Collective Contracts for Message-Passing Parallel Programs

Ziqing Luo(✉) and Stephen F. Siegel

University of Delaware, Newark, DE 19716, USA
{ziqing,siegel}@udel.edu

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 called 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. 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 model checker for checking contract satisfaction within small bounds on the number of processes.

Keywords: contract · message-passing · MPI · verification · collective

1 Introduction

Procedure contracts [27,46,47] are a well-known way to decompose program verification. In this approach, each procedure f is specified independently with pre- and postconditions or other invariants. To verify f, one needs only the contracts, not the implementations, of the procedures called by f.

Contract languages have been developed for many programming languages. These include the *Java Modeling Language* (JML) [38] for Java and the *ANSI C Specification Language* (ACSL) [10] 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 [18] with the WP plugin [9].

In this paper, we explore a procedure contract system for message-passing parallel programs, specifically for programs that use the Message-Passing Interface (MPI) [45], the de facto standard for high performance computing.

Our contracts apply to *collective-style* procedures in these programs. These are procedures f called by all processes and that are *communication-closed*: any message issued by a send statement in f is received by a receive statement in f, and vice-versa. The processes executing f coordinate in order to accomplish a coherent change in the global state. Examples include all of the standard blocking MPI collective functions [45, Chapter 5], but also many user-defined procedures,

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14682, pp. 44–68, 2024.
https://doi.org/10.1007/978-3-031-65630-9_3

such as a procedure to exchange ghost cells in a stencil computation. (We will use the term *collective* as shorthand for *collective-style* when there is no chance of ambiguity.) These procedures are typically specified informally by describing the effect they produce when called by all processes, rather than the effect of an individual process. They should be formally specified and verified in the same way.

Developers often construct applications by composing collective procedures. As examples, consider the Monte Carlo particle transport code OpenMC [53] (over 24K lines of C++/MPI code) and module `parcsr_ls` in the algebraic multigrid solver AMG [62] (over 35K lines of C/MPI code). Through manual inspection, we confirmed that every function in these codes that involves MPI communication is collective-style.

We begin in Sect. 2 with a toy message-passing language, so the syntax, semantics, and theoretical results can be stated and proved precisely. The main result is a theorem that justifies a method for verifying a collective procedure using only the contracts of the collective procedures called, as in the sequential case.

Section 3 describes changes needed to apply this system to C/MPI programs. We handle a significant subset of MPI that does not include `MPI_ANY_SOURCE` ("wildcard") receives. This means program behavior is largely independent of interleaving [55]. There are enough issues to deal with, such as MPI datatypes, input nondeterminism, and nontermination, that we feel it best to leave wildcards for a sequel. A prototype verification system for such programs, using the CIVL model checker, is described and evaluated in Sect. 4. Related work is discussed in Sect. 5. In Sect. 6, we wrap up with a discussion of the advantages and limitations of our system, and work that remains.

In summary, this paper makes the following contributions: (1) a contract theory for collective message-passing procedures, with mathematically precise syntax and semantics, (2) a theorem justifying a method for verifying that a collective procedure conforms to its contract, (3) a contract language for a large subset of MPI, based on the theory but also dealing with additional intricacies of MPI, and (4) a prototype verification tool for checking that collective-style MPI procedures conform to their contracts.

2 A Theory of Collective Contracts

2.1 Language

We describe a simple message-passing language MiniMP with syntax in Fig. 1. 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

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

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

Fig. 1. MiniMP syntax

return some value—even if the second argument of division is 0, e.g. Operators with '\', described below, occur only in the optional contract.

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 $\mathsf{PID} = \{0, \ldots, n-1\}$. A process can obtain its ID using the primitive `pid`; it can obtain n using `nprocs`.

The command "`send` *data* `to` *dest*" sends the value of *data* to the process with ID *dest*. There is one FIFO message buffer for each ordered pair of processes $p \to q$ and the effect of `send` is to enqueue the message on the buffer for which p is the ID of the sender and q is *dest*. The buffers are unbounded, so `send` never blocks. 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 1. The program of Fig. 2 has two procedures, both 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 is the sum of k and the original value of `x` on q. It also declares p cannot exit until q has entered. Procedure `f` calls `g` `nprocs` times. Its contract requires that all processes call `f`

```
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; } }
```

Fig. 2. `cyc`: a MiniMP program

with the same value for `k`. It ensures that upon return, the value of `x` is 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 f's program graph. A *state* specifies a process state for each process, as well as the state of channel $p \to q$ for all $p, q \in \mathsf{PID}$. The channel state is a finite sequence of integers, the buffered messages sent from p to q.

An *action* is a pair $a = \langle e, p \rangle$, where e is an edge $u \xrightarrow{\alpha} v$ in a program graph and $p \in \mathsf{PID}$. Action a 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 $s \xrightarrow{a} s'$ 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 $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$. The *length* of ζ, denoted $\mathsf{len}(\zeta)$, 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 $p \in \mathsf{PID}$. 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. Say f takes m formal parameters. Form a 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

In this section, we formulate conditions that correct collective procedures are expected to satisfy. Some of these reflect standard practice, e.g., collectives should be called in the same order by all processes, while others specify how a procedure conforms to various clauses in its contract. Ultimately, these conditions will be used to ensure that a simple "stub" can stand in for a collective call, which is the essential point of our main result, Theorem 1.

In formulating these conditions, we focus on the negative, i.e., we identify the earliest possible point in an execution at which a violation occurs. For example, if a postcondition states that on every process, `x` will be 0 when the function returns, then a postcondition violation occurs as soon as one process returns when its `x` has a non-zero value. There is no need to wait until every process has returned to declare that the postcondition has been violated. In fact, this allows us to declare a postcondition violation even in executions that do not terminate because some processes never return.

Fix a program P and integer $n \geq 1$. Let $\mathcal{C}$ be the set of names of collective procedures of P. Let ζ be an execution $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ of a procedure in P. For $i \in 1..\mathsf{len}(\zeta)$, let ζ^i denote the prefix of ζ of length i, i.e., the execution $s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} s_i$.

