Towards Deductive Verification of Message-Passing Parallel Programs

Ziqing Luo and Stephen F. Siegel

University of Delaware

November 12, 2018

Outline

Motivation

Background: Deductive Verification & Frama-C

Sequentialization: Invariant-Preserving Projection

Case Study: Cyclic Exchanger

Motivation

- MPI is still one of the popular APIs for developing HPC applications
    - Bernholdt, et al. *A Survey of MPI Usage in the US Exascale Computing Project*, 2018
- finite-state searching based tools only do bounded verification
    - e.g. CIVL, ISP, MOPPER ...
- few deductive verification tools for message-passing programs
    - e.g. ParTypes
- explore a new deductive approach to verify message-passing programs

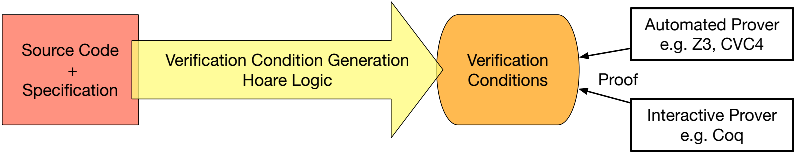
Background: Deductive Verification

```
{0 ≤ i}
  while (i > 0) {0 ≤ i}
    i--;
{i = 0}
```

Proof:

1	$\{0 \leq i \wedge 0 < i\}$ i = i - 1; $\{0 \leq i\}$	(assign)
2	$0 \leq i \wedge 0 < i \rightarrow 0 \leq i - 1$	[side]
3	$\{0 \leq i\}$ while (i > 0) i–; $\{0 \leq i \wedge i \leq 0\}$	(loop) 1, 2
4	$0 \leq i \wedge i \leq 0 \rightarrow i = 0$	[side]
5	$\{0 \leq i\}$ while (i > 0) i–; $\{i = 0\}$	(consequence) 3, 4

Background: Deductive Verification



Source Code + Specification

Verification Condition Generation
Hoare Logic

Verification Conditions

Proof

Automated Prover
e.g. Z3, CVC4

Interactive Prover
e.g. Coq

Verification Condition Generation

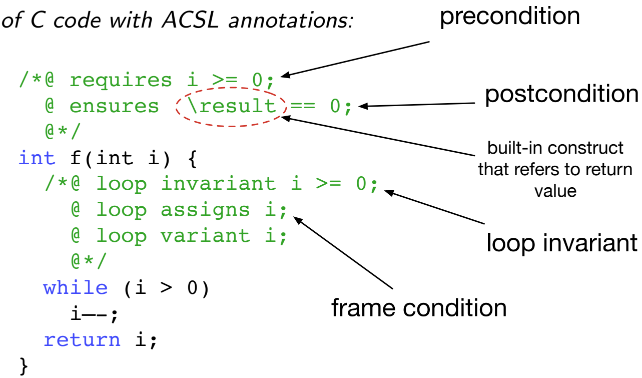
- automates the Hoare-style proof
- verification conditions are discharged by *theorem provers*
- users provide pre-/post-conditions and loop invariants

Background: Frama-C/WP & ACSL Annotations

Frama-C/WP is a deductive verification tool

- takes C programs with ACSL annotations
- ACSL = ANSI/ISO C Specification Language
- function contracts
- VCs → Why3 platform → multiple automated provers

an example of C code with ACSL annotations:



```
/*@ requires i >= 0;
  @ ensures \result == 0;
  @*/
int f(int i) {
  /*@ loop invariant i >= 0;
    @ loop assigns i;
    @ loop variant i;
    @*/
  while (i > 0)
    i--;
  return i;
}
```

precondition

postcondition

built-in construct that refers to return value

loop invariant

frame condition

Sequentialization & Global Invariant

- a global invariant Φ
    - an assertion over global states
    - provided by the user
- a sequential program
    - corresponds to a single generic process
    - a *reduction* (Lipton 1975) of the original message-passing program
        - group statements into atomic blocks
    - with inserted calls to `interleave()`
        - models the behavior of other processes
        - changes global state arbitrarily
        - preserves the global invariant
        - partial correctness

