Lazy-CSeq: A Lazy Sequentialization Tool for C ⋆
(Competition Contribution)

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
{oi2c11,et1m11,gennaro}@ecs.soton.ac.uk, bfischer@cs.sun.ac.za,
slatorre@unisa.it

Abstract. We describe a version of the lazy sequentialization schema by La Torre, Madhusudan, and Parlato that is optimized for bounded programs, and avoids the re-computation of the local state of each process at each context switch. Lazy-CSeq implements this sequentialization schema for sequentially consistent C programs using POSIX threads. Experiments show that it is very competitive.

1 Introduction

Sequentialization translates concurrent programs into (under certain assumptions) equivalent non-deterministic sequential programs and so reduces concurrent verification to its sequential counterpart. The widely used (e.g., in CSeq [2,3] or Rek [1]) sequentialization schema by Lal and Reps (LR) [6] considers only round-robin schedules with K rounds, which bounds the number of context switches between the different threads. LR first replaces the shared global memory by K indexed copies. It then executes the individual threads to completion, simulating context switches by non-deterministically incrementing the index. The first thread works with the initial memory guesses, while the remaining threads work with the values left by their predecessors. The initial guesses are also stored in a second set of copies; after all threads have terminated these are used to ensure consistency (i.e., the last thread has ended its execution in each round with initial guesses for the next round).

LR explores a large number of configurations unreachable by the concurrent program, due to the completely non-deterministic choice of the global memory copies and the late consistency check. The lazy sequentialization schema by La Torre, Madhusudan, and Parlato (LMP) [4,5] avoids this non-determinism, but at each context switch it re-computes from scratch the local state of each process. This can lead to verification conditions of exponential size when constructing the formula in a bounded model checking approach (due to function inlining). However, for bounded programs this re-computation can be avoided and the sequentialized program can instead jump to the context switch points. Lazy-CSeq implements this improved *bounded LMP schema* (bLMP) for sequentially consistent C programs that use POSIX threads.

⋆ This work was partially funded by the MIUR grant FARB 2011-2012, Università degli Studi di Salerno (Italy).

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 398–401, 2014.

2 Verification Approach

Overview. bLMP considers only round-robin schedules with K rounds. It further assumes that the concurrent program (and thus in particular the number of possible threads) is bounded and that all jumps are forward jumps, which are both enforced in Lazy-CSeq by unrolling. Unlike LR, however, bLMP does not run the individual threads to completion in one fell swoop; instead, it repeatedly calls the sequentialized thread functions in a round-robin fashion. For each thread it maintains the program locations at which the previous round's context switch has happened and thus the computation must resume in the next round. The sequentialized thread functions then jump (in multiple hops) back to these stored locations. bLMP also keeps the thread-local variables persistent (as `static`) and thus, unlike the original LMP, does not need to re-compute their values from saved copies of previous global memory states before it resumes the computation.

Data Structures. bLMP only stores and maintains, for each thread, a flag denoting whether the thread is active, the thread's original arguments, and an integer denoting the program location at which the previous context switch has happened. Since it does not need any copy of the shared global memory, heap allcotion needs no special treatment during the sequentialization and can be delegated entirely to the backend model checker.

Main Driver. The sequentialized program's main function orchestrates the analysis. It consists of a sequence of small code snippets, one for each thread and each round, that check the thread's active flag (maintained by Lazy-CSeq's implementation of the `pthread_create` and `pthread_join` functions), and, if this is set, non-deterministically increment the next context switch point `pc_cs` (which must be smaller than the thread's size), call the sequentialized thread function with the original arguments, and store the context switch point for the next round. Lazy-CSeq obtains from the unrolling phase the set of thread instances that the original concurrent program can possibly create within the given bounds. This allows the static construction of the main driver. Note that the choice of the context switch points in the driver is the only additional non-determinism introduced by the sequentialization.

```
if (active_tr[thr_idx] == 1) {
   pc_cs = pc[thr_idx] + nondet_uint();
   assume(pc_cs <= SIZE_<thr_idx>);
   thread_<thr_tdx>(thr_args[thr_idx]);
   pc[thr_idx] = pc_cs;
}
```

Thread Translation. The sequentialized program also contains a function for each thread instance (including the original `main`) identified during the unrolling phase. Within the function each statement is guarded by a check whether its location is before the stored location or after the next context switch non-deterministically chosen by the driver. In the former case, the statement has already been executed in a previous round, and the simulation jumps ahead one hop; in the latter case, the statement will be executed in a future round, and the simulation jumps to the thread's exit. Each jump target (corresponding either directly to a `goto` label or indirectly to a branch of an `if` statement) is also guarded by an additional check to ensure that the jump does not jump over the context switch. Since bLMP only explores states reachable in the

original concurrent program, `assert` statements need no special treatment during the sequentialization and can be delegated entirely to the backend model checker.

3 Architecture, Implementation, and Availability

Architecture. Lazy-CSeq is implemented as a source-to-source transformation tool in Python (v2.7.1). Like CSeq [2,3] and MU-CSeq [7] it uses the `pycparser` (v2.10, `github.com/eliben/pycparser`) to parse a C program into an abstract syntax tree (AST). However, in order to produce the right jump targets Lazy-CSeq unrolls all loops and replicates the thread functions. The sequentialized program can then be processed independently by any sequential verification tool for C. Lazy-CSeq has been tested with CBMC (v4.5, `www.cprover.org/cbmc/`) and ESBMC (v1.22, `www.esbmc.org`).

A small wrapper script bundles up translation and verification. It also invokes Lazy-CSeq repeatedly, with the parameters `-f2 -w2 -r2 -d135`, `-f4 -w4 -r1 -d145`, `-f16 -w1 -r1 -d220`, and `-f11 -w1 -r11 -d150`. Here `f` and `w` are the unwind bound for `for` (i.e. bounded) and `while` (i.e. potentially unbounded) loops, respectively, `r` is the number of rounds, and `d` is the depth option for the backend. We leave the analysis running to completion every time, without timeouts or memory limits. When the result is `TRUE`, the scripts restarts the analysis with the next set of parameters. As soon the script gets `FALSE`, it returns `FALSE`. Only if the analysis using the last set of parameters is finished and the results is `TRUE`, then the scripts returns `TRUE`.

Availability and Installation. Lazy-CSeq can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq/lazy-cseq-0.1.zip`; it also requires installation of the `pycparser`. It can be installed as global Python script. In the competition we only used CBMC as a sequential verification backend; this must be installed in the same directory as Lazy-CSeq.

Call. Lazy-CSeq should be called in the installation directory as follows:
`lazy-cseq.py -i<file> --spec<specfile> --witness<logfile>`

Strengths and Weaknesses. Since Lazy-CSeq is not a full verification tool but only a concurrency pre-processor, we only competed in the `Concurrency` category. Here it achieved a perfect score.

References

1. Chaki, S., Gurfinkel, A., Strichman, O.: Time-bounded analysis of real-time systems. In: FM-CAD, pp. 72–80 (2011)
2. Fischer, B., Inverso, O., Parlato, G.: CSeq: A Sequentialization Tool for C (Competition Contribution). In: Piterman, N., Smolka, S.A. (eds.) TACAS 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. 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)
5. La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing parameterized programs. In: FIT, EPTCS 87, pp. 34–47 (2012)
6. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009)
7. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings (Competition Contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 402–404. Springer, Heidelberg (2014)