CSeq: A Sequentialization Tool for C
(Competition Contribution)

Bernd Fischer[1,2], Omar Inverso[1], and Gennaro Parlato[1]

[1] Electronics and Computer Science, University of Southampton, UK
[2] Department of Computer Science, Stellenbosch University, South Afric
{b.fischer,oi2c11,gennaro}@ecs.soton.ac.uk

Abstract. Sequentialization translates concurrent programs into equivalent non-deterministic sequential programs so that the different concurrent schedules no longer need to be handled explicitly. It can thus be used as a *concurrency pre-processor* for many sequential program verification techniques. CSeq implements sequentialization for C and uses ESBMC as sequential verification backend [5].

1 Introduction

Sequentialization is a recent verification technique that translates a concurrent program into a non-deterministic sequential program that (under certain assumptions) behaves equivalently, so that the different concurrent schedules do not need to be explicitly handled during verification. It can be implemented as a source-to-source program transformation and can be used as a *concurrency pre-processor* for sequential program verification tools, which in principle makes it an attractive and general approach.

However, in practice, only a few tools exist, and most of them work on an idealized language such as Boolean programs, or on an intermediate representation level, which makes them unsuitable as concurrency pre-processors for third-party tools. With CSeq, we aim to close this gap, and to develop a sequentialization tool for the full C language.

2 Verification Approach

Lal/Reps Sequentialization Schema. CSeq largely follows the schema proposed by Lal and Reps [7], which replaces the control non-determinism inherent to concurrent programs by data non-determinism. More specifically, it translates a concurrent program (t_1 `||` t_2) into a sequential but non-deterministic program (t_1' `;` t_2' `;` c), which contains additional copies of the shared global memory; c checks any assumptions on these copies that have been made independently by the transformed threads t_1' and t_2'.

CSeq replaces each shared variable x by a k-indexed entry $x[k]$ in an array of size K where k is an auxiliary variable called the current round counter and K is the round bound. The transformed program then calls the thread functions sequentially, in the same order in which they are created. It simulates a context switch simply by non-deterministically increasing k up to the round bound K; if k grows beyond K, an early return is enforced (i.e., the thread is pre-empted). CSeq inserts this simulation code at all sequence points of the original program.

In this schema, the first thread accesses a fresh copy of the memory for each round, with non-deterministically chosen values, while the subsequent threads always continue with the state left by their predecessor at each round. The initial guesses are stored in a second copy $x'[k]$; at the end of the program c then checks that each round has ended with the guesses that are used in the next round, i.e., that $x[j] = x'[j+1]$ holds; simulations that do not satisfy this condition do not correspond to feasible runs, and are discarded.

Since infeasible runs are only discarded at the end, assertion and reachability checking need to be integrated with the sequentialization; in particular, in order to prevent false results, errors can only be reported after the checker c has run. CSeq thus replaces all assertions by conditionals that set an error variable that is tested by c. The same argument also applies to implicit safety properties such as array bounds violations, or nil pointer dereferences that are handled by the applied backend verification tool. In principle, these need to be translated into explicit checks, and their detection by the backend needs to be explicitly suppressed. However, CSeq does currently not support this.

Related Approaches. Sequentialization was originally developed for two threads and two context switches only by Qadeer and Wu [9], but was subsequently generalized by Lal and Reps to a fixed number of threads and a parameterized number of round-robin scheduling [7]. Later, LaTorre/Madhusadan/Parlato extended [7] to track only reachable configurations [10]. Further extensions allowed modelling of unbounded, dynamic thread creation [6, 3, 11], and dynamically linked data structures allocated on the heap [2]. Like CSeq, Rek [4] implements sequentialization for C via code-to-code transformation, but it is targeted at real-time systems and hard-codes a specific scheduling policy. Poirot [8] also verifies concurrent C programs via sequentialization, but it first translates them into Boogie and then implements the sequentialization transformation at the Boogie level.

3 Architecture, Implementation, and Availability

Architecture. CSeq is implemented as a source-to-source transformation tool in Python (V2.7.1). It uses the `pycparser` (v2.08) [1] 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 result can then be processed by any verification tool for C; a small script (`cseq-esbmc`) bundles up translation and verification by ESBMC.

Availability and Installation. CSeq can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq.zip`. It must be installed as global Python script; it also requires installation of the `pycparser`, and ESBMC (v1.20, which is available at `www.esbmc.org`) must be on the path to use the `cseq-esbmc` script.

Call. For the competition, CSeq should be called in the installation directory as follows: `./cseq-esbmc <file>`.

Limitations. CSeq is in the initial development and there are still some limitations on the structure of the programs that can be translated, and on the properties that can be checked. Currently we assume that the `main` function consists of an initialization stage, in which the variables are initialized and a known number of threads is created, followed

by a shut-down stage that includes all (if any) `pthread_join`'s. We currently do not support conditional waiting nor `pthread_join` and `pthread_exit` with return variables. We implemented a deadlock check, but do not use it for the competition, as it is not required by the benchmarks. Since heap-allocated memory is accessible to all threads, it needs to be treated similarly to global variables; CSeq does not support this yet. Lifting these restrictions, and in particular supporting dynamic memory, dynamic thread creation, and conditional waiting will require significant efforts.

We further assume that the declarations for the global variables precede those for all functions, that there are no static variables and no global multi-dimensional arrays, and that local variables cannot shadow global variables. We do not support switch statements, due to limitations in the `pycparser`. These limitations simplified our implementation and can be lifted relatively easily.

Sequentialization is in principle independent of the verification tool used as back-end, but the current version of CSeq is (tightly) integrated with ESBMC. Despite this, ESBMC's counterexamples are not yet translated back into the original concurrent program, although this is a purely mechanic process.

4 Results

Since CSeq is a concurrency pre-processor, we only competed in the `Concurrency` category. Here, CSeq did well, and correctly solved 11 of the 33 benchmarks, with no false results, winning the Silver medal. In particular, it scored better than ESBMC v1.20 with its built-in concurrency handling. The existing implementation limitations showed particularly prominently in the CIL-preprocessed benchmarks, which CSeq could not handle.

References

1. E. Bendersky. Pycparser. `http://code.google.com/p/pycparser/`.
2. M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science*, 7(4), 2011.
3. A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. *SAS*, LNCS 6887, pp. 129–145, 2011.
4. S. Chaki, A. Gurfinkel, and O. Strichman. Time-bounded analysis of real-time systems. *FMCAD*, pp. 72–80, 2011.
5. L. Cordeiro and B. Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. *ICSE*, pp. 331–240, 2011.
6. M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. *POPL*, pp. 411–422, 2011.
7. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.
8. S. Qadeer. Poirot - a concurrency sleuth. *ICFEM*, LNCS 6991, pp. 15, 2011.
9. S. Qadeer and D. Wu. KISS: keep it simple and sequential. *PLDI*, pp. 14–24, 2004.
10. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. *CAV*, LNCS 5643, pp. 477–492, 2009.
11. S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing parameterized programs. *FIT*, EPTCS 87, pp. 34-47, 2012.