Bounded Model Checking of Multi-threaded C Programs

Lazy-CSeq and MU-CSeq

presented by Vladimír Štill

ParaDiSe
Parallel & Distributed Systems Laboratory

Masaryk University

Brno, Czech Republic

7th March 2015

bounded model checking of parallel C via *sequentialization*

- ANSI C (C89) + `pthreads`
- multi-threaded programs translated to nondeterministic sequential programs
- bounded in the number of cycle iterations and recursion depth
  - cycle/function *unwinding*
  - each thread represented by a single function without cycles
- bounded and sequentialized program passed to a backend
  - CBMC is preferred, but both claim to work with multiple bounded model checkers

Lazy-CSeq [2, 1]

Lazy-CSeq

- exploits the simple structure of unwound programs
  - no function calls
  - no cycles $\rightarrow$ no back jumps
  - very simple control flow (`if`, forward `goto`, thread creation/joining)
  - every statement is executed at most once
- simulates all round-robin schedules
  - bound on the number of rounds
- the actual property violation detection left to the backend
- sequentialization preserves safety properties *within the bounds*

Assumption About the Input

- sequentially consistent memory access
- each statement ($\sim$ line of code) is atomic
- statically known thread function
    - i.e. no computation of thread function pointers
- *visible statement* – involve read or write operation of a shared variable

Sequentialization

1. functions which appear in `pthread_create` and `main` are copied as *thread entries*
   - every `pthread_create` corresponds to a new thread entry
   - we will denote them f_0 to f_n (f_0 = a copy of `main`)
2. loop and function calls are unwound in f_0 to f_n
   - except for calls to `pthread_*`
3. f_0 to f_n are instrumented to allow partial execution and to preserve state between invocations
   - all locals are turned into `static`
   - control flow is instrumented
4. new `main` is added
   - dispatches f_0 to f_n repeatedly in the round-robin fashion

Thread Control Flow Instrumentation

```
pthread_mutex_t m; int c = 0;
void *prod( void *b ) {
              int tmp = *((int*)b);
              pthread_mutex_lock( &m );
              if ( c > 0 )
                  c++;
              else {
                  c = 0;
                  while( tmp > 0 ) {
                      c++; tmp--;
                  }
              }
              pthread_mutex_unlock( &m );
              return 0;
}
```

```
pthread_mutex_t m; int c = 0;
void *prod( void *b ) {
              static int tmp = *((int*)b);
              pthread_mutex_lock( &m );
              if ( c > 0 )
                  c++;
              else {
                  c = 0;
                  while( tmp > 0 ) {
                      c++; tmp--;
                  }
              }
              pthread_mutex_unlock( &m );
              return 0;
}
```

Make variables `static`.

Thread Control Flow Instrumentation

```
pthread_mutex_t m; int c = 0;
void *prod( void *b ) {
              static int tmp = *((int*)b);
              pthread_mutex_lock( &m );
              if ( c > 0 )
                  c++;
              else {
                  c = 0;
                  if( !( tmp > 0 ) ) goto _l1;
                  c++; tmp--;
                  if( !( tmp > 0 ) ) goto _l1;
                  c++; tmp--;
                  assume( !( tmp > 0 ) );
              _l1: }
              pthread_mutex_unlock( &m );
```

Loop unwinding.

Thread Control Flow Instrumentation

```
#define J(A, B) if (pc[t] > A || A >= cs) goto B;
pthread_mutex_t m; int c = 0;
void *prod( void *b ) {
  0: J(0,1)   static int tmp = *((int*)b);
  1: J(1,2)   pthread_mutex_lock( &m );
  2: J(2,3)   if ( c > 0 )
  3: J(3,4)       c++;
              else {
  4: J(4,5)       c = 0;
                  if( !( tmp > 0 ) ) goto _l1;
  5: J(5,6)       c++; tmp--;
                  if( !( tmp > 0 ) ) goto _l1;
  6: J(6,7)       c++; tmp--;
                  assume( !( tmp > 0 ) );
              _l1: }
  7: J(7,8)   pthread_mutex_unlock( &m );
```

Add support for jumping over statements (invisible not considered).

Thread Control Flow Instrumentation

```
#define G(L) assume(cs >= G)
#define J(A, B) if (pc[t] > A || A >= cs) goto B;
pthread_mutex_t m; int c = 0;
void *prod( void *b ) {
  0: J(0,1)   static int tmp = *((int*)b);
  1: J(1,2)   pthread_mutex_lock( &m );
  2: J(2,3)   if ( c > 0 )
  3: J(3,4)       c++;
              else { G(4);
  4: J(4,5)       c = 0;
                  if( !( tmp > 0 ) ) goto _l1;
  5: J(5,6)       c++; tmp--;
                  if( !( tmp > 0 ) ) goto _l1;
  6: J(6,7)       c++; tmp--;
                  assume( !( tmp > 0 ) );
              _l1: G(7); }
  7: J(7,8)   pthread_mutex_unlock( &m );
```

Guard control flow validity.