Sequentialization & Global Invariant

- a global invariant Φ
  - an assertion over global states
  - provided by the user
- a sequential program
  - corresponds to a single generic process
  - a *reduction* (Lipton 1975) of the original message-passing program
    - group statements into atomic blocks
  - with inserted calls to `interleave()`
    - models the behavior of other processes
    - changes global state arbitrarily
    - preserves the global invariant
    - partial correctness

Invariant-Preserving Projection (IPP):

IPP preserves Φ iff the original program preserves Φ

Sequentialization & Global Invariant

- a global invariant Φ
    - an assertion over global states
    - provided by the user
- a sequential program
    - corresponds to a single generic process
    - a *reduction* (Lipton 1975) of the original message-passing program
        - group statements into atomic blocks
    - with inserted calls to `interleave()`
        - models the behavior of other processes
        - changes global state arbitrarily
        - preserves the global invariant
        - partial correctness

Invariant-Preserving Projection (IPP):

IPP preserves Φ iff the original program preserves Φ

To prove:

If Φ holds initially, it will hold after each atomic block.

An Example: Cyclic Exchanger

```
int rank, nprocs, nsteps;
double rbuf, sbuf;
#define LEFT(pid) ((pid)>0 ?  (pid)-1 :  nprocs-1)
#define RIGHT(pid) ((pid)<nprocs-1 ?  (pid)+1 :  0)
...
void exchange() {
  int t = 0;
  while (t < nsteps) {
    send(&sbuf, RIGHT(rank));
    recv(&rbuf, LEFT(rank));
    sbuf = rbuf;
    t++;
  }
}
```

- each process sends a value to its right “neighbor”
- assume that the send can buffer at least one message

Invariant-Preserving Projection

```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {

    send(&sbuf);
    //@ ghost sc[rank]++;

    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

Invariant-Preserving Projection

```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {

    send(&sbuf);
    //@ ghost sc[rank]++;

    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

auxiliary variables help
1. modeling message-passing
2. expressing properties

`chan`: message channels
`size`: message channel sizes
`sc` : send counters
`rc` : recv counters

Invariant-Preserving Projection

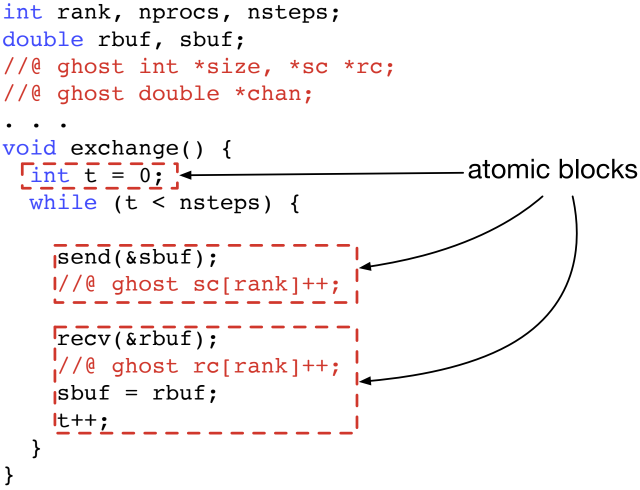
```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {

    send(&sbuf);
    //@ ghost sc[rank]++;

    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

keep track of send/recv

Invariant-Preserving Projection



```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {

    send(&sbuf);
    //@ ghost sc[rank]++;

    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

Invariant-Preserving Projection

```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {
    presend_interleave();
    send(&sbuf);
    //@ ghost sc[rank]++;
    prerecv_interleave();
    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

model the behavior of other processes

The Global Invariant of Cyclic Exchanger

```
/*@ axiomatic OracleSpec {
  @   logic double oralce(int t, int i);
  @   axiom oracle_ax: \forall int t,i; t > 0 ==>
  @           oracle(t-1, LEFT(i)) == oracle(t, i);
  @ }
 */

//@ . . .
//@ predicate inv1 = \forall int i; 0 <= i < nprocs ==>
                size[i] == 1 ==> chan[i] == oracle(sc[i]-1, i)

//@ . . .
//@ predicate inv2 = \forall int i; 0 <= i < nprocs ==>
                       rc[i] == sc[LEFT(i)] - size[LEFT(i)];
//@ . . .

