2015 30th IEEE/ACM International Conference on Automated Software Engineering

Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-Threaded C-Programs

Omar Inverso*, Truc L. Nguyen *, Bernd Fischer†, Salvatore La Torre‡ and Gennaro Parlato*
*Electronics and Computer Science, University of Southampton, UK
{oi2c11,tnl2g10,gennaro}@ecs.soton.ac.uk
†Division of Computer Science, Stellenbosch University, South Africa
bfischer@cs.sun.ac.za
‡Dipartimento di Informatica, Università degli Studi di Salerno, Italy
slatorre@unisa.it

***Abstract*— Lazy-CSeq is a context-bounded verification tool for sequentially consistent C programs using POSIX threads. It first translates a multi-threaded C program into a bounded nondeterministic sequential C program that preserves bounded reachability for all round-robin schedules up to a given number of rounds. It then reuses existing high-performance bounded model checkers as sequential verification backends. Lazy-CSeq handles the full C language and the main parts of the POSIX thread API, such as dynamic thread creation and deletion, and synchronization via thread join, locks, and condition variables. It supports assertion checking and deadlock detection, and returns counterexamples in case of errors. Lazy-CSeq outperforms other concurrency verification tools and has won the concurrency category of the last two SV-COMP verification competitions.**

I. Introduction

Bounded model checking (BMC) tools have successfully been used to analyze sequential software and to discover subtle errors in applications [1]. However, attempts to apply them naively to multi-threaded programs (e.g., [2]) face problems as the number of possible interleavings grows exponentially with the number of threads and statements, and a large number of specialized approaches based on partial order [3], [4], [5], [6], [7] or context-bounded analysis (CBA) [8], [9], [10], [11] methods have been developed. CBA methods limit the number of context switches they explore, which is empirically justified by work that has shown that errors manifest themselves within few context switches [12], and so fit well into the general BMC framework.

Lazy-CSeq is a context-bounded model checking tool for the verification of concurrent C programs. It is based on the technique of sequentialization [13], [8], [9], 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. The obtained sequential program can then be verified using different off-the-shelf sequential verification tools.

Lazy-CSeq is implemented as a source-to-source translation in the CSeq framework [14]. In contrast to the original CSeq tool [15], [16] that is based on a Lal/Reps-style sequentialization [8], Lazy-CSeq uses a different, *lazy* sequentialization [17], which aggressively exploits the structure of bounded programs and works well with BMC-based backends.

Lazy-CSeq's early prototypes [18], [17] already performed very well; in particular, they have won the concurrency category of the last two TACAS software verification competitions (SV-COMP) [19], [20]. Here we now describe how we have extended Lazy-CSeq into a full-fledged verification tool for sequentially consistent C programs using POSIX threads. Lazy-CSeq handles the full C language and the main parts of the POSIX thread API, such as dynamic thread creation and deletion, and synchronization via thread join, locks, and condition variables, and checks both built-in and user-defined assertions. We have extended Lazy-CSeq so that it can now also detect deadlocks and return counterexamples in case of any errors. We have further implemented a mechanism that allows users to control the schedule exploration, which can lead to better performance and can be used to implement different context-bounded analysis strategies, including bounding the number of context switches [11] and rounds [8].

With Lazy-CSeq we impact two different user groups within the broader software engineering community. First, for *software developers* (i.e., end-users), we provide a robust and well performing verification tool for a notoriously difficult verification problem. Second, for *verification tool developers*, we provide a front-end processor for concurrency handling that can easily be combined with different (sequential) verification tools.

The remainder of the paper is organized as follows. In the following two sections, we summarize the underlying sequentialization described in more detail in [17] and give a high-level overview of the CSeq framework and the Lazy-CSeq tool. In Section IV, we evaluate Lazy-CSeq on the SV-COMP 2015 benchmarks before we discuss related work in Section V, and finally conclude in Section VI.

