calls.

For `exchange`, nearly one million states are saved though its implementation involves only two MPI point-to-point calls. This is due to the generality of its contract. A process communicates with its left and right "neighbors" in this function. The contract assumes that the neighbors of a process can be any two processes—as long as each pair of processes agree on whether they are neighbors. Hence there is combinatorial explosion generating the initial states.

For each example, we made erroneous versions and confirmed that CIVL reports a violation or "unknown" result. These are also in the artifact repository.

5 RELATED WORK

The ideas underlying code contracts originate in the work of Floyd on formal semantics [33], the proof system of Hoare [41], the specification system Larch [36], and Meyer's work on Eiffel [59, 60]. Contract systems have been developed for many other languages, including Java [32, 46, 51], Ada [6], C# [8], and C [11, 24].

Verification condition generation (VCG) [7, 27, 32, 52] and symbolic execution [49, 50, 63] are two techniques used to verify that code conforms to a contract. *Extended static checking* is an influential VCG approach for Java [32, 46, 52]. Frama-C's WP plugin [10, 24] is a VCG tool for ACSL-annotated C programs, based on the Why platform [31]. The Kiasan symbolic execution platform [26] has been applied to both JML and Spark contracts [12].

Several contract systems have been developed for shared memory concurrency. In addition to VCC (Section 1), VeriFast is a deductive verifier for multithreaded C and Java programs [45]. Its contract language is based on concurrent separation logic [16]. These systems focus on issues, such as ownership and permission, that differ from those that arise in distributed computing.

For distributed concurrency, type-theoretic approaches based on *session types* [62, 66, 71] are used to describe communication protocols; various techniques verify an implementation conforms to a protocol. ParTypes [53] applies this approach to C/MPI programs using a user-written protocol that specifies the sequence of messages transmitted in an execution. Conformance guarantees deadlock-freedom for an arbitrary number of processes. However, ParTypes protocols cannot specify programs with wildcards or functional correctness, and they serve a different purpose than our contracts. Our goal is to provide a public contract for a collective procedure—the messages transmitted are an implementation detail that should remain "hidden" to the extent possible.

There are a number of correctness tools for MPI programs, including the dynamic model checkers ISP [73] and DAMPI [74], the static analysis tool MPI-Checker [29], and the dynamic analysis tool MUST [40]. These check for certain pre-defined classes of defects, such as deadlocks and incorrectly typed receive statements; they are not used to specify or verify functional correctness.

Ashcroft introduced the idea of verifying parallel programs by showing every atomic action preserves a global invariant [5]. This approach is applied to a simple message-passing program in [55] using Frama-C+WP and ghost variables to represent channels. The contracts are quite complicated; they are also a bespoke solution for a specific problem, rather than a general language. However, the approach applies to non-collective as well as collective procedures.

A parallel program may also be specified by a functionally equivalent sequential version [67]. This works for whole programs which consume input and produce output, but it seems less applicable to individual collective procedures.

Assume-guarantee reasoning [1, 28, 47, 48] is another approach that decomposes along process boundaries. This is orthogonal to our approach, which decomposes along procedure boundaries. It may be possible to combine these strategies, for example, by using collective contracts to specify collective procedures and to verify code that uses collectives, and then switch to assume-guarantee, session-type, or invariant approaches once non-collectives are used.

6 DISCUSSION

We have summarized a theory of contracts for collective procedures in a toy message-passing language. We have shown how this theory can be realized for C programs that use MPI using a prototype contract-checking tool. The approach is applicable to programs that use standard-mode blocking point-to-point operations, blocking MPI collective functions, multiple communicators, user-defined datatypes, pointers, pointer arithmetic, and dynamically allocated memory. We have used it to fully specify all of the MPI blocking collective functions, and several nontrivial user-defined collective functions.

MPI's nonblocking operations are probably the most important and widely-used feature of MPI not addressed here. In fact, there is no problem specifying a collective procedure that uses nonblocking operations, as long as the procedure completes all of those operations before returning. For such procedures, the nonblocking operations are another implementation detail that need not be mentioned in the public interface. However, some programs may use one procedure to post nonblocking operations, and another procedure to complete them; this is in fact the approach taken by the new MPI "nonblocking collective" functions [58, Sec. 5.12]. The new "neighborhood collectives" [58, Sec. 7.6] may also require new abstractions and contract primitives.

Our theory assumes no use of `MPI_ANY_SOURCE` "wildcard" receives. It is easy to construct counterexamples to Theorem 1 for programs that use wildcards. New conceptual elements will be required to ensure a collective procedure implemented with wildcards will always behave as expected.

Our prototype tool for verifying conformance to a contract uses symbolic execution and bounded model checking techniques. It demonstrates the feasibility of this approach, but can only "verify" with small bounds placed on the number of processes. It would be interesting to see if the verification condition generation (VCG) approach can be applied to our contracts, so that they could be verified without such bounds. This would require a kind of Hoare calculus for message-passing parallel programs, and/or a method for specifying and verifying a global invariant.

One could also ask for runtime verification of collective contracts. This is an interesting problem, as the assertions relate the state of multiple processes, so checking them would require communication.

REFERENCES

[1] Martín Abadi and Leslie Lamport. 1995. Conjoining specifications. *ACM Trans. Program. Lang. Syst.* 17, 3 (May 1995), 507–535. https://doi.org/10.1145/203095.201069

[2] Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. 2018. *Model Checking Procedural Programs.* Springer, Cham, Chapter 17, 541–572. https://doi.org/10.1007/978-3-319-10575-8_17

[3] Anonymous. 2023. *Collective Contracts for MPI: Extended Report, Artifact, and Source Code.* Technical Report 2023-0001. Anonymous Institution. https://github.com/Anonymous4Anonymous/MPI-Contract

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams. 1996. *Concurrent Programming in Erlang* (second ed.). Prentice Hall Europe, Herfordshire, UK.

[5] E. A. Ashcroft. 1975. Proving assertions about parallel programs. *J. Comput. Syst. Sci.* 10, 1 (Feb. 1975), 110–135. https://doi.org/10.1016/S0022-0000(75)80018-3

[6] John Barnes. 2003. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, Boston, MA, USA.

[7] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, Springer, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

[8] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and verification: the Spec# experience. *Commun. ACM* 54, 6 (2011), 81–91. https://doi.org/10.1145/1953122.1953145

[9] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

[10] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. 2020. WP Plug-in Manual: Frama-C 22.0 (Titanium). https://frama-c.com/download/frama-c-wp-manual.pdf

[11] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2012. ACSL: ANSI/ISO C Specification Language, Version 1.16. http://frama-c.com/download/acsl-1.16.pdf.

[12] Jason Belt, John Hatcliff, Robby, Patrice Chalin, David Hardin, and Xianghua Deng. 2011. Bakar Kiasan: Flexible Contract Checking for Critical Systems Using Symbolic Execution, See [15], 58–72. https://doi.org/10.1007/978-3-642-20398-5_6

[13] Massimo Bernaschi, Giulio Iannello, and Mario Lauria. 2002. Efficient Implementation of Reduce-scatter in MPI. In *Proceedings of the 10th Euromicro Conference on Parallel, Distributed and Network-based Processing* (Canary Islands, Spain) *(EUROMICRO-PDP'02).* IEEE Computer Society, Washington, DC, USA, 301–308. http://dl.acm.org/citation.cfm?id=1895489.1895529

[14] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas J. Naughton, III, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. 2018. *A Survey of MPI Usage in the U.S. Exascale Computing Project.* Technical Report 790. Oak Ridge National Lab. https://doi.org/10.2172/1462877

[15] Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). 2011. *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings.* LNCS, Vol. 6617. Springer. https://doi.org/10.1007/978-3-642-20398-5

[16] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science* 375, 1 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034 Festschrift for John C. Reynolds's 70th birthday.

[17] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). LNCS, Vol. 5674. Springer, Berlin, Heidelberg, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[18] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 2010. Local Verification of Global Invariants in Concurrent Programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (LNCS, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer, Berlin, Heidelberg, 480–494. https://doi.org/10.1007/978-3-642-14295-6_42

[19] MPI Community. 2020. Collective Synchronization. https://github.com/mpi-forum/mpi-issues/issues/257. Accessed Aug. 13, 2021.

[20] Correctness 17 2017. *Correctness'17: Proceedings of the First International Workshop on Software Correctness for HPC Applications* (Denver, CO, USA). ACM, New York, NY, USA. https://dl.acm.org/doi/proceedings/10.1145/3145344

[21] Correctness 18 2018. *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness).* IEEE Computer Society, Los Alamitos,

CA, USA. https://doi.org/10.1109/Correctness.2018.00004

[22] Correctness 19 2019. *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/Correctness49594.2019.00001

[23] Correctness 20 2020. *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/Correctness51934.2020.00001

[24] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C — A Software Analysis Perspective. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings (LNCS, Vol. 7504)*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer-Verlag, Berlin, Heidelberg, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[25] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[26] Xianghua Deng, Jooyong Lee, and Robby. 2006. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18–22 September 2006, Tokyo, Japan*. IEEE Computer Society, USA, 157–166. https://doi.org/10.1109/ASE.2006.26

[27] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. 1998. *Extended Static Checking*. Research Report 159. COMPAQ Systems Research Center. https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-159.pdf

[28] Juergen Dingel. 2003. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 138–148. https://doi.org/10.1109/ICSE.2003.1201195

[29] Alexander Droste, Michael Kuhn, and Thomas Ludwig. 2015. MPI-Checker: Static Analysis for MPI. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. ACM, New York, Article 3, 10 pages. https://doi.org/10.1145/2833157.2833159

[30] Robert D. Falgout and Ulrike Meier Yang. 2002. *hypre*: A Library of High Performance Preconditioners. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–641. https://doi.org/10.1007/3-540-47789-6_66.

[31] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where Programs Meet Provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP'13)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer-Verlag, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

[32] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/512529.512558

[33] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science* 19 (1967), 19–32.