Thread Dispatch

In the new `main`

- executes threads in the round-robin fashion with a fixed number of rounds
- each time a thread executes for a nondeterministically guessed number of steps
- executions of thread entry resumes where it left in last round thanks to control flow instrumentation (`J`) and saved `pc` values
- thread executes until its PC equals `cs` (context-switch point)
- keeps an array of active thread IDs, PCs for each thread

for K rounds:

- for each active thread t
  1. guess next context switch point (`cs`) nondeterministically
  2. run f_t
  3. set `pc[t] = cs`

- in principle very simple
- but wins SV-COMP since 2014 (together with MU-CSeq)
- works with many bounded model checkers
- supports deadlock detection, counterexamples
- ignores array bounds
- seems to support small part of C library (for example `malloc`, `strcpy`, but not `assert`, `qsort`)

Lazy-CSeq

- in principle very simple
- but wins SV-COMP since 2014 (together with MU-CSeq)
- works with many bounded model checkers
- supports deadlock detection, counterexamples
- ignores array bounds
- seems to support small part of C library (for example `malloc`, `strcpy`, but not `assert`, `qsort`)

How to Use It

- download at `http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html`
- requires CBMC (or BLITZ, ESBMC, LLBMC, …), Python 2, `pycparser`
  - CBMC needs to be in `PATH`
- `./cseq.py -i file.c`

Does It Work?

```c
int x;
void *thread( void *_ ) {
    ++x;
    return 0;
}
int main() {
    pthread_t t;
    pthread_create( &t, 0, thread, 0 );
    ++x;
    pthread_join( t, 0 );
    assert( x == 2 );
}
```

Does It Work?

```c
int x;
void *thread( void *_ ) {
    ++x;
    return 0;
}
int main() {
    pthread_t t;
    pthread_create( &t, 0, thread, 0 );
    ++x;
    pthread_join( t, 0 );
    assert( x == 2 );
}
```

```
$ python2 cseq.py --rounds=10 --unwind=10 \
                  --softunwindbound -i inc.c
warning: warnings on stderr from the backend)
inc.c, SAFE, 0.89
```

MU-CSeq [3, 4]

Memory Unwindings

MU-CSeq based on the idea of bounded memory unwindings

- targets C, but explained on a simplified language

Memory unwinding = a sequence of write operations into the shared memory

- guessed nondeterministically
- program scheduling must match MUs
- MU-CSeq bounds the number of writes into concurrently-accessed memory locations (*shared variables*)

Definitions I

n-memory unwinding M

- a sequence of writes $w_1 \dots w_n$ of shared variables
- each w_i is a triple (t_i, var_i, val_i)
  - t_i is the identifier of the thread which performed the write
  - var_i is the name of the written variables
  - val_i is the new value of var_i

Position in an *n*-memory unwinding *M*

- an index in the interval $[1, n]$

Definitions II

Execution of program P *conforms to* a memory unwinding M

- if the sequence of its writes in the shared memory exactly matches M

***Unfeasible* unwinding M for program P**

- no execution of P conforms to M

Simulation of Program P under Unwinding M

with thread number bound τ

- the aim is to simulate all runs of P which conforms to M
- threads communicate by
  - shared variables, which are in the unwinding
  - locks and thread creation/joining functions
- therefore, for the given unwinding M, threads can be executed sequentially
  - thread creation $\rightarrow$ function call
  - joins and locks prune infeasible runs

Simulation of Program P under Unwinding M

simulation of thread t

- keeps track of last position in M
- operations over non-shared variables are not changed
- a write of *val* to shared a variable *var* check that the closest entry in M for t is (t, *var*, *val*) (write of the same value to the same variable)
  - otherwise the run is abandoned
- a read of variable *var* nondeterministically guesses a position of write in the unwinding which writes to *var* between the last position of t in M and the position of next write from t and reads this value

Memory Unwinding Implementations

authors discuss several implementations

fine-grained MU

- all writes to shared variable are stored in MU
- this was presented so far

coarse-grained MU

- only some writes are visible in MU
- writes can be grouped together
  - *intra-thread* coarse-grained MU (grouping only in one thread)
  - *inter-thread* coarse-grained MU (writes from multiple threads can occur at a single MU position)
- nondeterministically selects which writes are visible to other threads

Coarse-Grained MU

Intra-Thread Coarse-Grained MU

- stores sequence of *clusters of writes*
  - thread id + partial mapping from shared variables to values
- simulation of thread t at position i:
  - if t does not write into memory at i it can only read;
  - otherwise:
    - the write must be to the variable in the mapping,
    - all the writes in the cluster must be matched before advancing to the next position
    - (the last written value to each variable in the cluster must match the mapping)

Inter-Thread Coarse-Grained MU

- multiple threads can be assigned to a single cluster/position
- unexposed writes can be seen by other threads of the cluster

Individual Memory Location Unwindings [4]

- separate unwinding for each individual shared memory location
  - for locations corresponding to scalar types or pointers
- timestamps of writes to recover global total order of writes
- supports dynamic memory and pointer arithmetics
- detailed description not available

MU-CSeq

- works with CBMC as a backend
- does loop and recursion bounding (but not context-switch bounding)
- winner of SV-COMP 2016 (beats Lazy-CSeq in speed)

How to Use It

- `http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html`
- needs CBMC + Python 2 + `pycparser`
- needs SV-COMP-style specification file (`ALL.prp`): `CHECK(init(main()), LTL(G ! call(__VERIFIER_error())))`
- `./mu-cseq.py -i file.c --spec ALL.prp`

Does it work?

- the transformation seems to be buggy
  - is not able to handle prefix increment (`++x`)
- seems to support very little of C library
- ignores memory errors
- but with postfix increment finds the bug omitted by Lazy-CSeq

Bibliography I

Omar Inverso, Truc L. Nguyen, Ermenegildo Tomasco, Bernd Fischer 2, Salvatore La Torre, and Gennaro Parlato.
Lazy-CSeq 1.0 (Competition Contribution), rejected,
`http://eprints.soton.ac.uk/387010/`.

Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato.
Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, chapter Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization, pages 585–602.
Springer International Publishing, Cham, 2014.

Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato.
Verifying concurrent programs by memory unwinding.
In *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, April 2015.

Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato.
MU-CSeq 0.4: Individual Memory Location Unwindings (Competition Contribution).
In *22st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), to appear*, 2016.