II. Lazy Sequentialization of Concurrent Programs

Sequentialization is based on a translation of the input program to a corresponding sequential program which is then analysed by an off-the-shelf backend verification tool for sequential programs. The key idea of such translations is to replace the control nondeterminism of the original program by data nondeterminism and to capture thread invocations by function calls. Lazy sequentialization methods in addition preserve the sequential ordering of the interleaved thread executions (preserving local invariants of the original program) and use much less data nondeterminism than other sequentializations, which can result in better performances of the backend tools.

A. Lazy sequentialization schema

We assume that a concurrent program P consists of $n+1$ functions $f_0, \ldots, f_n$ (where f_0 denotes the `main` function) and creates at most n threads respectively with start functions $f_1, \ldots, f_n$, respectively. Note that these assumptions can easily be enforced by bounding the programs in BMC fashion and cloning

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.108

```
bool the[T]={1,0,0};
int cs,ct,pc[T],size[T]={5,8,8,2,2};
#define J(A,B) if(pc[ct]>A||A>=cs) goto _##B;
pthread_mutex_t m0,m1; int x=1;

void T0(void *arg) {
          static int l;
 _0:J(0,1) pthread_mutex_lock(&m0);
 _1:J(1,2) pthread_mutex_lock(&m1);
 _2:J(2,3) l=x;
 _3:J(3,4) x=l+1;
 _4:J(4,5) pthread_mutex_unlock(&m0);
 _5:J(5,6) pthread_mutex_unlock(&m1);
 _6:         ;
}

void T1(void *arg) {
 _7:J(7,8)    pthread_mutex_lock(&m1);
 _8:J(8,9)    pthread_mutex_lock(&m0);
 _9:J(9,10)   x=3;
 _10:J(10,11) pthread_mutex_unlock(&m1);
 _11:J(11,12) pthread_mutex_unlock(&m0);
 _12:           ;
}

int main_thread() {
              static pthread_t t0,t1;
 _13:J(13,14) pthread_mutex_init(&m0,0);
 _14:J(14,15) pthread_mutex_init(&m1,0);
 _15:J(15,16) pthread_create(&t0,NULL,T0,0,1);
 _16:J(16,17) pthread_create(&t1,NULL,T1,0,2);
 _17:           ;
}

int main() {
for(r=1; r<=K; r++) {
 ct=0;
 if(active[ct]) {            // only active threads
   cs=pc[ct]+nondet_uint();  // next context switch
   assume(cs<=size[ct]);     // appropriate value?
   main_thread();            // thread simulation
   pc[ct]=cs;                // store context switch
 }
 .........
 ct=2;
 if(active[ct]) {
  .........
}} }
```

Fig. 1. Example program with injected control code.

the start functions if necessary. Since each start function is thus associated with at most one thread, we can identify threads and (start) functions.

For round-robin executions, we fix an arbitrary schedule ρ by permuting $f_0, \ldots, f_n$; in each round we execute an arbitrary number of statements from each function $f_0, \ldots, f_n$. For any fixed ρ our translation then guarantees that P fails an assertion in K rounds if and only if the sequentialized program P_K^{seq} fails the same assertion. Note that the translation thus preserves not only bounded reachability, but allows us to perform on the concurrent program all analyses supported by the sequential backend tool.

P_K^{seq} is composed of a new function `main` and a thread simulation function f_i^{seq} for each thread f_i in P. Fig. 1 shows (in black) a simple example program and (in gray) the extra code fragments injected by Lazy-CSeq. The program consists of two threads `T0` and `T1` that acquire two mutexes `m0` and `m1` in reverse order and can thus deadlock. Note that the sequential verification of P_K^{seq} relies on stubs provided by Lazy-CSeq. P_K^{seq} thus uses a slightly modified version of the Pthreads API. For example, the `pthread_create` stub takes an additional argument for the (statically known) id of the calling thread; see [17] for details.

The new `main` of P_K^{seq} is a driver that calls, in the order given by ρ, the functions f_i^{seq} for K complete rounds. For each thread it maintains the label at which the context switch was simulated in the previous round and where the computation must thus resume in the current round. Moreover, before each call to f_i^{seq}, the label at which the control will context-switch out is nondeterministically guessed.