[34] Ganesh Gopalakrishnan, Paul D. Hovland, Costin Iancu, Sriram Krishnamoorthy, Ignacio Laguna, Richard A. Lethin, Koushik Sen, Stephen F. Siegel, and Armando Solar-Lezama. 2017. *Report of the HPC Correctness Summit, January 25–26, 2017, Washington, DC*. Department of Energy, Washington, DC. https://doi.org/10.2172/1470989

[35] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. 2011. Formal analysis of MPI-based parallel programs. *Commun. ACM* 54, 12 (Dec. 2011), 82–91. https://doi.org/10.1145/2043174.2043194

[36] John V. Guttag, James J. Horning, and Jeannette M. Wing. 1985. The Larch Family of Specification Languages. *IEEE Software* 2, 5 (1985), 24–36. https://doi.org/10.1109/MS.1985.231756

[37] Les Hatton. 1997. The T Experiments: Errors in Scientific Software. *IEEE Computational Science & Engineering* 4, 2 (April 1997), 27–38. https://doi.org/10.1109/99.609829

[38] Les Hatton. 2012. Defects, Scientific Computation and the Scientific Method. In *Uncertainty Quantification in Scientific Computing*, Andrew M. Dienstfrey and Ronald F. Boisvert (Eds.). IFIP Advances in Information and Communication Technology, Vol. 377. Springer, Berlin, Heidelberg, 123–138. https://doi.org/10.1007/978-3-642-32677-6_8

[39] Les Hatton and Andy Roberts. 1994. How Accurate is Scientific Software? *IEEE Transactions on Software Engineering* 20, 10 (Oct. 1994), 785–797. https://doi.org/10.1109/32.328993

[40] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *International Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11–15, 2012*, Jeffrey K. Hollingsworth (Ed.). IEEE Computer Society Press, Los Alamitos, CA, USA, Article 30, 11 pages. https://doi.org/10.1109/SC.2012.79

[41] C. A.R̃. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[42] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. https://doi.org/10.1145/359576.359585

[43] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister. 2009. Software Standards for the Multicore Era. *Micro, IEEE* 29, 3 (May-June 2009), 40–51. https://doi.org/10.1109/MM.2009.48

[44] Marieke Huisman, Rosemary Monahan, Peter Müller, Wojciech Mostowski, and Mattias Ulbrich. 2017. *VerifyThis 2017: A Program Verification Competition*. Technical Report Karlsruhe Reports in Informatics 2017,10. Karlsruhe Institute of Technology, Faculty of Informatics. https://doi.org/10.5445/IR/1000077160

[45] Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-Grained Concurrency Specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 271–282. https://doi.org/10.1145/1926385.1926417

[46] Perry R. James and Patrice Chalin. 2010. Faster and More Complete Extended Static Checking for the Java Modeling Language. *Journal of Automated Reasoning* 44 (2010), 145–174. Issue 1-2. https://doi.org/10.1007/s10817-009-9134-9

[47] Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19–23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, Newcastle University, 321–332.

[48] C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619. https://doi.org/10.1145/69575.69577

[49] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7–11, 2003, Proceedings (LNCS, Vol. 2619)*, Hubert Garavel and John Hatcliff (Eds.). Springer-Verlag, Berlin, Heidelberg, 553–568. https://doi.org/10.1007/3-540-36577-X_40

[50] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[51] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (2006), 1–38. https://doi.org/10.1145/1127878.1127884

[52] K. Rustan M. Leino. 2001. Extended Static Checking: A Ten-Year Perspective. In *Informatics - 10 Years Back. 10 Years Ahead (LNCS, Vol. 2000)*, Reinhard Wilhelm (Ed.). Springer, Berlin, Heidelberg, 157–175. https://doi.org/10.1007/3-540-44577-3_11

[53] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-based verification of message-passing parallel programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25–30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, New York, NY, USA, 280–298. https://doi.org/10.1145/2814270.2814302

[54] Ziqing Luo and Stephen F. Siegel. 2018. Symbolic Execution and Deductive Verification Approaches to VerifyThis 2017 Challenges. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), Proceedings, Part II: Verification (LNCS, Vol. 11245)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, International Publishing, 160–178.

[55] Ziqing Luo and Stephen F. Siegel. 2018. Towards Deductive Verification of Message-Passing Parallel Programs. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, Ignacio Laguna and Cindy Rubio-González (Eds.). IEEE, USA, 59–68. https://doi.org/10.1109/Correctness.2018.00012

[56] Ziqing Luo, Manchun Zheng, and Stephen F. Siegel. 2017. Verification of MPI Programs Using CIVL. In *Proceedings of the 24th European MPI Users' Group Meeting* (Chicago, Illinois) *(EuroMPI '17)*. ACM, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/3127024.3127032

[57] Zeeya Merali. 2010. Computational Science: ...Error: Why scientific programming does not compute. *Nature* 467, 7317 (Oct. 2010), 775–777. https://doi.org/10.1038/467775a

[58] Message-Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard, Version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[59] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[60] Bertrand Meyer, Jean-Marc Nerson, and Masanobu Matsuo. 1987. EIFFEL: Object-Oriented Design for Software Engineering. In *ESEC '87, 1st European Software Engineering Conference, Strasbourg, France, September 9-11, 1987, Proceedings (LNCS, Vol. 289)*, Howard K. Nichols and Dan Simpson (Eds.). Springer, Berlin,

Heidelberg, 221–229. https://doi.org/10.1007/BFb0022115
[61] Michal Moskal. 2011. Verifying Functional Correctness of C Programs with VCC, See [15], 56–57. https://doi.org/10.1007/978-3-642-20398-5_5
[62] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. 2012. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *Objects, Models, Components, Patterns (LNCS, Vol. 7304)*, Carlo A. Furia and Sebastian Nanz (Eds.). Springer, Berlin, Heidelberg, 202–218. https://doi.org/10.1007/978-3-642-30561-0_15
[63] Corina Păsăreanu and Willem Visser. 2009. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Techol. Transf.* 11, 4 (Oct. 2009), 339–353. https://doi.org/10.1007/s10009-009-0118-1
[64] Michael Quinn. 2004. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, .
[65] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. 2015. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy* 82 (2015), 90–97. https://doi.org/10.1016/j.anucene.2014.07.048
[66] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-Passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 502–516. https://doi.org/10.1145/3314221.3322484
[67] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. 2008. Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. *ACM Transactions on Software Engineering and Methodology* 17, 2 (2008), Article 10, 1–34. https://doi.org/10.1145/1348250.1348256
[68] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. ACM, New York, Article 61, 12 pages. http://doi.acm.org/10.1145/2807591.2807635
[69] Stephen F. Siegel and Timothy K. Zirkel. 2011. Collective Assertions. In *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, January 23–25, 2011, Proceedings (LNCS, Vol. 6538)*, Ranjit Jhala and David Schmidt (Eds.). Springer, Berlin, Heidelberg, 387–402. https://doi.org/10.1007/978-3-642-18275-4_27
[70] Stephen F. Siegel and Timothy K. Zirkel. 2011. FEVS: A Functional Equivalence Verification Suite for High Performance Scientific Computing. *Mathematics in Computer Science* 5, 4 (2011), 427–435.
[71] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *PARLE'94 Parallel Architectures and Languages Europe (LNCS, Vol. 817)*, Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer, Berlin, Heidelberg, 398–413. https://doi.org/10.1007/3-540-58184-7_118
[72] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 865–878. https://doi.org/10.1145/3297858.3304069
[73] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7–14, 2008, Proceedings (LNCS, Vol. 5123)*, Aarti Gupta and Sharad Malik (Eds.). Springer, Berlin, Heidelberg, 66–79. https://doi.org/10.1007/978-3-540-70545-1_9
[74] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2010. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/SC.2010.7
[75] Ulrike Yang, Robert Falgout, and Jongsoo Park. 2017. Algebraic Multigrid Benchmark, Version 00. https://www.osti.gov//servlets/purl/1389816

A PROOF OF THEOREM 1

A.1 Preliminaries

We first establish some notation and vocabulary that will be used throughout this appendix.

Sequences. For a finite or infinite sequence η, write $|\eta|$ for the length of η. The elements of η are numbered from 1. For sequences η and ξ, $\eta \le \xi$ denotes that η is a prefix of, or equal to, ξ.

Associated transitions. Let P be a program and $\zeta = (s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots)$ an execution of a procedure in P. Suppose ζ is collective consistent. Choose $p \in \mathsf{PID}$ such that $T_q(\zeta) \le T_p(\zeta)$ for all $q \in \mathsf{PID}$. The *number of collective events in* ζ, denoted $\mathsf{ncol}(\zeta)$, is $|T_p(\zeta)|$. If $1 \le k \le \mathsf{ncol}(\zeta)$, the *$k$-th collective event of* ζ is the k-th symbol of $T_p(\zeta)$. By collective consistency, these definitions are independent of the choice of p.

For any integer $k \ge 1$, let

$$Q(\zeta, k) = \{p \in \mathsf{PID} \mid |T_p(\zeta)| \ge k\}.$$

A process p is in $Q(\zeta, k)$ if p performs at least k collective actions in ζ.

Let $k \in 1..\mathsf{ncol}(\zeta)$. Then $Q(\zeta, k) \ne \emptyset$. For each $p \in Q(\zeta, k)$, there is a unique $i \in 1..\mathsf{len}(\zeta)$ such that t_i is the k-th collective action of p in ζ. We say the transition $s_{i-1} \xrightarrow{t_i} s_i$ is *associated to the k-th collective event of* ζ.

Correctness. The following concepts are often grouped together so it will be convenient to give them a name:

Definition A.1. An execution of a procedure is *correct* if it is finite, does not deadlock, and has no collective errors.

A.2 Commuting Actions

The key to the proof is the ability to transpose commuting adjacent actions from different processes. We say transitions t_1 and t_2 are *adjoining* if the destination state of t_1 is the source state of t_2. The basic fact is captured by the following:

LEMMA A.2. *Let P be a program. Suppose $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2$ are adjoining transitions in P. Assume t_1 and t_2 are in different processes and it is not the case that t_1 is a send of a message received by t_2. Then there exists a state s_1' of P such that $s_0 \xrightarrow{t_2} s_1' \xrightarrow{t_1} s_2$ are adjoining transitions.*

PROOF. An assignment, call, or return updates the state of the invoking process only. A send or receive updates the states of the invoking process (the program counter) and a single message channel. An enabled receive in one process can never be disabled by an action from another process. Furthermore, the variables in a guard are local to one process (as there are no shared variables), so the truth value of a guard is not impacted by an action from another process. Therefore an assignment, call, return, or guard in one process commutes with any action from another process.

