Bounded Verification of Multi-Threaded Programs via Lazy Sequentialization

OMAR INVERSO, Gran Sasso Science Institute, Italy
ERMENEGILDO TOMASCO, Agenzia delle Entrate, Italy
BERND FISCHER, Stellenbosch University, South Africa
SALVATORE LA TORRE, Università degli Studi di Salerno, Italy
GENNARO PARLATO, Università degli Studi del Molise, Italy

Bounded verification techniques such as bounded model checking (BMC) have successfully been used for many practical program analysis problems, but concurrency still poses a challenge. Here we describe a new approach to BMC of sequentially consistent imperative programs that use POSIX threads. We first translate the multi-threaded program into a nondeterministic sequential program that preserves reachability for all round-robin schedules with a given bound on the number of rounds. We then reuse existing high-performance BMC tools as backends for the sequential verification problem. Our translation is carefully designed to introduce very small memory overheads and very few sources of nondeterminism, so that it produces tight SAT/SMT formulae, and is thus very effective in practice: our Lazy-CSeq tool implementing this translation for the C programming language won several gold and silver medals in the concurrency category of the Software Verification Competitions (SV-COMP) 2014–2021, and was able to find errors in programs where all other techniques (including testing) failed. In this paper we give a detailed description of our translation and prove its correctness, sketch its implementation using the CSeq framework, and report on a detailed evaluation and comparison of our approach.

CCS Concepts: • **Theory of computation → Program verification**; **Verification by model checking**; • **Software and its engineering → Formal software verification**; **Model checking**.

Additional Key Words and Phrases: concurrent program analysis, sequentialization

ACM Reference Format:
Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2021. Bounded Verification of Multi-Threaded Programs via Lazy Sequentialization. *ACM Trans. Program. Lang. Syst.* 1, 1 (August 2021), 49 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Concurrent programming is becoming ever more important as concurrent computer architectures such as multi-core processors are becoming ever more common. However, concurrent program analysis remains a stubbornly hard problem, since the number of possible interleavings grows exponentially with the number of the program's threads and statements. Approaches that analyze interleavings individually, such as concurrency testing [68] or a naïve application of bounded model checking (BMC) [25], struggle particularly to find concurrency bugs that manifest themselves only in a few of

Authors' addresses: Omar Inverso, Gran Sasso Science Institute, Italy, omar.inverso@gssi.it; Ermenegildo Tomasco, Agenzia delle Entrate, Italy, gildotomasco@gmail.com; Bernd Fischer, Stellenbosch University, South Africa, bfischer@sun.ac.za; Salvatore La Torre, Università degli Studi di Salerno, Italy, slatorre@unisa.it; Gennaro Parlato, Università degli Studi del Molise, Italy, gennaro.parlato@unimol.it.

the many interleavings. Approaches that use symbolic representations to analyze all interleavings collectively typically fare better.

In this paper, we develop and evaluate a new technique for *lazy sequentialization*, a symbolic approach for the analysis of concurrent programs with shared memory. It builds on the strengths of bounded model checking (BMC), which has been used successfully to discover subtle errors in sequential software [11]. In particular, our technique bounds the number of context switches it explores, which fits well into the general BMC framework. Such *context bounded analysis* (CBA) approaches [58, 63, 78] are empirically justified by work that has shown that errors typically manifest themselves within few context switches [67].

The general sequentialization approach, originally proposed by Qadeer and Wu [79], is based on the idea of translating concurrent programs into nondeterministic sequential programs that (under certain assumptions) behave equivalently, so that the different interleavings do not need be treated explicitly during verification. This allows the reuse of sequential verification *methods*, but since sequentializations can be implemented as code-to-code translations, it also allows the reuse of unchanged *tools* as sequential verification backends. However, the sequentialization translation alters the original program structure by injecting control code that represents an overhead for the backend. Therefore, the design of well-performing tools under this approach requires careful attention to the details of the translation. For example, eager sequentializations [35, 63, 89] guess the different values of the shared memory before the verification and then simulate (under this guess) each thread in turn. They can thus explore infeasible computations that need to be pruned away afterwards, which increases the state space. Lazy sequentializations [58] instead guess the context switch points and recompute the memory contents, and thus explore only feasible computations. This recomputation poses no problem for tools that compute function summaries [58, 59] since they can re-use the summaries from previous rounds, but it becomes a serious problem for BMC-based backends because it can lead to exponentially growing formula sizes [42].

In this paper, we design a new, surprisingly simple but effective lazy sequentialization schema that aggressively exploits the structure of bounded programs and works very well with BMC-based backends. The translation is carefully designed to introduce very small memory overheads and very few sources of nondeterminism, so that it produces simple formulas, and is thus very effective in practice. The sequentialized program simulates all bounded executions of the original program for a bounded number of rounds. It is composed of a main driver and an individual simulation function for each thread, where function calls and loops of the input program are inlined and unrolled, respectively. In each round, the main driver calls each thread simulation function; however, it keeps the values of the thread-local variables between the different function activation (by turning them into `static` variables). The thread simulation functions do not repeat all steps executed in the previous rounds but instead jump (in multiple hops) back to the stored program location where the previous round has finished. This avoids their recomputation and thus the exponentially growing formula sizes observed by Ghafari et al. [42]. The translation does not require built-in error checks or special dynamic memory allocation handling but can rely on the backend for these, because the analysis of the translated program explores only reachable states of the original input program.

We have implemented this sequentialization in the Lazy-CSeq tool that handles (1) the main parts of the POSIX thread API [54], such as dynamic thread creation and deletion, and synchronization via thread join, locks, and condition variables; and (2) the full C programming language with all its peculiarities such as different data types, dynamic memory allocation, and low-level programming features such as pointer arithmetic. Lazy-CSeq implements both bounding and sequentialization as source-to-source translations. The resulting sequential program can be given to *any* existing backend verification tool for sequential C programs. We have tested Lazy-CSeq with a wide variety of backends, including BLITZ [18], CBMC [23], CPAchecker [9], ESBMC [26], KLEE [15], LLBMC [65], SMACK [81], and Ultimate Automizer [46].

Lazy-CSeq has consistently performed very well in the annual Software Verification Competition SV-COMP. It won the concurrency category of SV-COMP14, SV-COMP15, SV-COMP20, and SV-COMP21, and came second in SV-COMP16 (where it was narrowly beaten by its sibling MU-CSeq [92], which uses a different sequentialization), SV-COMP17, and SV-COMP19.[1] Over the years, it outperformed mature tools with native concurrency support, such as CBMC [2], Ceagle [94], CIVL [97], CPAchecker, Divine [5], ESBMC, or SMACK. We have also evaluated Lazy-CSeq over further, hard concurrency problems. On some of these, all other approaches (including testing) failed and Lazy-CSeq was the only tool able to discover the bugs. These results justify the general sequentialization approach, and in contrast to the findings by Ghafari et al. [42], also demonstrate that a lazy translation can be more suitable for use in BMC than the more commonly applied LR-translation [31, 63], as Lazy-CSeq also outperforms by orders of magnitude our own LR-CSeq tool [34].

Contributions. This paper builds on and extends our previous work, where we introduced the translation and gave preliminary experimental results [52], and described the tool framework [51]. Here, we now give a more detailed description of the approach (including a formal definition of the sequentialization transformation and formal correctness proofs), and a more in-depth evaluation, including hard benchmarks that cannot be solved by any other approach.

In summary, in this paper we make the following contributions.

(1) We develop the first lazy sequentialization that is suitable for BMC-based verification backends. We describe its surprisingly simple high-level idea and give a precise translation.
(2) We give a formal correctness proof for this translation.
(3) We describe an implementation of this translation within the CSeq framework.
(4) We evaluate our implementation over a large set of standard benchmarks, as well as several hard benchmarks. We show that our implementation outperforms all other tools, and is the only one that can find bugs in all hard benchmarks.

Paper Organization. The remainder of the paper is organized as follows. We first introduce the syntax and the semantics of multi-threaded and bounded multi-threaded programs in Section 2. We give a detailed description of our translation, along with an informal correctness argument, and formalize this translation as a set of rewrite rules in Section 3. We prove its correctness in Section 4. We describe Lazy-CSeq's implementation in Section 5, and summarise our experimental evaluation in Section 6. We discuss related and future work in Sections 7 and 8, respectively, and finally end with our conclusions in Section 9.

2 SHARED-MEMORY MULTI-THREADED PROGRAMS

We use a simple imperative language for multi-threaded programs to simplify the presentation. It features dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. Thread communication is implemented via shared memory and modelled using global variables. However, our Lazy-CSeq tool works on the full C programming language. In this section and throughout this paper, we use the terms *multi-threaded program* and *concurrent program* interchangeably.