Collective Consistency. 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 $f \in \mathcal{C}$. Given an action a and $p \in \mathsf{PID}$, define string $T_p(a)$ as follows:

- if a is a call by p to some $f \in \mathcal{C}$, $T_p(a) = \mathsf{e}^f$ (a is called an *enter* action)

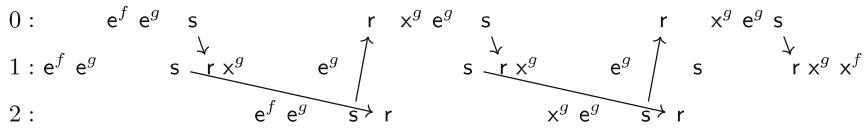

Fig. 3. Representation of a 3-process execution of $\mathtt{cyc}^f$ of Fig. 2. e^f = enter (call) f; x^f = exit (return from) f; s = send; r = receive. The execution has no collective errors and ends in a state with one buffered message sent from process 1 to process 2.

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

Now let $T_p(\zeta)$ be the concatenation $T_p(a_1)T_p(a_2)\cdots$. Hence $T_p(\zeta)$ records the sequence of collective actions—enter or exit actions—taken by p.

Definition 1. *An execution ζ is* collective consistent *if there is some $p \in \mathsf{PID}$ such that for all $q \in \mathsf{PID}$, $T_q(\zeta)$ 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 rest of this section, 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 $i \in 0..\mathsf{len}(\zeta)$ and $p \in \mathsf{PID}$, define $k = \mathsf{seg}_p(\zeta, i)$ to be the number of collective actions of p in $a_1, \ldots, a_i$. We say p *is in segment k at state i.*

Example 2. In program cyc of Fig. 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 $T_1(\zeta) = \mathsf{e}^f\mathsf{e}^g\mathsf{x}^g\mathsf{e}^g\mathsf{x}^g\mathsf{e}^g\mathsf{x}^g\mathsf{x}^f$ 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.

Precondition and Postcondition Violations. We now turn to the issue of evaluation of pre- and postconditions. Let f be a collective procedure in P with precondition $\mathsf{pre}(f)$ and postcondition $\mathsf{post}(f)$. 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 $\mathsf{pre}(f)$ and $\mathsf{post}(f)$. 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 $\mathsf{pre}(f)$ or $\mathsf{post}(f)$.

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

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

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

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

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 $a_{j(q)}$ is the r-th collective action of q in ζ. (As ζ is collective consistent, $a_{j(q)}$ 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 $[\![\mathsf{pre}(f)]\!]_{\alpha,q}$ holds for all $q \in \mathsf{PID}$. If no such α exists, we say a *precondition violation* occurs at step i.

Example 3. Consider program `cyc` of Fig. 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 a_i is an x^f action, define Q and $j(q)$ as above; for any $q \in Q$, $a_{j(q)}$ 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 $a_{k(q)}$ is the e^f action in q corresponding to $a_{j(q)}$, i.e., $a_{k(q)}$ is the call that led to the return $a_{j(q)}$. 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 $[\![\mathsf{post}(f)]\!]_{\alpha,\beta,q}$ holds for all $q \in \mathsf{PID}$.

Waitsfor Violations. We now explain the *waitsfor* contract clause. Assume again that a_i is an x^f action in process p, and that k is the index of the corresponding e^f action in p. The expression 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 $\mathsf{seg}_q(\zeta, i) < \mathsf{seg}_p(\zeta, k)$, i.e., p exits a collective call before q has entered it.

Correct Executions and Conformance to Contract. We can now encapsulate all the ways something may go wrong with collective procedures and their contracts:

Definition 2. *Let P be a program, $\zeta = s_0 \xrightarrow{a_1} s_1 \cdots$ an execution of a procedure in P, and $i \in 1..\mathsf{len}(\zeta)$. Let p be the process of a_i and $r = \mathsf{seg}_p(\zeta, i)$. 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*: a_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*: a_i is a receive of a message sent from a process q at a_j $(j < i)$ and $\mathsf{seg}_q(\zeta, j) > r$; or a_i is a send to q and $\mathsf{seg}_q(\zeta, i) > r$, or*
4. *an* unreceived message violation*: a_i is a collective action and there is an unreceived message sent to p from q at a_j $(j < i)$, and $\mathsf{seg}_q(\zeta, j) = r - 1$.*

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.

Definition 3. *An execution of a procedure is* correct *if it is finite, does not deadlock, and has no collective errors.*

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 in which (i) every process has one entry on its call stack, pointing to the start location of `main`, (ii) all channels are empty, and (iii) for all processes, the assignment to the global variables satisfies the precondition of f.

Definition 4. *A collective procedure f* conforms *(to its contract) if all executions of P^f from pre(f)-states are correct.*

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 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 $f, g \in \mathcal{C}$, we say f *uses* g if there is a path of positive length in the call graph from f to g on which any node other than the first or last is not in $\mathcal{C}$.

Given $f \in \mathcal{C}$, 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 correct then f conforms.*

Theorem 1 is the basis for the contract-checking tool described in Sect. 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 [43]. Here we summarize the main ideas of the proof. We assume henceforth that P is a collective recursion-free program.

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. We first observe that equivalence preserves most violations:

Lemma 1 *Let ζ and η be equivalent executions of a procedure f in P. Then*

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

If ζ 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 the error is *observable*. If the error is not observable, it is *internal*. 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 P. Let $g_1, g_2, \ldots$ be the sequence of collective procedures called from f. If a transition in region r (i.e., inside the call to g_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 f be 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.

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

Lemma 4 *Suppose 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 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 prove Theorem 1. Assume f is a collective procedure in P and all collective procedures used by f conform. Suppose f does not conform; we must show there is an incorrect simulation of f. As f does not conform, there is an incorrect execution ζ of P^f from a $pre(f)$-state. By Lemma 3, ζ is not observably correct. If ζ is finite or commits an observable collective error, Lemma 4 implies an incorrect simulation exists. Otherwise, Lemma 5 implies such a simulation exists. This completes the proof.

3 Collective Contracts for C/MPI

In Sect. 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 summarizes the syntax and semantics of each new contract primitive.

3.1 Background from MPI

In the toy language of Sect. 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 Sect. 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 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`, `MPI_DOUBLE`, etc. MPI provides 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 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.