A send from one process commutes with a send from another process as they access separate channels. Likewise, a send from one process p commutes with a receive from another process q if the destination of the send statement is not q or the source in the receive statement is not p.

So assume one of the statements is a send in process p to process q, and the other is a receive in process q from p. There are two cases:

(1) There is at least one buffered message in the $p \to q$ channel in state s_0. In this case, the send enqueues a new message onto the queue and the receive dequeues the oldest message in the queue, and it is clear these two actions commute.

(2) The $p \to q$ channel is empty in s_0. In this case, t_1 must be the send, and t_2 therefore receives the message enqueued by t_1, contradicting the hypothesis of this lemma.

□

Definition A.3. Let P be a program and f a procedure in P. Let $\sim$ be the smallest equivalence relation on the set of executions of f such that $\zeta \sim \eta$ if η is obtained from ζ by transposing two adjacent transitions satisfying the hypotheses of Lemma A.2. We say ζ is *equivalent* to η if $\zeta \sim \eta$.

Hence, if $\zeta \sim \eta$ then η can be obtained from ζ by performing a finite sequence of transpositions of commuting adjacent transitions.

Lemma A.4. *Let P be a program and f a procedure in P. Suppose ζ and η are executions of f and $\zeta \sim \eta$. Then all of the following hold:*

1. *ζ commits a consistency, precondition, postcondition, assigns, segment boundary, or unreceived message violation iff η commits a violation of one of those kinds.*
2. *ζ deadlocks iff η deadlocks.*
3. *ζ is finite iff η is finite.*

In the remainder of this section we sketch the proof of Lemma A.4. For the most part, the proof is straightforward, but tedious, since each kind of error must be considered in turn.

Consistency. The transitions involved in each transposition come from different processes. Therefore $T_p(\zeta) = T_p(\eta)$ for all $p \in \mathsf{PID}$. It follows that ζ is collective consistent iff η is collective consistent.

Hence if ζ has a consistency violation, we are done. So assume ζ (and therefore η) is collective consistent.

Precondition. Suppose $1 \le r \le \mathrm{ncol}(\zeta)$ and the r-th collective event of ζ is e^g where $g \in C$. For $q \in Q(\zeta, r)$, let $\sigma_{r,q}$ be the process state of process q just after executing the r-th collective action in q. Define $\alpha' \colon Q \to (V_f \to \mathbb{Z})$ by setting $\alpha'(q)(v)$ to be the value of v in $\sigma_{r,q}$. Hence α' is the partial g-valuation used to determine whether a precondition violation occurs when the last process in $Q(\zeta, r)$ to execute its r-th collective transition executes that transition.

We claim that a precondition violation involving the r-th collective event occurs in ζ iff there is no satisfying extension of α', i.e., an extension to a g-valuation α such that $[\![\mathrm{pre}(g)]\!]_{\alpha,p}$ holds for all $p \in \mathsf{PID}$. Indeed, if there is no such extension, then a precondition violation occurred with the last process in Q to execute its associated transition. Conversely, if a precondition violation occurs at some (possibly earlier) associated transition, then the set of processes used to determine a violation is some subset $Q' \subseteq Q$, the partial valuation used is $\alpha'|_{Q'}$, and there is no satisfying extension of $\alpha'|_{Q'}$. Since any extension of α' is an extension of $\alpha'|_{Q'}$, it follows there is no satisfying extension of α', proving the claim.

This means the question of whether a precondition violation for collective event r occurs in ζ depends only on the set of process states

$$\{\sigma_{r,q} \mid q \in Q(\zeta, r)\}.$$

This set is invariant under transposition of adjacent commuting transitions: while such a transposition may change the order in which processes execute their associated enter transitions, they do not change the final set of such processes or their local states at the time they enter. Hence a precondition violation occurs in ζ iff such a violation occurs in η.

Postcondition. The argument is the same as that for preconditions, but uses the final sets of process states in the prestate and poststate for the call.

Assigns. For $p \in \mathsf{PID}$ we may project the execution ζ onto p to obtain a "local execution" of process p, i.e., a sequence of process states and actions in p. This projection is invariant under transposition of commuting transitions. Since an assigns violation in process p depends only upon the sequence of process states and actions from p, ζ has an assigns violation iff η has an assigns violation.

Segment boundary. The first way a segment boundary violation could occur in ζ is that process p receives a message in segment a that was sent by process q in segment b, with $b > a$. A transposition does not change the segment number in which the send or receive occurs, and since a transposition will not be applied if the first transition is a send and the second a receive of the message sent, this kind of violation will be preserved by a transposition.

The second kind of segment boundary violation occurs when process p in segment a sends to q, when q is in segment b and $b > a$. The only way this can be affected by transposition is if the first transition is a collective action of q and the second transition is the send by p. After transposing, the send takes place when p is at segment a and q is at segment $b - 1$. If this is not also a segment boundary violation, then $a = b - 1$, i.e., the send takes place when p and q are at the same segment. But then the collective action of q results in an unreceived message violation.

Unreceived message. Suppose there is an unreceived message for p sent in segment $r-1$ when p executes its r-th collective action. The only way a transposition could change this condition is if the first transition is the send that enqueues the message, and the second transition is the collective action by p. After transposing, there is no unreceived message when p executes its r-th collective action. However, the send now commits a segment boundary violation: the send to p is executed when p has a higher segment number than that of the sending process.

Deadlock and finiteness. Deadlock is clearly preserved since ζ and η have the same final state. Transposition does not change the length of an execution, so in particular ζ is finite iff η is finite.

This completes the proof sketch for Lemma A.4.

The issue with waitsfor*.* The one kind of error not necessarily preserved by equivalence is a *waitsfor* violation. If all of the following hold:

- the first transition is an e^g in process p from state s_{i-1},
- the second transition is an x^g in process q from state s_i,
- $\mathrm{seg}_p(\zeta, i-1) = \mathrm{seg}_q(\zeta, i)$, and
- p is in the wait set for q,

then transposing results in a *waitsfor* violation where none existed before. However, if any of those conditions does not hold, the transposition cannot introduce a *waitsfor* violation.

A.3 Observable Correctness

Let P be a program with no collective recursion. Let C be the set of collective procedure names from P. Let $f \in C$ and assume all collective procedures used by f conform. Let $\tilde{C} \subseteq C$ consist of f and all collective procedures used by f. We call the elements of $\tilde{C}$ the *observable collective procedures.*

Let $\tilde{P}^f$ be the program that is the same as P^f except contracts are removed from all procedures in $C \setminus \tilde{C}$, so the collective procedures for $\tilde{P}^f$ are precisely the observable collective procedures. An *observable collective action* is a collective action involving $\tilde{C}$, i.e., an e^g or x^g action for $g \in \tilde{C}$. Note that any execution ζ of P^f is also an execution of $\tilde{P}^f$, but the notions of *segment* and *collective error* (and all the specific kinds of collective errors) depend on whether one considers ζ to be an execution of P^f or of $\tilde{P}^f$.

Definition A.5. Let ζ be an execution of P^f. Let $\tilde{\zeta}$ be the same execution, considered as an execution of $\tilde{P}^f$.

- We say ζ commits an *observable collective error* at step i if $\tilde{\zeta}$ commits a collective error at step i.
- If ζ commits a collective error at step i but $\tilde{\zeta}$ does not, then ζ commits an *internal collective error* at step i.
- We say ζ is *observably correct* if $\tilde{\zeta}$ is correct.

Let $\zeta = (s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots)$ be an execution of P^f.

It is not hard to see that if ζ is free of collective errors then so is $\tilde{\zeta}$. Intuitively, the segment decomposition of ζ refines that of $\tilde{\zeta}$, so, for example, a message that crosses a segment boundary in $\tilde{\zeta}$ also crosses a segment boundary in ζ. A pre- or postcondition, assigns, *waitsfor*, or unreceived message violation involving some $g \in \tilde{C}$ is also a violation in ζ, as $g \in C$.

Clearly, ζ is finite if and only if $\tilde{\zeta}$ is finite, and ζ deadlocks if and only if $\tilde{\zeta}$ deadlocks. Hence if ζ is correct, then ζ is observably correct.

At any state in ζ, the number of observable collective procedures on the call stack of a process is at most 2. In fact, for any $p \in \mathsf{PID}$, if $T_p(\tilde{\zeta})$ is finite then it is a prefix of a sequence of the form

$$\mathsf{e}^f \mathsf{e}^{g_1} \mathsf{x}^{g_1} \ldots \mathsf{e}^{g_m} \mathsf{x}^{g_m} \mathsf{x}^f,$$

where $g_1, \ldots, g_m$ are (not necessarily distinct) collective procedures used by f. If $T_p(\tilde{\zeta})$ is infinite, it has the form

$$\mathsf{e}^f \mathsf{e}^{g_1} \mathsf{x}^{g_1} \mathsf{e}^{g_2} \mathsf{x}^{g_2} \cdots .$$

If s is a state occurring on ζ, we say p is *at an internal state at s* if there are exactly two observable collective procedures on the call stack for p in s, otherwise p is *at an external state at s*.

Let $i \in 1..\mathrm{len}(\zeta)$, so $s_{i-1} \xrightarrow{t_i} s_i$ is a transition in ζ. Let p be the process of t_i. Suppose $r \geq 1$. We say t_i is *in region r* of ζ if $\mathrm{seg}_p(\tilde{\zeta}, i-1) = 2r$ or $\mathrm{seg}_p(\tilde{\zeta}, i) = 2r$. In other words, all transitions in p from e^{g_r} to x^{g_r}, inclusive, belong to region r. We say t_i is *strictly in region r* if $\mathrm{seg}_p(\tilde{\zeta}, i-1) = 2r$ and $\mathrm{seg}_p(\tilde{\zeta}, i) = 2r$, i.e., t_i is in region r but is not the initial e^{g_r} action nor the final x^{g_r} action (if the final action occurs in ζ). We say t_i is *post-region r* if $\mathrm{seg}_p(\tilde{\zeta}, i-1) > 2r$. We say t_i is *pre-region r* if $\mathrm{seg}_p(\tilde{\zeta}, i) < 2r$.

A.4 Extraction of executions

Fix a collective-recursion-free program P. Let f be a collective procedure of P, and ζ a finite execution of P^f with no observable collective error.