[1]In fact, the only instance where it did not win or place second was SV-COMP18, where we did not participate due to late rule changes.

$$
\begin{array}{lll}
P & ::= & (decl\texttt{;})^* \; (type\; p\; \texttt{(}\langle decl\texttt{,}\rangle^*\texttt{)}\; \texttt{\{}(\texttt{static?}\; decl\texttt{;})^*\, stmt\texttt{\}})^* \\
decl & ::= & type\; id(\texttt{[}e\texttt{]})? \\
type & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{mutex} \mid \texttt{void} \\
stmt & ::= & seq\texttt{;} \mid conc\texttt{;} \mid \texttt{\{}stmt^+\texttt{\}} \\
seq & ::= & \texttt{assume(}b\texttt{)} \mid \texttt{assert(}b\texttt{)} \mid x \texttt{:=} e \mid p\texttt{(}\langle e\texttt{,}\rangle^*\texttt{)} \mid \texttt{return}\; e? \\
 & & \mid \texttt{if(}b\texttt{)}\; \texttt{then}\; stmt\; (\texttt{else}\; stmt)? \mid \texttt{while(}b\texttt{)}\; \texttt{do}\; stmt \mid \texttt{goto}\; l \mid l\texttt{:skip} \\
conc & ::= & x \texttt{:=} y \mid y \texttt{:=} x \mid t \texttt{:= create}\; p\texttt{(}\langle e\texttt{,}\rangle^*\texttt{)} \mid \texttt{join}\; t \mid \texttt{init}\; m \mid \texttt{lock}\; m \mid \texttt{unlock}\; m \mid \texttt{destroy}\; m
\end{array}
$$

Fig. 1. Syntax of multi-threaded programs.

During the execution of a multi-threaded program, only one thread is *enabled* (i.e., allowed to make a transition) at any given time. Initially, only the main thread is enabled; new threads can be spawned by a thread creation statement. Once created, a thread is added to the pool of *active* threads. At a *context switch* the currently enabled thread is suspended but remains active, and one of the other active threads is resumed and becomes the new enabled thread. When a thread is resumed its execution continues either from the point where it was suspended or, if it becomes enabled for the first time, from the beginning.

All threads share the same address space: they can write to or read from global (or *shared*) variables of the program to communicate with each other. We assume the *sequential consistency* memory model: when a shared variable is updated its new valuation is immediately visible to all the other threads [64]. We further assume that each statement is atomic. Note, however, that it is always possible to decompose a statement into a sequence of atomic statements, each involving at most one shared variable [66].

2.1 Syntax

The syntax of multi-threaded programs is defined by the grammar shown in Figure 1. Terminal symbols are set in typewriter font. The notation $\langle n\ \texttt{t}\rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals `t`; *id* denotes a generic variable, x a local variable, y a shared variable, m a mutex, t a thread variable, p a procedure name, and l a label. We assume expressions e to be formed over local variables, Boolean literals `true` and `false`, and integer literals, which can be combined in a type-correct way using the standard Boolean and mathematical operators and the program's side-effect free functions. We adopt the usual square-bracket notation for arrays to indicate elements of fixed-sized vectors of scalar variables. We denote Boolean expressions by b.

A *multi-threaded program P* (see Figure 2(a) for an example) consists of a list of *global* variable declarations (i.e., *shared* variables), followed by a list of procedures. Shared variables are always initialized to a type-specific default value. We denote the default value by `0`, regardless of the type. Each procedure has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement. A local variable can be declared as `static`, as in C. Such `static` variables are also initialized to `0`. A statement is either a sequential, a concurrent, or a *compound* (i.e., a sequence of statements enclosed in braces) statement.

A *sequential statement seq* can be an `assume`- or `assert`-statement, an assignment, a call to a procedure that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional statement, a `while`-loop, a `goto`- or a `skip`-statement with a non-numerical label l. Note that by construction all jump targets are `skip`-statements because these are the only statements where we allow labels.

Sequential statements affect only the thread-local control flow and involve only local variables; the sequentialization does thus not need to simulate context switches at these [75]. Unlike global variables, local variables remain uninitialized after their declaration; therefore, until explicitly set by an appropriate assignment statement, they can nondeterministically assume any value allowed by their type. We also use the symbol `*` to denote the expression that nondeterministically evaluates to any possible value; hence, with `x:=*` we mean that `x` is assigned any possible value of its type domain.

A *concurrent statement conc* can be a concurrent assignment, a call to a thread routine, such as a thread creation or join, or a mutex operation (i.e., init, lock, unlock, and destroy). A concurrent assignment assigns a shared (resp. local) variable to a local (resp. shared) one. A thread creation statement t := `create` $p(e_1, \ldots, e_n)$ spawns a new thread from procedure p with expressions $e_1, \ldots, e_n$ as arguments. A thread join statement, `join` t, suspends the enabled thread until the thread identified by t terminates its execution, i.e., after the thread has executed its last statement. Lock and unlock statements respectively acquire and release a mutex. If the mutex is already acquired, the lock operation is blocking for the thread, i.e., the thread is suspended until the mutex is released and can then be acquired.

We assume that a valid program P satisfies the usual well-formedness and type-correctness conditions. We also assume that P contains a procedure `main`, which is the starting procedure of the only thread that exists in the beginning. We call this the *main thread*. We further assume that there are no calls to `main` in P and that no other thread can be created that uses `main` as starting procedure.

2.2 Semantics

A *thread configuration* of a multi-threaded program P is a triple $\langle locals, pc, stack \rangle$, where *locals* is a valuation of the thread's local variables, *pc* is the *program counter* that points to the next statement $P(pc)$ in P to be executed, and *stack* is a stack of procedure calls. A thread configuration is *initial* if *locals* is any valuation that assigns `0` to each static variable and an arbitrary value otherwise, *pc* is the program counter of the first statement of the thread, and *stack* is the empty stack (denoted by $\perp$ in the following). At a procedure call, the program counter of the caller and the current valuation of its local variables are pushed onto the stack, and the control moves to the initial location of the callee. At a procedure return, the top element of the stack is popped, and the local variables and the program counter are restored. Any other statement follows a standard *C-like semantics*.

A *thread identifier* is a non-negative integer. A *multi-threaded program configuration c* consisting of n active threads with identifiers $I = \{i_1, \ldots, i_n\}$ is a tuple of the form $\langle sh, en, \{th_i\}_{i \in I} \rangle$, where (1) *sh* is a valuation of the shared variables, (2) $en \in I \cup \{\dagger\}$ is either the identifier $i \in I$ of the *enabled thread* (i.e., the only active thread that is allowed to make a transition), or the token $\dagger$ to mark error configurations, and (3) th_i is the configuration of the thread with identifier i. A configuration $c = \langle sh, en, A \rangle$ is *initial* if *sh* is the default evaluation of the shared variables, $en = 0$ is the identifier of the main thread, and $A = \{th_0\}$, where th_0 is an initial configuration of `main` (i.e., the main thread); c denotes normal termination if $en \neq \dagger$ and $A = \emptyset$.

A *transition* of a multi-threaded program P from a configuration c to a configuration c', denoted by $c \rightarrow^i_P c'$, corresponds to the execution of a statement by the enabled thread (i.e., the thread with identifier $i = en$). If the statement being executed is sequential, only the configuration th_{en} of the enabled thread is updated, as usual. The execution of an `assert`-statement with a condition that evaluates to `false` causes a transition into an error configuration, from which no further transitions can be taken. The execution of an `assume`-statement allows a transition only if the condition evaluates to `true`. The execution of a `create`-statement adds (with a fresh identifier i) a new thread configuration to the configuration of the multi-threaded program, which consists of a valuation of the local variables the thread start function, the program counter pointing to its initial statement, and the empty stack. The execution of a `join`-statement allows a transition for the enabled thread only if the thread identified by the statement's argument t has already terminated its execution (i.e., $t \notin I$). The execution of a `lock`-statement allows a transition for the enabled thread only if the mutex m is not acquired by any thread; it leads to a new configuration where the value of m is set to en. The execution of an `unlock`-statement on a mutex m, held by the enabled thread, frees it. When the enabled thread terminates, its configuration th_{en} is removed from the program configuration. The enabled thread in c' is nondeterministically selected from the pool of active threads of c'. Note that for any schedule for P under which the execution of a `join`- and `lock`-statement blocks there is an equivalent execution with a different schedule under which the statement does not block. For example, any execution of P containing a `join` from a thread t_1 on a thread t_2 can always be captured by another execution where t_1 is pre-empted immediately before the `join` and is re-scheduled only after t_2 terminates (if at all). This re-scheduling does not affect the reachability of error states, which is what our approach is checking for. For the purpose of our proofs, we therefore assume without loss of generality that `join`- and `lock`-statements never block in an execution. We finally define $\rightarrow_P$ to be the union of all relations $\rightarrow^i_P$.

