Final Project

Techniques for Improving Software Productivity

Professor Mooly (Shmuel) Sagiv
Kalev Alpernas

305609588	Chen Shterental
309417418	Michael Palarya
304745375	Ben Sterenson

Project Part I - Debugging Peterson Algorithm (CBMC)

Peterson's algorithm is a concurrent programing algorithm for mutual exclusion that allows two processes to share a single resource without creating a conflict.

We will base the first part of our project on a code found on github, which implements Peterson's algorithm for two processes according to the pseudo code found in Wikipedia.
At first, we present the pseudo code:

```
bool flag[2] = {false, false};
int turn;

P0:     flag[0] = true;
P0_gate: turn = 1;
        while (flag[1] && turn == 1)
        {
            // busy wait
        }
        // critical section
        ...
        // end of critical section
        flag[0] = false;

P1:     flag[1] = true;
P1_gate: turn = 0;
        while (flag[0] && turn == 0)
        {
```

```
        // busy wait
    }
    // critical section
    ...
    // end of critical section
    flag[1] = false;
```

Notice that the variable `flag` from the pseudo code above is named in a different name, `interested`, in the implementation that we tested. In addition, on this implementation, the variable `cnt` is incremented by 1 on each entrance to the critical section, as you can see in the method `run`.

Using CBMC we will try to find bugs to check for correctness of the implementation and fix any issues discovered.

Peterson.c - Original code from https://gist.github.com/hyunhapark/a9f23beed205a67e43d1

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <signal.h>
#include <pthread.h>
#include <fcntl.h>

#define barrier() asm volatile ("" : : : "memory")

#define TRUE      (1)
#define FALSE     (0)
```

```
FILE *fp;
#define _CACHE_FLUSH fprintf(fp,"3")

static pthread_t tid[2];

volatile static int cnt = 0;

volatile static int interested[2] = {0, 0};
volatile static int turn = 0;

void acquire (int process) {
      int other = 1 - process;

      interested[process] = TRUE;
      turn = other;

      asm volatile ("mfence");
      while(interested[other] && turn==other);
}
void release (int process) {
      interested[process] = FALSE;
}

void *run (void *arg) {
      int p = *(int*)arg;
      int i;

      for(i=0;i<5000000;i++){
            acquire(p);
            cnt = cnt + 1;
```

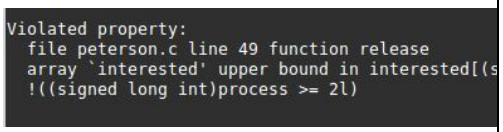
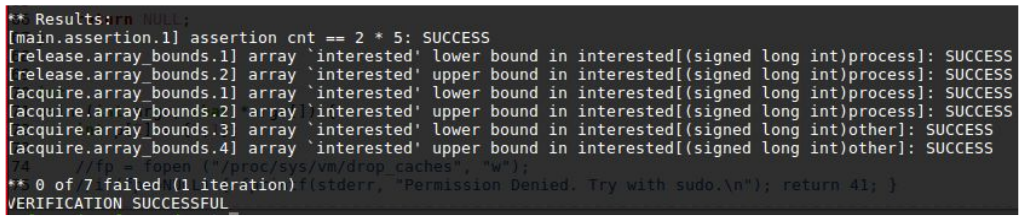
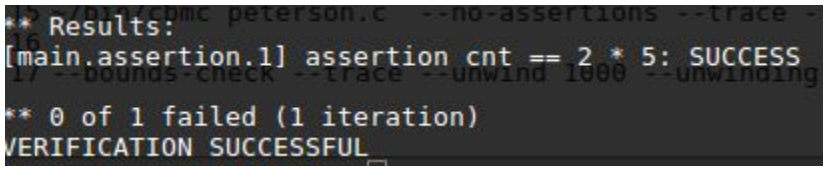
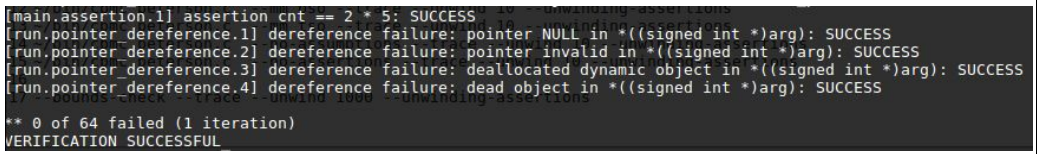
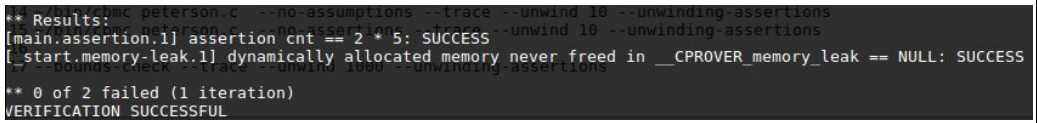
```
            release(p);
      }
      return NULL;
}