[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*∗

Fig. 4. Grammar for ACSL function contracts, extended for MPI. Details for standard ACSL clauses can be found in [10].

3.2 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 Fig. 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 4) with the adjustment that all of the collective errors defined in Definition 2 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*.

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.

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. Type `mpi_buf_t` is a struct 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; predicate `\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 $p + m +$ `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

In this section we describe a prototype tool we developed for MPI collective contract verification, and experiments applying it to various example codes. All experimental artifacts, including the tool source code, are available online [43].

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 [45, 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" [45]. This guarantee is reflected in line 13. "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

```
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 && count > 0};
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, \lambda integer t; \mpi_on(*SBUF, t));
14  behavior in_place:
15   assumes sbuf == MPI_IN_PLACE;
16   requires \mpi_agree(sbuf == MPI_IN_PLACE);
17   ensures
18    \mpi_reduce(*RBUF, 0, \mpi_comm_size, op, \lambda integer t; \mpi_on(\old(*RBUF), t));
19  disjoint behaviors; complete behaviors; */
20 int MPI_Allreduce(const void *sbuf, void *rbuf, int count, MPI_Datatype dt, MPI_Op op,
21                   MPI_Comm comm);
```

Fig. 5. The contract of the `MPI_Allreduce` function.

will be replaced by the output data." This option is 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 been discussed in the MPI Forum [17]. Our `MPI_Allreduce` contract declares, on line 9, that barrier synchronization occurs if `count` > 0, but other choices could be encoded.

In addition to the MPI collectives, we wrote contracts for a selection of user-defined collectives from the literature, including:

1. `exchange`: "ghost cell exchange" in 1d-diffusion solver [58]
2. `diff1dIter`: computes one time step in 1d-diffusion [58]
3. `dotProd`: parallel dot-product procedure from Hypre [23]
4. `matmat`: matrix multiplication using a block-striped decomposition [52]
5. `oddEvenIter`: odd-even parallel sorting algorithm [30,41].

We also implemented `cyc` of Fig. 2 in MPI with contracts.

Figure 6 shows the contract and the implementation for `dotProd`. The functions `hypre_MPI`$*$ 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 3) as well as the global result (line 13). Thus the contract is only

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

Fig. 6. The parallel `dotProd` function from Hypre [23], with contract.

valid if a real number model of arithmetic is used. This is a 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 [57] written in Java, because it provides a flexible intermediate verification language and it already has strong support for concurrency and MPI [44]. 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 Sect. 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 $\overline{P^f}$ (Sect. 2.4). This program has a driver that initializes the global variables and arguments for f to arbitrary values constrained only

function	states	prover	time(s)
g (cyc)	3,562	7	4
allreduceDR	7,390	15	5
f (cyc)	7,913	16	15
oddEvenIter	14,216	91	8
bcast	29,256	80	16
allreduce	14,174	64	16
dotProd	4,690	102	40
diff1dIter	4,762	130	100

function	states	prover	time(s)
allgather	14,606	356	32
reduce	118,278	54	46
scatter	125,900	394	69
gather	126,724	259	71
matmat	8,345	275	188
reduceScatterNC	264,215	259	214
reduceScatter	211,541	499	505
exchange	896,869	9659	478

Fig. 7. Verification performance for nprocs ≤ 5.