Let P be a multi-threaded program with configurations c and c'. A *run* or *execution* of P from c to c', denoted by $c \rightsquigarrow_P c'$, is any sequence of zero or more transitions $c_0 \rightarrow_P c_1 \rightarrow_P \cdots \rightarrow_P c_n$ where $c = c_0$ and $c' = c_n$. A configuration c' is *reachable* in P if $c \rightsquigarrow_P c'$ and c is an initial configuration of P.

A *context* of a thread with identifier i from c to c', denoted by $c \rightsquigarrow^i_P c'$, is any run $c_0 \rightarrow^i_P c_1 \rightarrow^i_P \cdots \rightarrow^i_P c_n$ where $c = c_0$ and $c' = c_n$. A run $\pi = c \rightsquigarrow_P c'$ is *k-context bounded* if it can be obtained by concatenating at most k contexts of P, i.e., there exist $c_0, c_1, \ldots, c_{k'}$ with $k' \leq k$, such that $\pi_j = c_{i-1} \rightsquigarrow^{i_j}_P c_i$ is a context (of some thread i_j), for any $j \in \{1, \ldots, k'\}$, and $\pi = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_{k'}$. We assume that context switches happen only when the control is at a concurrent statement or a thread terminates its execution since this simplifies the exposition but does not affect the reachability of error states [75].

A *schedule* of a program P is a fixed sequence ρ of thread indices. A run $c_0 \rightsquigarrow^{i_1}_P c_1 \rightsquigarrow^{i_2}_P \cdots \rightsquigarrow^{i_n}_P c_n$ of P is a *round* w.r.t. ρ (also called a *round-robin execution*), if $i_1, i_2, \ldots, i_n$ is a subsequence of ρ. A run is k *round-robin* w.r.t. ρ if it can be obtained by concatenating at most k-round-robin executions of P w.r.t. ρ.

2.3 Reachability

Let P be a multi-threaded program and k be a positive integer. The *reachability problem* asks whether there is a reachable error configuration of P. Similarly, the *k-context (respectively, k-round-robin) reachability problem* asks whether there exists an error configuration of P that is reachable through a k-context (respectively, k-round-robin) execution.

Example. Figure 2(a) shows an example of a multi-threaded program with a reachable assertion failure. It models a producer-consumer system, with two shared variables, a mutex `m`, and an integer `c` that stores the number of items that have been produced but not yet consumed.

The `main` function initializes the mutex and spawns two threads executing `P` (*producers*) and two threads executing `C` (*consumers*). Each producer acquires `m`, increments `c`,[2] and terminates by releasing `m`. Each consumer first checks

[2]Note that in order to simplify the presentation, we treat the increment and decrement statements as atomic, rather than decomposing them into sequences where each statement involves at most one shared variable access.

```
mutex m; int c;

void P(int n) {
  bool b; int l;
  l:=n;
  lock m;
  b:=(c>0);
  if(b) then
    c:=c+1;
  else {
    c:=0;
    while(l>0) do {
      c:=c+1;
      l:=l-1;
    }
  }




  unlock m;

}



void C() {
  bool b;
  b:=(c>0);
  assume(b);
  c:=c-1;
  b:=(c>=0);
  assert(b);

}



void main() {
  int p0,p1,c0,c1;
  c:=0;
  init m;
  p0 := create P(5);
  p1 := create P(1);
  c0 := create C();
  c1 := create C();
}
```

(a) original program

```
mutex m; int c;

void P_1(int n) {
     bool b; int l;
  0: skip; l:=n;
  1: lock m;
  2: b:=(c>0);
     if(b) then
  3:   c:=c+1;
     else {
  4:   c:=0;
       if(!(l>0)) then goto l1;
  5:   c:=c+1;
       l:=l-1;
       if(!(l>0)) then goto l1;
  6:   c:=c+1;
       l:=l-1;
       assume(!(l>0));
  l1:  skip;
     }
  7: unlock m;
  8: return;
}

void P_2(int n) {...}

void C_1() {
     bool b;
  0: b:=(c>0);
     assume(b);
  1: c:=c-1;
  2: b:=(c>=0);
     assert(b);
  3: return;
}

void C_2() {...}

void main_0() {
     int p0,p1,c0,c1;
  0: c:=0;
  1: init m;
  2: p0 := create P_1(5);
  3: p1 := create P_2(1);
  4: c0 := create C_1();
  5: c1 := create C_2();
}
```

(b) bounded program

```
#include "lazycseq.h"
bool active[5]={true,false,false,false,false};
uint cs,ct,arg[5],pc[5],size[5]={5,8,8,2,2};
void arg[5];
#define G(L)   assume(cs>L);
#define J(A,B) if(pc[ct]>A||A>=cs) goto B;
int m; int c;

void P_1^seq(int n) {
           static bool b=*; static int l=*;
  0:J(0,1) skip; l:=n;
  1:J(1,2) seq_lock(m);
  2:J(2,3) b:=(c>0);
           if(b) then {
  3:J(3,4)    c:=c+1;
           } else { G(3)
  4:J(4,5)    c:=0;
              if(!(l>0)) then goto l1;
  5:J(5,6)    c:=c+1;
              l:=l-1;
              if(!(l>0)) then goto l1;
  6:J(6,7)    c:=c+1;
              l:=l-1;
              assume(!(l>0));
  l1:         G(6) skip;
           } G(6)
  7:J(7,8) seq_unlock(m);
  8:       return;
}

void P_2^seq(int n) {...}

void C_1^seq() {
           static bool b;
  0:J(0,1) b:=(c>0);
           assume(b);
  1:J(1,2) c:=c-1;
  2:J(2,3) b:=(c>=0);
           assert(b);
  3:       return;
}

void C_2^seq() {...}

void main_0^seq() {
           static int p0,p1,c0,c1;
  0:J(0,1) c:=0;
  1:J(1,2) seq_init(m);
  2:J(2,3) seq_create(5,1); p0:=1;
  3:J(3,4) seq_create(1,2); p1:=2;
  4:J(4,5) seq_create(0,3); c0:=3;
  5:J(5,6) seq_create(0,4); c1:=4;
  6:       return;
}

void main() {. . .  see Figure 5 . . . }
```

(c) sequentialized program

Fig. 2. (a) Original multi-threaded producer-consumer program containing a reachable assertion failure. In the `main` thread, functions `P` and `C` are both used twice to spawn a thread. (b) Corresponding bounded multi-threaded program, resulting from applying standard transformations (with a loop unrolling bound of $n = 2$) to the original program. The functions P_1 and P_2 represent two distinct copies of the P-thread that was spawned twice in the original program. (c) Corresponding sequentialized program. The code injected by the source transformation is shown in grey. For succinctness, we use C-style initializers in declarations as well as macros.

$$\begin{aligned}
&P && ::= (\mathit{decl};)^{*}\ (\texttt{void}\ f_i\ \texttt{(}\langle \mathit{decl}\texttt{,}\rangle^{*}\texttt{)}\ \texttt{\{}(\texttt{static}?\ \mathit{decl};)^{*}\, \mathit{stmt}\texttt{\}})_{i=0,\ldots,n}\\
&\mathit{decl} && ::= \mathit{type}\ \mathit{id}(\texttt{[}e\texttt{]})?\\
&\mathit{type} && ::= \texttt{bool} \mid \texttt{int} \mid \texttt{mutex} \mid \texttt{void}\\
&\mathit{stmt} && ::= \mathit{seq}; \mid \mathit{conc}; \mid \texttt{\{}\mathit{stmt}^{+}\texttt{\}}\\
&\mathit{seq} && ::= \texttt{assume}(b) \mid \texttt{assert}(b) \mid x := e \mid n{:}\texttt{return} \mid \texttt{if}(b)\ \texttt{then}\ \mathit{stmt}\ (\texttt{else}\ \mathit{stmt})? \mid \texttt{goto}\ l \mid l{:}\texttt{skip}\\
&\mathit{conc} && ::= x := y \mid y := x \mid t := \texttt{create}\ f_i(\langle e\texttt{,}\rangle^{*}) \mid \texttt{join}\ t \mid \texttt{init}\ m \mid \texttt{lock}\ m \mid \texttt{unlock}\ m \mid \texttt{destroy}\ m \mid n{:}\mathit{conc}
\end{aligned}$$

Fig. 3. Syntax of bounded multi-threaded programs.

whether there are still elements not yet consumed; if so (i.e., the `assume`-statement on $\texttt{c} > 0$ holds), it decrements `c`, checks the assertion $\texttt{c} \geq 0$ and terminates. Otherwise it terminates immediately.

