CSeq: A Concurrency Pre-processor for Sequential C Verification Tools

Bernd Fischer
Division of Computer Science
Stellenbosch University, South Africa
bfischer@cs.sun.ac.za

Omar Inverso
Electronics and Computer Science
University of Southampton, UK
oi2c11@ecs.soton.ac.uk

Gennaro Parlato
Electronics and Computer Science
University of Southampton, UK
gennaro@ecs.soton.ac.uk

Abstract—**Sequentialization translates concurrent programs into equivalent nondeterministic sequential programs so that the different concurrent schedules no longer need to be handled explicitly. It can thus be used as a concurrency preprocessing technique for automated sequential program verification tools. Our CSeq tool implements a novel sequentialization for C programs using pthreads, which extends the Lal/Reps sequentialization to support dynamic thread creation. CSeq now works with three different backend tools, CBMC, ESBMC, and LLBMC, and is competitive with state-of-the-art verification tools for concurrent programs.**

I. Introduction

Concurrency makes program verification substantially harder, and many verification tools (e.g., LLBMC [16]) restrict themselves to sequential programs. An alternative to extending such tools one-by-one is to build a generic *concurrency pre-processor*, which reduces concurrent program verification to its sequential counterpart so that the existing sequential tools can be used unchanged. Here, we describe such a tool for the C programming language, CSeq. It is based on sequentialization [15] which 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.

CSeq supports the verification of sequential consistent C programs that use POSIX threads [12]. It is implemented as a source-to-source program transformation and can thus in principle be used with different sequential program verification tools.

Our first prototype [10] had several restrictions on the input code and only worked with the ESBMC bounded model checker [8], however the initial experiments already indicated that the approach is feasible. In particular, the combination of CSeq and ESBMC performed better in the `Concurrency` category of the TACAS Software Verification Competition than ESBMC's native concurrency handling [4]. Here, we extend CSeq to work with different backend tools, including CBMC [7] and LLBMC. We also remove many limitations; in particular we add support for type declarations and structures as well as dynamic thread creation. The tool can now indeed be used as a *generic* concurrency pre-processor, possibly matching the performance of state-of-the art concurrent verifiers.

II. Verification Approach

CSeq generally follows the Lal/Reps sequentialization schema [15], which replaces the control non-determinism inherent to concurrent programs by data non-determinism. More specifically, thread creations are replaced by calls to functions that simulate the corresponding threads to completion. The simulation creates K copies of the shared global memory to consider K context switches for each thread. The simulation of the first thread starts with first copy of the memory, which it keeps updating until it reaches the non-deterministically choosen context switch point. It then continues with the next copy until it reaches the next context switch point, and repeats this up to the bound K. The next thread then starts its simulation in each round with the copy of the memory left by its predecessor. Each copy of the memory thus represents the snapshot seen by the thread simulations executing during the corresponding round of a round-robin schedule. In order to ensure that the thread simulations work on consistent snapshots, the simulation stores the initial global memory guesses in a second copy, and checks at the end (i.e., after all simulation functions have terminated) that the last thread has ended its simulaton in each round with initial guesses for the next round.

A. Supported Program Structure

CSeq assumes that the input program can be divided into four blocks of code: `#include` statements, declarations of global variables, function definitions, and `main` function definition. Fig. 1 sketches this structure. Both global and local variables can be scalar, arrays or `struct`s; `typedef`s are supported too. We assume that the above blocks do not mix and that their order is as shown. The use of non-standard include files and of parameters for the `main` function is not supported yet, but these assumptions can be lifted relatively easily.

B. Supported Pthread Features

CSeq covers all the basic pthread functionalities: thread creation and join, locks and conditional waiting. In particular, we extended the Lal/Reps scheme so that dynamic thread creation is now supported as well, and thread creation statements can be at any point in the input code. However, passing return values from `pthread_join` and `pthread_exit` or arguments to `pthread_create` is not supported yet.

```
#include <pthread.h>
...

type_g1 g1; type_g2 g2; ...

f() {
  type_x1 x1; type_x2 x2; ...
  stmt_1;
  stmt_2;
  ...
}

...

main() {
  ...
}
```

Fig. 1. Structure of original programs.

```
#include <cseq.h>
...

type_g1 g1[K]; type_g2 g2[K]; ...             (i)

f() {
  type_x1 x1; type_x2 x2; ...
  cs(); if(ret) return; stmt_1[k];            (ii)
  cs(); if(ret) return; stmt_2[k];
  ...
}

...

main_thread() {
  ...
}

main() {
  type_g1 _g1[K]; type_g2 _g2[K]; ...         (iii)

  for(i=1;i++;i<K) {                          (iv)
    _g1[i]=g1[i];
    _g2[i]=g2[i];
    ...
  }

  t[0]=main_thread;                           (v)
  born[0]=0;
  for(i=0;i++;i<N) {
    if(born[i]>-1) {
      ret=0;
      k=born[i];
      t[i]();
    }
  }

  for(i=0;i++;i<K-1) {                        (vi)
    assume(_g1[i+1]==g1[i]);
    assume(_g2[i+1]==g2[i]);
    ...
  }

  assert(err==0);                             (vii)
}
```

Fig. 2. Structure of translated programs.