Each f_i^{seq} is essentially f_i with few lines of injected control code and with labels to denote the relevant context-switch points in the original code. When executed, each f_i^{seq} jumps (in multiple hops) to the saved position in the code and then restarts its execution until the label of the next context switch is reached. This is achieved by the `J`-macro. Context-switching at branching statements requires some extra care; see [17] for details. We also make the local variables persistent (i.e., `static`) such that we do not need to re-compute them when resuming suspended executions.

We make use of some additional data structures and variables to control the context-switching in and out of threads as described above. The data structures are parameterized over $\mathtt{T} \leq n$ which denotes the maximal number of threads activated in P executions. We keep track of the active threads, the arguments passed in each thread creation, the largest label used in each f_i^{seq}, the current label of each f_i^{seq}, and the index of and the context-switch point guessed for the currently executed thread.

Note that the control code we inject in the translation is designed such that each f_i^{seq} reads but does not write any of the additional data structures. This data is updated only in the main driver and in the portions of code simulating the API functions concerning thread creation and termination. This has the advantage of introducing fewer dependencies between the injected code and the original code, which typically leads to a better performance of the backend tool (e.g., for BMC backends this results in smaller formulas).

B. Deadlock check

Compared to the prototype described in [17], the Lazy-CSeq tool now uses an improved modeling and coverage of the Pthreads library, especially for mutexes. For example, it can now detect whether a mutex is used again after being destroyed. However, the main improvement in this respect is that Lazy-CSeq can now also check for deadlocks in the original concurrent program.

A *deadlock* is characterized by a subset of the threads (i) that are all blocked after trying to acquire a mutex that is held by another thread, and where (ii) the dependency chain between the waiting threads is cyclic. For example, for the program in Fig. 1 such a cycle (and thus a deadlock) occurs if `T0` acquires `m0`, then the context switches and `T1` successfully acquires `m1` but gets subsequently blocked when it tries to acquire `m0`, and the context switches back to `T0`, which gets blocked when it tries to acquire `m1`. Thus, there is a cyclic chain of length 2 where `T0` is the first thread and holds mutex `m0` (and is waiting for mutex `m1`), and `T1` is the second thread and holds mutex `m1` (and is waiting for mutex `m0`). Note that there may be other threads that are blocked by trying to acquire any of the mutexes held by any of the the threads in the chain, but are not required for the deadlock and thus do not need to be recorded in the chain.

Lazy-CSeq thus searches for deadlock conditions by non-deterministically guessing the chain on-the-fly while simulating the threads. This chain is modelled by an array of thread identifiers together with a single mutex identifier. The first position in the array contains the id of the thread that starts the cycle. Each subsequent position $i+1$ in the array contains the id of a thread that waits on a mutex held by the thread whose id is stored in the previous position i. The additional single mutex identifier denotes

the mutex on which the *second* thread in the array is blocked; this is stored when the second thread is entered into the chain. When a thread successfully acquires a mutex and the chain is still empty we non-deterministically decide to store its id in the first element of the array (thereby starting to search for a cycle) and continue with its simulation. When the mutex is already held by another thread, the simulation of the requesting thread is blocked; moreover, if the mutex is held by the thread stored at the end of the array we non-deterministically insert the id of the requesting thread at the current position in the array. We then test for a cycle over the waiting threads by checking whether the id of the last inserted thread is the same as the one stored in the first position of the array. When we release any mutex we also check that the first thread in the array does not release the mutex on which the second thread in the array is blocked and on which the deadlock eventually hinges. This ensures that the waiting threads cannot make progress before the simulation detects the cycle and thus correctness of the simulation.

III. Architecture, Implementation, and Availability

A. The CSeq framework

Lazy-CSeq is developed within the CSeq framework [14]. The framework builds on ideas from the original CSeq tool [16] but has been improved and fully re-engineered. It now provides support for quickly developing new sequentialization-based verification tools. To date, it has also been used to implement the MU-CSeq [21], [22], [23] and UL-CSeq [24] tools.