int main (int argc, char *argv[]) {
      int p[2] = {0,1};

      if( pthread_create(&tid[0], NULL, &run, &p[0]) ) { fprintf(stderr, "Thread Create failed(1).\n"); return 1;
}
      if( pthread_create(&tid[1], NULL, &run, &p[1]) ) { fprintf(stderr, "Thread Create failed(2).\n"); return 2;
}

      pthread_join(tid[0], NULL);
      pthread_join(tid[1], NULL);

      printf("cnt : %d\n",cnt);

      return 0;
}
```

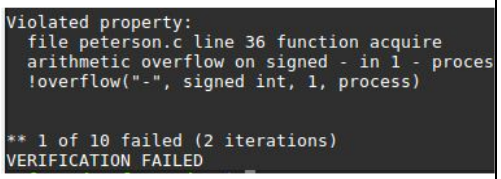
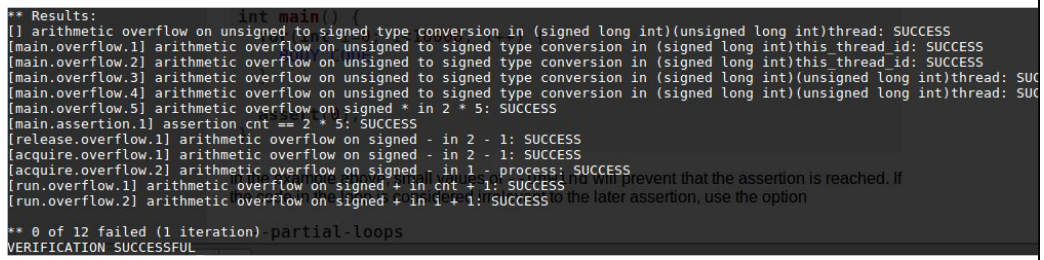
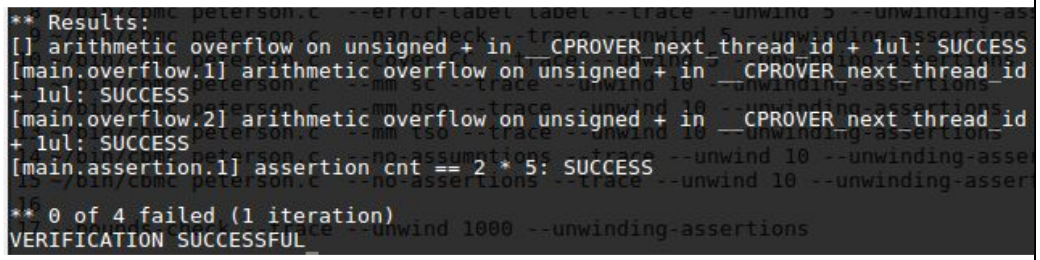
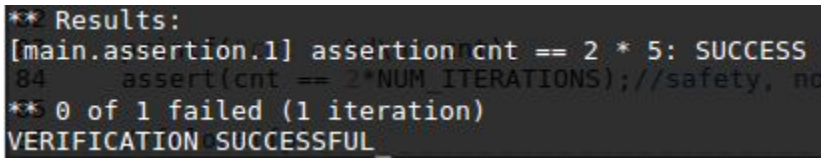
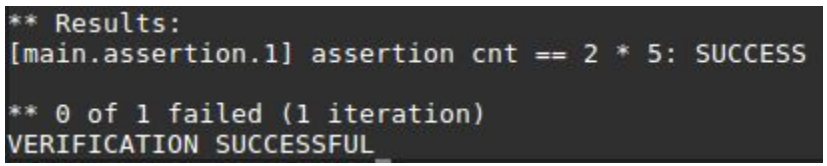
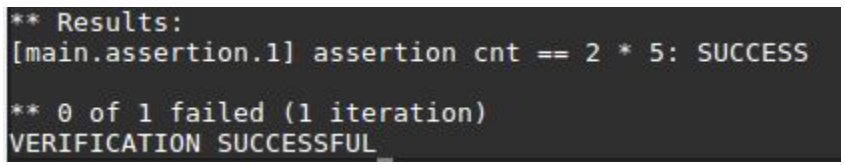
Compiling the code

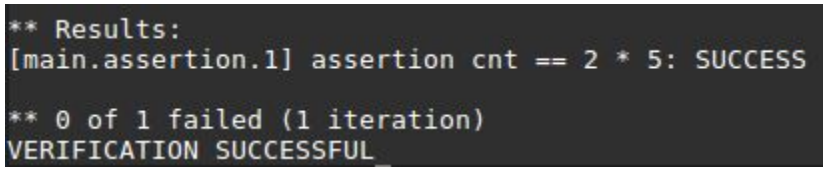
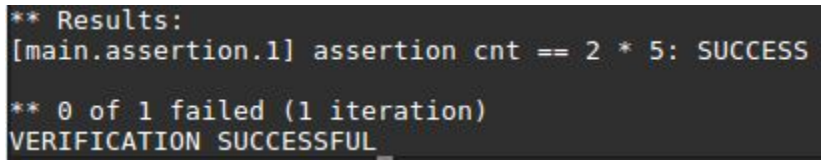
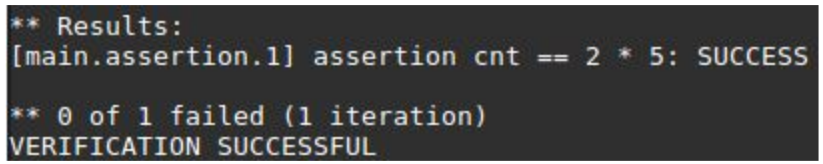
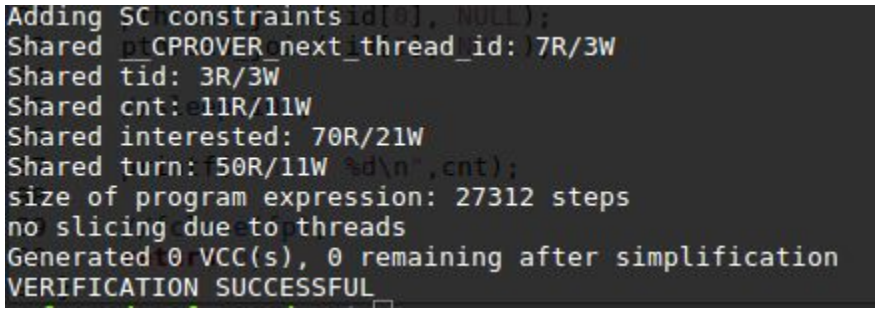
In order to compile the code - run the following command:

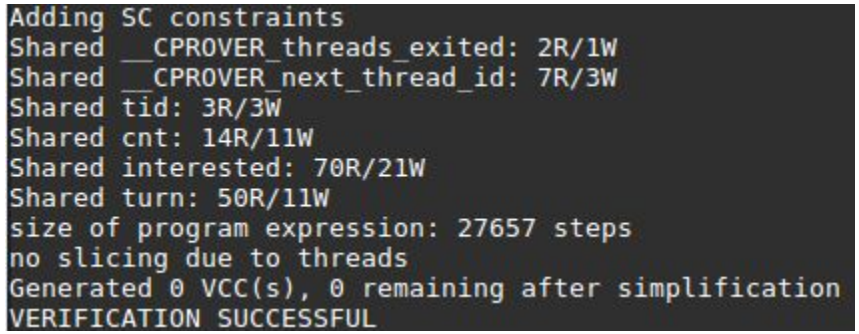
```
gcc ./peterson.c -pthread -o peterson
```

CBMC tests output table

Test Flag	Failure Reason	Success Message
bounds-check		
--div-by-zero-check	no bugs	
--pointer-check	no bugs	
--memory-leak-check	no bugs	

--signed-overflow-check		
--unsigned-overflow-check	no bugs	
--float-overflow-check	no bugs	
--error-label label	no bugs	
--nan-check	no bugs	

--mm sc	no bugs	
--mm pso	no bugs	
