MU-CSeq 0.3: Sequentialization by Read-Implicit and Coarse-Grained Memory Unwindings*
(Competition Contribution)

Ermenegildo Tomasco[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 di Salerno, Italy

Abstract. We describe a new CSeq module that implements improved algorithms for the verification of multi-threaded C programs with dynamic thread creation. It is based on sequentializing the programs according to a guessed sequence of write operations in the shared memory (memory unwinding, MU). The original algorithm (implemented in MU-CSeq 0.1) stores the values of all shared variables for each write (read-explicit fine-grained MU), which requires multiple copies of the shared variables. Our new algorithms store only the writes (read-implicit MU) or only a subset of the writes (coarse-grained MU), which reduces the memory footprint of the unwinding and so allows larger unwinding bounds.

1 Introduction

Sequentializations translate concurrent programs into sequential ones while preserving a given verification property (e.g., reachability). They reuse sequential verification tools and offer many advantages, such as the ability to focus on the concurrency aspects of a language, to quickly experiment with different approaches, and to build robust verification tools with less effort. We develop the CSeq tool as a modular sequentialization framework [1, 2] for multi-threaded C programs with dynamic thread creation. It contains modules for the Lal/Reps scheme [5], a lazy sequentialization scheme aimed at bounded model checking [3, 4], and a memory unwinding scheme [6].

A *memory unwinding* (MU) is an explicit representation of the write operations into the shared memory as a sequence that contains for each write the writing thread, the variable, and the written value. We can vary which writes are represented and thus exposed to the other threads, which leads to different strategies with different performance characteristics. In a *fine-grained* MU *every* write operation is represented explicitly and individually. In a *coarse-grained* MU we only represent a subset of the writes and group together multiple writes (by exposing for each group only the last write for each variable). In an *intra-thread* MU the writes in one group are all executed by one thread; the writes not represented can thus be seen as having been superseded by subsequent writes in the same context. In an *inter-thread* MU the writes in one group can come from different threads, thus summarizing the effect of multiple context switches.

* Partially supported by EPSRC EP/M008991/1, INDAM-GNCS 2014, and MIUR-FARB 2012-2014 grants. Contact author: Ermenegildo Tomasco, et1m11@ecs.soton.ac.uk.

2 Verification Approach

Overview. Our approach can be seen as an *eager sequentialization* of the original concurrent program P over the unwound memory. We first guess an *n-memory unwinding* of P, i.e., a sequence $w_1 \dots w_n$ identifying the threads, the shared variables and the values involved in the write operations of P. We then simulate all runs of P that are *compatible* with this guess. For the simulation, each thread is translated into a simulation function where write and read accesses over the shared memory are replaced by operations over the unwound memory. The simulation functions are executed sequentially, starting from the main function; each thread creation is translated into a call to the corresponding simulation function. All context switches are implicitly simulated through the MU. We adapt this general sequentialization scheme with different implementations, in particular for the functions to read from / write into the shared memory and for dynamic thread creation. The details can be found in [7].

Fine-grained MU. In this approach, all the writes of a P run are considered meaningful to the other threads and thus exposed. We store each of them individually in the memory unwinding, with three arrays reporting respectively for each position the writing thread, the variable name and the written value. For an efficient implementation of the MU API functions, we also store some additional data such as the index of the last write performed in the simulation and a table containing, for each position and thread t, the position of next write of t in the memory unwinding.

We distinguish between the *read-explicit* and the *read-implicit* schemes. In the first case, all shared variables are replicated at each position of the sequence; we used this schema in MU-CSeq 0.1 [6]. Its main feature is that the value of each shared variable can be read directly at each step. It thus trades memory consumption for a simple logic in the implementation of the MU API. In the second scheme, at each position in the sequence, we copy only the shared variable that is modified by the corresponding write. The implementation of the MU API becomes more involved, but it yields an effective gain when the number of shared variables is large compared to the number of writes. We have also mixed the two schemes into a third one that is read-explicit for scalar variables and read-implicit for the arrays.

Coarse-grained MU. In this approach, we store at each position of the sequence a partial mapping from the shared variables to values, with the meaning that the variables in the domain of the mapping are modified from the previous position and the value given by the mapping is their value at this position. A variable that is modified at position $i + 1$ could also be modified between positions i and $i + 1$ by other writes that are not exposed in the sequence. Thus, by exposing only some of the writes of a run (1) we restrict the number of possible runs that can match a MU (in fact, the unexposed writes cannot be read externally, and thus some possible interleavings of the threads are ruled out) and (2) we handle larger number of writes by nondeterministically deeming only some of them as interesting for the other threads.

In this approach we also distinguish between the cases in which either only one (*intra-thread coarse-grained* MU) or multiple (*inter-thread coarse-grained* MU) threads are allowed to modify the variables. Both variants can be realized as read-implicit, read-explicit and mixed schemes.

3 Architecture, Tool setup, and Configuration

Architecture. MU-CSeq 0.3 is implemented as source-to-source transformations in Python, 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), and then traverses the AST to construct a sequentialized version, as outlined above. The resulting program can be processed independently by any verification tool for C, but we have only tested MU-CSeq 0.3 with CBMC (v4.9 revision 4648, `www.cprover.org/cbmc/`). For the competition we use a wrapper script that bundles up the translation and calls CBMC for verification. The wrapper returns the output from CBMC.

We use a simple syntactic analysis of the program to determine which schema and parameters we use. In particular, if the program contains arrays we use the mixed fine-grained MU with parameters `-w25 -t10 -f2 -u2 -thl0`; here `w` (resp., `t`) is the bound on the number of write operations (resp., of spawned threads), `f` is the unwind bound for `for` and `u` is the unwind bound for the remaining loops, and `thl` is the bound on the number of threads that are spawned in any *while*-loop. If the program contains more than 30 assignments but no loop, or a `pthread_create` inside a *for*-loop, we switch to the inter-thread coarse-grained MU, with parameters `-w2 -t52 -f52 -u1 -thl0`. In all other cases we use again the first schema but with parameters `-w23 -t10 -f12 -u1 -thl3`. We use a timeout of 850 seconds, and interpret the single case where this timeout applies as *true*.

Availability and Installation. MU-CSeq 0.3 is available at `http://users.ecs.soton.ac.uk/gp4/cseq/mu-cseq-0.3.zip`; it also requires installation of the `pycparser`. CBMC must be installed in the same directory as MU-CSeq.

Call. MU-CSeq should be called in the installation directory as follows: `mu-cseq.py -i` *`file`* `--spec` *`specfile`* `--witness` *`logfile`*.

Strengths and Weaknesses. MU-CSeq participates only in the concurrency category. It returns the correct answers for all problems in this category, but is slower than Lazy-CSeq, thus winning the Silver medal.

References

1. B. Fischer, O. Inverso, G. Parlato. CSeq: A Sequentialization Tool for C (Competition Contribution). *TACAS*, LNCS 7795, pp. 616-618, 2013.
2. B. Fischer, O. Inverso, 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. A. Lal, T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.
6. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, G. Parlato. MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings (Competition Contribution). *TACAS*, LNCS 8413, pp. 402-404, 2014.
7. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, G. Parlato. Verifying Concurrent Programs by Memory Unwinding. *TACAS*, this volume, 2015.