C. State Replication and Statement Rewriting

CSeq replaces each global variable `g` by a `k`-indexed entry `g[k]` in an array of size `K` where `k` is an auxiliary variable called the current round counter and `K` is the round bound (see Fig. 2 (*i*)); we use the notation `stmt[k]` to denote the statement resulting from this replacement (e.g., Fig. 2 (*ii*)). For each global variable `g` CSeq also inserts a second copy `_g[]` (see Fig. 2 (*iii*)) that contains the guesses that the first thread uses in each round; note that only the guesses for the second and subsequent rounds are copied into the first copy, to prevent overwriting the initializations done by the original program (see Fig. 2 (*iv*)). In addition to the state replication, all pthread-specific statements and types as well as calls to `assert` and `assume` are mapped into equivalent CSeq-specific code. The code for all the extra functions and data structures is added at the top of the sequentialized file, and it is instrumented according to the verification backend specified at the moment of the translation.

D. Thread Creation, Synchronization and Execution

CSeq models the status of each thread and each lock as an integer variable. `cseq_create` (which replaces `pthread_create`) simply inserts a pointer to the thread function into an array `t` of size `N` declared in `cseq.h` and records the round in which the thread was created in an array `born`; the pointer is then used later on to start the simulation of the threads in the round stored in `born`. The initial `main` function is handled as a thread created in round 0 (`main_thread` in Fig. 2) and its pointer is inserted as the first item in the array of threads, to start the simulation (see Fig. 2 (*v*)). `cseq_exit` simply updates the thread status while `cseq_join` uses an `assume` statement on the thread status to prune away simulations in which the thread has yet not terminated. The mutex lock and unlock operations similarly set and check the lock variable.

E. Context Switches

The sequentialized program simulates the threads in the order in which they are created via `cseq_create`. It simulates a context switch by non-deterministically increasing `k` up to the round bound `K-1`. If `k` reaches `K-1`, non-deterministically an early exit can be enforced (i.e., the thread is pre-empted). CSeq inserts this simulation code (as shown in Fig. 3) at all sequence points of the original program threads (see Fig. 2 (*ii*)).

```
cs() {
  unsigned int j;
  assume(k+j < K);    // j==0 --> no context switch
  k =+ j;
  if (k == K-1 && nondet()) ret = 1; // preemption
}
```

Fig. 3. Context switch simulation.

F. Consistency Check

The first simulated thread, in each round, accesses a fresh copy of the memory with non-deterministically chosen values, while the subsequent threads continue with the state left by their predecessor. The initial guesses are stored in `_g[]`; at the end of the simulation (see Fig. 2 (vi)) we check that each round has ended with the guesses that are used in the next round; simulations that do not satisfy this condition do not correspond to feasible runs, and are discarded.

G. Error Detection

Since infeasible runs are only discarded at the end, assertion checking must be integrated with the sequentialization; in particular, in order to prevent false results, errors can only be reported after the checker has run. CSeq thus replaces all assert statements by conditionals that set an error variable `err` and exit from the thread. The error variable is checked at the end of the simulation (see Fig. 2 (vii)).

III. Architecture, Implementation, and Availability

A. Architecture

CSeq is implemented as a source-to-source transformation tool in Python (V2.7.1). It uses the `pycparser` (v2.08) [3] 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 independently by any of the supported verification tools for C.

B. Availability and Installation

Our prototype can be downloaded from http://users.ecs.soton.ac.uk/gp4/cseq/cseq-0.5.zip. It requires installation of the `pycparser`. More information is available in the `README` file in the installation package at the URL above. The project's homepage is http://users.ecs.soton.ac.uk/gp4/cseq.html.

C. Call

CSeq is a simple command line tool (`cseq.py`) that reads the specified input file and writes the translated file to the standard output. The translation process can be controlled using the following command line parameters:

`-t`<*n*> sets maximum number of threads
`-r`<*n*> sets maximum number of rounds
`-f`<*fmt*> sets output format (default `cbmc`).

In addition, we provide a wrapper script (`cseq-feeder.py`) which invokes the verification backend specified by the `-f` option on the sequentialized file. An additional option is used to set the unwinding limit for the back-end:

`-u`<*n*> sets unwinding limit for model-checking.

