Lazy-CSeq 1.0 [⋆]
(Competition Contribution)

Omar Inverso[1], Truc L. Nguyen[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. Sequentialization translates concurrent programs into (under certain assumptions) equivalent nondeterministic sequential programs and so reduces concurrent verification to its sequential counterpart. In previous work, we have developed and implemented in the Lazy-CSeq tool a lazy sequentialization schema for bounded programs that introduces very small memory overheads and very few sources of nondeterminism and is thus very effective in practice [1, 2].
The current version of Lazy-CSeq adds deadlock detection, counterexample generation, and explicit schedule control. It also implements an improved version of the original schema, which uses an optimized representation of the context switch points and eagerly guesses these, but retains its other characteristics. Experiments show that these optimizations lead to some performance gains.

1 Verification Approach

Overview. Lazy-CSeq 1.0 follows the same general verification approach as before, 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 a sequential program P' that simulates all computations that P can execute in round-robin schedules with K rounds. P' consists of a main driver function and a simulation function for each thread instance (including the original `main`) identified during the unrolling phase. P' is finally verified using a verification backend for sequential programs.

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) are now kept separate for each thread, which allows us to use 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 all round lengths, and ensures that the guesses are smaller

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

than the corresponding thread sizes. In our experience this leads to simpler verification conditions than the original approach, where the individual run lengths were guessed right before the corresponding sequentialized thread function were called. 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.

Deadlock Detection. Lazy-CSeq 1.0 now also supports deadlock detection (although this property is not required by any of the benchmarks). It uses an array of thread identifiers to represent the thread dependency chain, and non-deterministically guesses the chain on the fly while simulating the threads; see [3] for details.

Explicit Schedule Control. Lazy-CSeq 1.0 now also allows users to control the schedule exploration round-by-round; in earlier versions in each round all threads were scheduled and always in the same order. Note that even when the schedule is fixed context switches can still happen at any time. We can use this new feature to guide the analysis when some specific facts on scheduling are known, or on complex problems where even analyzing a single round would require too many resources. Specific schedules can help to trim down the main driver and result in smaller verification conditions with consequent performance improvements.

2 Software Architecture

Lazy-CSeq 1.0 is implemented as a source-to-source transformation tool in Python (v2.7.9) within the CSeq framework. However, while this year's version implements the same schema as last year's version, the implementation itself has changed substantially, due to a complete framework refactoring [3]. The framework now consists of independent modules that can be configured and composed easily. Lazy-CSeq is implemented as CSeq configuration of eighteen modules, which include (*i*) the frontend processing module, which is based on the `pycparser` (v2.14, `github.com/eliben/pycparser`); (*ii*) eight simple transformation modules to rewrite the input program in steps into a progressively simplified syntax; (*iii*) four translators for program flattening to produce a bounded program (see [2]); (*iv*) two modules implementing the sequentialization algorithm that produces a backend-independent sequentialized file (see [2]); (*v*) a standard program instrumentation to instrument the sequentialized file for a specific backend; and (*vi*) two wrappers for backend invocation and user report generation or counterexample translation. The new framework also provides a line and variable mapping between the original program and its sequentialized counterpart, which we used to implement a counterexample generation that was missing from previous versions.

The new framework simplifies the integration of new sequential backends (e.g., P. Gonzalez de Aledo's integration with the Forest tool [4]), but Lazy-CSeq 1.0 remains tightly integrated with CBMC and uses its non-standard bitvector data type. However, due an internal error in CBMC we had to regress to CBMC v4.6.

3 Tool Setup and Configuration

Availability and Installation. Lazy-CSeq can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq/lazy-cseq-1.0-svcomp16.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. The wrapper script for the tool on the BenchExec repository is `lazycseq.py`.

Call. Lazy-CSeq only participates in the concurrency category. It should be called in the installation directory as follows: `lazy-cseq.py -i<file> --spec<specfile> --witness<logfile>`. This small wrapper script bundles up translation and verification and repeatedly calls Lazy-CSeq, with the following parameters:

`--unwind-for=1 --unwind-for-max=50 --unwind-while=1 --rounds=2 --depth=800`,

`--unwind-for=2 --unwind-while=2 --rounds=2 --depth=300`,

`--unwind-for=4 --unwind-while=4 --rounds=1 --depth=150`,

`--unwind-for=16 --unwind-while=1 --rounds=1 --depth=350`, and

`--unwind-for=1 --unwind-for-max=11 --unwind-while=1 --rounds=11 --depth=400`.

Here `unwind-for` and `unwind-for-max` are soft and hard unwind bounds for bounded (i.e., `for`) loops, `unwind-while` the unwind bound for potentially unbounded (i.e., `while`) loops, respectively, `rounds` is the number of rounds, and `depth` is the depth option for the backend. 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`.

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 we achieved a perfect score. Compared to Lazy-CSeq 0.6c, we achieved a speedup of approx. 60% over all unsafe benchmarks, but this is skewed by two instances that time out using v0.6c but are solved very quickly using v1.0; discounting these two instances we achieved a speedup of approx. 14%.

References

1. 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.
2. 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.
3. O. Inverso, T.L. Nguyen, B. Fischer, S. La Torre, G. Parlato. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-Threaded C-Programs. *ASE Tool Demonstration*, 2015.
4. P. Gonzalez de Aledo, P. Sanchez Espeso. FramewORk for Embedded System verification (Competition Contribution). *TACAS*, LNCS 9035, pp. 429-431, 2015.