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);
```

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 of these implementations are more advanced: `allreduceDR` implements `MPI_Allreduce` using a double recursive algorithm; `reduceScatterNC` implements `MPI_Reduce_scatter` using an algorithm optimized for noncommutative reduction operations [12].

We applied our prototype to these collective implementations, using the contracts described in Sect. 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. For the CIVL configuration, we specified two theorem provers to be used in order: (1) CVC4 [8] 1.8, and (2) Z3 [49] 4.8.17, each with a timeout of two seconds.

Results are given in Fig. 7. For each problem, we give the number of states saved by CIVL, the number of calls to the theorem provers, and the total verification time in seconds, rounded up to the nearest second.

The times range from 4 seconds to 8 and a half 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 takes significantly longer 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.

5 Related Work

The ideas underlying code contracts originate in the work of Floyd on formal semantics [26], the proof system of Hoare [29], the specification system Larch [27], and Meyer's work on Eiffel [46,47]. Contract systems have been developed for many other languages, including Java [25,32,38], Ada [5], C# [7], and C [10,18].

Verification condition generation (VCG) [6,25,39] and symbolic execution [35,36,51] are two techniques used to verify that code conforms to a contract. *Extended static checking* is an influential VCG approach for Java [25,32,39]. Frama-C's WP plugin [9,18] is a VCG tool for ACSL-annotated C programs, based on the Why platform [24]. The Kiasan symbolic execution platform [20] has been applied to both JML and Spark contracts [11].

Several contract systems have been developed for shared memory concurrency. The VCC verifier [15,16,48] takes a contract approach, based on object invariants in addition to pre- and postconditions, to shared-memory concurrent C programs. VeriFast is a deductive verifier for multithreaded C and Java programs [31]. Its contract language is based on concurrent separation logic [14]. 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* [50,54,59] are used to describe communication protocols; various techniques verify an implementation conforms to a protocol. ParTypes [40] 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.

Several recent approaches to the verification of distributed systems work by automatically transforming a message-pasing program to a simplified form. One of these takes a program satisfying *symmetric nondeterminism* and converts it to a sequential program, proving deadlock-freedom and enabling verification of other safety properties [4]. Another does the same for a more general class of distributed programs, but requires user-provided information such as an "invariant action" and an abstraction function [37]. A related approach converts an asynchronous *round-based* message-passing program, with certain user-provided annotations, to a synchronous form [19]. This technique checks that each round is *communication-closed*, a concept that is similar to the idea of collective-style procedures. It is possible that these approaches could be adapted to verify that collective-style procedures in an MPI program conform to their contracts.

There are a number of correctness tools for MPI programs, including the dynamic model checkers ISP [60] and DAMPI [61], the static analysis tool MPI-Checker [22], and the dynamic analysis tool MUST [28]. 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 [3]. This approach is applied to a simple message-passing program in [42] 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 [56]. This works for whole programs which consume input and produce output, but it seems less applicable to individual collective procedures.

Assume-Guarantee Reasoning. [1,21,33,34] is another approach that decomposes along process boundaries. This is orthogonal to our approach, which decomposes along procedure boundaries.

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 [45, Sec. 5.12]. The new "neighborhood collectives" [45, 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.

Acknowledgements. We are grateful to the anonymous reviewers for providing valuable advice on the presentation of the results in this paper and for pointing out important related work. This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, under award DE-SC0021162. Support was also provided by U.S. National Science Foundation awards CCF-1955852, CCF-1319571, and CCF-2019309.

References

1. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. **17**(3), 507–535 (1995). https://doi.org/10.1145/203095.201069
2. Alur, R., Bouajjani, A., Esparza, J.: Model Checking Procedural Programs, chap. 17, pp. 541–572. Springer, Cham (2018).https://doi.org/10.1007/978-3-319-10575-8_17
3. Ashcroft, E.A.: Proving assertions about parallel programs. J. Comput. Syst. Sci. **10**(1), 110–135 (1975). https://doi.org/10.1016/S0022-0000(75)80018-3
4. Bakst, A., Gleissenthall, K.v., Kıcı, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. Proc. ACM Program. Lang. **1**(OOPSLA) (2017). https://doi.org/10.1145/3133934
5. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley, Boston (2003)
6. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2005). https://doi.org/10.1007/11804192_17

7. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011). https://doi.org/10.1145/1953122.1953145
8. Barrett, C., et al.: CVC4. In: International Conference on Computer Aided Verification, pp. 171–177. Springer, Heidelberg (2011). http://dl.acm.org/citation.cfm?id=2032305.2032319
9. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP plug-in manual: frama-C 22.0 (Titanium) (2020). https://frama-c.com/download/frama-c-wp-manual.pdf
10. Baudin, P., et al.: ACSL: ANSI/ISO C Specification Language, version 1.16 (2020). http://frama-c.com/download/acsl-1.16.pdf
11. Belt, J., Hatcliff, J., Robby, Chalin, P., Hardin, D., Deng, X.: Bakar Kiasan: flexible contract checking for critical systems using symbolic execution. In: Bobaru et al. [13], pp. 58–72.https://doi.org/10.1007/978-3-642-20398-5_6
12. Bernaschi, M., Iannello, G., Lauria, M.: Efficient implementation of reduce-scatter in MPI. In: Proceedings of the 10th Euromicro Conference on Parallel, Distributed and Network-Based Processing (EUROMICRO-PDP 2002), pp. 301–308. IEEE Computer Society, Washington (2002). http://dl.acm.org/citation.cfm?id=1895489.1895529
13. Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.): NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, 18–20 April 2011. Proceedings, LNCS, vol. 6617. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5
14. Brookes, S.: A semantics for concurrent separation logic. Theoret. Comput. Sci. **375**(1), 227–270 (2007). https://doi.org/10.1016/j.tcs.2006.12.034. Festschrift for John C. Reynolds's 70th Birthday
15. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
16. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 480–494. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_42
17. Community, M.: Collective Synchronization (2020). https://github.com/mpi-forum/mpi-issues/issues/257. Accessed 13 Aug 2021
18. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C—a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
19. Damian, A., Drăgoi, C., Militaru, A., Widder, J.: Communication-closed asynchronous protocols. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification,pp. 344–363. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_20
20. Deng, X., Lee, J., Robby: 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, pp. 157–166. IEEE Computer Society, USA (2006). https://doi.org/10.1109/ASE.2006.26
