Artifact Evaluation Abstract

Collective Contracts for Message-Passing Parallel Programs

Ziqing Luo and Stephen F. Siegel

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

1 Artifact Information

- Package DOI: https://dx.doi.org/10.5281/zenodo.10938740
- SHA256 check sum:
  68db25b3541fdf15656c8aa17ba99b450a7569395622279b6358daeb2c2c7d4d

2 Artifact Summary

The paper introduces a contract language for C/MPI programs, a method for verifying such programs, and a prototype tool that implements that method. The paper also presents the results of applying the prototype to a set of examples. Each example is a C/MPI program with contracts. The artifact package includes source code of the prototype and those examples. Reviewers can build the tool from the source code, reproduce our verification results, and apply the tool to their own examples.

The package contains a Docker image, in which all the dependencies have been installed. To run the image, type the following commands in the command line after installing Docker:

```
docker load <mpi_contract_docker_image.tar.gz
docker run -it mpi_contract
```

All the materials are placed under `/workspace`. The `README` file explains what is included in the image, how to build the prototype from source, and how to run the experiments. We made the process simple through `Makefiles`.

The prototype utilizes two SMT solvers: Z3 and CVC4. Configurations for the two solvers are specified in the file `/root/.sarl` and these configurations are similar to those used in the reported experiment. Reviewers are free to change timeouts or the order of the two solvers in the file. Our tool invokes the second prover only after the first prover fails to solve a query or times out. Altering the configuration will affect the verification time.

3 Badge Claim

We claim all three badges: Artifact Available, Artifact Functional, and Artifact Reusable.

4 Creating New Examples

In this part, we walk through the process of creating a new C/MPI example of a contracted collective-style function from a simple sequential one, as a tutorial for reviewers to test our artifact for reusability. We encourage the reviewers to use our paper as a reference for the syntax and semantics of the contract language.

```
#pragma CIVL ACSL // tells CIVL to parse the annotations as ACSL
/*@
  requires n > 0;
  requires \valid(a + (0 .. n-1)) && \valid(b + (0 .. n-1));
  assigns  \nothing;
  ensures  \forall int i; 0 <= i < n ==> \result >= a[i] + b[i];
  ensures  \exists int i; 0 <= i < n && \result == a[i] + b[i];
 */
int maxSum(int * a, int * b, int n) {
  int max = a[0] + b[0];

  /*@ loop invariant 1 <= i <= n;
      loop invariant \forall int j; 0 <= j < i ==> max >= a[j] + b[j];
      loop invariant \exists int j; 0 <= j < i && max == a[j] + b[j];
      loop assigns max, i;
   */
  for (int i = 1; i < n; i++) {
    if (max < a[i] + b[i])
      max = a[i] + b[i];
  }
  return max;
}
```

Fig. 1. File `maxSum.c`: specification and definition of sequential function `maxSum`

We start with the simple sequential C function `maxSum` in Figure 1. This function takes two buffers of `int`s, assuming they have the same size $n \geq 1$. It computes the maximum value of the set $\{a[i] + b[i] \mid 0 \leq i < n\}$.

The specification of this function occurs in formatted comments in the ACSL language, which our paper extends. Refer to the ACSL manual [1] for more information.

The specification includes *loop invariants*. This is because of the presence of a *for* loop. Without it, our verification tool would require n to be bounded. (Loop invariants are a standard part of the ACSL specification language.)

The formatted comment preceding the function definition is the function contract. It consists of a precondition, a frame condition, and a postcondition.

Preconditions are specified using the `requires` clause. They state an assumption about the inputs to the function. In our example, we assume the two input buffers are valid to read and they do have the same positive size n.

Frame conditions are specified with `assigns` clauses. They specify an upper bound on the set of memory locations that will be modified by the function. For `maxSum`, the function modifies no memory location that is visible to its callers.

