Contracts for Message-Passing Programs

Ziqing Luo, Advisor: Stephen F. Siegel
University of Delaware

Funding for verifying MPI-based C programs is supported by U.S. National Science Foundation under the award, CCF-1319571.

1. Abstract

Automatic formal verification of parallel programs is expensive because of the state explosion problem. A potential way to overcome this problem is to **divide and conquer**. If program components, such as procedures, could be verified independently, the verification problem could be decomposed into smaller, more manageable tasks. There is a well-known way to do this for **sequential programs** using code contracts. Many programming languages adopt code contracts to improve the robustness of applications. A variety of verification tools are developed to verify that programs satisfy their contracts. The contract approach has been extended to deal with shared-memory concurrency as well.

We are interested in developing a contract system for collective-style MPI functions. The system provides a formal **specification language** which will be used by programmers to specify the behaviors of, and decompose a program into, collective-style functions. The system verifies a program by verifying each decomposed function independently.

2. Traditional Procedure Contracts

- Procedure contracts were introduced in [1]. Contracts are based on the notion of a Hoare Triple [2]:

  {P} S {Q}

- Precondition P and postcondition Q are predicates. S is a procedure. The contract means "if P is true before S is executed, then Q is true after S is executed".

Procedure Contracts and Program Correctness

- Contracts enable **modular verification**. To verify that a program is correct, each procedure p can be verified independently. When verifying p, one assumes that all procedures q called by p satisfy their contracts. So only the contract of q (and not its implementation) is used when verifying p. If one verifies every procedure in this way, the program is correct.

ANSI/ISO C Specification Language (ACSL)

- ACSL [3] is an annotation-based specification language for C programs. It can be used to specify function contracts as well as other concepts, such as loop invariants. The following is a simple ACSL example:

```
int result;
/*@ requires \valid(a) && \valid(b);
  @ assigns result;
  @ ensures  \valid(a) && \valid(b);
  @ behavior eq:
  @   assumes *a == *b;
  @   ensures result == 1;
  @ behavior neq:
  @   assumes *a != *b;
  @   ensures result == 0;
  */
void equals(int *a, int *b) {
  result = *a == *b ? 1 : 0;
}
```

requires clause: specifies preconditions.
ensures clause: specifies postconditions.
assigns clause: any variable not listed has same value before and after.
behavior: a contract component activated only when the assumed expression holds.
\valid: predicate indicating a pointer can be safely dereferenced.

3. Collective Contract Semantics

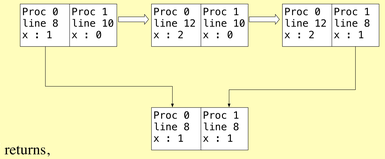
- A collective-style function will be called by all processes in a communicator. Messages sent from the function will be received within that function.
- A collective contract allows the user to specify behavior in an intuitive "bulk-synchronous" style, even when executions are not necessarily bulk-synchronous.
- When specifying the behavior of a collective-style function, the user can conceptually assume that all processes are calling the function simultaneously.
- A collective contract will be interpreted at two *collective states*. A collective pre-state and a collective post-state.
- Conceptually, a collective state is formed by taking a snapshot of each process state as control passes through a certain location. The snapshots are then joined together to form the collective state.

- Consider procedure `f` below:

```
int x = 0;
void f() {
  if (rank == 0) {
    x = 1;
    send(x, 1); // send x to proc 1
  } else if (rank == 1)
    recv(x, 0); // recv x from proc 0
}
int main() {
  f();
  x = 2;
}
```

- An intuitive specification might state: "when `f` returns, processes 0 and 1 have the same value for `x`." This is true — as long as "when `f` returns" is interpreted in the collective post-state.

Execution States



Collective Post-State

4. Collective Contract Language

- An extension of ACSL.
- An **`\mpi_collective`** specifies a *collective contract*. A collective contract is associated with a communication universe, which is identified by a communicator and a constant indicating for point-to-point or collective communication mode.

```
<contract clauses>
\mpi_collective(comm, mode):
  <contract clauses>
```

- A collective contract **implicitly** asserts: If there is no buffered messages in the collective pre-state, there is no buffered messages in the collective post-state.

New Primitives Added to ACSL

New Terms

`\on`(e, *rank*):	the value of expression e on a process of rank *rank*.
`\mpi_comm_size`:	a constant denoting the number of processes in the communicator.
`\mpi_comm_rank`:	a constant denoting the rank of this process in the communicator.
`\mpi_extent`(*datatype*):	the extent of an MPI datatype.
`\mpi_offset`(buf, *count*, *datatype*):	a pointer value: `(char *)`buf`+`*count*`*\mpi_extent`(*datatype*).
`\mpi_region`(buf, *count*, *datatype*):	the memory region containing a sequence of *count* objects of a C type corresponding to *datatype* whose first element is pointed to by buf.