21. Dingel, J.: Computer-assisted assume/guarantee reasoning with VeriSoft. In: Proceedings of the 25th International Conference on Software Engineering (ICSE

2003), pp. 138–148. IEEE Computer Society, Washington (2003).https://doi.org/10.1109/ICSE.2003.1201195
22. Droste, A., Kuhn, M., Ludwig, T.: MPI-checker: static analysis for MPI. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM 2015), pp. 3:1–3:10. ACM, New York (2015). https://doi.org/10.1145/2833157.2833159
23. Falgout, R.D., Yang, U.M.: *hypre*: a library of high performance preconditioners. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) Computational Science—ICCS 2002, pp. 632–641. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47789-6_66
24. Filliâtre, J.C., Paskevich, A.: Why3: where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP 2013), pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
25. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, 17–19 June 2002, pp. 234–245. Association for Computing Machinery, New York (2002). https://doi.org/10.1145/512529.512558
26. Floyd, R.W.: Assigning meanings to programs. Math. Aspects Comput. Sci. **19**, 19–32 (1967)
27. Guttag, J.V., Horning, J.J., Wing, J.M.: The Larch family of specification languages. IEEE Softw. **2**(5), 24–36 (1985). https://doi.org/10.1109/MS.1985.231756
28. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI runtime error detection with MUST: advances in deadlock detection. In: Hollingsworth, J.K. (ed.) International Conference on High Performance Computing Networking, Storage and Analysis, SC 2012, Salt Lake City, 11–15 November 2012, pp. 30:1–30:11. IEEE Computer Society Press, Los Alamitos (2012). https://doi.org/10.1109/SC.2012.79
29. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
30. Huisman, M., Monahan, R., Müller, P., Mostowski, W., Ulbrich, M.: VerifyThis 2017: A Program Verification Competition. Tech. Rep. Karlsruhe Reports in Informatics 2017, 10, Karlsruhe Institute of Technology, Faculty of Informatics (2017). https://doi.org/10.5445/IR/1000077160
31. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 271–282. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/1926385.1926417
32. James, P.R., Chalin, P.: Faster and more complete extended static checking for the Java Modeling Language. J. Automat. Reason. **44**, 145–174 (2010). https://doi.org/10.1007/s10817-009-9134-9
33. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577
34. Jones, C.B.: Specification and design of (parallel) programs. In: Mason, R.E.A. (ed.) Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, 19–23 September 1983, pp. 321–332. North-Holland/IFIP, Newcastle University (1983)

35. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) 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, 7–11 April 2003, Proceedings. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_40
36. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
37. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020), pp. 227–242. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3385412.3385980
38. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006). https://doi.org/10.1145/1127878.1127884
39. Leino, K.R.M.: Extended static checking: a ten-year perspective. In: Wilhelm, R. (ed.) Informatics - 10 Years Back. 10 Years Ahead. LNCS, vol. 2000, pp. 157–175. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44577-3_11
40. López, H.A., et al.: Protocol-based verification of message-passing parallel programs. In: Aldrich, J., Eugster, P. (eds.) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Part of SPLASH 2015, Pittsburgh, 25–30 October 2015, pp. 280–298. ACM, New York (2015). https://doi.org/10.1145/2814270.2814302
41. Luo, Z., Siegel, S.F.: Symbolic execution and deductive verification approaches to VerifyThis 2017 challenges. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), Proceedings, Part II: Verification. LNCS, vol. 11245, pp. 160–178. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-030-03421-4_12
42. Luo, Z., Siegel, S.F.: Towards deductive verification of message-passing parallel programs. In: Laguna, I., Rubio-González, C. (eds.) 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness), pp. 59–68. IEEE (2018). https://doi.org/10.1109/Correctness.2018.00012
43. Luo, Z., Siegel, S.F.: Artifact of "Collective contracts for message-passing parallel programs" (2024). https://doi.org/10.5281/zenodo.10938740
44. Luo, Z., Zheng, M., Siegel, S.F.: Verification of MPI programs using CIVL. In: Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI 2017), pp. 6:1–6:11. ACM, New York (2017). https://doi.org/10.1145/3127024.3127032
45. Message-Passing Interface Forum. MPI: A Message-Passing Interface standard, version 3.1 (2015). https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
46. Meyer, B.: Applying "Design by Contract." IEEE Comput. **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
47. Meyer, B., Nerson, J.M., Matsuo, M.: EIFFEL: object-oriented design for software engineering. In: Nichols, H.K., Simpson, D. (eds.) ESEC 1987. LNCS, vol. 289, pp. 221–229. Springer, Heidelberg (1987). https://doi.org/10.1007/BFb0022115
48. Moskal, M.: Verifying functional correctness of C programs with VCC. In: Bobaru et al. [13], pp. 56–57 (2011). https://doi.org/10.1007/978-3-642-20398-5_5

49. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
50. Ng, N., Yoshida, N., Honda, K.: Multiparty session C: safe parallel programming with message optimisation. In: Furia, C.A., Nanz, S. (eds.) Objects, Models, Components, Patterns. LNCS, vol. 7304, pp. 202–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30561-0_15
51. Păsăreanu, C., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Techol. Transf. **11**(4), 339–353 (2009). https://doi.org/10.1007/s10009-009-0118-1
52. Quinn, M.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill (2004)
53. Romano, P.K., Horelik, N.E., Herman, B.R., Nelson, A.G., Forget, B., Smith, K.: OpenMC: a state-of-the-art Monte Carlo code for research and development. Ann. Nucl. Energy **82**, 90–97 (2015). https://doi.org/10.1016/j.anucene.2014.07.048
54. Scalas, A., Yoshida, N., Benussi, E.: Verifying message-passing programs with dependent behavioural types. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019), pp. 502–516. Association for Computing Machinery, New York (2019).https://doi.org/10.1145/3314221.3322484
55. Siegel, S.F., Avrunin, G.S.: Modeling wildcard-free MPI programs for verification. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2005), pp. 95–106. Association for Computing Machinery, New York (2005).https://doi.org/10.1145/1065944.1065957
56. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM Trans. Softw. Eng. Methodol. **17**(2), 1–34 (2008). https://doi.org/10.1145/1348250.1348256
57. Siegel, S.F., et al.: CIVL: the Concurrency Intermediate Verification Language. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015), pp. 61:1–61:12. ACM, New York (2015). http://doi.acm.org/10.1145/2807591.2807635
58. Siegel, S.F., Zirkel, T.K.: FEVS: a functional equivalence verification suite for high performance scientific computing. Math. Comput. Sci. **5**(4), 427–435 (2011). https://doi.org/10.1007/s11786-011-0101-6
59. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Maritsas, D., Philokyprou, G., Theodoridis, S. (eds.) PARLE 1994 Parallel Architectures and Languages Europe. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58184-7_118
60. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_9
61. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., Supinski, B.R.d., Schulz, M., Bronevetsky, G.: 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 2010), pp. 1–10. IEEE Computer Society, Washington (2010). https://doi.org/10.1109/SC.2010.7
62. Yang, U., Falgout, R., Park, J.: Algebraic Multigrid Benchmark, Version 00 (2017). https://www.osti.gov//servlets/purl/1389816

Distributed Systems