D. Limitations

Since heap-allocated memory is accessible to all threads, it needs to be treated similarly to global variables; however, this is an unsolved problem and CSeq does not support this yet. Implicit safety properties such as array bounds violations or nil pointer dereferences that are handled by the applied backend verification tool must be translated into explicit assertions, and their detection by the backend must be explicitly suppressed, in order to prevent false results; again, CSeq does not support this yet. The counterexample provided by the backend to the wrapper script refers to the sequentialized code and is not translated back to the original input code. However, this can be done easily by mapping back line numbers, reverting the state replication, and rearranging the order of the statuses in the counterexample, shuffled by non-determinism.

IV. Experimental Evaluation

A. Setup

We evaluated CSeq over 24 benchmarks taken from the `pthread_atomic` and `pthread` sections of the `Concurrency` category of the TACAS Software Verification Competition [4], with a total of approximately 2200 lines of code. The 10 benchmarks that end on "_unsafe" contain an error that we encoded as `assert(0)`. We used CSeq to translate the benchmarks into all supported formats and then used CBMC (v4.5),[1] ESBMC (v1.22),[2] and LLBMC (v2012.2a)[3] to verify the translated programs. Note that we used the original C files, not the CIL-preprocessed versions.

We also evaluated the native concurrency handling of CBMC[4] [1] and of Threader (c0.92) [11], the fastest verifier in the Concurrency category at the Software Verification competition [4].[5] Here we ran CBMC with the same setting for the context switch bound and Threader on the CIL-preprocessed versions. All tools were run on an otherwise idle Gentoo Linux standard PC with 12GB of memory and an Intel Xeon CPU with 2.67GHz. The timeout was set to 400s. For a comparison between CSeq and ESBMC's native concurrency handling, see [10].

B. Results

Table I summarizes the results. Here N and K denote the number of threads and rounds, respectively, used for the sequentialization translation. U denotes the unwinding bound for bounded model-checking (not used by Threader). Times are given in seconds; for the sequentialized versions they also include CSeq's runtime, which is generally neglible (approx. 0.1secs). *TO* denotes timeout. The time of the fastest tool for each benchmark is shown in bold; the time of the fastes CSeq backend is shown in cursive.

CSeq fails to translate five benchmarks due to the restrictions already mentioned: on the first file due to non-standard include files, on the last file because of dynamic memory allocation (the tool finds a call to `malloc` and rejects the file) and on the other files because the passing of parameters to `main` is not handled correctly at the moment. Threader fails on both the original CIL-preprocessed versions for the queue test cases.

[1]cbmc --unwind ⟨U⟩ --no-unwinding-assertions ⟨file⟩.c

[2]esbmc --unwind ⟨U⟩ -–64 --no-slice --no-bounds-check --no-div-by-zero-check --no-pointer-check --no-unwinding-assertions --z3-ir ⟨file⟩.c

[3]clang -c -g -I. -emit-llvm ⟨file⟩.c -o ⟨file⟩.bc;
llbmc --max-loop-iterations=⟨U⟩ --ignore-missing-function-bodies -no-overflow-checks -no-div-by-zero-checks --no-max-loop-iterations-checks ⟨file⟩.bc

[4]cbmc --unwind ⟨U⟩ --error-label ERROR --no-unwinding-assertions ⟨file⟩.c

[5]threader.sh ⟨file⟩.cil.c

TABLE I
COMPARISON OF SEQUENTIALIZATION AND NATIVE CONCURRENCY HANDLING. * - PROGRAM REJECTED. † - INTERNAL ERROR.

	Sequentialized version					Concurrent version	
	N	K	U	CBMC	ESBMC	CBMC	Threader
dekker_safe	2	3	5	4.7	*4.2*	**0.5**	**0.5**
lamport_safe	2	3	5	52.0	*23.0*	**7.1**	63.8
peterson_safe	2	3	5	*0.6*	0.8	**0.3**	7.4
rw_lock_safe	4	2	5	*1.6*	2.8	**0.6**	1.8
rw_lock_unsafe	4	2	5	-†	*4.5*	**0.4**	2.6
scull_safe	-	-	5	-†	-†	**1.5**	171.2
szymanski_safe	2	3	5	*0.9*	1.1	**0.6**	21.3
time_var_mutex_safe	2	3	5	*1.0*	2.1	**0.7**	7.2
fib_longer_safe	2	7	7	*23.4*	TO	18.1	**11.2**
fib_longer_unsafe	2	7	7	*7.2*	TO	**3.0**	10.1
fib_safe	2	6	6	**6.6**	65.2	6.8	7.7
fib_unsafe	2	6	6	*6.3*	45.8	**0.5**	6.8
indexer_safe	-	-	130	-†	-†	TO	**0.7**
lazy_unsafe	3	2	7	*0.9*	1.8	**0.5**	0.7
queue_safe	2	2	5	144.5	**10.1**	71.6	-†
queue_unsafe	2	2	5	*249.2*	TO	**86.0**	-†
reorder_2_unsafe	-	-	8	-†	-†	6.2	**2.4**
reorder_5_unsafe	-	-	8	-†	-†	6.5	**3.5**
stack_safe	2	2	5	**8.7**	TO	64.7	TO
stack_unsafe	2	2	5	*9.2*	TO	**3.7**	144.4
stateful_safe	2	2	5	0.8	**0.7**	0.9	3.9
stateful_unsafe	2	2	5	**0.7**	1.1	**0.7**	0.8
sync_safe	2	2	5	4.5	**1.4**	4.9	2.5
twostage_3_unsafe	-	-	5	-*	-*	28.6	**24.1**

