MU-CSeq 0.4: Individual Memory Location Unwindings (Competition Contribution)

Ermenegildo Tomasco[1], Truc L. Nguyen[1], Omar Inverso[1,2], Bernd Fischer[3], Salvatore La Torre[4], and Gennaro Parlato[1](✉)

[1] Electronics and Computer Science, University of Southampton, Southampton, UK
gennaro@ecs.soton.ac.uk
[2] Gran Sasso Science Institute, L'Aquila, Italy
[3] Division of Computer Science, Stellenbosch University, Stellenbosch, South Africa
[4] Dipartimento di Informatica, Università di Salerno, Salerno, Italy

Abstract. We present the MU-CSeq tool for the verification of multi-threaded C programs with dynamic thread creation, dynamic memory allocation, and pointer arithmetic. It is based on sequentializing the programs over the new notion of *individual memory location unwinding* (IMU). IMU is derived from the notion of memory unwinding that has been implemented in the previous versions of MU-CSeq. The main concepts of IMU are: (1) the use of multiple write sequences, one for each individual shared memory location that is *effectively* used in the executions and (2) the use of memory addresses rather than variable names in the operations on the shared memory, which requires a separate table to map write sequences but supports pointer arithmetic.

1 Verification Approach

MU-CSeq 0.4 follows the sequentialization approach to verification. Its idea is to translate, using a code-to-code translation that preserves the verification property of interest, a concurrent program into a sequential one, which is then analyzed using a symbolic sequential verification tool.

In MU-CSeq 0.4 we have implemented a sequentialization based on the novel notion of *individual memory location unwindings* (IMU). IMU is derived from the concept of memory unwinding that has been implemented in the previous versions of MU-CSeq [2,3]. A *memory unwinding* (MU) is an explicit representation of the sequence of write operations into the shared memory performed by the threads. Each element of the sequence represents a write operation characterized by the identifier of the writing thread, the variable identifier, and the written value. The sequentialized program first guesses the values in the MU using non-determinism–supported by symbolic verification tools–and then simulates each thread against the MU. If each thread matches its memory writes in

Partially supported by EPSRC EP/M008991/1 grant, INDAM-GNCS 2015 grant and MIUR-FARB 2013–2015 grants.

DOI: 10.1007/978-3-662-49674-9_65

the MU then their sequential simulation corresponds to a valid execution of the original concurrent program (see [2] for more details).

IMU improves on MU by providing a separate memory unwinding for each individual shared memory location corresponding to a scalar type or a pointer. To recreate a global total order over the shared memory writes we associate a *timestamp* (i.e., a distinct natural number) with each write in each individual MU. This is crucial for the correctness of the simulation since it is used to synchronize the simulation of the individual threads (otherwise the distinct MUs can give rise to many total orders).

Another important feature of the new encoding is to associate each memory location with its physical memory address. When a read or write operation is performed using a memory address, e.g., `*p=3` for a pointer variable `p`, we first search for the location corresponding to the value of `p` and then simulate the read/write operation as we would do for scalar variables (for which the locations are statically known).

This new representation of the writes has several good features when used in combination with sequential BMC verification tools. In particular, the use of the individual MU simplifies the simulation of read and write operations resulting in much smaller verification conditions and verification time. In fact, for each memory access, the formula now only contains an encoding of the corresponding individual sequence and not the whole sequence of writes. Although the high level idea is simple, we observe that the underlying reasoning for IMU is more involved than MU.

Another advantage of IMU is that it gives a simple and effective way to support dynamic memory allocation and pointer arithmetics. This feature was not implemented in previous versions of MU-CSeq as it requires convoluted simulation functions resulting in a blowup of the verification time of the sequential BMC backend analysis.

IMU not only improves MU as we have mentioned above but also simplifies the development of new sequentialization schemes for other interesting properties of concurrent programs such as data-race and deadlock detection as well as weak memory models including TSO and PSO.