--mm tso	no bugs	
--no-assumptions	no bugs	

--no-assertions	no bugs	

Fixes for CBMC failed tests table

Test Flag	Failure	Fix
bounds-check	Violated property: file peterson.c line 49 function release array `interested' upper bound in interested[(signed long int)process] !((signed long int)process >= 2l)	We managed to fix both bugs using the same method - ● defined MAX_PROCESS to indicate number of threads ● added the if statement: if (process >= 0 && process <= MAX_PROCESS) (to both functions: acquire and release before accessing interested[process])
signed-overflow-check	Violated property: file peterson3.c line 35 function acquire arithmetic overflow on signed - in 1 - process !overflow("-", signed int, 1, process)	

Remarks considering the tests

- We defined unwinding of depth 5 because it was taking far too long to run for each flag with any unwinding higher than 5.
- We didn't use the --unwinding-assertions flag because when we tried to use it, the execution has failed due to insufficient iterations. We believe this is caused by the while-loop that is running within a for-loop in the 'acquire' method.

Peterson.c - Fixed code

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <signal.h>
#include <pthread.h>
#include <fcntl.h>

#define barrier() asm volatile ("" : : : "memory")

#define TRUE  (1)
#define FALSE (0)

FILE *fp;
#define _CACHE_FLUSH fprintf(fp,"3")
#define NUM_ITERATIONS 5
#define MAX_PROCCS 2

static pthread_t tid[2];

volatile static int cnt = 0;

volatile static int interested[2] = {0, 0};
volatile static int turn = 0;


void acquire (int process) {
```

```
        if(process >=0 && process<=MAX_PROCCS-1){
                int other = 1 - process;

                interested[process] = TRUE;
                turn = other;

                asm volatile ("mfence");

                while(interested[other] && turn==other);
        }
}

void release (int process) {
        if(process >=0 && process<=MAX_PROCCS-1){
                interested[process] = FALSE;
        }
}

void *run (void *arg){
        int p = *(int*)arg;
        int i;

        for(i=0;i<NUM_ITERATIONS;i++){
                acquire(p);
                cnt = cnt + 1;
                release(p);
        }

        return NULL;
}

int main (int argc, char *argv[]) {
        int p[2] = {0,1};

        if( pthread_create(&tid[0], NULL, &run, &p[0]) ) { fprintf(stderr, "Thread Create failed(1).\n"); return 1; }
```

```
        if( pthread_create(&tid[1], NULL, &run, &p[1]) ) { fprintf(stderr, "Thread Create failed(2).\n"); return 2; }

        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
        printf("cnt : %d\n",cnt);

        return 0;
}
```