#define inv (. . . inv1 && inv2 && . . .)
```

The Global Invariant of Cyclic Exchanger

```
/*@ axiomatic OracleSpec {
  @   logic double oralce(int t, int i);
  @   axiom oracle_ax: \forall int t,i; t > 0 ==>
  @           oracle(t-1, LEFT(i)) == oracle(t, i);
  @ }
 */
```

express computation

```
//@ . . .
//@ predicate inv1 = \forall int i; 0 <= i < nprocs ==>
                size[i] == 1 ==> chan[i] == oracle(sc[i]-1, i)

//@ . . .
//@ predicate inv2 = \forall int i; 0 <= i < nprocs ==>
                       rc[i] == sc[LEFT(i)] - size[LEFT(i)];
//@ . . .

#define inv (. . . inv1 && inv2 && . . .)
```

The Global Invariant of Cyclic Exchanger

express properties related to
message channels

```
/*@ axiomatic OracleSpec {
  @   logic double oralce(int t, int i);
  @   axiom oracle_ax: \forall int t,i; t > 0 ==>
  @           oracle(t-1, LEFT(i)) == oracle(t, i);
  @ }
 */

//@ . . .
//@ predicate inv1 = \forall int i; 0 <= i < nprocs ==>
                  size[i] == 1 ==> chan[i] == oracle(sc[i]-1, i)

//@ . . .
//@ predicate inv2 = \forall int i; 0 <= i < nprocs ==>
                         rc[i] == sc[LEFT(i)] - size[LEFT(i)];
//@ . . .

#define inv (. . . inv1 && inv2 && . . .)
```

The Global Invariant of Cyclic Exchanger

```
/*@ axiomatic OracleSpec {
  @   logic double oralce(int t, int i);
  @   axiom oracle_ax: \forall int t,i; t > 0 ==>
  @           oracle(t-1, LEFT(i)) == oracle(t, i);
  @ }
 */

//@ . . .
//@ predicate inv1 = \forall int i; 0 <= i < nprocs ==>
                size[i] == 1 ==> chan[i] == oracle(sc[i]-1, i)

//@ . . .
//@ predicate inv2 = \forall int i; 0 <= i < nprocs ==>
                       rc[i] == sc[LEFT(i)] - size[LEFT(i)];
//@ . . .

#define inv (. . . inv1 && inv2 && . . .)
```

express synchronization among procs

ACSL Annotations

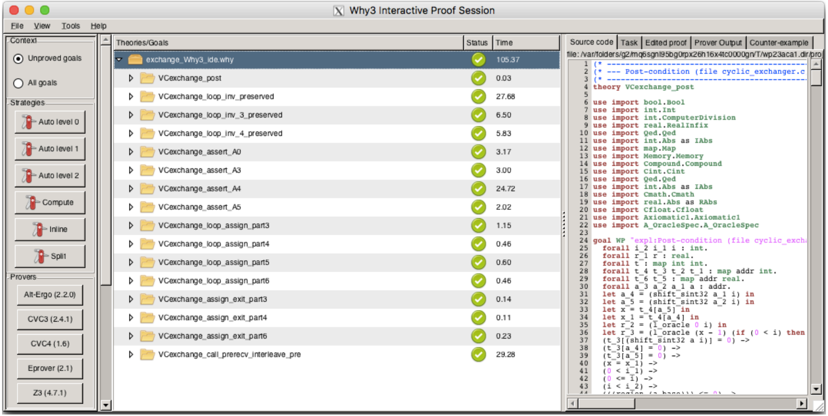
```
/*@ requires \valid(x) && sizes[rank] == 0 && 0 <= rank < nprocs;
  @ assigns chans[rank], sizes[rank];
  @ ensures chans[rank] == *x && sizes[rank] == 1; */
void send(double * x);
```

```
/*@ requires inv;
  @ assigns sizes[0..nprocs-1], chans[0..nprocs-1];
  @ assigns sc[0..nprocs-1], rc[0..nprocs-1];
  @ ensures sc[rank]==\old(sc[rank]) && rc[rank]==\old(rc[rank]);
  @ ensures sizes[rank]==0 && chans[rank]==\old(chans[rank]) && inv; */
void presend_interleave(void);
```