mypyvy: A Research Platform for Verification of Transition Systems in First-Order Logic

James R. Wilcox[1], Yotam M. Y. Feldman[2], Oded Padon[3], and Sharon Shoham[2(✉)]

[1] University of Washington, Seattle, USA
[2] Tel Aviv University, Tel Aviv-Yafo, Israel
sharon.shoham@gmail.com
[3] VMware Research, Palo Alto, USA

Abstract. `mypyvy` is an open-source tool for specifying transition systems in first-order logic and reasoning about them. `mypyvy` is particularly suitable for analyzing and verifying distributed algorithms. `mypyvy` implements key functionalities needed for safety verification and provides flexible interfaces that make it useful not only as a verification tool but also as a research platform for developing verification techniques, and in particular invariant inference algorithms. Moreover, the `mypyvy` input language is both simple and general, and the `mypyvy` repository includes several dozen benchmarks—transition systems that model a wide range of distributed and concurrent algorithms. `mypyvy` has supported several recent research efforts that benefited from its development framework and benchmark set.

1 Introduction

`mypyvy` is an open-source[1] research platform for automated reasoning about symbolic transition systems expressed in first-order logic. A chief design goal for `mypyvy` is to lower the barrier to entry for developing new techniques for solver-aided analysis and verification of transition systems. As a result, `mypyvy`'s modeling language is simple and close to the underlying logical foundation, and the tool is designed as a collection of reusable components, making it easy to experiment with new verification techniques.

The main application domain of `mypyvy` is verification of complex distributed algorithms. Following prior work [32,33], transition systems in `mypyvy` are expressed in uninterpreted first-order logic (i.e., without theories). Using uninterpreted first-order logic is motivated by the experience that solvers often struggle when theories (e.g., arithmetic, arrays, or algebraic data types) are combined with quantifiers. Quantifiers are essential for describing distributed algorithms (e.g., to state properties about all messages in the network), but theories can often be avoided, yielding improved automation.

[1] https://github.com/wilcoxjay/mypyvy.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14682, pp. 71–85, 2024.
https://doi.org/10.1007/978-3-031-65630-9_4

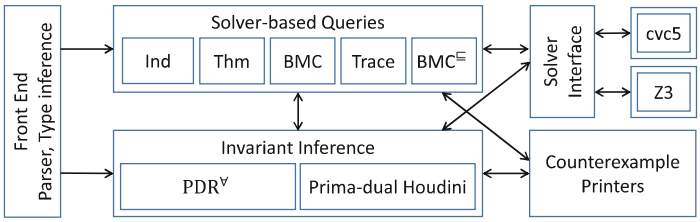

Fig. 1. Main components of `mypyvy`.

`mypyvy` consists of a language for expressing transition systems directly as logical formulas but in a convenient manner (Sect. 2), a tool for reasoning about such systems, and a collection of benchmarks accumulated over the last few years (Sect. 2.1). Figure 1 depicts `mypyvy`'s components, which are divided to solver-based queries (Sect. 3) and invariant inference algorithms (Sect. 4). Solver-based queries such as inductiveness checking and bounded model checking are answered by translating them into satisfiability checks that are sent to external first-order solvers. These queries are used as basic building blocks for developing invariant inference algorithms. `mypyvy` includes an implementation of two such algorithms: PDR$^{\forall}$ [21] and Primal-dual Houdini [34]. `mypyvy`'s internals are designed with the goal of making it easy to build on (Sect. 5). `mypyvy` interacts with multiple solvers, and currently supports Z3 [13] and cvc5 [2]. To present counterexamples (states, transitions, or traces) in a user-friendly way, `mypyvy` supports custom printers that simplify and improve readability of counterexamples.

`mypyvy` is not just the sum of the analyses currently available; it is a platform for doing research in automated verification. Several projects (including ongoing ones) use the `mypyvy` foundation and benchmark suite to build new invariant inference techniques, user interfaces for verification and exploration, and, most recently, liveness verification techniques (Sect. 6).

`mypyvy`'s first-order modeling is inspired by Ivy [30,33], which promoted the idea of modeling distributed systems in the EPR decidable fragment of first-order logic. Ivy includes a rich and modular high-level imperative specification language, as well as mechanisms for creating executable implementations, specification-based testing, liveness verification, and more. As a result, Ivy's syntax, semantics, and code base are more complicated than what would be ideal for enabling rapid exploration of new techniques. In contrast, `mypyvy`'s focus on transition systems, with a simple syntax and semantics, makes it especially suited for enabling verification research.[2] Moreover, `mypyvy`'s code base is intentionally designed, documented, and typed (using Python's support for type annotations), to make it easy to build on and extend.

[2] There are current open-source efforts to automatically translate Ivy to `mypyvy` [9,36], which would allow Ivy users to benefit from `mypyvy`'s algorithms.

Broadly, mypyvy has three target audiences:

1. Researchers interested in modeling and verifying distributed algorithms. mypyvy offers a user-friendly input language, several queries that assist in developing models of distributed algorithms, readable counterexamples, and access to a variety of automatic verification algorithms.
2. Researchers developing verification techniques, and invariant inference in particular. mypyvy offers a starting point for implementing new algorithms on top of a developer-friendly code base. mypyvy includes many useful building blocks, and has already been successfully used in several research projects.
3. Researchers looking for benchmarks for various verification tasks. mypyvy includes a significant set of transition systems (and their invariants), which can serve as benchmarks for invariant inference or other verification tasks.

2 Modeling Language

We present mypyvy through a simple example of modeling and analyzing a toy consensus protocol.[3] To get started, the user first expresses a transition system in mypyvy's input language, which is a convenient syntax for (many-sorted) uninterpreted first-order logic. A mypyvy model of the toy consensus protocol is shown in Fig. 2. In this protocol, each node *votes* for a single value, and once a majority or *quorum* of nodes vote for the same value a *decision* takes place. Because majorities intersect, the protocol ensures that at most one value is decided on. Modeling an algorithm or system of interest as a transition system in first-order logic may involve some abstraction, e.g., modeling majorities as abstract quorums such that every two quorums intersect [31].

States. The first step is to choose the types over which the transition system is defined. In the fashion of first-order logic, the basic types are *uninterpreted sorts* (mypyvy does not use SMT theories). In the example, we use the sorts node, value, and quorum to represent the nodes that participate in the distributed system, the values they choose from, and the sets of nodes that suffice for a decision (we abstract majorities following [4,32]). The state of the system is modeled by variables which can be *constants* (individuals), *relations*, or *functions*, whose domains are constructed from the aforementioned sorts. Each state variable is either `immutable`, which means it does not change throughout an execution of the system, or `mutable`, which means it may change with each transition. In the example, all state variables are relations. An immutable relation member denotes membership of a node in a quorum. The other relations are mutable: v records votes of nodes for values, b tracks which nodes already voted, and d records decisions.

[3] While not useful as a consensus protocol, this example does illustrate important aspects from proofs of complex, widely used consensus protocols like Paxos [25].

