MU-CSeq: Sequentialization of C Programs by Shared 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
{et1m11,oi2c11,gennaro}@ecs.soton.ac.uk, bfischer@cs.sun.ac.za,
slatorre@unisa.it

Abstract. We implement a new sequentialization algorithm for multi-threaded C programs with dynamic thread creation as a new CSeq module. The novel basic idea of this algorithm is to fix (by a nondeterministic guess) the sequence of write operations in the shared memory and then simulate the behavior of the program according to any scheduling that respects this choice. Simulation is done thread-by-thread and the thread creation mechanism is replaced by function calls.

1 Introduction

Sequentialization translates a concurrent program into a corresponding sequential one while preserving a given verification property (e.g., reachability). The idea is to reuse in the domain of concurrent programs the technology developed for the analysis of sequential programs. This simplifies and speeds up the development of robust tools for concurrent programs. It also allows the designers to focus only on the concurrency aspects and provides them with a framework in which they can quickly check the effectiveness of their solutions. A sequentialization tool can be designed as a front-end for a number of analysis tools that share the same input language, and thus many alternatives are immediately available.

We design a new sequentialization algorithm for multi-threaded C programs with dynamic thread creation. Its main novelty is the idea of *memory unwinding* (MU). We fix (by a nondeterministic guess) the sequence of write operations in the shared memory and then simulate the behavior of the program according to any scheduling that respects this choice. We can then use of the number of writes in the shared memory as a parameter of the bounded analysis, which is orthogonal to considering the number of context switches underlying previous research on sequentializations based on the notion of bounded context-switching (e.g., [10,6,7,2,1,8,9]). Moreover, MU-CSeq naturally accommodates the simulation of dynamic thread creation by function calls.

We implement MU-CSeq as a new module of the tool CSeq [3,4]. Other modules of CSeq implement the Lal/Reps algorithm [6] and a lazy-sequentialization scheme aimed to exploit bounded model checking [5].

* This work was partially funded by the MIUR grant FARB 2011-2012, Università degli Studi di Salerno (Italy).

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 402–404, 2014.

2 Verification Approach

Overview. MU-CSeq translates a multi-threaded C program P, into a standard C program P'. The source-to-source translation is parameterized over the number of writes N_w in the shared memory and the maximum number of threads N_t. The overall scheme consists of guessing a sequence σ of N_w writes and then simulating any execution of P that matches σ. The simulation is done thread-by-thread, starting from the original main function; when a new thread is created the simulation of the current thread is suspended until the simulation of the new thread has ended. When the number of threads passes the bound N_t, each new thread creation operation is just ignored.

Modules of P'. The main function of P' is in charge of guessing a consistent sequence of writes σ and starting the simulation of P. P' has a function for each function (including the main) and each thread of P. The translation of P modules into the corresponding modules of P' consists of: 1) adding a few lines of control code to handle creation and execution of threads, and 2) replacing the reads and writes in the shared memory with calls to *_read* and *_write* functions, respectively.

Guessing the Sequence of Writes. We use a global two-dimensional array *_mem* that corresponds to the temporal unwinding of the shared memory according to the memory updates. Here, each column corresponds to an updating event (i.e., a *write*) in σ and each row corresponds to a variable. The entry *_mem*[*i,j*] contains the value of the i-th shared variable after the j-th write in σ. We use a second global array *_sigma* to store for each write the involved variable and the thread that has executed the write. To guess the writes, we assign non-deterministic values to these arrays. The main function of P' then uses assume statements to check the consistency of the values stored in the guessed arrays before starting the simulation of P.

Accessing Global Memory. On executing each thread t, we store in a variable *thr_pos* the index of the last executed write in σ. This variable is updated by *_read* and *_write*. On calling *_write* for the assignment *x*=*e*, *thr_pos* is updated to the corresponding index and then *_mem*[*x,thr_pos*]=*e* is checked. By calling *_read* for reading variable x, first *thr_pos* is nondeterministically updated to any index between its current value and the next write in σ by t, and then *_mem*[*x,thr_pos*] is returned.

Thread Creation and Execution. Thread creation and execution are implemented as function calls in P'. Thus, if a thread t_2 is created from a thread t_1, the simulation of t_1 stops until the call to t_2 has terminated. Before the simulation of t_2 starts, the current value of *thr_pos* is stored in a local variable such that when t_2 has terminated, the simulation of t_1 restarts from this index. Accordingly, the simulation of t_2 starts from the current value of *thr_pos*. When either the last statement of thread t_2 is reached, or a write after the last guessed write for t_2 is executed, or an index greater than N_w is guessed for a read, then all the calls of thread t_2 are returned, including the call that has started the thread simulation. After the return we check that all write operations that t_2 has to execute actually happened.

3 Architecture, Tool Setup, and Configuration

Architecture. Our sequentialization is implemented as a source-to-source transformation in Python (v2.7.1), within the CSeq tool. It uses `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 the sequentialized version, as outlined above. The resulting program can be processed independently by any verification tool for C. MU-CSeq has been tested with CBMC (v4.2, `www.cprover.org/cbmc/`) and ESBMC (v1.22, `www.esbmc.org`). For the competition we use a wrapper script that bundles up the translation and calls CBMC for verification. We use the parameters `-w24 -t17 -f17 -unwind1 -depth4000 -MaxThreadCreate3`, where `w` (resp., `t`) is the bound on the number of write operations (resp., of spawned threads), `f` is the unwind bound for `for` and `unwind` is the unwind bound for the remaining loops, `depth` is the depth option for the backend, and `MaxThreadCreate` is the bound on the number of threads that are spawned in any `while` loop. No timeouts or memory limits are used in the analysis. The wrapper returns the output from CBMC.

Availability and Installation. MU-CSeq can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq/mu-cseq-0.1.zip`; 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 MU-CSeq.

Call. The tool should be called in the installation directory as follows:

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

Strengths and Weaknesses. Since MU-CSeq is not a full verification tool but only a concurrency pre-processor, we only competed in the `Concurrency` category. Here it achieved a perfect score.

References

1. Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011)
2. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: POPL, pp. 411–422 (2011)
3. Fischer, B., Inverso, O., Parlato, G.: CSeq: A Sequentialization Tool for C (Competition Contribution). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 616–618. Springer, Heidelberg (2013)
4. Fischer, B., Inverso, O., Parlato, G.: CSeq: A Concurrency Pre-Processor for Sequential C Verification Tools. In: ASE, pp. 710–713 (2013)
5. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: A Lazy Sequentialization tool for C (Competition Contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 398–401. Springer, Heidelberg (2014)
6. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009)
7. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
8. La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing parameterized programs. In: FIT, EPTCS 87, pp. 34–47 (2012)
9. La Torre, S., Parlato, G.: Scope-bounded Multistack Pushdown Systems: Fixed-Point, Sequentialization, and Tree-Width. In: FSTTCS. LIPIcs, vol. 18, pp. 173–184 (2012)
10. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI, pp. 14–24 (2004)