The mutex ensures that at any point of the computation at most one producer is operating. However, the assertion can still be violated since there are two consumer threads, whose behaviors can be freely interleaved: with $\texttt{c} = 1$, both consumers can pass the assumption, so that both decrement `c` and one of them will write the value -1 back to `c`, and thus violate the assertion.

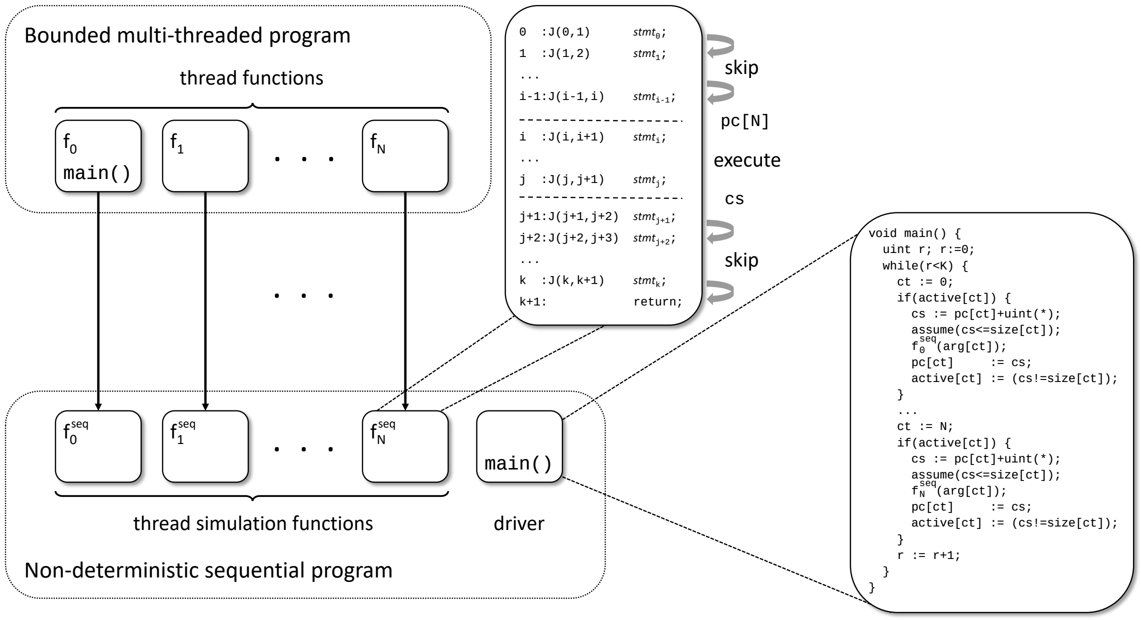
2.4 Bounded multi-threaded programs

BMC tools work on *bounded* programs, i.e., programs that are syntactically guaranteed to terminate after a bounded number of transitions. This intuitively means that there are no loops or (recursive) function calls and that all `goto`-statements describe forward jumps. The bounded version of a program can be obtained by applying standard program transformations such as loop unwinding and function inlining [28].

In this section we introduce the multi-threaded version of bounded programs. More precisely, *bounded multi-threaded programs* are defined by the syntax given in Figure 3; this syntax defines a variant of the multi-threaded programs defined by the full syntax in Figure 1. A bounded multi-threaded program consists of a finite number of thread start functions f_i; each f_i is a bounded function containing no loops and no function calls and all `goto`-statements must define forward jumps. Thus, we simplify the notation introduced in Section 2.2 and denote a configuration of a thread simply as a pair of the form $\langle \mathit{locals}, \mathit{pc}\rangle$.

We impose that each f_i (for $i > 0$) is used exactly once as start function in a `create`-statement to spawn a new thread, and that there exists a function f_0 corresponding to the original program's `main` thread. The first statement of each of these functions must either be a concurrent statement or a labeled `skip`-statement. The last statement of each of these functions must be a labeled `return`-statement, and this must be the only occurrence of a `return`-statement in each function.

We further impose that all concurrent statements in a function (as well as each function's first statement resp. final `return`-statement) are labeled with a numerical label n, such that the labels in each function start from 0 and increase by 1 according to the program order; any other label of the program must be non-numerical. This labeling restriction is required by our sequentialization to reposition the program counter after a simulated context switch. Note that numerical labels can only be targets of `goto`-statements introduced by our sequentialization. We use the term *visible statement* to denote any statement with a numerical label. We define an auxiliary function ℓ_P on the program counters in a program P such that $\ell_P(pc)$ is the preceding label, or more specifically, the largest numerical label of any statement


Bounded multi-threaded program
thread functions
f0 main()
f1
fN
skip
pc[N]
execute
cs
skip
thread simulation functions
driver
Non-deterministic sequential program

```
0  :J(0,1)      stmt0;
1  :J(1,2)      stmt1;
...
i-1:J(i-1,i)    stmti-1;
--------------------
i  :J(i,i+1)    stmti;
...
j  :J(j,j+1)    stmtj;
--------------------
j+1:J(j+1,j+2)  stmtj+1;
j+2:J(j+2,j+3)  stmtj+2;
...
k  :J(k,k+1)    stmtk;
k+1:            return;
```

```
void main() {
  uint r; r:=0;
  while(r<K) {
    ct := 0;
    if(active[ct]) {
      cs := pc[ct]+uint(*);
      assume(cs<=size[ct]);
      f0seq(arg[ct]);
      pc[ct]     := cs;
      active[ct] := (cs!=size[ct]);
    }
    ...
    ct := N;
    if(active[ct]) {
      cs := pc[ct]+uint(*);
      assume(cs<=size[ct]);
      fNseq(arg[ct]);
      pc[ct]     := cs;
      active[ct] := (cs!=size[ct]);
    }
    r := r+1;
  }
}
```

Fig. 4. Code-to-Code translation overview.

with program counter pc' within the same thread function as pc such that $pc' \le pc$. Note that ℓ_P is well-defined because the first statement in each thread function is labeled with 0. We are using $\ell_P^{max}(f_i)$ to denote the largest numerical label occurring in the thread simulation function f_i.

Furthermore, to simplify the presentation, we assume that each thread simulation function has a single argument of type `int` (although our implementation supports arbitrary argument lists). We also assume that the identifier of a thread with start function f_i is i.

Note that a general multi-threaded program can easily be transformed into a bounded multi-threaded program obeying all of the above restrictions by a series of simple source-to-source transformations such as loop unwinding, function inlining, and function cloning. Figure 2(b) gives an example of a bounded multi-threaded program that is obtained from the multi-threaded program from Figure 2(a) with an unwinding bound of 2. Note that we get two separate copies of each of the thread functions P and C, since the original program spawns two producer and two consumer threads.

We define a (bounded) *sequential program* as a (bounded) multi-threaded program that does not use any concurrent statements other than concurrent assignments but may use numerical labels at all statements.

3 LAZY SEQUENTIALIZATION FOR BOUNDED MULTI-THREADED PROGRAMS

We now describe our code-to-code translation from a bounded multi-threaded program P to a sequential program $P_{k,\rho}^{seq}$ that simulates all k-round-robin executions of P w.r.t. ρ, for any given $k > 0$ and schedule ρ. We fix a program P consisting of $n + 1$ functions $f_0, \ldots, f_n$, where f_0 denotes the `main` function, as shown in the upper half of Figure 4. We recall that by definition, P contains n calls to `create`, which spawn (at most) n threads using as start functions $f_1, \ldots, f_n$, respectively. Our translation guarantees that P fails an assertion in a k-round-robin execution w.r.t. ρ if and only if $P^{seq}_{k,\rho}$ fails the *same* assertion. Furthermore, the translation allows us to perform on the bounded multi-threaded program all the reachability analyses that are supported by the sequential backend tool.

$P^{seq}_{k,\rho}$ is composed of a new function `main` and a *thread simulation function* f^{seq}_i for each thread f_i in P, as shown in the lower half of Figure 4. The new `main` of $P^{seq}_{k,\rho}$ calls, in the order given by ρ, the functions f^{seq}_i for k complete rounds (see Figure 5 for details). For each thread it maintains the numerical label at which the context switch was simulated in the previous round and where the computation must thus resume in the current round.

Each f^{seq}_i is essentially f_i, with each call to a thread routine such as `create`, `join`, `lock`, or `unlock` replaced by a call to a corresponding simulation function and few lines of additional control code that re-positions the program counter. When executed, each f^{seq}_i 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 (recall that the numerical labels in f_i identify the relevant context switch points in the original code, i.e., the visible statements). We make all local variables persistent (i.e., `static`), so that their valuations are retained across invocations of the thread simulation functions, which are regular functions in $P^{seq}_{k,\rho}$; hence, we do not need to re-compute them when resuming suspended executions.