Suppose $r \geq 1$ and there is some action in ζ in region r. Let $g = g_r$. We now describe how one can extract an execution of P^g from region r of ζ. It will be shown that this extracted execution starts from a *pre*(g)-state.

First, suppose

$$s_{i-1} \xrightarrow{t_i} s_i \xrightarrow{t_{i+1}} s_{i+1}$$

are two consecutive transitions in ζ, t_i and t_{i+1} are in two different processes, t_i is post-region r, and t_{i+1} is in region r. Let p be the process of t_i and q that of t_{i+1}. We have

$$\mathrm{seg}_p(\tilde{\zeta}, i) \geq \mathrm{seg}_p(\tilde{\zeta}, i-1) > 2r \geq \mathrm{seg}_q(\tilde{\zeta}, i+1).$$

It follows that t_i cannot be a send of a message that is received by t_{i+1}, else a segment boundary violation occurs in $\tilde{\zeta}$ at step $i+1$, contradicting the assumption that $\tilde{\zeta}$ is collective error-free. By Lemma A.2, we may transpose t_i and t_{i+1} to yield an execution ξ equivalent to ζ.

We claim that ξ is also free of observable collective errors. By Lemma A.4, we only need to show that $\tilde{\xi}$ (i.e., ξ considered as an execution of $\tilde{P}^f$) has no *waitsfor* violation. According to the comments following the proof of Lemma A.4, such a violation can be introduced only if $\mathrm{seg}_p(\tilde{\zeta}, i-1) = \mathrm{seg}_q(\tilde{\zeta}, i)$. However, in the transformation above, $\mathrm{seg}_p(\tilde{\zeta}, i-1) > \mathrm{seg}_q(\tilde{\zeta}, i+1) \geq \mathrm{seg}_q(\tilde{\zeta}, i)$.

Similarly, if t_i is in region r and t_{i+1} is pre-region r, it can be seen that the two transitions commute, and the resulting execution is free of observable collective errors.

As ζ is finite, by a finite number of transpositions of these two kinds, one can transform ζ to an equivalent execution, free of observable collective errors, which is the concatenation $\eta \circ \theta \circ \phi$, where all transitions in η are pre-region r, all transitions in θ are in region r, and all transitions in ϕ are post-region r. Let s be the final state of η, which is also the initial state of θ. Note that all channels are empty in s: otherwise, an observable unreceived message violation would occur.

The "extracted" execution of P^g is formed by modifying θ. Let $\chi = (s'_0 \xrightarrow{t'_1} s_1 \xrightarrow{t'_2} \cdots)$ be a copy of θ which we will modify in a sequence of steps. First, remove the portion of the call stack below the call to g in every state occurring in χ. After this modification, every process has an empty call stack in the initial state s'_0. Then, insert an entry on the bottom of each stack in every state corresponding to the *main* function in P^g.

Let V_g be the global variables of P^g, which consists of the global variables of P together with a set of fresh variables corresponding to the formal parameters of g. A state of P^g must specify, for each $p \in \mathsf{PID}$, the value of each global variable of P^g on process p, i.e., a g-valuation $\alpha\colon \mathsf{PID} \to (V_g \to \mathbb{Z})$.

Let Q be the set of processes that have at least one transition in region r of ζ; the processes of Q are at a location just before a call to g in s. For $q \in Q$ and formal parameter x, let $\alpha(q)(x)$ be the result of evaluating the corresponding actual argument in the call on process q at s. For v a global of P, let $\alpha(q)(v)$ be the value of v in the process state for q at s.

At this point, we have defined a partial g-valuation $Q \to (V_g \to \mathbb{Z})$, which is the valuation used to check the precondition of g when the last process in Q to enter region r of ζ enters. Since ζ has no observable collective error, no precondition violation occurs at that step, i.e., the partial g-valuation can be extended to a full g-valuation $\alpha\colon \mathsf{PID} \to (V_g \to \mathbb{Z})$ for which $[\![\mathrm{pre}(g)]\!]_{\alpha,p}$ holds for all $p \in \mathsf{PID}$. Use valuation α to specify the value of all global variables in s_0'; note that the resulting s_0' is a *pre*(g)-state.

The processes not in Q never execute in χ, so those process states remain the same at every state in χ. In fact the values assigned to the variables of processes not in Q are irrelevant as they are never read nor modified and therefore have no impact on the extracted execution χ. They were only needed to ensure that the initial state of the extracted execution satisfies the definition of a *pre*(g)-state.

We summarize the discussion above in the following:

Lemma A.6. *Let P be a collective-recursion-free program, f a collective procedure of P, and ζ a finite execution of f with no observable collective error. Let $r \geq 1$ and Q the set of processes that have at least one transition in region r of ζ. Assume $Q \neq \emptyset$. Let g be the observable collective procedure associated to region r of ζ. For $p \in Q$, let $\sigma_p \xrightarrow{\tau_p} \sigma_p'$ be the first transition in ζ from process p in region r. Then there is an execution χ of P^g with initial state s_0' such that*

(1) *s_0' is a pre(g)-state,*
(2) *for $p \in Q$, the value assigned to global variable v of P in process p of s_0' is the value assigned to v by process p in σ_p,*
(3) *for $p \in Q$, the value assigned to formal parameter x of g in process p of s_0' is the value assigned to x by process p in σ_p', and*
(4) *the sequence of actions in χ is the sequence of region r actions in ζ, in the same order.*

We next show that internal errors in ζ are preserved in an extracted execution.

Lemma A.7. *Assume ζ is an observably correct execution of collective procedure f in a collective-recursion-free program P. Let $g_1, g_2, \ldots$ be the sequence of collective procedures called from f. If a transition in region r of ζ commits an internal collective error then the execution of P^{g_r} extracted from region r of ζ is incorrect.*

Proof. Write $\zeta = (s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots)$. Let $g = g_r$. Let χ be the execution of P^g extracted from ζ. By Lemma A.6, χ preserves the order of all region r transitions in ζ.

Suppose ζ commits a consistency violation at step i, and t_i is in region r of ζ. We will see that χ commits a consistency violation. Let p be the process of t_i.

Suppose t_i is the first action of p in region r of ζ, i.e., the e^g action entering the region. The string $T_p(\zeta^i)$ has the following form:

$$T_p(\zeta^i) = \mathsf{e}^f \mathsf{e}^{g_1} \cdots \mathsf{x}^{g_1} \cdots \mathsf{e}^{g_{r-1}} \cdots \mathsf{x}^{g_{r-1}} \mathsf{e}^g.$$

For a consistency violation to occur at step i, there must be some other process q with

$$T_q(\zeta^i) = \mathsf{e}^f \mathsf{e}^{g_1} \cdots \mathsf{x}^{g_1} \cdots \mathsf{e}^{g_{r-1}} \cdots \mathsf{x}^{g_{r-1}} \mathsf{e}^h \cdots,$$

where h is an observable collective procedure used by f and $h \neq g$. But this is an observable consistency violation, contradicting the assumption that ζ is observably correct.

So t_i must occur after the e^g transition in p, and there is some process q, strings μ and ν, and distinct collective action symbols c and c' such that

$$T_p(\zeta) = \mu \mathsf{e}^g \nu c$$
$$T_q(\zeta) = \mu \mathsf{e}^g \nu c' \cdots .$$

Both the c and c' transitions occur in region r of ζ. But then

$$T_p(\chi) = \mathsf{e}^g \nu c$$
$$T_q(\chi) = \mathsf{e}^g \nu c' \cdots ,$$

and χ has a consistency violation, as required.

So assume that ζ is collective consistent. It follows that χ is collective consistent: each $T_p(\chi)$ is the substring of $T_p(\zeta)$ starting with the e^g and ending with the corresponding x^g (if the x^g occurs in $T_p(\zeta)$). Now we consider the other kinds of violations.

If a precondition or postcondition violation occurs at transition i in ζ in region r, then the same violation occurs at the corresponding transition in χ: the process states are exactly the same except for the additional stack entries on the bottom of the stacks in ζ, which have no effect on the valuations used to evaluate pre- or postconditions.

An *assigns* or *waitsfor* violation in ζ in region r depends only on entrance and exit actions made by processes in region r, and the process states immediately preceding or following those actions. Since the order of the region r actions in ζ is exactly the same as the order of the actions in χ, these violations carry over to χ.

The first kind of segment boundary violation in region r of ζ occurs when a process p in segment a of region r receives a message that was sent by a process q in segment b, where $a < b$. If the send occurred post-region r, then the violation is observable, contradicting the assumption that ζ is observably correct. So the send must occur in region r, and cross a segment boundary within region r, and therefore the exact same violation occurs in χ. A similar argument applies to the second kind of segment boundary violation.

If t_i is an exit action by process p when there is an unreceived message for p sent from a previous segment of ζ, then the same happens at the corresponding transition in χ. □

An immediate corollary is:

Lemma A.8. *Let P be a collective-recursion-free program and f a collective procedure of P. Assume all collective procedures used by f conform. Let ζ be an execution of P^f. Then ζ is correct if and only if ζ is observably correct.*

Proof. If ζ is correct then ζ is observably correct by the comments following Definition A.5.

So assume ζ is observably correct but not correct; we will arrive at a contradiction. It follows from Definition A.5 that ζ has an internal collective error. Let r be the region of the transition committing the first internal collective error of ζ, and g the associated collective procedure used by f. Let χ be the execution of P^g extracted from region r of ζ. By Lemma A.6, the initial state of χ is a *pre*(g)-state. By Lemma A.7, χ is incorrect. But this contradicts the assumption that g conforms. □

The following is an application of the notion of extracted execution. We use the concept to show that calling a conforming collective procedure cannot lead to error, assuming the call did not commit a precondition violation.

Lemma A.9. *Suppose P is a collective-recursion-free program, f is a collective procedure of P, and all collective procedures used by f conform. Let $\zeta = (s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots)$ be an execution of P^f. Suppose $1 \leq i < \mathrm{len}(\zeta)$, ζ^i is observably correct, p is the process of t_{i+1}, and p is at an internal state at s_i. Then ζ does not commit a collective error at step $i+1$.*

Proof. Say t_{i+1} is in region r of ζ, inside a call to the collective procedure g used by f. Let χ be the execution of P^g extracted from region r of ζ^i. Let s'_0 be the initial state of χ. By Lemma A.6, s'_0 is a *pre*(g)-state.