The framework comprises several modules that are either *translators* that implement source-to-source transformations of C programs, or *wrappers* that work on generic strings and are used for general-purpose tasks that do not produce source code. Each tool within CSeq is identified by a *configuration* that corresponds to a sequence of translators followed by a sequence of wrappers. Fig. 2 sketches the configuration for Lazy-CSeq.

A verification tool takes as input the file containing the source code of the concurrent C program to analyze and the list of verification parameters. For Lazy-CSeq, the verification parameters are the number of *rounds*, the *unwinding depth* and the acronym of the backend tool. The input parameters are passed to the appropriate modules, additionally the first module takes as input also the input source file and then the output of each module is fetched as input to the following module. The output of the last module in the sequence is the analysis outcome.

The first translator is always a *merger*: the input source code is merged with external sources pulled in by the `#include` directives. The last translator is typically an *instrumenter*, which instruments the output according to the backend tool (as explained below). The purpose of the wrappers is to interact with the backend tool and interpret its answer at the end of the analysis, in particular, we have a `cex` module that is responsible for tracking back the counter-example generated by the backend tool on the input source code, and thus output the counter-example.

Translators run in two steps: (1) the input code is parsed in order to build the abstract syntax tree (AST), the symbol table, and other data structures; (2) the AST is recursively traversed and un-parsed back into a string that corresponds to the output C code. This mechanism is built on top of `pycparser`, a parser for C99 that uses `PLY`, an implementation of Lex-Yacc, and it is implemented by conveniently overriding `pycparser`'s AST-based pretty-printer, so that the output code is transformed while visiting the AST. In particular, the transformation is made on-the-fly by directly changing the output generated by AST subtree visits rather than altering the structure of the AST itself. Other source-to-source translation tools [25] use instead rewrite rules. String-based source transformations are in contrast more intuitive and require a less steep learning curve, and combined with Python's flexibility it is relatively easy to implement complex code transformations quickly. String-based rewriting is also used in the ROSE framework [26].

The CSeq framework also provides a *line-mapping* functionality that is independent from the specific translation performed and is a useful support for the counterexample generation. The idea is to keep track of the location in the source code where each line of the output was translated from. During the generation of the output, translators automatically create maps from output to input, in a similar way to how the C Preprocessor (CPP) uses line control information when merging multiple source files, to keep track of which lines comes from which source file. However, rather than inserting explicit `#line` directives in the source code (like CPP does) the information is stored as a table which maps output lines back to input lines (note that each input line may generate several output lines, for instance after unfolding a loop). At the end of the last translation, it is possible to track line numbers back to the output of the first module. For the first module (merger), since there might be multiple input files (due to the `#include` directives), we map output line numbers to pairs of the form (*linenumber, filename*).

Instrumenting the code for a specific backend is in itself a quite simple standalone transformation undertaken by the instrumentation module. It consists in replacing the primitives for handling non-determinism (that are backend-independent and potentially injected at any point by any module) with backend-specific statements. This involves three kinds of statements: (1) variable assignment statements to nondeterministic values using `nondet_int`, `nondet_long`, etc., (2) restrictions of non-determinism using `assume`, (3) explicit condition checks using `assert`. This requires a simple renaming of the function calls, or inserting ad-hoc functions definition, depending on whether or not the desired verification backend natively models all of the above. The size of a backend integration is therefore usually less than 10 lines; however, the CBMC default backend exploits CBMC's bitvectors to optimize the representation of the program counters and is thus more complicated.