```
/*@ requires inv && sizes[rank]==0 && sbuf==oracle(0, rank);
  @ requires 0<nsteps && sc[rank]==0 && rc[rank]==0;
  @ assigns chans[0..nprocs-1], rbuf, sbuf, sizes[0..nprocs-1];
  @ assigns rc[0..nprocs-1], sc[0..nprocs-1];
  @ ensures rbuf == oracle(nsteps-1, LEFT(rank));*/
void exchange() {
  ...
}
```

In total, we wrote 54 lines of ACSL annotations for 17 lines of code

Discharging Verification Conditions



Theories/Goals	Status	Time
exchange_Why3_ide.why	✓	105.37
VCexchange_post	✓	0.03
VCexchange_loop_inv_preserved	✓	27.68
VCexchange_loop_inv_3_preserved	✓	6.50
VCexchange_loop_inv_4_preserved	✓	5.83
VCexchange_assert_A0	✓	3.17
VCexchange_assert_A3	✓	3.00
VCexchange_assert_A4	✓	24.72
VCexchange_assert_A5	✓	2.02
VCexchange_loop_assign_part3	✓	1.15
VCexchange_loop_assign_part4	✓	0.46
VCexchange_loop_assign_part5	✓	0.60
VCexchange_loop_assign_part6	✓	0.46
VCexchange_assign_exit_part3	✓	0.14
VCexchange_assign_exit_part4	✓	0.11
VCexchange_assign_exit_part6	✓	0.23
VCexchange_call_prerecv_interleave_pre	✓	29.28

file: /var/folders/g2/mq6sgnl95bg0rpx26h16x4tc0000gn/T/wp23aca1.dir/pro

```
(* ----------------------------------------------------
(* --- Post-condition (file cyclic_exchanger.c
(* ----------------------------------------------------
theory VCexchange_post

use import bool.Bool
use import int.Int
use import int.ComputerDivision
use import real.RealInfix
use import Qed.Qed
use import int.Abs as IAbs
use import map.Map
use import Memory.Memory
use import Compound.Compound
use import Cint.Cint
use import Qed.Qed
use import int.Abs as IAbs
use import Cmath.Cmath
use import real.Abs as RAbs
use import Cfloat.Cfloat
use import Axiomatic1.Axiomatic1
use import A_OracleSpec.A_OracleSpec

goal WP "expl:Post-condition (file cyclic_excha
  forall i_2 i_1 i : int.
  forall r_1 r : real.
  forall t : map int int.
  forall t_4 t_3 t_2 t_1 : map addr int.
  forall t_6 t_5 : map addr real.
  forall a_3 a_2 a_1 a : addr.
  let a_4 = (shift_sint32 a_1 i) in
  let a_5 = (shift_sint32 a_2 i) in
  let x = t_4[a_5] in
  let x_1 = t_4[a_4] in
  let r_2 = (l_oracle 0 i) in
  let r_3 = (l_oracle (x - 1) (if (0 < i) then
  (t_3[(shift_sint32 a i)] = 0) ->
  (t_3[a_4] = 0) ->
  (t_3[a_5] = 0) ->
  (x = x_1) ->
  (0 < i_1) ->
  (0 <= i) ->
  (i < i_2) ->
```

Deadlock Freedom

for process i, it is either at ...

- a non-communication statement
    - will not be blocked
- a send statement
    - will not be blocked iff size[pid] = 0
- a recv statement
    - will not be blocked iff size[LEFT(pid)] = 1

Express Program Locations

```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {
    presend_interleave();
    send(&sbuf);
    //@ ghost sc[rank]++;
    prerecv_interleave();
    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

first block, local:
sc[rank] - rc[rank] = 0

Another global invariant: $\forall i.\ \mathrm{sc}[i] - \mathrm{rc}[i] = 0 \lor \mathrm{sc}[i] - \mathrm{rc}[i] = 1$

Express Program Locations

```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {
    presend_interleave();
    send(&sbuf);
    //@ ghost sc[rank]++;
    prerecv_interleave();
    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

second block, before the send:
sc[rank] - rc[rank] = 0

Another global invariant: $\forall i.\ sc[i] - rc[i] = 0 \lor sc[i] - rc[i] = 1$

Express Program Locations