Case 1: action $t = t_{i+1}$ is enabled at the final state s'_f of χ. Let χ' be the execution of χ extended by the single action t. Since g conforms and χ' is an execution of P^g from a *pre*(g)-state, χ' is correct.

If t commits a consistency, precondition, postcondition, assigns, *waitsfor*, or unreceived message violation in ζ, the same violation is committed by t in χ', contradicting the fact that χ' is correct. Suppose t commits a segment boundary violation in ζ. If the violation is of the second kind, i.e., p sends a message to a process q in a later segment of ζ, then it is also a segment violation of the second kind in χ': even if, in χ', q has terminated, q is at least one segment after p when t executes. Again, this contradicts the fact that χ' is correct.

If the segment boundary violation is of the first kind then t is a receive by p of a message from q when q is at a higher segment in ζ. As we are assuming t is enabled at s'_f, the send must also occur in χ', and t commits a segment boundary violation in χ'.

Case 2: t is not enabled at s'_f. Then t is a receive statement in p with source q, and the channel $p \longrightarrow q$ is empty in s'_f. Since t is enabled at the final state of ζ^i, the matching send in q must occur in ζ^i, and since that transition does not occur in χ, it must occur in ζ after q exits region r. Hence at s'_f, q has terminated. But this means that no extension of χ' can terminate: p is permanently blocked at the receive with empty channel. Hence any maximal extension of χ' is incorrect: either infinite, or finite ending in deadlock. This contradicts the assumption that g conforms. ◻

A.5 Existence of simulations

Assume P is a collective-recursion-free program with collective procedure f, and ζ is an execution of P^f with no observable collective error. Then one can form a simulation η (i.e., an execution of $\overline{P^f}$) from ζ by essentially deleting all transitions within regions and replacing each with a single *havoc* followed by *wait* statement. We now make this precise.

Say $\zeta = (s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots)$. For each $i \geq 0$, we will define a finite simulation η_i, such that η_i is a prefix of or equal to η_{i+1}. We will let η be the simulation which is the limit of the η_i. We will also establish the following invariant: for all $i \geq 0$, the final state s'_i of η_i is related to s_i as follows: for any $p, q \in \mathsf{PID}$, the state of the $p \rightarrow q$ channel in s'_i is obtained by deleting from the corresponding channel in s_i any messages sent from a process at an internal state. Moreover, if p is at an external state at step i of ζ, the the process of state of p at s_i is identical to the process state of p at s'_i.

For $i = 0$, η_0 is the execution of length 0 starting at the initial state of $\overline{P^f}$ corresponding to s_0.

Assume $i \geq 0$ and η_i has been defined. Let p be the process of t_{i+1}. If p is at an internal state at both s_i and s_{i+1} (i.e., t_{i+1} is strictly within some region), let $\eta_{i+1} = \eta_i$. The invariant on channels is maintained because (i) if t_{i+1} is a send, the message sent is from an internal region, and (ii) if t_{i+1} is a receive then the message received was sent within the region, as ζ is free of collective errors.

If p is at an external state at s_i, η_{i+1} is obtained by appending t_{i+1} and the resulting state to η_i (this includes the case where t_{i+1} enters some region). If p is at an internal state at s_i and at an external state at s_{i+1}, i.e., t_{i+1} exits a region, proceed as follows: let g be the collective procedure being exited. Then η_{i+1} is obtained from η_i by adding the following three transitions to η_i:

(1) A *havoc* transition which assigns to the variables in g's *assigns* list the values they have in s_i. The fact that ζ has no collective error guarantees that these values will not cause a postcondition violation and are therefore allowed in a simulation.
(2) A *wait* transition on g's wait set. The latter must be enabled in η, otherwise ζ would have an observable *waitsfor* violation.
(3) The x^g transition.

After the exit, the process state for the executing process p in s'_{i+1} is identical to the process state of p in s_{i+1}, maintaining the claimed invariant.

Recall that infinite executions are considered erroneous. Here we show that certain infinite executions will be detected by simulations.

Lemma A.10. *Suppose P is a collective-recursion-free program, f is a collective procedure of P, and all collective procedures used by f conform. If ζ is an infinite execution of P^f with no observable collective error then there exists an incorrect simulation of f.*

Proof. Suppose ζ has only a finite number of regions. Since ζ is infinite, there is some region r with an infinite number of transitions from ζ. Let g be the collective procedure associated to region r. Let m be the index of the last transition in ζ to enter region r. For each $i \geq m$, let χ_i be the execution of P^g extracted from region r of ζ^i. By Lemma A.6, each χ_i starts from the same initial state. Moreover the action sequence of χ_{i+1} is either the same sequence as that of χ_i, or extends that sequence by one action. It follows that for each i, χ_i is a (not necessarily strict) prefix of χ_{i+1}, and, for an infinite number of i, χ_i is a strict prefix of χ_{i+1}. Let χ be the limit of the χ_i. Then χ is an infinite execution of P^g from a *pre*(g)-state, contradicting the assumption that g conforms.

So the regions in ζ increase without bound. That means there are an infinite number of transitions from external states, so the simulation formed from ζ is infinite. ◻

The following shows that collective errors and deadlocks are detected by simulations:

Lemma A.11. *Suppose P is a collective-recursion-free program, f is a collective procedure of P, all collective procedures used by f conform, and ζ is an execution of P^f. If ζ has an observable collective error or ends in deadlock then there exists an incorrect simulation of f.*

Proof. Let $i \in 0..\mathrm{len}(\zeta)$ be the maximal integer for which ζ^i has no observable collective error. The final state of ζ^i is s_i. Let η be the simulation generated from ζ^i. Let s' be the final state of η.

Case 1: s_i is not deadlocked. Then $t = t_{i+1}$ commits an observable collective error in ζ. Let p be the process of t_{i+1}. By Lemma A.9, p is at an external state at s_i. Hence t is also an enabled transition in the simulation state s'.

If t commits an observable consistency violation, then it must be an e^g action for some procedure g used by f. (It cannot be an x^g action, since those are executed from an internal state.) Then action t causes the same collective error from s'.

Likewise, if t commits an observable postcondition, assigns, *waitsfor*, or unreceived message violation, then t must be an x^f action, and the same action is enabled at s', and commits the same collective error.

Likewise, if t commits an observable precondition violation in a call to procedure g, then it commits the same violation at simulation state s'.

Suppose t commits a segment boundary violation. Then t is a send or receive by p when the matching process q is in a later segment in ζ. Since p is at an external state at s_i, q is also at a later segment in η at s'. Hence t commits a segment boundary violation from s'.

Case 2: s_i is deadlocked. Then $\zeta = \zeta^i$ has no observable collective error, there is some process that is not terminated at s_i, and all non-terminated processes are blocked at receive statements with empty channels at s_i. Suppose all the non-terminated processes are at external states at s_i. Then the simulation state s' is also deadlocked, and therefore incorrect.

So suppose there is some process p at an internal state at s_i. Say p is in a region corresponding to a call to collective procedure g used by f. Consider the execution χ extracted from this region of ζ. By Lemma A.6, χ starts at a *pre*(g)-state. If there were an action enabled at the final state of χ, then that same action would be enabled at s_i; hence χ is also deadlocked. But then χ is an incorrect execution of conforming procedure g from a *pre*(g)-state, a contradiction. □

Finally, we can prove the main theorem:

Proof of Theorem 1. Suppose ζ is an incorrect execution. By Lemma A.8, ζ is not observably correct. If ζ is finite, Lemma A.11 implies an incorrect simulation exists. If ζ is infinite, Lemma A.10 implies an incorrect simulation exists. □

B CONTRACTS OF MPI COLLECTIVE FUNCTIONS AND MPI_REDUCE_LOCAL

B.1 MPI_Allgather

```
#define SBUF         \mpi_buf(sbuf, scount, stype)
#define SBUF_OF(id)  \mpi_on(SBUF, (id))
#define RBUF         \mpi_buf(rbuf, rcount * \mpi_comm_size, rtype)
#define RBUF_OF(id)  (\mpi_buf(rbuf, rcount, rtype) + (id)*rcount)
#define SSIG         (\mpi_sig(stype) * scount)
#define SSIG_OF(id)  \mpi_on(SSIG, (id))
#define RSIG         (\mpi_sig(rtype) * rcount)
/*@ mpi uses comm;
    mpi collective(comm):
      requires \mpi_nonoverlapping(rtype) && rcount >= 0 && \valid(RBUF);
      requires \separated(RBUF, { SBUF | int i; sbuf != MPI_IN_PLACE });
      assigns  *RBUF;
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior not_in_place:
        assumes sbuf != MPI_IN_PLACE;
        requires \mpi_agree(sbuf != MPI_IN_PLACE);
        requires \forall int i; 0 <= i < \mpi_comm_size
                   ==> SSIG == \mpi_on(RSIG, i);
        requires scount >= 0 && \valid_read(SBUF);
        ensures  \forall int i; 0 <= i < \mpi_comm_size
                   ==> *RBUF_OF(i) == \mpi_on(*SBUF, i);
     behavior in_place:
        assumes sbuf == MPI_IN_PLACE;
        requires \mpi_agree(sbuf == MPI_IN_PLACE);
        requires \forall int i; 0 <= i < \mpi_comm_size
                   ==> RSIG == \mpi_on(RSIG, i);
        ensures  \forall int i; 0 <= i < \mpi_comm_size
                   ==> *RBUF_OF(i) == \old(\mpi_on(*RBUF_OF(i), i));
     disjoint behaviors;
     complete behaviors;
 */
int MPI_Allgather(const void *sbuf, int scount, MPI_Datatype stype,
                  void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm);
```