B. The Lazy-CSeq tool

The Lazy-CSeq tool is a CSeq configuration of eighteen modules, that can be conceptually grouped into the following categories (see Fig. 2):

1) the source merging module;
2) eight simple transformation modules to rewrite the input program in steps with a progressively simplified syntax, so to simplify the complex transformations occurring later in the sequence;
3) four translators for program flattening to produce a bounded program (see [17]);
4) two modules implementing the sequentialization algorithm that produces a backend-independent sequentialized file (see [17]);
5) standard program instrumentation to instrument the sequentialized file for a specific backend;

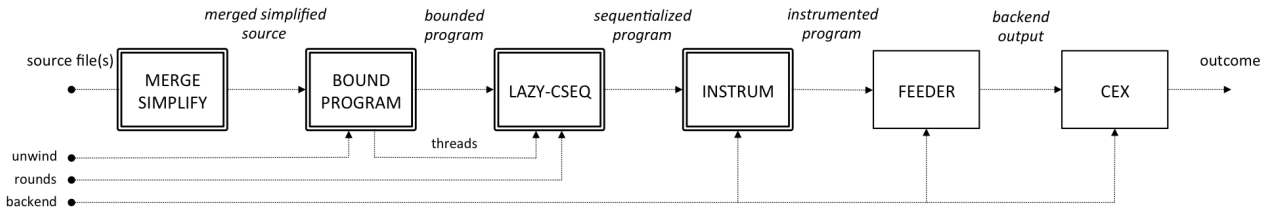

Fig. 2. Configuration sequence of Lazy-CSeq. Double framed boxes denote modules composed of multiple submodules.

6) two wrappers for backend invocation and user report generation or counterexample translation.

Compared to the features of the prototype described in [17], Lazy-CSeq's new features are *deadlock checking* (see Section 2), *counter-example generation*, and *scheduling selection*. We briefly describe the latter two here.

The counterexample generation feature of Lazy-CSeq tackles one of the main usability limitations of sequentialization-based tools, namely that when an error is found the error trace is too hard to follow as the counterexample produced by the backend actually refers to the sequentialized file. Lazy-CSeq instead generates counterexamples that refer to the actual input code.

The main task here consists in translating the backend's counterexample by tracing back line numbers to their corresponding input coordinates, and then showing the amended states in the same order. For this we use the line-mapping feature provided by CSeq framework. We also insert additional concurrency-specific details, to show schedules, thread creations, lock operations and the like.

Counterexample generation in Lazy-CSeq is currently supported only for the default backend. However, we stress that the line-mapping facility provided by the framework CSeq is general, backend-independent and translation-independent, and thus can be used for any backend.

For an n-thread program, our prototype from [17] fixes a schedule ρ of the threads (this corresponds to the order in which the threads get created) and explores all computations up to a number of rounds r specified as input parameter, where in each round threads are scheduled according to ρ. In the Lazy-CSeq tool we now allow to specify some schedule restrictions for each round. Namely, we can choose for each round if all threads can be scheduled (denoted with `+`), or only a thread from a set of threads can be scheduled (we list the threads with numbers separated by commas). To separate rounds we use "`:`". For example, for two rounds, "`+:+`" denotes the scheduling from [17], for "`1,2:+`" only the first and second thread (in order of creation) can be scheduled at the first round, filtering out any other possible choice (note that the main thread is always in first set), "`1:3:2`" indicates an explicit schedule. Note that even when the schedule is fixed context-switch points can still happen at any time. Scheduling selection can be useful to guide the analysis when some specific facts on scheduling are known, or on complex problems where even analyzing a single round would require too many resources. In fact the translation is tailored to the specific sub-set of possible schedules, and the trimmed-down main driver results in smaller verification conditions.

C. Usage.

Lazy-CSeq can be invoked with the command `cseq.py -i input.c` to analyze the input file `input.c` and check for reachable error states determined by an `ERROR` label, an assertion failure, or incorrect use of locks, using the default analysis parameters and the default backend. Deadlock checking is off by default and can be enabled with `--deadlock`.

The analysis parameters are the loop unwinding depth and the number of rounds. Their default value is `1` for both and can be changed with `--unwind k` and `--rounds k`, respectively. The default backend is CBMC and it can be changed using `--backend b` where *b* is one of the following:

- bounded model-checkers: `blitz` [27], `cbmc` [28], `esbmc` [2], `llbmc` [29]
- abstraction-based tools: `cpachecker` [30], `satabs` [31]
- symbolic testing tools: `klee` [32]

Support for bounded model-checking is mature, while abstraction-based and testing backends are only supported experimentally at this stage.