New Predicates

`\mpi_agree`(e):	a predicate asserting that all processes have the same value for expression e.
`\mpi_valid`(buf, *count*, *datatype*):	a predicate asserting that the pointer buf points to the first element of a sequence of *count* objects of a C type corresponding to *datatype*.
`\mpi_equals`(*region0*, *region1*):	A predicate asserting that the two memory regions *region0* and *region1* have the exact same value.

New Clause

`waitsfor` R:	a clause expressing that a process cannot exit the function without all processes listed in R entering the function first.

5.Example: Ghost cell exchange

```
int left, right, nxl;
double * u;

/*@ \mpi_collective(MPI_COMM_WORLD, P2P):
  @   requires nxl > 0 && \mpi_valid(u, nxl + 2, MPI_DOUBLE);
  @   requires left != \mpi_comm_rank && right != \mpi_comm_rank;
  @   requires left == MPI_PROC_NULL || (0 <= left && left < \mpi_comm_size);
  @   requires right == MPI_PROC_NULL || (0 <= right && right < \mpi_comm_size);
  @   ensures  \mpi_valid(u, nxl + 2, MPI_DOUBLE);
  @   behavior hasLeftNeighbor:
  @     assumes  left != MPI_PROC_NULL;
  @     requires \mpi_comm_rank == \on(left, right);
  @     assigns  \mpi_region(u, 1, MPI_DOUBLE);
  @     ensures  u[0] == \on(left, u[nxl]);
  @     waitsfor left;
  @   behavior hasRightNeighbor:
  @     assumes  right != MPI_PROC_NULL;
  @     requires \mpi_comm_rank == \on(right, left);
  @     assigns  \mpi_region(&u[nxl+1], 1, MPI_DOUBLE);
  @     ensures  u[nxl + 1] == \on(right, u[1]);
  @     waitsfor right;
  @*/
void exchange_ghost_cells() {
  MPI_Sendrecv(&u[1], 1, MPI_DOUBLE, left, 0, &u[nxl+1], 1, MPI_DOUBLE,
               right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Sendrecv(&u[nxl], 1, MPI_DOUBLE, right, 0, &u[0], 1, MPI_DOUBLE,
               left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

a collective-style function using point-to-point communication in `MPI_COMM_WORLD`

`u` must point to a region containing `nxl+2` doubles.

my rank in `MPI_COMM_WORLD`

I am my right neighbor's left neighbor.

My right ghost cell holds the value of my right neighbor's left-most non-ghost cell.

I cannot exit until my right neighbor enters the function.

Collective-style procedures can call point-to-point or other non-collective functions, but they cannot be further decomposed.

I cannot be a neighbor of myself.

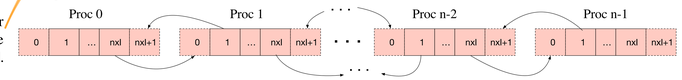
the size of `MPI_COMM_WORLD`

the value of variable `right` on my left neighbor

This contract section will be activated only if `right != MPI_PROC_NULL;`

I will not change values of memory locations other than the two ghost cells.

The precondition is the weakest assumption needed to prove the postcondition. Stronger assumptions, like "the rank of my left neighbor is my rank minus one" are not required, and may not alway hold.



References:

[1] B. Meyer. Applying 'Design by Contract'. *IEEE Computer*, 25, 10 (Oct 1992), 40 – 51.

[2] C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM. ACM*, 12, 10 (October 1969), 576 – 580.

[3] Baudin, et al. ACSL: ANSI C Specification Language. version 1.12 (2017), https://frama-c.com/acsl.html.

[4] Siegel, et al. CIVL: the Concurrency Intermediate Verification Language. In *Proceeding of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '15), ACM, New York, USA, Article 61, 12 pages.

6. Verification of Collective Contracts

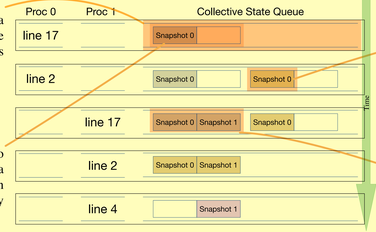
- A verification tool based on CIVL [4].
- Model checking and symbolic execution.
- automatically transforms a program to a modular verification form and feeds the result to the verifier

```
/*@ function contracts
 */
function g() {
  // body-of-g ...
}

function h() {
  // body-of-h ...
}

/*@ function contracts
 */
function f() {
  g();
  h();
}

main() { ... }
```

Code Transform

```
function g() {
  collective_assert (precondition);
  ... // refresh memory locations
  collective_assume (postcondition);
}

function h() {
  // body-of-h ...
}

function f() {
  g();
  h();
}

function driver_f() {
  collective_assume (precondition);
  f();
  collective_assert (postcondition);
}

main() { driver_f(); }
```

CIVL maintains a queue of collective states



Process snapshots are inserted

The first process to enter/exit a collective function creates a new entry

The last process to enter/exit a collective function dequeues a complete entry