B.2 MPI_Allgatherv

```
#define SBUF        \mpi_buf(sbuf, scount, stype)
#define RBUF_OF(i)  (\mpi_buf(rbuf, rcounts[(i)], rtype) + displs[(i)])
#define SSIG_OF(i)  \mpi_on(\mpi_sig(stype) * scount, (i))
#define RSIG_OF(i)  \mpi_sig(rtype) * rcounts[(i)]
/*@ mpi uses comm;
    mpi collective(comm):
      requires \valid_read(rcounts + (0 .. \mpi_comm_size-1));
      requires \valid_read(displs + (0 .. \mpi_comm_size-1));
      requires \forall int i; 0 <= i < \mpi_comm_size ==> rcounts[i] >= 0;
      requires \forall int i; 0 <= i < \mpi_comm_size ==> \valid(RBUF_OF(i));
      requires \mpi_nonoverlapping(rtype);
      requires \forall int i, j; 0 <= i < j < \mpi_comm_size
                 ==> (displs[i] + rcounts[i] <= displs[j]) ||
                     (displs[j] + rcounts[j] <= displs[i]);
      requires \separated({ RBUF_OF(i) | int i; 0 <= i < \mpi_comm_size },
                          {{ SBUF | int i; sbuf != MPI_IN_PLACE },
                           rcounts + (0 .. \mpi_comm_size-1),
                           displs + (0 .. \mpi_comm_size-1)});
      assigns {*RBUF_OF(i) | int i; 0 <= i < \mpi_comm_size};
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior not_in_place:
        assumes sbuf != MPI_IN_PLACE;
        requires \mpi_agree(sbuf != MPI_IN_PLACE);
        requires \valid_read(SBUF) && scount >= 0;
```

```
        requires \forall int i; 0 <= i < \mpi_comm_size
                   ==> RSIG_OF(i) == SSIG_OF(i);
        ensures \forall int i; 0 <= i < \mpi_comm_size
                   ==> *RBUF_OF(i) == \mpi_on(*SBUF, i);
      behavior in_place:
        assumes sbuf == MPI_IN_PLACE;
        requires \mpi_agree(sbuf == MPI_IN_PLACE);
        requires \forall int i; 0 <= i < \mpi_comm_size
                   ==> \mpi_agree(RSIG_OF(i));
        ensures  \forall int i; 0 <= i < \mpi_comm_size
                   ==> *RBUF_OF(i) == \old(\mpi_on(*RBUF_OF(i), i));
      disjoint behaviors;
      complete behaviors;
 */
int MPI_Allgatherv(const void *sbuf, int scount, MPI_Datatype stype,
                   void *rbuf, const int *rcounts, const int *displs,
                   MPI_Datatype rtype, MPI_Comm comm);
```

B.3 MPI_Allreduce

```
#define SBUF \mpi_buf(sbuf, count, datatype)
#define RBUF \mpi_buf(rbuf, count, datatype)
#define AGREE(x)  \mpi_agree((x))
/*@
   mpi uses comm;
   mpi collective(comm):
     requires \valid(RBUF) && count >= 0 && AGREE(count) && AGREE(datatype)
              && AGREE(op);
     requires \mpi_nonoverlapping(datatype);
     requires \separated(RBUF, { SBUF | int i; sbuf != MPI_IN_PLACE });
     assigns *RBUF;
     ensures AGREE(*RBUF);
     waitsfor {i | int i; 0 <= i < \mpi_comm_size};
     behavior not_in_place:
       assumes sbuf != MPI_IN_PLACE;
       requires AGREE(sbuf != MPI_IN_PLACE);
       requires \valid_read(SBUF);
       ensures  \mpi_reduce(*RBUF, 0, \mpi_comm_size, op,
                            \lambda integer t; \mpi_on(*SBUF, t));
     behavior in_place:
       assumes sbuf == MPI_IN_PLACE;
       requires AGREE(sbuf == MPI_IN_PLACE);
       ensures  \mpi_reduce(*RBUF, 0, \mpi_comm_size, op,
                            \lambda integer t; \mpi_on(\old(*RBUF), t));
   disjoint behaviors;
   complete behaviors;
*/
int MPI_Allreduce(const void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

B.4 MPI_Alltoall

```
#define SBUF       \mpi_buf(sbuf, scount * \mpi_comm_size, stype)
#define RBUF       \mpi_buf(rbuf, rcount * \mpi_comm_size, rtype)
#define SBUF_OF(i) (\mpi_buf(sbuf, scount, stype) + (i) * scount)
#define RBUF_OF(i) (\mpi_buf(rbuf, rcount, rtype) + (i) * rcount)
/*@ mpi uses comm;
    mpi collective(comm):
      requires \valid(RBUF) && rcount >= 0 && \mpi_nonoverlapping(rtype);
      requires \separated(RBUF, { SBUF | int i; sbuf != MPI_IN_PLACE });
      assigns  *RBUF;
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior not_in_place:
```

```
        assumes  sbuf != MPI_IN_PLACE;
        requires \mpi_agree(sbuf != MPI_IN_PLACE) && \valid_read(SBUF);
        requires scount >= 0 &&
                 scount * \mpi_sig(stype) == rcount * \mpi_sig(rtype);
        ensures  \forall int i, j; 0 <= i < \mpi_comm_size && j == \mpi_comm_rank
                   ==> *RBUF_OF(i) == \mpi_on(*SBUF_OF(j), i);
      behavior in_place:
        assumes  sbuf == MPI_IN_PLACE;
        requires \mpi_agree(sbuf == MPI_IN_PLACE);
        ensures \forall int i, j; 0 <= i < \mpi_comm_size && j == \mpi_comm_rank
                  ==> *RBUF_OF(i) == \old(\mpi_on(*RBUF_OF(j), i));
     disjoint behaviors;
     complete behaviors;
 */
int MPI_Alltoall(const void *sbuf, int scount, MPI_Datatype stype,
                 void *rbuf, int rcount, MPI_Datatype rtype,
                 MPI_Comm comm);
```

B.5 MPI_Alltoallv

```
#define SBUF_OF(i)  (\mpi_buf(sbuf, scounts[(i)], stype) + sdispls[(i)])
#define RBUF_OF(i)  (\mpi_buf(rbuf, rcounts[(i)], rtype) + rdispls[(i)])
/*@ mpi uses comm;
    mpi collective(comm):
      requires \valid_read(rcounts + (0 .. \mpi_comm_size-1));
      requires \valid_read(rdispls + (0 .. \mpi_comm_size-1));
      requires \forall int i; 0 <= i < \mpi_comm_size ==> \valid(RBUF_OF(i));
      requires \forall int i; 0 <= i < \mpi_comm_size ==> rcounts[i] >= 0;
      requires \forall int i, j; 0 <= i < j < \mpi_comm_size ==>
                 (rdispls[i] + rcounts[i] <= rdispls[j]) ||
                 (rdispls[j] + rcounts[j] <= rdispls[i]);
      requires \mpi_nonoverlapping(rtype);
      requires \separated({ RBUF_OF(i) | int i; 0 <= i < \mpi_comm_size },
                          { { {SBUF_OF(i), scounts + i, sdispls + i} |
                          int i; 0 <= i < \mpi_comm_size && sbuf != MPI_IN_PLACE },
                          rcounts + (0 .. \mpi_comm_size-1),
                          rdispls + (0 .. \mpi_comm_size-1) });
      assigns {*RBUF_OF(i) | int i; 0 <= i < \mpi_comm_size-1};
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior not_in_place:
        assumes sbuf != MPI_IN_PLACE;
        requires \mpi_agree(sbuf != MPI_IN_PLACE);
        requires \forall int i; 0 <= i < \mpi_comm_size ==> \valid_read(SBUF_OF(i));
        requires \valid_read(scounts + (0 .. \mpi_comm_size-1));
        requires \valid_read(sdispls + (0 .. \mpi_comm_size-1));
        requires \forall int i; 0 <= i < \mpi_comm_size ==> scounts[i] >= 0;
        requires \forall int i, j; 0 <= i < \mpi_comm_size &&
                 j == \mpi_comm_rank ==>
                   \mpi_on(scounts[j] * \mpi_sig(stype), i) ==
                   rcounts[i] * \mpi_sig(rtype);
        ensures \forall int i, j; 0 <= i < \mpi_comm_size &&
                j == \mpi_comm_rank ==>
                  *RBUF_OF(i) == \mpi_on(*SBUF_OF(j), i);
      behavior in_place:
        assumes sbuf == MPI_IN_PLACE;
        requires \mpi_agree(sbuf == MPI_IN_PLACE);
        requires \forall int i, j; 0 <= i < \mpi_comm_size &&
                 j == \mpi_comm_rank ==>
                   \mpi_on(rcounts[j] * \mpi_sig(rtype), i) ==
                   rcounts[i] * \mpi_sig(rtype);
        ensures  \forall int i, j; 0 <= i < \mpi_comm_size &&
                 j == \mpi_comm_rank ==>
                   *RBUF_OF(i) == \old(\mpi_on(*RBUF_OF(j), i));
```

```
    disjoint behaviors;
    complete behaviors;
 */
int MPI_Alltoallv(const void *sbuf, const int *scounts, const int *sdispls,
                  MPI_Datatype stype,
                  void *rbuf, const int *rcounts, const int *rdispls,
                  MPI_Datatype rtype, MPI_Comm comm);
```

B.6 MPI_Alltoallw

