Lazy-CSeq 2.0: Combining Lazy Sequentialization with Abstract Interpretation*

(Competition Contribution)

Truc L. Nguyen[1], Omar Inverso[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. Lazy sequentialization has emerged as one of the most effective techniques to find bugs in concurrent programs. However, the size of the shared global and thread-local state still poses a problem for further scaling. We therefore use abstract interpretation to minimize the representation of the concurrent program's state variables. More specifically, we run the Frama-C abstract interpretation tool over the sequentialized program output by Lazy-CSeq to compute over-approximating intervals for all (original) state variables and then exploit CBMC's bitvector support to reduce the number of bits required to represent these in the sequentialized program. We demonstrate that this leads to substantial performance gains on complex instances.

1 Verification Approach

Overview. In recent editions of the software verification competition [1, 5, 9, 10], as well as in complex industrial case studies [11], sequentialization has proven to be a very effective program verification approach expecially for bug-hunting purposes. However, the size of the shared global and thread-local state still poses a problem for further scaling. In an experiment [11], we manually reduced the size of the state variables to the minimum required to find the bug (three bits in the case of safestack), which lead to a 20x speed-up. This clearly indicates the potential benefits of such a reduction.

Here, we automate this reduction and integrate abstract interpretation into the lazy sequentialization described in [6], in order to minimize the representation of the concurrent program's state variables, and to scale up sequentialization to more complex concurrent verification tasks. This integration of abstract interpretation is the main novelty of Lazy-CSeq 2.0 over previous versions [5, 8].

More specifically, we use abstract interpretation to over-approximate the intervals of all variables of the sequentialized program P' corresponding to a given a concurrent program P. Then, we replace in P' the original state variables from P with bitvectors of sizes sufficient to safely represent them; these bitvectors are often much smaller than

* Partially supported by EPSRC EP/M008991/1, INDAM-GNCS 2016, and MIUR-FARB 2014-2016 grants. Contact author: Gennaro Parlato, gennaro@ecs.soton.ac.uk.

the original data types. Finally, the resulting sequential program P'' is verified using an off-the-shelf verification backend for sequential programs. In Lazy-CSeq 2.0, we rely on Frama-C [2] as abstract interpretation framework for the interval analysis and CBMC [3] as sequential verification backend with native support for bitvectors.

In more detail, we first transform the input concurrent program P into a bounded concurrent program by inlining the functions and unwinding the loops up to a given depth. Then, we sequentialize this program by bounding the number of rounds of thread executions; in each round all threads are executed at most once and always in the same order [6]. The resulting non-deterministic sequential program P' simulates all computations that P can execute in the given number of round-robin schedules and loop unwinding depth. Program flattening guarantees that in P' there is a bounded number of threads, that each statement is executed at most once, and that all jumps are forward. P' consists of a main driver function and a simulation function for each thread instance (including the original `main`) identified during the unrolling phase.

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, Lazy-CSeq 2.0 also maintains, for each thread, the length of each round. An 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 introduced by the translation. Further, as mentioned above, the original state variables of P are represented in P'' using bitvectors of a possibly more compact size, safely over-approximated using abstraction-based interval analysis.

Main Driver. The main function of P' consists of two phases. The first phase simply guesses all round lengths, and ensures that the guesses are smaller 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 functions were called. The second phase consists of a sequence of small code snippets, one for each thread and each round, that (if the thread's active flag is on) 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.

2 Software Architecture

Lazy-CSeq 2.0 is implemented as a source-to-source transformation tool in Python (v2.7.9) within the CSeq framework [4, 7], which consists of independent modules that

can be configured and composed easily. In particular, it is implemented as CSeq configuration of about twenty modules, which include (*i*) the frontend processing module, which is based on the `pycparser` (v2.14, `github.com/eliben/pycparser`); (*ii*) simple transformation modules to rewrite the input program in steps into a progressively simplified syntax; (*iii*) translators for program flattening to produce a bounded program [6]; (*iv*) two modules implementing the sequentialization algorithm and that produce a backend-independent sequentialized file [6]; (*v*) wrappers for the abstract interpretation backend and for transforming the program's state variables into bitvectors of compact size, exploiting the over-approximated intervals; (*vi*) a standard program instrumentation to adapt the sequentialized file for a specific backend; and (*vii*) wrappers for backend invocation and user report generation or counterexample translation.