The option `--rounds` uses standard round-robin schedules as in [17]. This can be replaced with restricted schedules using `--schedule r1:...:rn`, which gives schedule restrictions for `n` rounds, as described above.

Counterexample generation is disabled by default but can be enabled when using the default backend with `--cex`. Alternatively, `--linemap` will show a table of the line maps across all source transformation steps, one row for each output line, one column for each transformation.

D. Availability and Installation.

Lazy-CSeq is available as open source software under BSD license. Our tool can be downloaded from http://users.ecs.soton.ac.uk/gp4/cseq/files/lazy-cseq-1.0.tar.gz. 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/. A demo video of the tool is available at http://users.ecs.soton.ac.uk/gp4/cseq/files/lazy-cseq-1.0-demo.mov.

IV. Experimental Results

We have compared Lazy-CSeq-1.0 using CBMC (v5.1) as a backend against CBMC (v5.1) itself and CSeq-0.5. CBMC uses partial orders to symbolically model concurrency [7], while CSeq [16] is based on an eager sequentialization implementing a variant of LR [8] and uses CBMC (v5.1) as sequential backend.

We have used the set of benchmarks from the Concurrency category of the Software Verification Competition (SV-COMP'15) held at TACAS [20]. These are widespread benchmarks, and many state-of-the-art analysis tools have been trained on them

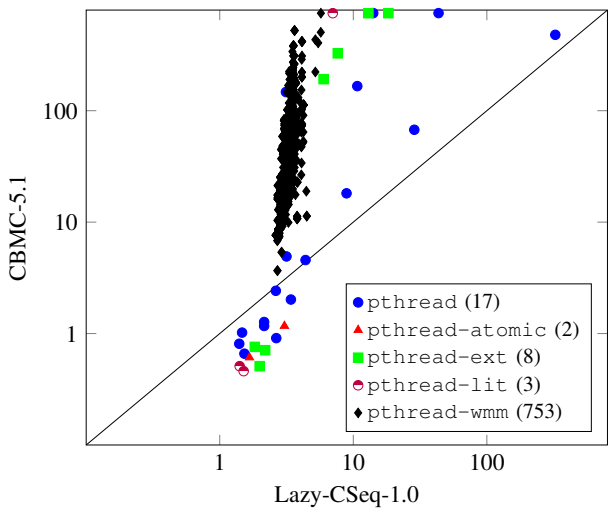

Fig. 3. Lazy-CSeq-1.0 versus CBMC-5.1.

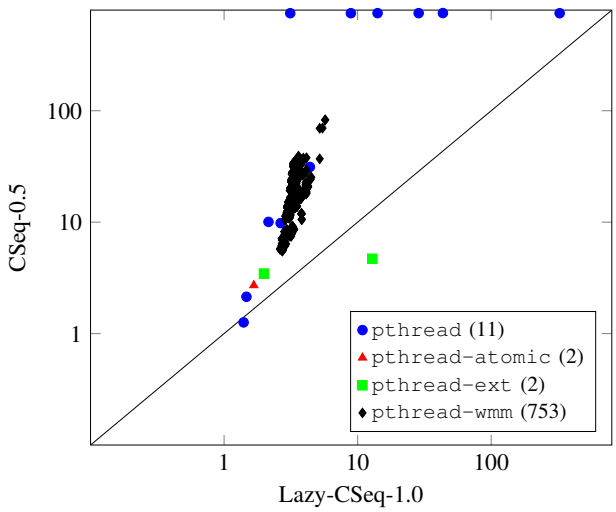

Fig. 4. Lazy-CSeq-1.0 versus CSeq-0.5.

(i.e., CBMC); in addition, they offer a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms.

Since we use a BMC tool as a backend, and BMC can in general not prove correctness, but can only certify that an error is not reachable within the given bounds, we thus conducted the experiments only on the 783 unsafe files of the 993 files in the whole benchmark set, with a total of approx. 240K lines of code.