Figure 2(c) shows the resulting sequentialized program for the bounded Producer-Consumer example shown in Figure 2(b). The parts in black correspond to the unwound original program, those in light grey are injected to achieve the desired sequentialization.

We now describe our translation in a top-down fashion. We start by describing the (global) auxiliary variables used in the translation in Section 3.1. Then, we give the details of function `main` of $P^{seq}_{k,\rho}$ in Section 3.2, and illustrate how to construct each f^{seq}_i from f_i in Section 3.3. Finally, we discuss how the thread routines are simulated in Section 3.4. We convey an informal correctness argument as we go along. The formal translation is given in Section 3.5 while Section 4 contains a formal proof of correctness.

3.1 Auxiliary Data Structures

Let `N` be a symbolic constant denoting the maximal number of threads in the program, i.e., $n + 1$. During the simulation of P, the sequentialized program $P^{seq}_{k,\rho}$ maintains the following data structures:

- `bool active[N]` tracks whether a thread is active, i.e., has been created but not yet terminated. Initially, only `active[0]` is `true` since f^{seq}_0 simulates the `main` function of P;
- `int arg[N]` stores the argument used for thread creation (recall that for simplicity we have assumed a single argument with an implicit call-by-reference semantics in Section 2.4);
- `int size[N]` stores the largest numerical label $\ell^{max}_P(f_i)$ for each thread simulation function f_i;
- `int pc[N]` stores the label of the last context switch point for each thread simulation function;
- `int ct` tracks the identifier of the thread currently under simulation;
- `int cs` contains the (pre-guessed) numerical label at which the next context switch for thread `ct` will be simulated.

Note that the thread simulation functions f^{seq}_i read but do not write any of the above data structures. `N` and `size[]` are constants computed from the bounded program and remain unchanged during the simulation. `arg[]` is set by `seq_create` (that simulates `create`) and remains unchanged once set. `active[]` is set by `seq_create` and unset by the main driver as described in the next section. `pc[]`, `ct`, and `cs` are updated by the main driver following the mechanism shown in the next section.

```
void main() {
    uint r; r:=0;                          // round counter
    while(r<K) {                           // driver loop
        ct := 0;                           // simulate thread f_0
        if(active[ct]) {
            cs := pc[ct]+uint(*);          // (1) choose next context switch point
            assume(cs<=size[ct]);          // (2) ... and constrain it
            f_0^seq(arg[ct]);              // (3) call thread simulation function
            pc[ct]     := cs;              // (4) store context switch point
            active[ct] := (cs!=size[ct]);
        }
        . . .
        ct := N;                           // simulate thread f_N
        if(active[ct]) {
            cs := pc[ct]+uint(*);
            assume(cs<=size[ct]);
            f_N^seq(arg[ct]);
            pc[ct]     := cs;
            active[ct] := (cs!=size[ct]);
        }
        r := r+1;
    }
}
```

Fig. 5. Main driver.

3.2 Main Driver

Figure 5 shows the code of the function `main` in $P^{seq}_{k,\rho}$. It drives the simulation: each iteration of the loop simulates an entire round of a computation of P. To simulate each thread $f_{\texttt{ct}}$ we invoke the corresponding simulation function $f^{seq}_{\texttt{ct}}$ with the argument `arg[ct]` that was originally used to create the thread. The order in which the functions are called corresponds to the round-robin schedule ρ, which we assume for simplicity to be the sequence $\langle 0, \ldots, n \rangle$.

For each active thread the driver thus executes the following steps:

(1) nondeterministically guess the label for the next context switch point and store it in `cs`,
(2) check that the value is appropriate,
(3) call the thread simulation function to simulate the thread from `pc[ct]` through to `cs`,
(4) store `cs` in `pc[ct]` (used in the next round to restart the computation from this label), and
(5) set `active[ct]` to `false` if `cs` is the label of the `return`-statement (i.e., the simulation of the thread is completed).

The choice of an appropriate value for `cs` is simplified by the bounded structure of P, more precisely, by the fact that all jumps are forward. We can thus pick any value for `cs` that is between the value stored in `pc[ct]` (corresponding to the case that the thread will not make any progress, hence skips the round) and the largest numerical label in $f^{seq}_{\texttt{ct}}$ (which corresponds to the last possible context switch point in the code of the corresponding thread $f_{\texttt{ct}}$). Note that this guess is the only source of nondeterminism introduced by our translation.

3.3 Thread Translation

Here we describe how each function f_i representing a thread in P is converted into a thread simulation function f^{seq}_i in $P^{seq}_{k,\rho}$.

Persistence of Thread Local Storage. Each thread f_i in P is simulated in $P^{seq}_{k,\rho}$ by repeated calls to the thread simulation function f^{seq}_i; each invocation executes a fragment of the code according to the context switch points that are guessed in the `main` function. f^{seq}_i therefore needs to maintain the thread-local state between consecutive invocations. We achieve this by imposing that each thread-local variable's storage class is `static`. However, since local variables are considered uninitialized right after their declaration (i.e., can take any value from their respective domains), while static variables are initialized to `0` by default, we explicitly assign a nondeterministic value to such local variables. For instance, the local variable declaration `int l;` in Figure 2(b) is turned into `static int l:=*;`. This directly applies to all primitive types and can be done at the level of the components for arrays and structured types.

Thread Pre-emption and Resumption. When a function f^{seq}_i is called for the first time, it starts its execution from the beginning. In the subsequent calls, it must skip over the statements already executed in previous calls, in order to resume the simulation from its last context switch point. When the control reaches the label guessed for the next context switch, it must return without executing any further statements. Different solutions exist to implement this using `goto`-statements and distinct labels associated with every meaningful context switch point in the code. We tried to use a multiplexer at the top of the thread's body, implemented with a `switch` and a series of `goto`-statements, to jump over the statements already executed, directly to the starting label. We also injected additional code at the context switch label to return immediately when the thread is pre-empted. However, this schema has performed poorly in our experiments, possibly because it introduces complex control flow branching, and consequently complex formulas.

In contrast, the schema we present here scales well when used together with BMC backends, although it may look counterintuitive at first. Recall that bounded multi-threaded programs use consecutive natural numbers as labels on the visible statements in program order, starting with 0 for the first statement in each function. We then use these labels and `goto`-statements in a way that avoids complex branching in the control flow. Specifically, right after each numerical label i (except for the last one) we inject (via a macro `J`) a conditional jump of the form

`if(pc[ct]>`i `||` i`>=cs) then goto` j`;` (J macro)

in front of the original statement. Note that `J` is always called with arguments i and i+1 (see Figure 2(c)). When the thread simulation function tries to execute statements before the context switch label of the previous round, or after the guessed context switch label, the condition becomes true, and the control thus jumps to the next label without executing actual statements of the thread. This achieves the positioning of the control at the program counter corresponding to `pc[ct]` with potentially multiple hops, and similarly, the fall-through to the last statement of the thread (which is by assumption always a `return`) when the guessed context switch label is reached. Note that whenever the control is between these two labels the injected condition becomes false and the statements of $f^{seq}_{\texttt{ct}}$ are executed as in the original thread. Figure 4 illustrates this.

As an example, consider the sequentialized program in Figure 2(c), and assume that P^{seq}_1 is called (i.e., `ct` = 1) with the previous context switch point at label 2 (i.e., `pc[1]` = 2) and the next context switch point at label 6 (i.e., `cs` = 6). At label 0, the condition of the injected `if`-statement is true, thus the `goto`-statement is executed and the control jumps to label 1. There the condition is again true, and the control jumps to label 2. Now, the condition check fails, thus the underlying code is executed, up to label 6. Here, the condition of the injected `if`-statement holds again, so the control jumps first to label 7, and then to label 8, thus reaching the `return` statement without executing any other code of the thread.

```
void P_1(int n) {
     bool b; int l;
   0: skip; l:=n;
   1: lock m;
   2: b:=(c>0);
     if(b) then
   3:   c:=c+1;
     else {
   4:   c:=0;
        if(!(l>0)) then goto l1;
   5:   c:=c+1;
        l:=l-1;
        if(!(l>0)) then goto l1;
   6:   c:=c+1;
        l:=l-1;
        assume(!(l>0));
  l1:   skip;
     }
   7: unlock m;
   8: return;
}
```

(a) bounded program

```
void P_1^seq(int n) {
    static bool b=*; static int l=*;
  0:/*J(0,1)*/ if(pc[ct]>0||0>=cs) then goto 1; skip; l:=n;
  1:/*J(1,2)*/ if(pc[ct]>1||1>=cs) then goto 2; seq_lock(m);
  2:/*J(2,3)*/ if(pc[ct]>2||2>=cs) then goto 3; b:=(c>0);
                                              if(b) then {
  3:/*J(3,4)*/ if(pc[ct]>3||3>=cs) then goto 4;    c:=c+1;
                                              } else {/*G(3)*/ assume(cs>3);
  4:/*J(4,5)*/ if(pc[ct]>4||4>=cs) then goto 5;    c:=0;
                                                 if(!(l>0)) then goto l1;
  5:/*J(5,6)*/ if(pc[ct]>5||5>=cs) then goto 6;    c:=c+1;
                                                 l:=l-1;
                                                 if(!(l>0)) then goto l1;
  6:/*J(6,7)*/ if(pc[ct]>6||6>=cs) then goto 7;    c:=c+1;
                                                 l:=l-1;
                                                 assume(!(l>0));
  l1:                                            /*G(6)*/ assume(cs>6); skip;
                                              }/*G(6)*/ assume(cs>6);
  7:/*J(7,8)*/ if(pc[ct]>7||7>=cs) then goto 8; seq_unlock(m);
  8:                                          return;
}
```

(b) sequentialized program

Fig. 6. (a) Code of thread function P_1 (see Figure 2 for the full code). (b) Code of corresponding thread simulation function $\mathsf{P}_1^{\mathsf{seq}}$. Code injected by the sequentialization transformation is shown in grey; J- and G-macros are shown as comments before their expansion.