Due to CSeq's source-to-source translation architecture, we can use Frama-C as a black box. We simply run it over the sequentialized program and extract, for each state variable, the intervals estimated at the end of P'. Since these over-approximate the size requried to hold the variables' values at any given program point, the bitvector transformation can simply compute bitvector sizes from the upper bounds of these intervals.

3 Tool Setup and Configuration

Availability and Installation. Lazy-CSeq 2.0 can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq/lazy-cseq-2.0-svcomp17.tar.gz`. It can be installed as global Python script. It requires installation of the `pycparser`, CBMC (v5.6), and Frama-C (Aluminium version); CBMC must be installed in the same directory as the Python script. For convenience, our archive contains the required CBMC and Frama-C versions. The wrapper script for the tool on the BenchExec repository is `lazycseqabs.py`.

Call. Lazy-CSeq 2.0 only participates in the concurrency category. It should be called in the installation directory using a wrapper script as follows:

```
lazy-cseq-abs.py -i<file> --spec<specfile> --witness<logfile>.
```

Note that Lazy-CSeq 2.0 produces a witness in a CBMC-like textual format, since there is no witness format for concurrent programs. The wrapper script bundles up translation and verification and calls Lazy-CSeq 2.0 six times, with different parameters and bounds. As soon as it detects a reachable error condition within the given bounds, it reports `FALSE` and terminates; otherwise it continues with the next set of parameters otherwise. If the last invocation reports no reachable error conditions, the script returns `TRUE`.

4 Strengths and Weaknesses

Since Lazy-CSeq 2.0 is not a full verification tool but only a concurrency pre-processor, we only competed in the `Concurrency` category.

Lazy sequentialization has already proven to be effective, especially in a bug-hunting setting, in recent editions of the software verification competition. The strength of this year's approach is in the compact bitblasting induced by the combined use of abstract

interpretation's interval analysis and bitvector support. This can indeed provide significant analysis speedups on complex problems. In particular, interval analysis turns out to be quite lightweight yet quite accurate even on such problems, perhaps due to the particularly simple structure of the sequentialized programs. In practice, the interval analysis requires only a few hundreds of milliseconds to a few seconds, and overall verification times can improve by tens of seconds.

The intervals of the program's state variables are safely over-approximated, to minimize the number of bits needed for their representation while avoiding overflow problems. This enabled us to correctly solve all benchmarks.

On the other hand, one possible weakness of our approach is that a judicious choice of bounding parameters is essential, because it is ultimately based on bounded model-checking. This is not really a problem in the competition setting, where fine-tuning of the parameters is possible during the training phase.

References

1. D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (Report on SV-COMP 2016). *TACAS*, LNCS 8413, pp. 887–904, 2016.
2. G. Canet, P. Cuoq, B. Monate. A Value Analysis for C Programs. *SCAM*, pp. 123–124, 2009.
3. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *TACAS*, LNCS 2988, pp. 168–176, 2004.
4. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-Processor for Sequential C Verification Tools. *ASE*, pp. 710-713, 2013.
5. 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.
6. 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.
7. 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*, pp. 807–812, 2015.
8. T.L. Nguyen, B. Fischer, S. La Torre, G. Parlato. Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs. *ATVA*, LNCS 9938, pp. 174–191, 2016.
9. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, G. Parlato. Verifying Concurrent Programs by Memory Unwinding. *TACAS*, LNCS 9035, pp. 551–565, 2015.
10. E. Tomasco, T.L. Nguyen, O. Inverso, B. Fischer, S. La Torre, G. Parlato. MU-CSeq 0.4: Individual Memory Location Unwindings (Competition Contribution). *TACAS*, LNCS 9636, pp. 938-941, 2016.
11. E. Tomasco, T.L. Nguyen, O. Inverso, B. Fischer, S. La Torre, G. Parlato. Lazy sequentialization for TSO and PSO via shared memory abstractions. *FMCAD*, pp. 193–200, 2016.