Postconditions are specified with `ensures` clauses. They describe the output of the function when it returns. For the example, the function should guarantee that the returned value is no less than $a[i] + b[i]$ for any `i` in $[0, n - 1]$, and that it in fact equals $a[i] + b[i]$ for some i. The built-in keyword `\result` represents the returned value.

If this code is saved in file `maxSum.c` then it can be verified using our tool as follows:

```
civl verify -input_mpi_nprocs=1 -mpiContract=maxSum -loop maxSum.c
```

We specified 1 MPI process because the program is sequential. The option `-loop` is needed, as loop invariants are used.

With a verified `maxSum`, we now consider how to parallelize it using MPI. In MPI, there is no shared memory; each process may be thought of as a separate computer running the same program. (Communication takes place through calls to message-passing functions.) We assume that each process has a complete copy of the arrays a and b. In the parallel version, each process is responsible for some unique portion of the data. The process will compute a local result from its portion and then communicate with other processes to compute a global result. This is a typical pattern in MPI.

Following this idea, we create an MPI version of `maxSum`, called `maxSumPar`. The function definition is in Figure 2. The function uses an auxilliary collective function `maxPar`. All processes call `maxPar` with an integer argument `sval`; the function returns the maximum of these values to all processes. This function is specified but not implemented in this example. This illustrates one of the main points about the contract approach: in order to verify `maxSumPar`, we need only the specifications, and not the implementations, of functions called by `maxSumPar`.

The function `maxSumPar` is also collective, i.e., called by all processes. The preconditions state the following assumptions: the number of processes is n, the memory region pointed to by a (or b) is allocated and holding at least n `ints`, and all processes have the same values for the two arrays. As in the sequential case, the postconditions state that the function returns the maximum $a[i] + b[i]$.

Last but not least, the two contracts specify something about the synchronization behavior of the functions. In both cases, the claim, encoded in the `waitsfor` clause, is that no process can exit (return from) a call to the function, until every process has entered (invoked) the function. Hence each of these functions acts as a global synchronization point or *barrier*. In the case of `maxSumPar`, this is another claim that will be checked by the verifier.

One now can verify `maxSumPar` for up to 5 MPI processes using the following command:

```
civl verify -input_mpi_nprocs_lo=1 \
  -input_mpi_nprocs_hi=5 -mpiContract=maxSumPar maxSumPar.c
```

```
/*@
   mpi uses comm;
   mpi collective(comm):
     assigns \nothing;
     ensures \mpi_agree(\result);
     ensures \forall int i; 0 <= i < \mpi_comm_size ==>
               \result >= \mpi_on(sval, i);
     ensures \exists int i; 0 <= i < \mpi_comm_size &&
               \result == \mpi_on(sval, i);
     waitsfor {i | int i; 0 <= i < \mpi_comm_size};
 */
int maxPar(int sval, MPI_Comm comm);

/*@
  mpi uses MPI_COMM_WORLD;
  mpi collective(MPI_COMM_WORLD):
    requires n > 0;
    requires \mpi_comm_size == n;
    requires \forall int i; 0 <= i < n ==>
               \mpi_agree(a[i]) && \mpi_agree(b[i]);
    requires \valid(a + (0 .. n-1)) && \valid(b + (0 .. n-1));
    assigns  \nothing;
    ensures  \forall int i; 0 <= i < n ==> \result >= a[i] + b[i];
    ensures  \exists int i; 0 <= i < n && \result == a[i] + b[i];
    waitsfor {i | int i; 0 <= i < n};
 */
int maxSumPar(int * a, int * b, int n) {
  int rank, local;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // gets the unique PID
  local = a[rank] + b[rank];
  return maxPar(local, MPI_COMM_WORLD);
}
```

Fig. 2. File `maxSumPar.c`: specification of function `maxPar`, specification and definition of function `maxSumPar`.

References

1. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C Specification Language, https://frama-c.com/html/acsl.html, accessed April. 7, 2024