We have performed the experiments on an otherwise idle machine with a Xeon W3520 2.6GHz processor and 12GB of memory, running a Linux operating system with 64-bit kernel 3.0.6. We set a 10GB memory limit and a 750s timeout for the analysis of each subject. For each tool and file, we set the parameters to the minimum value needed to expose the error.

The experiments for CBMC and CSeq-0.5 are summarized by the scatter plots (with logarithmic axes) shown in Figure 3 and Figure 4, respectively. All tools report the correct answers. Both CBMC and CSeq time out on 6 files. Furthermore, CSeq rejects 5 files and returns "unknown" on 10 files (due to translation errors and bugs in the tool). The experiments clearly show that Lazy-CSeq outperforms both CBMC and CSeq, except on a handful of small files in which CBMC is faster. Overall, Lazy-CSeq is about 6x and 20x faster than CSeq and CBMC, respectively.

V. Related Work

In addition to the already cited work there is further related research that we briefly discuss here. Sequentialization was originally developed for two threads and two context switches by Qadeer and Wu [13], but was subsequently generalized by Lal and Reps to a fixed number of threads and a parameterized number of round-robin scheduling [8]. Later, LaTorre/Madhusadan/Parlato extended this work to track only reachable configurations [9]. Further extensions allowed modelling of unbounded, dynamic thread creation [33], [34], [35], [36], and dynamically linked data structures allocated on the heap [37]. Poirot [38] 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 [39] 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 becoming a prominent approach to find bugs in concurrent programs. It ensures fast prototyping of analysis tools by reusing existing sequential program tools. Our Lazy-CSeq tool is quite robust and competitive with state-of-the-art tools as shown by the experiments. It also allows for analysis with different specifications (reachability, deadlock check) and different technologies (depending on the choice of the backend). When used with CBMC as backend, our tool also generates counterexamples for the input program; we are not aware of other tools based on sequentializations that support this. Another interesting feature is the possibility of refining the scheduling of threads by entering a scheduling expression as a parameter that can be useful to guide the analysis when some specific facts on scheduling are known or just to restrict the search.

As future directions, we plan to extend the CSeq framework with more sequentialization algorithms, support for other classes of concurrent programs (embedded programs, distributed programs) and counterexample generation modules for other backends.

Acknowledgements. This work was partially supported by EPSRC EP/M008991/1, INDAM-GNCS 2014 and MIUR-FARB 2012-2014 grants.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, LNCS 1579, 1999, pp. 193–207.
[2] L. C. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," in *ICSE*, 2011, pp. 331–340.
[3] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *CAV*, LNCS 3576, 2005, pp. 82–97.
[4] M. K. Ganai and A. Gupta, "Efficient Modeling of Concurrent Systems in BMC," in *SPIN*, LNCS 5156 , 2008, pp. 114–133.
[5] N. Sinha and C. Wang, "On Interference Abstractions," in *POPL*, 2011, pp. 423–434.

