Unbounded Lazy-CSeq: A Lazy Sequentialization Tool for C with unboundedly many Context Switches

(Competition Contribution)*

Truc L. Nguyen[1], Omar Inverso[1], Ermenegildo Tomasco[1], Bernd Fischer[2], Salvatore La Torre[3], and Gennaro Parlato[1]

[1] Electronics and Computer Science, University of Southampton, UK
[2] Division of Computer Science, Stellenbosch University, South Africa
[3] Dipartimento di Informatica, Università degli Studi di Salerno, Italy

Abstract. We describe a new CSeq module for the verification of multi-threaded C programs with dynamic thread creation. This module implements a variation of the *lazy sequentialization* algorithm implemented in Lazy-CSeq. The main novelty is that we do not bound here the number of round-robin schedules in an execution and the loops that do not contain a thread creation statement are not unwound (and thus unboundedly many iterations can be considered). As in Lazy-CSeq the number of thread creations per run is bounded as well as the depth of recursive calls. For the experiments we use CPAChecker as backend.

The description below follows the general SVCOMP requirements, but is the same as for SVCOMP'15 (and this paper is not meant to become part of the proceedings).

1 Introduction

The tool CSeq [2, 3] is a modular framework for the verification of multi-threaded C programs with dynamic thread creation that is based on sequentialization: the input concurrent program is translated into a corresponding sequential one and then the verification is performed on this last one by using existing verification tools. Modules of CSeq implement *eager* sequentialization schemes [6, 8, 9] and lazy sequentialization schemes targeted to bounded model checking [4, 5].

The module Lazy-CSeq [5] implements a lazy sequentialization for bounded programs that avoids the recomputation of local states of the first lazy scheme [7] and is allows to explore the runs of the starting concurrent program up to a bounded number of context switches (arranged in rounds of a round-robin schedule). The new module UL-CSeq does not bound the number of rounds and can handle also unbounded programs. In particular, we still bound the number of threads in a run and the depth of the recursion in recursive function calls, however we keep the cycles (i.e., we do not unroll them) if they do not contain thread creation statements. The resulting program has a finite control flow graph and thus is suitable for the tool CPAChecker [1] that we use in our experiments.

* Corresponding author: Truc L. Nguyen, tnl2g10@soton.ac.uk.

2 Verification Approach

Overview. Our sequentialization scheme schedules the different threads in a round-robin fashion until all the threads terminate. It runs the threads across unboudedly many rounds. We bound the number of possible threads in the program and this is indirectly achieved by unrolling the loops that contain thread creation statements. The overall structure of the translation has a main driver and a function for each thread. The purpose of the driver is to call in an infinite `while`-loop the thread functions according to a round-robin schedule repeatedly. At each iteration a whole round of contexts (one for each thread) is executed.

For each thread, we maintain the program locations at which the previous round's context switch has happened and thus the computation must resume in the next round. To ensure the correctness of resuming from previous context switch, we also keep a global variable to store the current mode in the thread simulation: resume, execute, or suspend. To avoid the recomputation of the local states when a thread is resumed, we declare the local variables of class static (i.e., persistent) and keep track of the program counter for each thread.

Heap allocation needs no special treatment during the sequentialization and can be delegated entirely to the backend model checker.

Main driver. The following is the main driver with bound of two on thread creations:

```
while (1){
  __cs_simulate = 0;  /*thread simulation mode is on resume*/
  main_thread();             /*main thread*/
  __cs_simulate = 0;
  if (__cs_activethread_1) thr_1(__cs_args_1);   /*thread 1*/
  __cs_simulate = 0;
  if (__cs_activethread_2) thr_2(__cs_args_2);   /*thread 2*/
}
```

Thread translation. The sequentialized program also contains a function (*thread function*) for each thread instance (including the original main). The code shared by multiple threads is duplicated for each of them such that each thread has its own code.

In the translation, we inject a guard for each statement to control the resume, execution and suspension of each thread. The injected code is

```
if (__cs_simulate == 1 ||        /*execute*/
    (__cs_simulate == 0 && __cs_pc_1 == curr_pc)){/*resume*/
  __cs_simulate = 1;
  if (__VERIFIER_nondet_bool()){  /*context switch guess*/
        __cs_pc_1 = curr_pc;      /*save program location*/
        __cs_simulate = 2;    }       /*suspend this thread*/
  else {   /* execute statement */  }
}
```

This control code on resuming makes the control to skip all the statements up to the program counter at the last context switch. On positioning at the corresponding statement, the mode changes to execution, and the statements are executed until a context switch happens, and then the mode changes to suspend. In this mode, we skip the instructions until returning to the main driver. Context switching are nondeterministically guessed in the execution mode before each statement is executed.

`If`- and `while`-statements also require to inject a similar code to guard the condition.

3 Architecture, Implementation, and Availability

Architecture. UL-CSeq is implemented as a source-to-source transformation tool in Python (v2.7+). It uses the `pycparser` (v2.14, `https://github.com/eliben/pycparser`) to parse a C program into an abstract syntax tree (AST). The sequentialized program can then be processed independently by any sequential verification tool for C. UL-CSeq has been tested with CPAChecker (v1.4, `http://cpachecker.sosy-lab.org/`).

A small script bundles up translation and verification. This script first invokes the translation which sequentializes a concurrent program into a sequential one, then it calls CPAChecker to analyze the sequentialized program. The script returns TRUE (safe) or FALSE (unsafe) according to the analysis of CPAChecker.

Availability and Installation. UL-CSeq can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq/ul-cseq-svcomp16.tar.gz`; it also requires installation of the `pycparser`. In the competition we used CPAChecker as a sequential verification backend; this must be installed in the directory of UL-CSeq. CPAChecker also requires the installation of Java Runtime Environment. For the competition, a compressed version of CPAChecker is included, and it can be used when unzipped. The wrapper script for the tool on the BenchExec repository is `ulcseq.py`.

Call. For the competition, UL-CSeq should be called in the installation directory as follows:

```
./ul-cseq.py -i<file> --spec<specfile> --witness<logfile>
```

Since UL-CSeq is not a full verification tool but only a concurrency pre-processor, we only compete in the `Concurrency` category.

References

1. D. Beyer and M. Erkan Keremoglu CPAchecker: A Tool for Configurable Software Verification. *CAV*. LNCS 6806, pp. 184-190, 2011.
2. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Sequentialization Tool for C (Competition Contribution). *TACAS*. LNCS 7795, pp. 616-618, 2013.
3. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-Processor for Sequential C Verification Tools. *ASE*, pp. 710-713, 2013.
4. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, G. Parlato. Lazy-CSeq: A Lazy Sequentialization tool for C (Competition Contribution). *TACAS*, LNCS 8413, pp. 398-401, 2014.
5. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, G. Parlato. Bounded Model Checking of Multi-Threaded C Programs via Sequentialization *CAV*, LNCS 8559, pp. 585-602, 2014.
6. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.
7. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. *CAV*, LNCS 5643, pp. 477–492, 2009.
8. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, G. Parlato. MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings (Competition Contribution). *TACAS*, LNCS 8413, pp. 402-404, 2014.
9. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, G. Parlato. Verifying Concurrent Programs by Memory Unwinding. *TACAS*, LNCS 9035, pp. 551-556, 2015.