Overall, the native concurrency handling is faster than sequentialization, but the time difference is generally reasonably small; moreover, for some benchmarks CSeq even outperforms the native concurrency handling. Within CSeq, CBMC slightly outperforms ESBMC as backend, but again the differences are small, and may be caused by the fact that we have mainly used CBMC for testing during development, and as a result the code generated from CSeq is now somewhat optimized for that specific backend.

V. Related Work

Sequentialization was originally developed for two threads and two context switches by Qadeer and Wu [18], but was subsequently generalized by Lal and Reps to a fixed number of threads and a parameterized number of round-robin scheduling [15]. Later, LaTorre/Madhusadan/Parlato extended this work to track only reachable configurations [13]. Further extensions allowed modelling of unbounded, dynamic thread creation [9], [5], [14], and dynamically linked data structures allocated on the heap [2]. Poirot [17] also verifies concurrent C programs via sequentialization, but it first translates them into Boogie and then implements the sequentialization transformation at the Boogie level, and can thus not be used as a generic concurrency preprocessor. Moreover Poirot uses a different, Windows-based concurrency library, not immediately comparable to the POSIX thread api. Rek [6] implements sequentialization for C via code-to-code transformation, but it is targeted at real-time systems and hard-codes a specific scheduling policy.

VI. Conclusions

Sequentialization is based on a relatively straightforward idea, but due to different backend tool idiosyncrasies and corner cases, a generic sequentialization tool is not easy to build. Nevertheless, the current results are encouraging. In addition to lifting the mentioned restrictions and limitations, we plan to integrate further backends, in particular CEGAR-based tools, into CSeq, to provide support for heap-allocated memory, and to automatically map the counterexample back to the original program. Further performance improvement of our prototype is certainly possible. The key is to reduce the non-determinism to speed-up the verification. This can be achieved (a) by optimising the static code and in particular the `cs()` function, which being duplicated at each program statement is extremely critical, (b) by reducing the context switch attempts: for example, when a statement does not access global memory, there is no need to wrap that statement between two context switch attempts.

References

[1] J. Alglave, D. Kroening, and M. Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. *CAV*, LNCS 8044, pp. 141–157, 2013.

[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] E. Bendersky. Pycparser. http://code.google.com/p/pycparser/.

[4] D. Beyer. TACAS 2013 Competition on Software Verification. http://sv-comp.sosy-lab.org/2013/

[5] A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. *SAS*, LNCS 6887, pp. 129–145, 2011.

[6] S. Chaki, A. Gurfinkel, and O. Strichman. Time-bounded analysis of real-time systems. *FMCAD*, pp. 72–80, 2011.

[7] E. M. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. *TACAS*, *LNCS* 2988, pp. 168–176, 2004.

[8] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. *ICSE*, pp. 331–240, 2011.

[9] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. *POPL*, pp. 411–422, 2011.

[10] B. Fischer, O. Inverso, and G. Parlato. CSeq: A Sequentialization Tool for C (Competition Contribution). *TACAS*, *LNCS* 7795, pp. 616–618, 2013.

[11] C. Popeea, A. Rybalchenko. Threader: A Verifier for Multi-threaded Programs (Competition Contribution). *TACAS*, *LNCS* 7795, pp. 633-636, 2013.

[12] ISO/IEC: Information technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, ISO/IEC/IEEE 9945:2009. ISO (2009).

[13] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. *CAV*, LNCS 5643, pp. 477–492, 2009.

[14] S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing parameterized programs. *FIT*, EPTCS 87, pp. 34-47, 2012.

[15] 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.

[16] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. *VSTTE*, LNCS 7152, pp. 146–161, 2012.

[17] S. Qadeer. Poirot - a concurrency sleuth. *ICFEM*, LNCS 6991, pp. 15, 2011.

[18] S. Qadeer and D. Wu. KISS: keep it simple and sequential. *PLDI*, pp. 14–24, 2004.