Handling Control-flow Branching. Some care must be taken in handling branches and jumps in the control flow, to avoid simulating infeasible computations. Consider for example the `if-then-else` in P_1 of Figure 6(a), and assume that `pc[ct]` = 2 and `cs` = 3 when executing the corresponding code of Figure 6(b), i.e., in this round the sequentialized program is assumed to simulate (feasible) control flows between labels 2 and 3. However, if $\mathsf{c} \leq 0$ (and thus `b` evaluates to `false`), then the program's control flow goes from label 2 into the `else`-branch right. Without the `assume`-statement asserted by the macro `G(3)`, the control flow would reach label 4, and the condition in the `if`-statement inserted by `J(4,5)` would be tested. Since this would hold, the control flow would then slide through to label 8, and return to the `main` driver, which would then set `pc[ct]` to 3. In the next round, the computation would then duly resume from this label—which in the underlying multi-threaded execution is unreachable! Similar problems can occur when the context switch label is in the body of the `else`-branch, and with `goto`-statements.

Note that we could fix this problem by assigning `pc` in the called function rather than in the `main` driver. However, this would require to inject at each possible context switch point an assignment to `pc` guarded by a nondeterministic choice. This has performed poorly in our experiments. The main reason for this is that the control code is spread "all over" and thus even small increments of its complexity may significantly increase the complexity of the formulae computed by the backend tools.

We therefore simply prune away simulations that store unreachable labels in `pc`. For this, we inject (via a macro `G`) a simple guard of the form

$$\texttt{assume(cs > } j\texttt{);} \qquad (\mathsf{G}\text{ macro})$$

where j is largest numerical label up the point where the guard is injected. We insert such guards at all control flow locations that are target of an explicit or implicit jump, i.e., right at the beginning of each `else`-block, right after the `if`-statement, and right after any non-numerical label in the actual code of the simulated thread that can be the target of a `goto`-statement of the starting program (see Figure 6 for examples).

We now show informally that this adjustment indeed suffices to prune away all simulations involving such spurious control flows. The formal correctness proof is given in the next section. Consider first the case of `goto`-statements. We assume without loss of generality that the statement's execution is feasible in the multi-threaded program and that the target's label is in the code after the intended context switch point `cs`; note that in this case `cs` is unreachable because the control flow jumps over the corresponding label. However, when resuming the thread in the simulation we would instead restart from `cs`, thus ending up into a spurious computation. But since a `G`-macro is inserted at the jump's target with an argument that is greater or equal to `cs`, and because all `goto`-statements describe forward jumps, the injected assumption fails, and this simulation gets correctly pruned away.

The argument for `if`-statements requires two cases that follow the same lines as discussed for the `goto`-statements. First consider that the planned context switch point `cs` is in the `then`-branch but the `if`-condition evaluates to `false`. Then the guard at the beginning of the `else`-statement fails, because the argument passed to the `G`-macro here corresponds to the last label in the `then`-branch and thus has at least the value of `cs`, so that the injected `assume`-statement fails. In the symmetric scenario (i.e., the planned context switch point is in the `else`-branch but the `if`-condition evaluates to `true`), the `assume`-statement guard injected by the guard after the `if`-statement will fail because the value `cs` is guaranteed to be less or equal than the last label in the `else`-branch, which is used as argument in the `G`-macro. Note that the `J`-macro at the last context switch point in the `else`-branch jumps over this guard so that it never prunes feasible control flows.

We stress that though the guess of the context switch points is done eagerly and we thus need to prune away infeasible guesses, the simulation of the input program is still done lazily. In fact, the computation is pruned as soon as we pass the guessed context switch before executing any other statement of the original program. Thus, all the statements of the input program executed until that point correspond to a prefix of a feasible computation of the input program.

3.4 Simulation of Thread Routines

For each thread routine we provide a verification stub, i.e., a simple standard function that simulates the original implementation for verification purposes. Figure 7 shows the stubs for the routines used in this paper. Our implementation provides stubs for further thread routines in the `pthread` API, which are typically equally straightforward.

In `seq_create` we simply set the thread's `active` flag and store the argument to be passed (later, from the main driver) to the thread simulation function. The `seq_create` stub uses an additional integer argument `tid` as thread identifier that is statically added during the translation, and is set to the index i of the thread function f_i used in the corresponding `create`-statement.

Under the semantics of multi-threaded programs, a thread invoking `join t` cannot progress unless `t` is terminated. In the simulation, a thread is terminated (and the driver sets the corresponding `active`-entry to `false`) if it has reached the thread's last numerical label. The stub `seq_join` thus uses an `assume`-statement with the condition `active[tid]=false` to prune away any simulation that involves switching to a thread that is stuck at a statement `join t`. This is aligned with the semantics of `join`-statements given in Section 2.2.

For mutexes, we need to know whether they are free or already destroyed, or which thread holds them otherwise. We thus model mutexes as integers, and define two constants `FREE` and `DESTROYED` that have values different from any possible thread index. When we initialize or destroy a mutex we assign it with the appropriate constant. If we want to lock a mutex we assert that it is not destroyed and then check whether it is free before we assign to it the index of the thread that has invoked `lock`. Similarly to the case of `join`, according to the semantics given in Section 2.2, we abort

```
void seq_init(int m) {
  m := FREE;
}

void seq_destroy(int m) {
  m := DESTROYED;
}

void seq_lock(int m) {
  assert(m != DESTROYED);
  assume(m == FREE);
  m := ct;
}

void seq_unlock(int m) {
  assert(m == ct);
  m := FREE;
}
```

```
void seq_create(int arg, int tid) {
  active[tid] := true;
  arg[tid] := arg;
}

int seq_join(int tid) {
  assume(active[tid] == false);
}
```

Fig. 7. Thread simulation stubs.

the simulation if the lock is held by another thread. If a thread executes `unlock`, we first assert that the lock is held by the invoking thread and then set it to `FREE`.

3.5 Rewrite Rules

Here we formalize the general translation informally described above. This is done using rewrite rules defined over the syntax of bounded multi-threaded programs. Let P be a bounded multi-threaded program and $P^{seq}_{k,\rho}$ be a sequentialized program for P. $P^{seq}_{k,\rho}$ is obtained from P by applying the translation $[\![\cdot]\!]_{k,\rho}$ defined by the rewrite rules given in Figure 8. $[\![P]\!]_{k,\rho}$ starts by including a header file `lazycseq.h` that contains the declarations of the simulation functions for the thread routines shown in Figure 7; the corresponding definitions are contained in `lazycseq.c`. This is followed by the declaration of the auxiliary data structures used by the sequentialization (see Section 3.1), followed by the declaration of the original global variables, which remains unchanged. All thread functions are first sequentialized (as discussed in Section 3.3) and then appended to the program. Finally, the main driver constructed according to ρ is appended (see Figure 5 for the structure of the main driver with $\rho = \langle 0, \ldots, n \rangle$).

The translation rules for most statement types are straightforward. Slightly more involved are the rules for the `if`- and (labeled) `skip`-statements where the `G`-macros are inserted, and the (labeled) concurrent statements where the `G`-macros are inserted (both as sketched in Section 3.3). Note that the labels ℓ_1, ℓ_2 and ℓ denote the largest numerical label up to the point where the corresponding statements textually occur in the program.

Figure 2(c) shows the result of the translation map $[\![\cdot]\!]_{k,\rho}$ of Figure 8 applied to the bounded multi-threaded program shown in Figure 2(b).