Project Part II - Generalizing Peterson Algorithm (Dafny)

While experimenting with Peterson Two-Processes algorithm for mutual exclusion, we were looking to find an inductive invariant which proves that any two threads cannot enter the critical section simultaneously in a N-Processes problem.

This will basically generalize the Peterson algorithm for a general N processes.
Researching the topic led us to the Dafny-based proof of correctness for the simplified version of the algorithm, as was presented at **Washington University on January 9, 2017 - http://homes.cs.washington.edu/~jrw12/SharedMem.html**

The pseudocode used in the original proof was:

T1	**T2**
flag1 = true	**flag2 = true**
victim = 1	**victim = 2**
while (flag2 && victim == 1) {}	**while (flag1 && victim == 2) {}**
// critical section	**// critical section**
flag1 = false	**flag2 = false**

Whenever a process is interested in entering the critical section, it raises its own flag, i.e. flag1 = true and defines itself as the 'victim' which basically tells the other process that it is waiting and not yet entering the critical section, thus clearing the way for the other process to go ahead.

The original proof goes as follows:

```
// There are six possible PCs for each thread, each corresponding to
// the beginning of one atomic step.
datatype PC = SetFlag | SetVictim | CheckFlag | CheckVictim | Crit | ResetFlag

method Peterson()
decreases *
{
```

```
  // Thread ids will be ints 0 or 1.
  // Below, we will use 1-t to get the "other" tid from a tid t.

  // We store some variables in arrays so that we can write the code
  // for both threads at once, indexed by the thread id.
  var flag := new bool[2];
  flag[0] := false;
  flag[1] := false;

  // The algorithm is correct regardless of the initial value of
  // victim, so we initialize in nondeterministically.
  var victim;
  if * {
    victim := 0;
  } else {
    victim := 1;
  }

  var pc := new PC[2];
  pc[0] := SetFlag;
  pc[1] := SetFlag;


  while true
    // Invariant 1: whenever t's flag is false, its PC is at SetFlag.
    //
    // Comment this out to see that it is required by Invariant 2.
    invariant forall t :: 0 <= t < 2 && !flag[t] ==> pc[t] == SetFlag

    // Invariant 2: If t holds the lock and 1-t is trying to get it,
```

```
    //               then victim == 1-t. (victim != t works too!).
    //
    // Comment this out to see that it is required by the mutual
    // exclusion invariant below!
    //
    // Writing the invariant like this, using t' to represent the
    // other thread, avoids a trigger matching loop.
    invariant forall t, t' :: 0 <= t < 2 ==> t' == 1-t ==>
        (pc[t] == Crit || pc[t] == ResetFlag) ==>
        (pc[t'] == CheckVictim || pc[t'] == CheckFlag ||
         pc[t'] == Crit || pc[t'] == ResetFlag) ==>
        victim != t

     // This is mutual exclusion: It is never the case that both
     // threads are in the critical section.
     //
     // This is directly implied by Invariant 2, since if both PCs are
     // Crit, then the victim cannot be equal to either thread.
     //
     // Dafny is clever enough to infer an additional loop invariant,
     // namely 0 <= victim < 2, which is required to complete the proof.
    invariant !(pc[0] == Crit && pc[1] == Crit)

    decreases *
  {
    // Each iteration of the loop simulates one atomic step of execution.

    // First, nondeterministically select the thread that gets to step.
    var t;
    if * {
```

```
      t := 0;
    } else {
      t := 1;
    }

    // Now, branch on the selected thread's current PC and execute the
    // corresponding atomic step.
    //
    // Note that each step only reads or writes one shared variable.
    match pc[t] {
      case SetFlag     => flag[t] := true; pc[t] := SetVictim;
      case SetVictim   => victim := t; pc[t] := CheckFlag;
      case CheckFlag   => pc[t] := if flag[1-t] then CheckVictim else Crit;
      case CheckVictim => pc[t] := if victim == t then CheckFlag else Crit;
      case Crit        => pc[t] := ResetFlag;
      case ResetFlag   => flag[t] := false; pc[t] := SetFlag;
    }
  }
}
```

We were convinced that simple adjustments are sufficient to generalize the algorithm for a N-Processes scenario. The pseudocode that we were trying to prove that it maintained safety (mutual exclusion), was:

Ti
flag_i = true
victim = i
while ((exists j. j!=i && flag_j) &&
victim == i) {}
// critical section
flag_i = false