```
1  sort node
2  sort value
3  sort quorum
4
5  immutable relation member(node, quorum)
6  axiom forall Q1, Q2. exists N.
7           member(N, Q1) & member(N, Q2)
8
9  mutable relation v(node, value)
10 mutable relation b(node)
11 mutable relation d(value)
12
13 init forall N, V. !v(N,V)
14 init forall N. !b(N)
15 init forall V. !d(V)
16
17 transition vote(n: node, x: value)
18   modifies v, b
19   !b(n) &
20   (forall N, V.
21     v'(N, V) <-> v(N, V) | (N = n & V = x)) &
22   (forall N. b'(N) <-> b(N) | N = n)
23
24 transition decide(x: value)
25   modifies d
26   (exists Q. forall N. member(N, Q) -> v(N, x)) &
27   (forall V. d'(V) <-> d(V) | V = x)
28
29 safety [agreement] forall X, Y. d(X) & d(Y) -> X = Y
30 invariant [decision_quorums] forall X. d(X) ->
31   exists Q. forall N. member(N, Q) -> v(N, X)
32 invariant [unique_votes] forall N, X, Y.
33       v(N, X) & v(N, Y) -> X = Y
34 invariant [voting_bit] forall N, X. v(N, X) -> b(N)
35
36 zerostate theorem forall Q. exists N. member(N, Q)
37 onestate theorem unique_votes & decision_quorums -> agreement
38 twostate theorem forall N, X.
39       voting_bit & vote(N, X) -> voting_bit'
40
41 unsat trace {
42   vote
43   vote
44   vote
45   decide
46   decide
47   assert !safety
48 }
49
50 sat trace {
51   any transition
52   assert exists N, V. v(N,V)
53   decide
54   assert exists V. d(V)
55 }
```

Fig. 2. The toy consensus example in mypyvy.