4 CORRECTNESS

In this section, we give a formal proof of correctness of the lazy sequentialization schema defined in Section 3. Specifically, we prove the following theorem:

$$\left\llbracket\begin{array}{l}(decl;)^* \\ (\texttt{void}\ f_i(\langle decl,\rangle^*) \\ \ \{(\texttt{static?}\ decl;)^*\, stmt\})_{i=0,\ldots,n}\end{array}\right\rrbracket_{k,\rho} \mathrel{\widehat{=}} \begin{array}{l}\texttt{\#include "lazycseq.h"} \\ \texttt{bool active[N]:=[true,false,...,false];} \\ \texttt{uint cs,ct,pc[N],size[N]:=[...];} \\ \texttt{int arg[N];} \\ \texttt{\#define G(L) assume(cs>L);} \\ \texttt{\#define J(A,B) if(pc[ct]>A||A>=cs) goto B;} \\ (decl;)^* \\ (\texttt{void}\ f_i^{seq}(\langle decl,\rangle^*) \\ \ \{(\texttt{static}\ decl;)^*\llbracket stmt\rrbracket_{k,\rho}\})_{i=0,\ldots,n} \\ \texttt{main()\{}\text{... see Figure 5 ...}\texttt{\}}\end{array}$$

$$\llbracket \texttt{assume}(b)\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{assume}(b)$$

$$\llbracket \texttt{assert}(b)\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{assert}(b)$$

$$\llbracket x := e\rrbracket_{k,\rho} \mathrel{\widehat{=}} x := e$$

$$\llbracket n : \texttt{return}\rrbracket_{k,\rho} \mathrel{\widehat{=}} n : \texttt{return}$$

$$\left\llbracket\begin{array}{l}\texttt{if}(b)\ \texttt{then}\ stmt_1 \\ \texttt{else}\ stmt_2\end{array}\right\rrbracket_{k,\rho} \mathrel{\widehat{=}} \begin{array}{l}\{\texttt{if}(b)\ \texttt{then}\ \llbracket stmt_1\rrbracket_{k,\rho} \\ \ \texttt{else}\ \{\texttt{G}(\ell_1);\llbracket stmt_2\rrbracket_{k,\rho}\};\ \texttt{G}(\ell_2);\}\end{array}$$

$$\llbracket \texttt{goto}\ l\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{goto}\ l$$

$$\llbracket l : \texttt{skip}\rrbracket_{k,\rho} \mathrel{\widehat{=}} \begin{cases}\{\texttt{0: J(0,1); skip}\} & \text{if } l = \texttt{0} \\ \{l : \texttt{G}(\ell);\ \texttt{skip}\} & \text{otherwise}\end{cases}$$

$$\llbracket x := y\rrbracket_{k,\rho} \mathrel{\widehat{=}} x := y$$

$$\llbracket y := x\rrbracket_{k,\rho} \mathrel{\widehat{=}} y := x$$

$$\llbracket t := \texttt{create}\ f_i(e)\rrbracket_{k,\rho} \mathrel{\widehat{=}} \{\ \texttt{seq\_create}(e,i);\ t := i\ \}$$

$$\llbracket \texttt{join}\ t\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{seq\_join}(t)$$

$$\llbracket \texttt{init}\ m\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{seq\_init}(m)$$

$$\llbracket \texttt{lock}\ m\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{seq\_lock}(m)$$

$$\llbracket \texttt{unlock}\ m\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{seq\_unlock}(m)$$

$$\llbracket \texttt{destroy}\ m\rrbracket_{k,\rho} \mathrel{\widehat{=}} \texttt{seq\_destroy}(m)$$

$$\llbracket n{:}\ conc\rrbracket_{k,\rho} \mathrel{\widehat{=}} \{n{:}\,\texttt{J}(n, n{+}1);\ \llbracket conc\rrbracket_{k,\rho}\}$$

$$\llbracket \{stmt^+\}\rrbracket_{k,\rho} \mathrel{\widehat{=}} \{\llbracket stmt\rrbracket^+_{k,\rho}\}$$

Fig. 8. Rewrite rules for the lazy sequentialization. Note that the labels ℓ_1, ℓ_2 and ℓ denote the largest numerical label up to the point where the corresponding statements textually occur in the program.

Theorem 4.1 (Correctness). *Let P be a bounded multi-threaded program, ρ be a schedule of P, and k be a positive integer. P fails an assertion through a k-round-robin execution w.r.t. ρ if and only if $\llbracket P\rrbracket_{k,\rho}$ fails an assertion.*

Proof Strategy. We prove the two directions of Theorem 4.1 respectively in Section 4.1 and Section 4.2. For the forward direction, we prove a key property in Lemma 4.3: any k-round-robin execution of P w.r.t. ρ can be *simulated* by the sequentialized program $\llbracket P\rrbracket_{k,\rho}$. For the backward direction, we also prove a key property in Lemma 4.5: every execution of $\llbracket P\rrbracket_{k,\rho}$ ending with a *relevant* configuration, i.e., a configuration that is *equivalent* to a non-error P configuration,

```
...
void C1() {
    bool b;


  0:
  ① b:=(c>0);
    assume(b);
  1:
    c:=c-1;
  2:
    b:=(c>=0);
    assert(b);
  3:
    return;
}
...
```

(a) bounded program

```
  void main() {
  ...
  // first round                              // second round
  ...                                          ...
  ct := 3;      // simulate C1                 ct := 3;      // simulate C1
  if(active[ct]) {                             if(active[ct]) {
    cs := pc[ct]+uint(*);                        cs := pc[ct]+uint(*);
    assume(cs<=size[ct]);                        assume(cs<=size[ct]);
25: if(pc[ct]>0||0>=cs) then goto 26;      58: if(pc[ct]>0||0>=cs) then goto 59;
  ② b:=(c>0);                                ③ b:=(c>0);
      assume(b);                                   assume(b);
26: if(pc[ct]>1||1>=cs) then goto 27;      59: if(pc[ct]>1||1>=cs) then goto 60;
      c:=c-1;                                      c:=c-1;
27: if(pc[ct]>2||2>=cs) then goto 28;      59: if(pc[ct]>2||2>=cs) then goto 61;
      b:=(c>=0);                                   b:=(c>=0);
      assert(b);                                   assert(b);
28:                                        61:
      skip;                                        skip;
    pc[ct]     := cs;                            pc[ct]     := cs;
    active[ct] := (cs!=size[ct]);                active[ct] := (cs!=size[ct]);
  }                                            }
  ...                                          ...
}
```

(b) inlined sequentialized program

Fig. 9. PC mapping example. (a) shows part of the bounded multi-threaded program P, see Figure 2 for the full program; (b) shows the corresponding part of the inlined sequentialized program $\widehat{P}_{k,\rho}$ for $k = 2$, with the code for the first (resp. second) round on the left (resp. right). Injected control code is shown in gray; note that the inlining process updates the labels used in the J-macros but leaves the values used in the guards unchanged. Corresponding labels and statements are shown on the same line. ①, ②, and ③ denote program counters (which are distinct from the numerical labels). We have $pc\_map_{P,k}(①, 1) = ②$ and $pc\_map_{P,k}(①, 2) = ③$, corresponding to the first (resp. second) occurrence of the relevant statement `b:=(c>0);` in $\widehat{P}_{k,\rho}$. Note that a similar mapping is also used in Lazy-CSeq to trace counter-examples back to the original (unbounded) program.

captures a (prefix of a) k-round-robin execution of P w.r.t. ρ. Before doing so, let us introduce some assumptions and notations that will allow us to simplify the presentation.

Assumptions and Notations. Henceforth we assume that P is a bounded multi-threaded program with thread functions $f_0, \dots, f_n$, and that the local variables of all f_i's and the global variables of $[\![P]\!]_{k,\rho}$ have distinct names; this is without loss of generality because variables can be renamed.

We use $\widehat{P}_{k,\rho}$ to denote $[\![P]\!]_{k,\rho}$ with (i) the `while`-loop of the main driver unrolled k times (once for each round), (ii) all $n + 1$ thread simulation functions inlined into the main driver, (iii) the local variables of $[\![P]\!]_{k,\rho}$ lifted to global variables, and (iv) the initial statements of the `main`-function initializing these lifted variables with nondeterministic values. Note that the resulting program is syntactically well-formed because the local variables have different names by assumption, and because the inlining renames apart the numerical labels and updates the jump targets. Note further that $\widehat{P}_{k,\rho}$ is semantically equivalent to $[\![P]\!]_{k,\rho}$ because the initialization of the lifted variables captures the semantics of local variables, and because the inlining happens after the macro expansion, the numerical labels used in J- and G-macros are renamed consistently, so that the simulation remains unchanged; in fact, after the initialization phase $\widehat{P}_{k,\rho}$ and $[\![P]\!]_{k,\rho}$ can execute the same transitions. Since $\widehat{P}_{k,\rho}$ consists by definition only of one thread that has no local variables, the only meaningful components of a configuration of $\widehat{P}_{k,\rho}$ are the shared variable valuation and the program counter. Thus, we simplify the notation introduced in Section 2.2 and denote a configuration of $\widehat{P}_{k,\rho}$ simply as $\widehat{c} = \langle \widehat{sh}, \widehat{pc} \rangle$, where $\widehat{sh}$ is a shared variable valuation, and $\widehat{pc}$ is a program counter.

