Unbounded Lazy-CSeq: A Lazy Sequentialization Tool for C Programs with Unbounded Context Switches*

(Competition Contribution)

Truc L. Nguyen[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
tnl2g10@soton.ac.uk

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 now support an unbounded number of context switches and allow unbounded loops, while the number of allowed threads still remains bounded. This is achieved by a modified sequentialization transformation and the use of the CPAchecker as sequential verification backend.

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 concurrent input program is translated into a corresponding sequential program, which is then verified using existing verification tools for sequential programs. Modules of CSeq implement different *eager* sequentialization schemes [2,3,7,8] 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 [6]. It allows us to explore all runs of the original concurrent program up to a bounded number of context switches (arranged in rounds of a round-robin schedule). The new module UL-CSeq described here removes two limitations of this schema: it no longer bounds the number of rounds, and it can handle unbounded programs. In particular, while we still bound the number of threads in a run and the depth of the recursion in recursive function calls we keep the loops (i.e., we do not unroll them), as long as 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.

2 Verification Approach

Overview. Our sequentialization scheme bounds the number of possible threads in the program, which is achieved indirectly by finite unrolling of the loops that contain thread

* Partially supported by EPSRC grant no. EP/M008991/1, INDAM-GNCS 2014 grant and MIUR-FARB 2012-2014 grants.

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 460–462, 2015.
DOI: 10.1007/978-3-662-46680-3_45

creation statements. It runs the threads for an unbouded number of rounds, scheduling them in a round-robin fashion until all the threads terminate. The overall structure of the sequentialized program thus has a main driver and a simulation function for each thread. The purpose of the driver is to repeatedly call, in an infinite `while`-loop, the thread simulation functions according to a round-robin schedule. In each iteration an entire 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 each thread's current mode (i.e., resume, execute, or suspend) in the simulation: To avoid the recomputation of the local states when a thread is resumed, we declare its local variables as `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.

Thread Translation. The sequentialized program also contains a *thread simulation 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, and in particular, its own copy of the thread-local variables.

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

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

On resuming, this control code makes the function to skip all statements up to the program counter value 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 switches are nondeterministically guessed in the execution mode before each statement is executed. `If`- and `while`-statements also require the injection of similar code to guard the control flow conditions.

3 Architecture, Implementation, and Availability

Architecture. UL-CSeq is implemented as a source-to-source transformation tool in Python (v2.7.1). It uses the `pycparser` (v2.10, `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 (v1.3.4, `http://cpachecker.sosy-lab.org/`).

A small script bundles up translation and verification. The script first invokes the translation which sequentializes the concurrent program, and then calls the CPAchecker to analyze the sequentialized program as follows: `cpa.sh -timelimit 86400 -heap 12000M -preprocess -stats -predicateAnalysis -outputpath` *`output`*. The script returns TRUE (safe) or FALSE (unsafe) according to the analysis of CPAchecker.

Availability and Installation. UL-CSeq can be downloaded from this link `http://users.ecs.soton.ac.uk/gp4/cseq/ul-cseq-svcomp15.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.

Call. Since UL-CSeq is not a full verification tool but only a concurrency pre-processor, we only compete in the `Concurrency` category. Here, it should be called in the installation directory as follows: `./UL-CSeq.py -i` *`file`* `--spec` *`specfile`* `--witness` *`logfile`*.

Strengths and Weaknesses. UL-CSeq's main strength compared to Lazy-CSeq and MU-CSeq is that, due to the use of the CPAchecker as backed, a TRUE result now represents an actual correctness proof (at least if the number of threads in the program is bounded), and not just a failure to find an error. Its main weakness is that this is slower than the approach taken in Lazy-CSeq and MU-CSeq, resulting in a relatively large number of timeouts, and a lower overall score. Moreover, we still need to bound the number of threads a priori.

References

1. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
2. Fischer, B., Inverso, O., Parlato, G.: CSeq: A Sequentialization Tool for C. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 616–618. Springer, Heidelberg (2013)
3. Fischer, B., Inverso, O., Parlato, G.: CSeq: A Concurrency Pre-Processor for Sequential C Verification Tools. In: ASE, pp. 710–713 (2013)
4. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: A Lazy Sequentialization Tool for C. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 398–401. Springer, Heidelberg (2014)
5. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 585–602. Springer, Heidelberg (2014)
6. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
7. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 402–404. Springer, Heidelberg (2014)
8. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying Concurrent Programs by Memory Unwinding. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 551–565. Springer, Heidelberg (2015)