Lazy-CSeq 0.6c: An Improved 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

Abstract. We describe an improved version of the bounded lazy sequentialization schema, and its implementation within the CSeq framework for sequentially consistent C programs using POSIX threads. The new schema uses an optimized representation of the context switch points and eagerly guesses these, but retains the other characteristics of the original lazy schema. Experiments show that the optimizations lead to substantial performance gains.

1 Introduction

Sequentialization translates concurrent programs into (under certain assumptions) equivalent nondeterministic sequential programs and so reduces concurrent verification to its sequential counterpart. The first general sequentialization schema for an arbitrary but bounded number of rounds was proposed by Lal and Reps (LR) [6]. This schema uses one copy of the shared memory for each round; the initial values of all memory copies are nondeterministically guessed in the beginning (*eager* exploration), while the transition between rounds (i.e., context switch points) are guessed *lazily* during the execution of the sequential program. LR is widely used but since it induces large formulas (due to the repeated memory copies) with a high degree of nondeterminism (due to the eager guess of the inital values of the copies), lazy schemas [5] have found interest. In particular, we have developed a lazy sequentialization schema for bounded programs that is carefully designed to introduce very small memory overheads and very few sources of nondeterminism, so that it produces simple formulas, and is thus already very effective in practice [3, 4].

In this paper, we describe an improved version of this bounded lazy sequentialization schema, and its implementation within the CSeq framework for sequentially consistent C programs using POSIX threads. The new schema uses an optimized representation of the context switch points and eagerly guesses these, but retains the other characteristics of the original lazy schema. Experiments show that the optimizations lead to substantial performance gains.

⋆ Contact author: Omar Inverso, oi2c11@ecs.soton.ac.uk.

2 Verification Approach

Overview. The verification follows the same approach as before [3, 4], i.e., the concurrent program P is bounded and unrolled so that there is only a bounded number of possible threads, that each statement is executed at most once, and that all jumps are forward jumps, and then translated into into a sequential program P' that simulates all computations that P can execute in round-robin schedules with K rounds. P' is finally verified using CBMC (v4.9 revision 4648, `www.cprover.org/cbmc/`).

P' consists of a main driver function and a simulation function for each thread instance (including the original `main`) identified during the unrolling phase. The driver maintains, for each thread, the program location at which the previous round's context switch has happened and thus the computation must resume in the next round. In addition, it also maintains the schedule, in form of a variable that, for each thread and each round, contains the number of visible statements that the thread is allowed to execute in the corresponding round. The driver orchestrates the simulation by first guessing the run lengths, and then repeatedly calling the simulation functions in a round-robin fashion. The simulation functions jump (in multiple hops) back to the stored locations context switch locations and execute the allowed number of visible statements.

The translation also converts thread-local variables into `static` variables, so that the simulation functions do not need to re-compute their values from saved copies of previous global memory states before they resume the computation.

Data Structures. P' stores and maintains, for each thread, a flag denoting whether the thread is active, the thread's original arguments, and the program location at which the previous context switch has happened. In addition, the new version also maintains, for each thread, the length of each round. One important optimization is that all variables in P' that refer to program locations (i.e., the context switch locations, the round lengths, and the current program counters) can now be kept separate for each thread, which allows us to use CBMC's bitvectors with different sizes as data types, and so to reduce the memory overhead induced by the translation.

Main Driver. The sequentialized program's main function consists of two phases. The first phase simply guesses the round lengths, and ensures that the guesses are smaller than the corresponding thread sizes. Note that this is the only additional nondeterminism introduced by the sequentialization. The second phase consists of a sequence of small code snippets, one for each thread and each round, that check the thread's active flag and, if this is set, set the next context switch point, call the sequentialized thread function with the original arguments, and store the context switch point for the next round.

Thread Translation. Within the simulation function for each thread instance, each statement is guarded by a check whether its location is before the stored location or after the guessed next context switch. 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.

3 Architecture, Implementation, and Availability

Architecture. Lazy-CSeq 0.6c is implemented as a source-to-source transformation tool in Python (v2.7.5) within the CSeq framework. This 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 (in principle) be processed independently by any sequential verification tool for C, but Lazy-CSeq 0.6c has been more tightly integrated with CBMC (v4.9, revision 4648), and uses the non-standard bitvector data type provided by this version.

A small wrapper script bundles up translation and verification. It also invokes Lazy-CSeq repeatedly, with the parameters `-f1 -F50 -w1 -r2 -d800`, `-f2 -w2 -r2 -d200`, `-f4 -w4 -r1 -d150`, `-f16 -w1 -r1 -d350`, and `-f1 -F11 -w1 -r11 -d300`. Here `f` and `F` are soft and hard unwind bounds for `for` (i.e. bounded) loops, `w` the unwind bound for `while` (i.e. potentially unbounded) loops, respectively, `r` is the number of rounds, and `d` is the depth option for the backend. We set a timeout of 200s for the first four calls and 800s for the last one. The script starts by invoking Lazy-CSeq with the first set of parameters. As soon as the tool detects a reachable error condition within the given bounds, the script reports `FALSE` and terminates; the analysis restarts with the next set of parameters otherwise. If the last invocation reports no reachable error conditions, the script returns `TRUE`.

Availability and Installation. Lazy-CSeq can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq/newseq-0.6c.tar.gz`; 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>`

References

1. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Sequentialization Tool for C (Competition Contribution). *TACAS*. LNCS 7795, pp. 616-618, 2013.
2. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-Processor for Sequential C Verification Tools. *ASE*, pp. 710-713, 2013.
3. 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.
4. 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.
5. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. *CAV*, LNCS 5643, pp. 477–492, 2009.
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.