```
int rank, nprocs, nsteps;
double rbuf, sbuf;
//@ ghost int *size, *sc *rc;
//@ ghost double *chan;
. . .
void exchange() {
  int t = 0;
  while (t < nsteps) {
    presend_interleave();
    send(&sbuf);
    //@ ghost sc[rank]++;
    prerecv_interleave();
    recv(&rbuf);
    //@ ghost rc[rank]++;
    sbuf = rbuf;
    t++;
  }
}
```

third block, before the recv:
sc[rank] - rc[rank] = 1

Another global invariant: $\forall i.\ \mathrm{sc}[i] - \mathrm{rc}[i] = 0 \vee \mathrm{sc}[i] - \mathrm{rc}[i] = 1$

Deadlock Freedom

for process i, it is either at ...

- a non-communication statement
    - sc[i] - rc[i] = 0
    - size[pid] = 0 since no send has been invoked
- a send statement
    - sc[i] - rc[i] = 0
    - not blocked iff size[pid] = 0
- a recv statement
    - sc[i] - rc[i] = 1
    - not blocked iff size[LEFT(pid)] = 1

Deadlock Freedom

To prove: the global invariant implies deadlock freedom …

```
#define LEFT(pid) ((pid) > 0 ? (pid) - 1 : nprocs - 1)
```

$$
\begin{array}{l}
\text{nprocs} > 0 \wedge \\
\forall i.\ (0 \le \text{size}[i] \le 1 \wedge \\
\qquad 0 \le \text{sc}[i] \le \text{nsteps} \wedge \\
\qquad 0 \le \text{rc}[i] \le \text{nsteps} \wedge \\
\qquad \text{rc}[i] = \text{sc}[\text{LEFT}(i)] - \text{size}[\text{LEFT}(i)] \wedge \\
\qquad \text{sc}[i] - \text{rc}[i] = 0 \vee \text{sc}[i] - \text{rc}[i] = 1) \\
\rightarrow \\
\exists i.\ (\text{sc}[i] - \text{rc}[i] = 0 \wedge \text{size}[i] = 0) \vee \\
\quad\ (\text{sc}[i] - \text{rc}[i] = 1 \wedge \text{size}[\text{LEFT}(i)] = 1)
\end{array}
$$

Deadlock Freedom

To prove: the global invariant implies deadlock freedom ...

```
#define LEFT(pid) ((pid) > 0 ? (pid) - 1 : nprocs - 1)
```

$$
\begin{array}{l}
\mathrm{nprocs} > 0 \wedge \\
\forall i.\ (0 \leq \mathrm{size}[i] \leq 1 \wedge \\
\qquad 0 \leq \mathrm{sc}[i] \leq \mathrm{nsteps} \wedge \\
\qquad 0 \leq \mathrm{rc}[i] \leq \mathrm{nsteps} \wedge \\
\qquad \mathrm{rc}[i] = \mathrm{sc}[\mathrm{LEFT}(i)] - \mathrm{size}[\mathrm{LEFT}(i)] \wedge \\
\qquad \mathrm{sc}[i] - \mathrm{rc}[i] = 0 \vee \mathrm{sc}[i] - \mathrm{rc}[i] = 1) \\
\rightarrow \\
\exists i.\ (\mathrm{sc}[i] - \mathrm{rc}[i] = 0 \wedge \mathrm{size}[i] = 0) \vee \\
\qquad (\mathrm{sc}[i] - \mathrm{rc}[i] = 1 \wedge \mathrm{size}[\mathrm{LEFT}(i)] = 1)
\end{array}
$$

- To the best of our knowledge, no automated prover can prove this formula.
- We **proved** it: 1) by hand, see the paper; 2) by using CVC4 with a bound 200 on nprocs

Conclusion & Future Work

Conclusion:

1) a new approach to deductively verify message-passing programs

2) we proved the following for `cyclic exchanger` using mechanized tools:

1. functional correctness (for $0 < nprocs$)
2. deadlock freedom (for $0 < nprocs \leq 200$)
3. termination (assuming deadlock freedom)

Conclusion & Future Work

Future Work:

- generalize the approach
    - stencil-based programs (e.g. diffusion)
- automates the transformation
    - code transformer
    - pre-defined libraries
- try other deductive verification frameworks
    - verbosity in Frama-C/WP, e.g. pointer aliasing