```
#define SBUF_OF(i) (\mpi_buf(sbuf, scounts[(i)], stypes[(i)]) + sdispls[i])
#define RBUF_OF(i) (\mpi_buf(rbuf, rcounts[(i)], rtypes[(i)]) + rdispls[i])
/*@ mpi uses comm;
    mpi collective(comm):
      requires \valid_read(rcounts + (0 .. \mpi_comm_size-1));
      requires \valid_read(rdispls + (0 .. \mpi_comm_size-1));
      requires \valid_read(rtypes + (0 .. \mpi_comm_size-1));
      requires \forall int i; 0 <= i < \mpi_comm_size
                 ==> \valid(RBUF_OF(i)) && rcounts[i] >= 0;
      requires \forall int i; 0 <= i < \mpi_comm_size
                 ==> \mpi_nonoverlapping(rtypes[i]);
      requires \forall int i, j; 0 <= i < j < \mpi_comm_size
                 ==> (rdispls[i] + rcounts[i] <= rdispls[j]) ||
                     (rdispls[j] + rcounts[j] <= rdispls[i]);
      requires \separated({RBUF_OF(i) | int i; 0 <= i < \mpi_comm_size},
                          { {{SBUF_OF(i), scounts + i, sdispls + i, stypes + i} |
                             int i; 0 <= i < \mpi_comm_size},
                             rcounts + (0 .. \mpi_comm_size-1),
                             rdispls + (0 .. \mpi_comm_size-1),
                             rtypes +  (0 .. \mpi_comm_size-1)});
      assigns  { *RBUF_OF(i) | int i; 0 <= i < \mpi_comm_size};
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior not_in_place:
        assumes  sbuf != MPI_IN_PLACE;
        requires \mpi_agree(sbuf != MPI_IN_PLACE);
        requires \valid_read(scounts + (0 .. \mpi_comm_size-1));
        requires \valid_read(sdispls + (0 .. \mpi_comm_size-1));
        requires \valid_read(stypes + (0 .. \mpi_comm_size-1));
        requires \forall int i; 0 <= i < \mpi_comm_size ==>
                   \valid_read(SBUF_OF(i)) && scounts[i] >= 0;
        requires \forall int i, j; 0 <= i < \mpi_comm_size &&
                   j == \mpi_comm_rank ==>
                     \mpi_on(\mpi_sig(stypes[j]) * scounts[j], i) ==
                     \mpi_sig(rtypes[i]) * rcounts[i];
        ensures  \forall int i, j; 0 <= i < \mpi_comm_size &&
                   j == \mpi_comm_rank ==>
                     *RBUF_OF(i) == \mpi_on(*SBUF_OF(j), i);
      behavior in_place:
        assumes  sbuf == MPI_IN_PLACE;
        requires \mpi_agree(sbuf == MPI_IN_PLACE);
        requires \forall int i, j; 0 <= i < \mpi_comm_size &&
                   j == \mpi_comm_rank ==>
                     \mpi_on(\mpi_sig(rtypes[j]) * rcounts[j], i) ==
                     \mpi_sig(rtypes[i]) * rcounts[i];
        ensures  \forall int i, j; 0 <= i < \mpi_comm_size &&
                   j == \mpi_comm_rank ==>
                     *RBUF_OF(i) == \old(\mpi_on(*RBUF_OF(j), i));
    disjoint behaviors;
    complete behaviors;
 */
int MPI_Alltoallw(const void *sbuf, const int scounts[], const int sdispls[],
                  const MPI_Datatype stypes[],
                  void *rbuf, const int rcounts[], const int rdispls[],
```

```
                  const MPI_Datatype rtypes[], MPI_Comm comm);
```

B.7 MPI_Barrier

```
/*@ mpi uses comm;
    mpi collective(comm):
      requires \true;
      assigns  \nothing;
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
 */
int MPI_Barrier(MPI_Comm comm);
```

B.8 MPI_Bcast

```
#define BUF \mpi_buf(buf, count, datatype)
#define AGREE(x) \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires 0 <= root < \mpi_comm_size && AGREE(root);
      requires AGREE(count * \mpi_sig(datatype)) && 0 <= count;
      requires \mpi_nonoverlapping(datatype);
      ensures AGREE(*BUF);
      behavior root:
        assumes \mpi_comm_rank == root;
        requires \valid_read(BUF);
        assigns \nothing;
      behavior nonroot:
        assumes \mpi_comm_rank != root;
        requires \valid(BUF);
        assigns *BUF;
        waitsfor root;
      complete behaviors;
      disjoint behaviors;
   */
int MPI_Bcast(void * buf, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm);
```

B.9 MPI_Exscan

```
#define SBUF \mpi_buf(sbuf, count, datatype)
#define RBUF \mpi_buf(rbuf, count, datatype)
#define AGREE(x) \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires count >= 0 && AGREE(count) && AGREE(datatype) && AGREE(op);
      requires \mpi_nonoverlapping(datatype);
      requires sbuf != MPI_IN_PLACE ==> \valid_read(SBUF);
      requires sbuf == MPI_IN_PLACE || \mpi_comm_rank > 0
                 ==> \valid(RBUF);
      waitsfor {i | int i; 0 <= i < \mpi_comm_rank-1};
      behavior zero:
        assumes \mpi_comm_rank == 0;
        assigns  \nothing;
      behavior others:
        assumes \mpi_comm_rank > 0;
        requires \separated(RBUF, {SBUF | int i; sbuf != MPI_IN_PLACE});
        assigns *RBUF;
        ensures \mpi_reduce(*RBUF, 0, \mpi_comm_rank, op,
                  \lambda integer t;
                    \mpi_on(sbuf != MPI_IN_PLACE ? *SBUF : \old(*RBUF), t));
    disjoint behaviors;
    complete behaviors;
 */
int MPI_Exscan(const void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
```

```
               MPI_Op op, MPI_Comm comm);
```

B.10 MPI_Gather

```
#define SBUF         \mpi_buf(sbuf, scount, stype)
#define SBUF_OF(id)  \mpi_on(SBUF, (id))
#define RBUF         \mpi_buf(rbuf, rcount * \mpi_comm_size, rtype)
#define RBUF_OF(id)  (\mpi_buf(rbuf, rcount, rtype) + (id)*rcount)
#define SSIG         (\mpi_sig(stype) * scount)
#define SSIG_OF(id)  \mpi_on(SSIG, (id))
#define RSIG         (\mpi_sig(rtype) * rcount)
#define AGREE(x)     \mpi_agree((x))
/*@
  mpi uses comm;
  mpi collective(comm) :
    requires AGREE(root) && 0 <= root < \mpi_comm_size;
    behavior root:
      assumes  \mpi_comm_rank == root;
      requires \valid(RBUF) && \mpi_nonoverlapping(rtype);
      requires \forall int id; 0 <= id < \mpi_comm_size
                 ==> id != root ==> SSIG_OF(id) == RSIG;
      requires sbuf == MPI_IN_PLACE || \valid_read(SBUF);
      requires sbuf != MPI_IN_PLACE ==> (SSIG == RSIG);
      requires \separated(RBUF, { SBUF | int i; sbuf != MPI_IN_PLACE });
      assigns  *RBUF;
      ensures  sbuf != MPI_IN_PLACE ==> *RBUF_OF(root) == *SBUF;
      ensures  sbuf == MPI_IN_PLACE ==>
                 *RBUF_OF(root) == \old(*RBUF_OF(root));
      ensures  \forall int id; 0 <= id < \mpi_comm_size ==>
                 id != root ==>
                   *RBUF_OF(id) == *SBUF_OF(id);
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
    behavior not_root:
      assumes  \mpi_comm_rank != root;
      requires \valid_read(SBUF);
      assigns  \nothing;
      waitsfor \nothing;
  disjoint behaviors;
  complete behaviors;
*/
int MPI_Gather(const void* sbuf, int scount, MPI_Datatype stype,
               void* rbuf, int rcount, MPI_Datatype rtype,
               int root, MPI_Comm comm);
```

B.11 MPI_Gatherv

```
#define SBUF \mpi_buf(sbuf, scount, stype)
#define RBUF(i) (\mpi_buf(rbuf, rcounts[i], rtype) + displs[i])
#define AGREE(x) \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires AGREE(root) && 0 <= root < \mpi_comm_size;
      behavior root:
        assumes  \mpi_comm_rank == root;
        requires \mpi_nonoverlapping(rtype);
        requires \valid_read(rcounts + (0 .. \mpi_comm_size-1));
        requires \valid_read(displs + (0 .. \mpi_comm_size-1));
        requires \forall int i; 0 <= i < \mpi_comm_size
                   ==> rcounts[i] >= 0;
        requires \forall int i; 0 <= i < \mpi_comm_size
                   ==> \valid(RBUF(i));
        requires \forall int i,j; 0 <= i < j < \mpi_comm_size
```

```
                   ==> (displs[i] + rcounts[i] <= displs[j] ||
                        displs[j] + rcounts[j] <= displs[i]);
        requires \forall int i; 0 <= i < \mpi_comm_size && i != root ==>
                    \mpi_on(scount*\mpi_sig(stype), i) ==
                    rcounts[i]*\mpi_sig(rtype);
        requires sbuf != MPI_IN_PLACE ==> \valid_read(SBUF) &&
                   scount*\mpi_sig(stype) ==
                   rcounts[root]*\mpi_sig(rtype);
        requires \separated( {RBUF(i) | int i; 0 <= i < \mpi_comm_size},
                      { {{SBUF, &scount, &stype} | int i; sbuf != MPI_IN_PLACE},
                        rcounts + (0 .. \mpi_comm_size-1),
                        displs + (0 .. \mpi_comm_size-1)} );
        assigns { *RBUF(i) | int i; 0 <= i < \mpi_comm_size };
        ensures sbuf != MPI_IN_PLACE ? *RBUF(root) == *SBUF :
                                       *RBUF(root) == \old(*RBUF(root));
        ensures \forall int i; 0 <= i < \mpi_comm_size && i != root
                  ==> *RBUF(i) == \mpi_on(*SBUF, i);
        waitsfor { i | int i; 0 <= i < \mpi_comm_size};
      behavior non_root:
        assumes  \mpi_comm_rank != root;
        requires \valid_read(SBUF);
        assigns  \nothing;
        waitsfor \nothing;
      complete behaviors;
      disjoint behaviors;*/
int MPI_Gatherv(const void* sbuf, int scount, MPI_Datatype stype,
                void* rbuf, const int rcounts[], const int displs[],
                MPI_Datatype rtype, int root, MPI_Comm comm);
```

B.12 MPI_Reduce

```
#define SBUF \mpi_buf(sbuf, count, datatype)
#define RBUF \mpi_buf(rbuf, count, datatype)
#define AGREE(x) \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires AGREE(root) && 0 <= root < \mpi_comm_size;
      requires 0 <= count && \mpi_nonoverlapping(datatype);
      requires AGREE(count) && AGREE(datatype) && AGREE(op);
      behavior root:
        assumes  \mpi_comm_rank == root;
        requires \valid(RBUF);
        requires sbuf == MPI_IN_PLACE || \valid_read(SBUF);
        requires sbuf != MPI_IN_PLACE ==> \separated(RBUF, SBUF);
        assigns  *RBUF;
        ensures  \mpi_reduce(*RBUF, 0, \mpi_comm_size, op,
                             \lambda integer t;
                               \mpi_on(sbuf != MPI_IN_PLACE ? *SBUF :
                                                              \old(*RBUF), t));
        waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior non_root:
        assumes  \mpi_comm_rank != root;
        requires \valid_read(SBUF);
        assigns  \nothing;
        waitsfor \nothing;
      disjoint behaviors;
      complete behaviors;
 */
int MPI_Reduce(const void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
```

B.13 MPI_Reduce_local