[6] ——, "Staged Concurrent Program Analysis," in *SIGSOFT FSE*, 2010, pp. 47–56.
[7] J. Alglave, D. Kroening, and M. Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software," in *CAV*, LNCS 8044, 2013, pp. 141–157.
[8] A. Lal and T. W. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," *Formal Methods in System Design*, vol. 35, no. 1, pp. 73–97, 2009.
[9] S. La Torre, P. Madhusudan, and G. Parlato, "Reducing Context-Bounded Concurrent Reachability to Sequential Reachability," in *CAV*, LNCS 5643, 2009, pp. 477–492.
[10] S. La Torre, P. Madhusudan, and G. Parlato, "Analyzing recursive programs using a fixed-point calculus," in *PLDI*, 2009, pp. 211–222.
[11] S. Qadeer and J. Rehof, "Context-Bounded Model Checking of Concurrent Software," in *TACAS*, LNCS 3440, pp. 93–107.
[12] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," in *PLDI*, 2007, pp. 446–455.
[13] S. Qadeer and D. Wu, "KISS: Keep It Simple and Sequential," in *PLDI*, 2004, pp. 14–24.
[14] CSeq framework, *http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html*.
[15] B. Fischer, O. Inverso, and G. Parlato. CSeq: A Sequentialization Tool for C (Competition contribution). *TACAS*, *LNCS* 7795, pp. 616–618, 2013.
[16] B. Fischer, O. Inverso, and G. Parlato, "CSeq: A Concurrency Pre-processor for Sequential C Verification Tools," in *ASE*, 2013, pp. 710–713.
[17] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded model checking of multi-threaded C programs via lazy sequentialization," in *CAV*, LNCS 8559, pp. 585–602.
[18] ——, "Lazy-CSeq: A Lazy Sequentialization Tool for C - (Competition contribution)," in *TACAS*, LNCS, 8413, 2014, pp. 398–401.
[19] D. Beyer, "Status report on software verification - (Competition summary SV-COMP 2014)," in *TACAS*, LNCS 8413, 2014, pp. 373–388.
[20] ——, "Software verification and verifiable witnesses - (Report on SV-COMP 2015)," in *TACAS*, LNCS 9035, pp. 401–416.
[21] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato, "Verifying concurrent programs by memory unwinding," in *TACAS*, LNCS 9035, 2015, pp. 551–565.
[22] ——, "MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings - (Competition contribution)," in *TACAS*, 2014, pp. 402–404.
[23] ——, "MU-CSeq 0.3: Sequentialization by Read-Implicit and Coarse-Grained Memory Unwindings - (Competition contribution)," in *TACAS*, 2015, pp. 436–438.
[24] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato, "Unbounded Lazy-CSeq: A lazy sequentialization tool for C programs with unbounded context switches - (Competition contribution)," in *TACAS*, LNCS 9035, 2015, pp. 461–463.
[25] I. D. Baxter, C. W. Pidgeon, and M. Mehlich, "Dms®: Program transformations for practical scalable software evolution," in *ICSE*, 2004, pp. 625–634.
[26] D. J. Quinlan, M. Schordan, B. Philip, and M. Kowarschik, "The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications," in *LCPC*, LNCS 2624, 2001, pp. 383–394.
[27] C. Y. Cho, V. D'Silva, and D. Song, "BLITZ: Compositional Bounded Model Checking for Real-world Programs," in *ASE*, 2013, pp. 136–146.
[28] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, LNCS 2988, 2004, pp. 168–176.
[29] S. Falke, F. Merz, and C. Sinz, "The Bounded Model Checker LLBMC," in *ASE*, 2013, pp. 706–709.
[30] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *CAV*, LNCS 6806, 2011, pp. 184–190.
[31] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: sat-based predicate abstraction for ANSI-C," in *TACAS*, LNCS 3440, 2005, pp. 570–574.
[32] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.
[33] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-bounded Scheduling," in *POPL*, 2011, pp. 411–422.
[34] S. La Torre, P. Madhusudan, and G. Parlato, "Model-Checking Parameterized Concurrent Programs Using Linear Interfaces," in *CAV*, LNCS 6174, 2010, pp. 629–644.
[35] A. Bouajjani, M. Emmi, and G. Parlato, "On Sequentializing Concurrent Programs," in *SAS*, LNCS 6887, 2011, pp. 129–145.
[36] S. La Torre, P. Madhusudan, and G. Parlato, "Sequentializing Parameterized Programs," in *FIT*, EPTCS 87, 2012, pp. 34–47.
[37] M. F. Atig, A. Bouajjani, and S. Qadeer, "Context-bounded analysis for concurrent programs with dynamic creation of threads," *Logical Methods in Computer Science*, vol. 7, no. 4, 2011.
[38] S. Qadeer, "Poirot - A Concurrency Sleuth," in *ICFEM*, LNCS 6991, 2011, p. 15.
[39] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-bounded Analysis of Real-time Systems," in *FMCAD*, 2011, pp. 72–80.