```
> mypyvy verify consensus.pyv

checking init:
 implies invariant agreement..ok.
checking transition vote:
 preserves invariant agreement..ok.
checking transition decide:
 preserves invariant agreement..no!

counterexample:
  universes:
    sort node (1): node0
    sort quorum (1): quorum0
    sort value (2): value0 value1

  immutable:
    member(node0,quorum0)

  state 0:
    d(value1)
    v(node0,value0)

  state 1:
    d(value0)
    d(value1)
    v(node0,value0)

error consensus.pyv: invariant
agreement is not preserved by
transition decide
```

Fig. 3. A counterexample to induction (CTI) for the toy consensus protocol's safety property without additional invariants.

Axioms. mypyvy allows the user to define a "background theory" over the immutable symbols, which restricts the state space, via **axiom** declarations. In the example, the property that any two quorums intersect (abstracting majorities) is expressed as an axiom for the member relation (line 6). (The sorts of quantified variables are omitted in formulas since mypyvy infers them automatically.) Another common background theory that is useful when modeling distributed protocols in mypyvy is a total order, which can be used to abstract the natural numbers in first-order logic (e.g., to model rounds or indices).

Initial States. The initial states are defined as those that satisfy all `init` declarations. In the example, these declare that all mutable relations are initially empty (lines 13 to 15).

Transitions. The transitions of the system are expressed by `transition` declarations. The semantics is that each transition executes atomically and can modify the system's state. Transitions can have parameters, which are local variables that are assigned nondeterministically whenever the transition is executed. The example has two transitions: vote(n, x) and decide(x) (lines 17 to 27). An important design choice of `mypyvy` is that the user specifies transitions by explicitly writing logical formulas. Each transition is defined over two states: variables in the usual notation refer to the state *before* the transition is applied (*pre-state*), and primed variables refer to the state *after* the transition (*post-state*). Pre-conditions are encoded as conjuncts in the formula about the pre-state; for example, vote requires that the node has not already voted by specifying !b(n). Post-conditions are encoded as conjuncts about the post-state, relating it to the pre-state; for example, vote specifies that the relation b is updated to include exactly the same nodes as before in addition to n. Writing transitions directly through formulas offers great flexibility, but in order to write these formulas succinctly, a transition starts with a `modifies` clause that declares which mutable state variables are changed by it. For any mutable state component *not* in the modifies clause, `mypyvy` implicitly adds a conjunct encoding that the component does not change. Formally, the transition relation is the disjunction of the formulas from each of the transitions, where parameters are existentially quantified.

Safety. Finally, the user may specify safety properties using first-order formulas in `safety` declarations. The agreement safety property in the example (line 29) states that at most one value is decided. A safety property holds if it is satisfied by every state that is reachable from an initial state via a sequence of transitions.

2.1 Benchmarks

The `mypyvy` repository includes over 30 transition systems collected over the years. Some of these were translated from Ivy, while others were directly modeled in `mypyvy`. The benchmarks model a variety of distributed and concurrent algorithms, including consensus algorithms, networking algorithms, and cache coherence protocols. The variety of benchmarks, which also vary in complexity, is useful for evaluating and experimenting with new verification techniques. Additional details can be found in the paper's artifact [39].

3 Satisfiability-Based Queries

Once a transition system is specified, `mypyvy` supports several satisfiability-based queries over it, which are directly translated to satisfiability checks and handed off to solvers (currently Z3 [13] and cvc5 [2] are supported). These queries are

useful building blocks for developing more advanced solver-aided algorithms, and for users who are interested in analyzing specific systems (especially during the model development process). For most queries, `mypyvy` provides counterexamples based on satisfying models obtained from solvers. And while solvers are not guaranteed to terminate, `mypyvy` makes it easy to follow the EPR fragment restrictions, which ensures termination.

3.1 Queries

Inductiveness Checking. `mypyvy` allows the user to add `invariant` declarations to prove safety by induction. These are first-order formulas, whose conjunction (together with the safety properties) forms a candidate inductive invariant. Figure 2 lists three supporting invariants (lines 30 to 34). The most common query in `mypyvy` is to check if the candidate invariant is inductive. When translating an inductiveness check to the solver, `mypyvy` splits it into one solver query per (transition, invariant) pair. In our experience, splitting the disjunction outside the solver improves performance and reliability, and, best of all, improves transparency for the user when one of the cases is more problematic (e.g., takes a long time).

Theorems. In addition to invariants, which are meant to hold in all reachable states of the transition system, `mypyvy` supports checking `theorem` declarations, which specify first-order formulas that are expected to be valid modulo the background theory (i.e., axioms). `zerostate` theorems refer to immutable state variables only, `onestate` theorems may refer to the mutable state variables as well, and `twostate` theorems involve two states, similarly to `transition` declarations. In the toy consensus example, a `zerostate` theorem (line 36) is used to state that quorums cannot be empty (follows from the quorum intersection axiom); a `onestate` theorem (line 37) is used to state that, given the background theory, the `unique_votes` and `decision_quorums` invariants imply the `agreement` safety property; and a `twostate` theorem (line 39) is used to check that the `voting_bit` invariant is preserved by the vote transition.

Bounded Model Checking (BMC). It is often useful to explore (un)reachability of a safety violation via BMC. Given a transition system and a safety property, BMC asks, "Is there a counterexample trace with $\leq k$ transitions?" BMC is implemented in the usual way, by unrolling the transition relation.

Trace Queries. Trace queries allow the user to explore the possible executions of the system in a more targeted way than BMC. This is useful both when the user is interested only in specific scenarios, and when BMC does not scale to sufficient depth. As an illustration, in a model of a distributed system with many protocol steps, BMC may only reasonably scale to a small depth, say 5 transitions, but many interesting behaviors of the system may not occur until at least 10 or 15 transitions. In Fig. 2, lines 41 to 48 show a query for the nonexistence of an execution trace that starts with three vote transitions, followed by two decide

transitions, and then reaches a safety violation. `mypyvy` translates such a query to a first-order formula that is checked for unsatisfiability.

As a complement of trace queries that are expected to be unsatisfiable (specified by the `unsat` keyword), it is also useful to make `sat` trace queries that are expected to be satisfiable, demonstrating that some behaviors are indeed possible.[4] For example, lines 50 to 54 show a query expecting the existence of a trace that starts with any transition after which there exists a vote, followed by a decide transition after which there exists a decision. (That is possible when the number of nodes is 1.) Such satisfiable trace queries are especially useful for detecting *vacuity bugs*, where, due to a modeling error, some transitions mistakenly cannot execute, potentially making the system erroneously safe.

Relaxed Bounded Model Checking ($BMC^{\sqsubseteq}$). So far we discussed *concrete* traces. `mypyvy` can also search for *relaxed* counterexample traces of a bounded depth. A relaxed trace consists of a sequence of interleaved transitions and "relaxation steps", where some elements get deleted from the structure. As shown in [21], a relaxed counterexample trace that starts at an initial state and ends in a safety violation *proves* that there is no universally quantified inductive invariant that implies safety. This is the case in the toy consensus example—a relaxed counterexample trace found by `mypyvy` for this example is provided in the paper's artifact [39]. The key to implementing relaxed BMC queries is encoding universe reduction between states. `mypyvy` does so by introducing a mutable unary relation active for each sort and using it as a guard in every quantifier, effectively restricting the universe in each state to the "active" part. Relaxation steps are then modeled by adding a relax transition where each active relation in the post-state is a subset of the corresponding one in the pre-state (expressed as a universally quantified formula); all other state variables are unmodified over the active part. Finally, a relaxed BMC query is encoded similarly to a BMC query (with the added relax transitions), except that, due to the use of different active universes, the axioms are asserted not only at the beginning of the trace but also after every (relaxation) step, together with assertions requiring that the active universe contains the constants and is closed under functions.

3.2 Counterexamples

When a query fails (except for a `sat trace` query), it is because the formula sent to the solver was satisfiable. In such cases, `mypyvy` obtains a model from the solver and displays a *counterexample*—which can be a state, a transition, or a trace, depending on the failing query. For example, when inductiveness checking fails, it returns either a 1-state model demonstrating a violation of safety at an initial state, or a 2-state model demonstrating a counterexample to induction (CTI). As

[4] `mypyvy` uses solver queries to generate executions of the transition system. A solver is needed due to `mypyvy`'s flexible and abstract modeling language. More imperative modeling languages, e.g. that of Ivy, admit execution/simulation without solvers, which can be useful for invariant inference as well [40,42]. Such simulation can also be implemented for a fragment of `mypyvy`'s language.

another example, when BMC finds an execution that violates safety, it returns a k-state model providing a counterexample trace. Figure 3 shows a CTI (2-state model) for the toy consensus protocol when the invariants supporting the safety property are omitted. In general, `mypyvy` displays a k-state model by first listing the universe of each sort and the interpretations of the immutable symbols (`member` in our example). Then, for each of the k states, the interpretations of the mutable symbols in that state are printed. For relations, by default `mypyvy` only prints positive literals, i.e., the tuples that are in the relation.

Annotations, Plugins, and Custom Printers. In some cases, the default counterexample printing of `mypyvy` is not as readable as it could be. For example, if one of the sorts in the transition system is totally ordered (using a binary relation and suitable axioms), it would make sense to name the elements of that sort according to the total order. To improve the readability of counterexamples, `mypyvy` supports custom formatting via *printer plugins* and *annotations*. Every declaration in `mypyvy` can be tagged with *annotations*, which have no inherent meaning, but can be detected by plugins, e.g., to cause things to be printed differently. For example, the declaration `sort round @printed_by(ordered_by_printer, le)` invokes the `ordered_by_printer` plugin and tells `mypyvy` that the sort `round` should be printed in the order given by the `le` relation. `mypyvy` provides several other custom printers, including one for printing sorts that represent sets of elements coming from another sort. Users can also implement their own custom printing plugins in Python.

`mypyvy` also supports a handful of other annotations. `@no_print` instructs `mypyvy` not to print a sort, relation, constant, or function at all, which can be useful either because of a custom printer for another symbol, or temporarily because the model is large and the symbol is irrelevant to the current debugging session. `@no_minimize` is used to instruct `mypyvy`'s model minimizer not to minimize elements of a certain sort or relation. The annotation framework is extensible, and we expect more uses for it to come up.

3.3 Decidability and Finite Counterexamples via EPR

In general, `mypyvy` does not restrict the quantifier structure used in formulas, nor the signatures of state variables. As a result, the first-order formulas that encode different queries in `mypyvy` are not guaranteed to reside in any decidable fragment and solvers may diverge. However, a common practice when working with `mypyvy` is to use the effectively propositional (EPR) [35,37] fragment of first-order logic, which imposes certain restrictions on functions and quantifier alternations. To encode a system in EPR (i.e., ensure that formulas generated for all queries are in EPR), the user can rely on recently developed methodologies [32,38]. For example, the toy consensus example of Fig. 2 is in EPR. Satisfiability of EPR is decidable, and reliably checked by solvers. EPR enjoys a small-model property, which implies queries have finite counterexamples (if any). Solver reliability and finite counterexamples are key enablers for more advanced algorithms (e.g., invariant inference) that make thousands of solver queries and