Below is the product of our work, with all of the changes marked in **Yellow** -

```
// There are six possible PCs for each thread, each corresponding to
// the beginning of one atomic step.
datatype PC = SetFlag | SetVictim | CheckFlag | CheckVictim | Crit | ResetFlag

method Peterson(n:int, vic:int, starting_thread:int)
decreases *
requires 2 <= n < 10
requires 0 <= vic < 2
requires 0 <= starting_thread < n
{
  // Thread ids will be ints 0 or 1.
  // Below, we will use 1-t to get the "other" tid from a tid t.

  // We store some variables in arrays so that we can write the code
  // for both threads at once, indexed by the thread id.

  var flag := new bool[n];
  var i := 0;
  while i < n
  {
    flag[i] := false;
    i := i + 1;
  }


  // The algorithm is correct regardless of the initial value of
  // victim, so we initialize in nondeterministically.
  var victim := vic;
```

```
  //var victim;
  //if * {
  //  victim := 0;
  //} else {
  //  victim := 1;
  //}

  var pc := new PC[n];
  var j := 0;
  while j < n
//for all j' which is smaller than j, j' is in SetFlag
(1)   invariant forall j' :: 0 <= j' < n && j' < j ==> pc[j'] == SetFlag
  {
    pc[j] := SetFlag;
    j := j + 1;
  }

  while true
    // Invariant 1: whenever t's flag is false, its PC is at SetFlag.
    //
    // Comment this out to see that it is required by Invariant 2.
     //SetFlag is the only pc in which flag[t] can be false.
(2)    invariant forall t :: 0 <= t < n && !flag[t] ==> pc[t] == SetFlag
    // Invariant 2: If t holds the lock and 1-t is trying to get it,
    //              then victim == 1-t. (victim != t works too!).
    //
    // Comment this out to see that it is required by the mutual
    // exclusion invariant below!
    //
    // Writing the invariant like this, using t' to represent the
```

```
    // other thread, avoids a trigger matching loop.
(3)     invariant forall t, t' :: 0 <= t < n  && 0 <= t' < n && t != t' ==>
         (pc[t] == Crit || pc[t] == ResetFlag) ==>  // if t holds the lock
         (pc[t'] == CheckVictim || pc[t'] == CheckFlag ||   // then if t' is trying to get it
          pc[t'] == Crit || pc[t'] == ResetFlag) ==>
         victim != t  // then victim != t

     // This is mutual exclusion: It is never the case that both
     // threads are in the critical section.
     //
     // This is directly implied by Invariant 2, since if both PCs are
     // Crit, then the victim cannot be equal to either thread.
     //
     // Dafny is clever enough to infer an additional loop invariant,
     // namely 0 <= victim < 2, which is required to complete the proof.
     // invariant !(pc[0] == Crit && pc[1] == Crit)
//safety, checking that no two threads enter the critical section at the same time
(4)  invariant forall t, t' :: ( (0 <= t < n && 0 <= t' < n && pc[t] == pc[t'] && pc[t] == Crit)==>
      i == j )     decreases *
  {
    // Each iteration of the loop simulates one atomic step of execution.

    // First, nondeterministically select the thread that gets to step.
    var t := starting_thread;
    // Now, branch on the selected thread's current PC and execute the
    // corresponding atomic step.
    //
    // Note that each step only reads or writes one shared variable.
    match pc[t] {
      case SetFlag     => flag[t] := true; pc[t] := SetVictim;
```

```
      case SetVictim   => victim := t; pc[t] := CheckFlag;
      //case CheckFlag   => pc[t] := if flag[1-t] then CheckVictim else Crit;
      case CheckFlag   => pc[t] := if (exists t' :: 0 <= t' < n && t' != t && flag[t']) then
CheckVictim else Crit;
(5)    case CheckVictim => pc[t] := if victim == t then CheckFlag else Crit;
      case Crit        => pc[t] := ResetFlag;
      case ResetFlag   => flag[t] := false; pc[t] := SetFlag;
    }
  }
}
```

Digging deeper into our offered solution, and after our experiments with Dafny, we have realized that the **Red (4)** invariant , i.e. the safety invariant, that checks for mutual exclusion, was not inductive. Namely, we realized that our new algorithm doesn't guarantee mutual exclusion and is not sufficient for generalizing Peterson's algorithm, and some better planned strategy is necessary, e.g. the **Filter algorithm** that is based on turn-counting.

We realized that the very important **(4)** invariant, which checks for the actual mutual exclusion by demanding that no two processes enter the critical section at the same time, will always fail for `n > 2` because of the **Blue (5)** transaction rule, that enters the critical section after verifying that the victim (i.e. delayed process) is not the current one.

Having as little as 3 processes running simultaneously will generate the scenario where, without loss of generality, Victim := thread A, and both B and C threads are allowed into the critical section by **(5)** (because B and C are both not equal to thread A), thus not maintaining the **(4)** invariant and obviously contradicting our solution and proving it to be wrong.

We believe that although we didn't manage to prove an algorithm's correctness, refuting it was equally beneficial to our academic experience. In addition, we found three inductive invariants **(in Pink(1),(2),(3))** of our modified algorithm, and it was also an interesting process to find them. We explained these invariants in the comments of our modified Dafny code above that are marked light blue. Indeed, By commenting out the **(4)** invariant, we proved using Dafny that these three invariants are inductive.