2 Software Architecture

The sequentializations in MU-CSeq 0.4 are implemented as source-to-source transformations in Python (v2.7.9), within the re-factored CSeq framework [4]. This uses the `pycparser` (v2.14, http://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 0.4 with CBMC (v5.2, www.cprover.org/cbmc/). For the competition we use a wrapper script that bundles up the translation, calls CBMC for verification, and returns its output.

Our tool takes the following options: `w` is the bound on the number of write operations for each location, `f` is the unwind bound for *for*-loops, `u` is the unwind bound for the remaining loops, `b` is the number of bits used for shared variables and memory addresses, `p` is the number of tracked locations that are stored on the heap, `m` is the maximal number of malloc invocations, `v` is the bound on the number of lock/unlock operations on single locations, `ml` is the bound on the number of lock/unlock operations on the whole memory, and `thl` is the bound on the number of threads that are spawned in any *while*-loop.

We use a simple syntactic analysis of the program to determine which schema and parameters we use in the competition. If the program contains more than 30 assignments but no loops, or a `pthread_create` inside a constant bounded *for*-loop, we use the inter-thread coarse-grained MU with parameters `-w2 -f52 -u1 -b7` (for the MU scheme, `w` actually denotes the length of the overall sequence of writes). Otherwise we use the IMU scheme with the following parameters:

`-w7 -u1 -f2 -b12 -p5 -v6 -ml7 -m3 -thl3`, for programs with arrays;
`-w7 -u2 -f2 -b12 -p2 -v6 -ml7 -m3 -thl3`, if the program contains thread-local variables;
`-w`*<c1>* `-u1 -f`*<c1>* `-b17 -p2 -v6 -ml7 -m3 -thl3`, if the program's *for*-loops are upper bounded by a constant *<c1>* and do not contain `pthread_create`;
`-w6 -u1 -f2 -b7 -p2 -v6 -ml7 -m3 -thl3`, otherwise.

All parameter values were empirically determined. We use a timeout of 70 s, and interpret the cases where this timeout applies as *true*.

3 Tool Setup and Configuration

Availability and Installation. MU-CSeq 0.4 is available at http://users.ecs.soton.ac.uk/gp4/cseq/mu-cseq-0.4.zip; it also requires installation of the `pycparser`. CBMC must be installed in the same directory as MU-CSeq. The wrapper script for the tool on the BenchExec repository is `mu-cseq.py`.

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

Strengths and Weaknesses. Since MU-CSeq 0.4 is not a full verification tool but only a concurrency preprocessor, we only competed in the `Concurrency` category. Here we achieved a full score, with an overall runtime of circa 45 min for all benchmarks in the category. Compared to MU-CSeq 0.3 [2], the new version achieved a substantial speedup over most of the benchmarks, as shown by the scatter plot in Fig. 1.

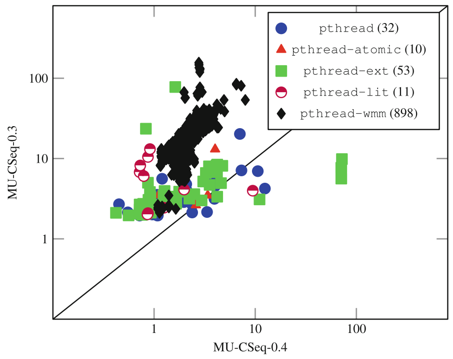

Fig. 1. Comparison of MU-CSeq v0.3 and v0.4.

References

1. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded c programs via lazy sequentialization. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 585–602. Springer, Heidelberg (2014)
2. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying concurrent programs by memory unwinding. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 551–565. Springer, Heidelberg (2015)
3. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: MU-CSeq 0.3: sequentialization by read-implicit and coarse-grained memory unwindings. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 436–438. Springer, Heidelberg (2015)
4. Inverso, O., Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: a context-bounded model checking tool for multi-threaded C-programs. In: ASE Tool Demonstration, pp. 807–812 (2015)