```
#define INBUF     \mpi_buf(inbuf, count, datatype)
#define INOUTBUF  \mpi_buf(inoutbuf, count, datatype)
/*@ requires \valid(INBUF) && \valid(INOUTBUF);
    requires count >= 0;
    requires \mpi_nonoverlapping(datatype);
    requires \separated(INOUTBUF, INBUF);
    assigns *INOUTBUF;
    ensures \mpi_reduce(*INOUTBUF, 0, 2, op, \lambda integer t;
                        t == 0 ? *INBUF : \old(*INOUTBUF));
 */
int MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count,
                     MPI_Datatype datatype, MPI_Op op);
```

B.14 MPI_Reduce_scatter

```
#define MY_RANK           \mpi_comm_rank
#define SCOUNT            \sum(0, \mpi_comm_size-1, \lambda int k0; rcounts[k0])
#define COUNTS(i)         (i==0?0:\sum(0, (i-1), \lambda int k1; rcounts[k1]))
#define SBUF              \mpi_buf(sbuf, SCOUNT, datatype)
#define MY_SBUF_BLK(i)    (\mpi_buf(sbuf, rcounts[(i)], datatype) + COUNTS(i))
#define MY_RBUF_AS_SBUF_BLK(i) (\mpi_buf(rbuf, rcounts[(i)], datatype) + COUNTS(i))
#define RBUF_AS_SBUF      \mpi_buf(rbuf, SCOUNT, datatype)
#define MY_RBUF           \mpi_buf(rbuf, rcounts[MY_RANK], datatype)
#define AGREE(x)          \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires \valid_read(rcounts + (0 .. \mpi_comm_size-1));
      requires rcounts[MY_RANK] >= 0 && AGREE(datatype) &&
               AGREE(SCOUNT) && AGREE(op);
      requires \forall int i; 0 <= i < \mpi_comm_size
                 ==> AGREE(rcounts[i]);
      requires \mpi_nonoverlapping(datatype);
      requires \separated(MY_RBUF, { rcounts + (0 .. \mpi_comm_size-1),
                                    { SBUF | int i; sbuf != MPI_IN_PLACE }});
      assigns *MY_RBUF;
      ensures \forall int i; i == \mpi_comm_rank ==>
                 \mpi_reduce(*MY_RBUF, 0, \mpi_comm_size, op,
                   \lambda integer t;
                      \mpi_on(sbuf != MPI_IN_PLACE ? *MY_SBUF_BLK(i) :
                                                     \old(*MY_RBUF_AS_SBUF_BLK(i)), t));
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior not_in_place:
        assumes  sbuf != MPI_IN_PLACE;
        requires \valid_read(SBUF);
        requires \valid(MY_RBUF);
      behavior in_place:
        assumes sbuf == MPI_IN_PLACE;
        requires \valid(RBUF_AS_SBUF);
    disjoint behaviors;
    complete behaviors;
 */
int MPI_Reduce_scatter(const void *sbuf, void *rbuf, const int *rcounts,
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

B.15 MPI_Reduce_scatter_block

```
#define SBUF    \mpi_buf(sbuf, rcount * \mpi_comm_size, datatype)
#define MY_SBUF_BLK (\mpi_buf(sbuf, rcount, datatype) + \mpi_comm_rank * rcount)
#define MY_RBUF  \mpi_buf(rbuf, rcount, datatype)
#define RBUF_AS_SBUF \mpi_buf(rbuf, rcount * \mpi_comm_size, datatype)
#define AGREE(x) \mpi_agree((x))
/*@ mpi uses comm;
```

```
    mpi collective(comm):
      requires \valid(MY_RBUF) && rcount >= 0 && AGREE(rcount) &&
               AGREE(datatype) && AGREE(op);
      requires \mpi_nonoverlapping(datatype);
      assigns  *MY_RBUF;
      ensures  \mpi_reduce(*MY_RBUF, 0, \mpi_comm_size, op,
                           \lambda integer t;
                             \mpi_on(sbuf != MPI_IN_PLACE ? *MY_SBUF_BLK :
                                                            \old(*MY_RBUF), t));
      waitsfor {i | int i; 0 <= i < \mpi_comm_size};
      behavior not_in_place:
        assumes sbuf != MPI_IN_PLACE;
        requires AGREE(sbuf != MPI_IN_PLACE);
        requires \valid_read(SBUF);
      behavior in_place:
        assumes sbuf == MPI_IN_PLACE;
        requires AGREE(sbuf == MPI_IN_PLACE);
        requires \valid(RBUF_AS_SBUF);
    disjoint behaviors;
    complete behaviors;
 */
int MPI_Reduce_scatter_block(const void *sbuf, void *rbuf, int rcount,
                             MPI_Datatype datatype, MPI_Op op,
                             MPI_Comm comm);
```

B.16 MPI_Scan

```
#define SBUF \mpi_buf(sbuf, count, datatype)
#define RBUF \mpi_buf(rbuf, count, datatype)
#define AGREE(x) \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires \valid(RBUF) && count >= 0;
      requires AGREE(count) && AGREE(datatype) && AGREE(op);
      requires \mpi_nonoverlapping(datatype);
      requires sbuf != MPI_IN_PLACE ==> \valid_read(SBUF);
      requires \separated(RBUF, {SBUF | int i; sbuf != MPI_IN_PLACE});
      assigns  *RBUF;
      ensures \mpi_reduce(*RBUF, 0, \mpi_comm_rank + 1, op,
                          \lambda integer t;
                            \mpi_on(sbuf != MPI_IN_PLACE ? *SBUF :
                                                           \old(*RBUF), t));
      waitsfor {i | int i; 0 <= i < \mpi_comm_rank};
 */
int MPI_Scan(const void *sbuf, void *rbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

B.17 MPI_Scatter

```
#define SBUF_OF(id) (\mpi_buf(sbuf, scount, stype) + (id) * scount)
#define SBUF        \mpi_buf(sbuf, scount * \mpi_comm_size, stype)
#define RBUF        \mpi_buf(rbuf, rcount, rtype)
#define SSIG        (scount*\mpi_sig(stype))
#define RSIG        (rcount*\mpi_sig(rtype))
#define AGREE(x)    \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires AGREE(root) && 0 <= root < \mpi_comm_size;
      behavior root:
        assumes \mpi_comm_rank == root;
        requires scount >= 0 && \valid_read(SBUF);
        requires rbuf == MPI_IN_PLACE || \valid(RBUF);
        requires rbuf != MPI_IN_PLACE ==> SSIG == RSIG &&
```

```
                   \mpi_nonoverlapping(rtype);
        requires rbuf != MPI_IN_PLACE ==> \separated(RBUF, SBUF);
        assigns  {*RBUF | int i; rbuf != MPI_IN_PLACE};
        ensures  rbuf != MPI_IN_PLACE ==> *RBUF == *SBUF_OF(root);
        waitsfor \nothing;
      behavior nonroot:
        assumes \mpi_comm_rank != root;
        requires rcount >= 0 && \valid(RBUF) && \mpi_nonoverlapping(rtype);
        requires RSIG == \mpi_on(SSIG, root);
        assigns  *RBUF;
        ensures  \forall int i; i == \mpi_comm_rank ==>
                   *RBUF == \mpi_on(*SBUF_OF(i), root);
        waitsfor root;
    disjoint behaviors;
    complete behaviors;
 */
int MPI_Scatter(const void* sbuf, int scount, MPI_Datatype stype,
                void* rbuf, int rcount, MPI_Datatype rtype, int root,
                MPI_Comm comm);
```

B.18 MPI_Scatterv

```
#define SBUF_OF(id) (\mpi_buf(sbuf, scounts[(id)], stype) + displs[(id)])
#define RBUF        \mpi_buf(rbuf, rcount, rtype)
#define SSIG_OF(id) (\mpi_sig(stype) * scounts[(id)])
#define RSIG        (\mpi_sig(rtype) * rcount)
#define AGREE(x)    \mpi_agree((x))
/*@ mpi uses comm;
    mpi collective(comm):
      requires AGREE(root) && 0 <= root < \mpi_comm_size;
      behavior root:
        assumes  \mpi_comm_rank == root;
        requires \valid_read(scounts + (0 .. \mpi_comm_size-1));
        requires \valid_read(displs + (0 .. \mpi_comm_size-1));
        requires \forall int i; 0 <= i < \mpi_comm_size
                   ==> scounts[i] >= 0;
        requires \forall int i, j; 0 <= i < j < \mpi_comm_size
                   ==> (displs[i] + scounts[i] <= displs[j]) ||
                       (displs[j] + scounts[j] <= displs[i]);
        requires \forall int i; 0 <= i < \mpi_comm_size &&
                 i != root
                   ==> \valid_read(SBUF_OF(i));
        requires rbuf != MPI_IN_PLACE ==> \valid_read(SBUF_OF(root)) && \valid(RBUF) &&
                                          SSIG_OF(root) == RSIG                       &&
                                          \mpi_nonoverlapping(rtype);
        requires rbuf != MPI_IN_PLACE ==>
                   \separated(RBUF, {{ SBUF_OF(i) | int i; 0 <= i < \mpi_comm_size },
                                       scounts + (0 .. \mpi_comm_size-1),
                                       displs + (0 .. \mpi_comm_size-1)});

        assigns  {*RBUF | int i; rbuf != MPI_IN_PLACE};

        ensures  rbuf != MPI_IN_PLACE ==> *RBUF == *SBUF_OF(root);
        waitsfor \nothing;
      behavior nonroot:
        assumes  \mpi_comm_rank != root;
        requires  rcount >= 0 && \valid(RBUF) && \mpi_nonoverlapping(rtype);
        requires  \forall integer i; i == \mpi_comm_rank ==>
                   RSIG == \mpi_on(SSIG_OF(i), root);
        assigns  *RBUF;
        ensures  \forall integer i; i == \mpi_comm_rank ==>
                   *RBUF == \mpi_on(*SBUF_OF(i), root);
        waitsfor {root | int i; 0 <= i < \mpi_comm_size};
```

```
    disjoint behaviors;
    complete behaviors;
 */
int MPI_Scatterv(const void *sbuf, const int *scounts, const int *displs,
                 MPI_Datatype stype, void *rbuf, int rcount,
                 MPI_Datatype rtype, int root, MPI_Comm comm);
```