In the proofs, we need to match configurations of P with those of $\widehat{P}_{k,\rho}$. We recall that when transforming P into $[\![P]\!]_{k,\rho}$ each thread function of P is replaced with the corresponding simulation function, which is in turn obtained by replacing each original statement of the thread function with a block of statements that contains either the original statement or, in case of a call to a thread routine, the call to the corresponding thread simulation stub (see Figure 8 for details). We refer to these statements as the *relevant* statements of $[\![P]\!]_{k,\rho}$. Due to the inlining of the simulation functions in the translation from $[\![P]\!]_{k,\rho}$ to $\widehat{P}_{k,\rho}$, each relevant statement is copied k times in $\widehat{P}_{k,\rho}$, once for each round of computation; note that in the inlining the single `return`-statement in each thread function is replaced by a `skip`-statement. We still refer to such copies as relevant statements. We define a map $pc\_map_{P,k}(pc, r)$ that relates statements of P with the corresponding relevant statements of $\widehat{P}_{k,\rho}$ as follows: for a program counter pc of a thread function f of P and round number $r \in [0, k-1]$, $pc\_map_{P,k}(pc, r)$ denotes the program counter of the relevant statement of $\widehat{P}_{k,\rho}$ that (1) corresponds to the statement at program counter pc and (2) occurs in the r-th inlined copy of f. Figure 9 gives an example.

A configuration c of P and a configuration $\widehat{c}$ of $\widehat{P}_{k,\rho}$ are *equivalent* if the valuations of the variables in P coincide in c and $\widehat{c}$, and the valuation of the auxiliary control variables in $\widehat{c}$ is *consistent* with c, i.e., the valuation of `active` is consistent with the threads that are present in c, the valuation of `ct` identifies the currently enabled thread, the valuation of `pc[i]` matches the numerical label at the program counter in c for each thread i except possibly for the enabled one, and the program counter of $\widehat{c}$ corresponds to that of the enabled thread in c. Formally:

*Definition 4.2 (*Equivalent Configurations*).* Let P be a bounded multi-threaded program with shared variables V and local variables L_i for each thread function f_i where V and all L_i are pairwise disjoint, and $c = \langle sh, en, \{th_i\}_{i \in I}\rangle$ be a configuration of P with $th_i = \langle locals_i, pc_i\rangle$ for any of the active threads $i \in I \subseteq [0, n]$. Further, let $k > 0$ and $\widehat{c} = \langle \widehat{sh}, \widehat{pc}\rangle$ be a configuration of the sequentialized program $\widehat{P}_{k,\rho}$. For a round number $r \in [0, k-1]$, c is *r-equivalent to* $\widehat{c}$, denoted $c \equiv_r \widehat{c}$, if the following holds:

(1) (a) $\widehat{sh}(v) = sh(v)$ for all $v \in V$, and (b) $\widehat{sh}(l) = locals_i(l)$ for all $i \in [0, n]$ and $l \in L_i$;
(2) for any $i \in [0, n]$, $\widehat{sh}(\texttt{active}[i]) = \texttt{true}$ iff $i \in I$;
(3) $\widehat{sh}(\texttt{ct}) = en$;
(4) for any $i \in (I \setminus \{en\})$, $\widehat{sh}(\texttt{pc}[i]) = \ell_P(pc_i)$;
(5) $\widehat{pc} = pc\_map_{P,k}(pc_{en}, r)$. □

A configuration $\widehat{c} = \langle \widehat{sh}, \widehat{pc}\rangle$ of $\widehat{P}_{k,\rho}$ is *relevant* if there exists a program counter pc of P and a round number $r \in [0, k-1]$ such that $\widehat{pc} = pc\_map_{P,k}(pc, r)$, i.e. $\widehat{pc}$ is the program counter of a relevant statement. Note that if $c \equiv_r \widehat{c}$, then $\widehat{c}$ must be relevant.

For the sake of simplicity, in the rest of the section we fix a schedule $\rho = \langle 0, \ldots, n\rangle$. Note that any other schedule could be accommodated by renaming the functions.

Let $\pi = c_0 \rightarrow_P c_1 \rightarrow_P \cdots \rightarrow_P c_m$, with $c_i = \langle sh_i, en_i, A_i\rangle$ and $A_i = \{\langle locals_i^j, pc_i^j\rangle_{j \in I_i}\}$, be a k-round-robin execution (w.r.t. ρ) of P, where $locals_i^j$ is the valuation of the local variables of the j-th thread after the i-th execution step.

We can identify a new round in π by the fact that the identifier en_i of the enabled thread in configuration c_i is smaller than its counterpart en_{i-1} in the predecessor configuration c_{i-1}. We thus define a function $round_\pi(i)$ that describes the *minimal round-robin bound* for each prefix of π (i.e., the minimal number of rounds required to reach the last configuration c_i in this prefix). Formally, $round_\pi : [0,m] \longrightarrow [0,k-1]$ is defined as:

$$round_\pi(i) = \begin{cases} 0 & \text{if } i \in [0,1]; \\ round_\pi(i-1)+1 & \text{if } i \in [2,m] \text{ and } en_i < en_{i-1}; \\ round_\pi(i-1) & \text{otherwise (i.e., } i \in [2,m] \text{ and } en_i \geq en_{i-1}). \end{cases}$$

We also use a function $cs_\pi(i)$ to denote the numerical label at which the enabled thread in configuration c_i will context switch out (this is explicitly captured by the value of cs during the simulation of P by $\widehat{P}_{k,\rho}$), or in the case of the last context of π where no context switch will occur, the largest numerical label of the thread function f_{en_m} enabled in the final configuration c_m, i.e., $\ell_P^{max}(f_{en_m})$. Formally, $cs_\pi : [0,m] \longrightarrow N$ is defined as follows:

$$cs_\pi(i) = \begin{cases} \ell_P^{max}(f_{en_m}) & \text{if } i = m; \\ \ell_P(pc_{i+1}^{en_i}) & \text{if } i < m \text{ and } en_i \neq en_{i+1}; \\ cs_\pi(i+1) & \text{otherwise.} \end{cases}$$

4.1 Simulation of P executions by $\widehat{P}_{k,\rho}$

Here we prove that every k-round-robin execution of P w.r.t. ρ can be simulated by $\widehat{P}_{k,\rho}$, and thus also by $[\![P]\!]_{k,\rho}$. Intuitively, starting from equivalent configurations, P and $\widehat{P}_{k,\rho}$ execute the "same" statements and reach equivalent configurations. However, since $\widehat{P}_{k,\rho}$ contains additional control code, each transition of P may correspond to a sequence of transitions of $\widehat{P}_{k,\rho}$. In our proof (by induction), we consider each of those sequences as split into two parts: the first part corresponds to the simulation in $\widehat{P}_{k,\rho}$ of a single statement in P, which may end at a non-relevant configuration, and a (possibly empty) second part that reaches the next relevant configuration in $\widehat{P}_{k,\rho}$.

Lemma 4.3. *Let P be a bounded multi-threaded program with thread functions $f_0, \ldots, f_n$, $\rho = \langle 0, \ldots, n \rangle$, k be a positive integer, and c_0 be an initial configuration of P. For every k-round-robin execution*

$$\pi = c_0 \longrightarrow_P c_1 \longrightarrow_P \cdots \longrightarrow_P c_m$$

of P w.r.t. ρ, there is an initial configuration $\widehat{c}_I$ of $\widehat{P}_{k,\rho}$ and an execution

$$\widehat{\pi} = \widehat{c}_I \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_0 \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_1 \rightsquigarrow_{\widehat{P}_{k,\rho}} \cdots \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_m$$

of $\widehat{P}_{k,\rho}$ such that $c_i \equiv_{round_\pi(i)} \widehat{c}_i$, for every $i \in [0,m]$.

Proof. For $i \in [0,m]$, let $c_i = \langle sh_i, en_i, A_i \rangle$ with $A_i = \{\langle locals_i^h, pc_i^h \rangle\}_{h \in I_i}$, and $\widehat{c}_i = \langle \widehat{sh}_i, \widehat{pc}_i \rangle$. The proof is by induction over $j \in [0,m]$ for the following stronger property $\mathcal{P}(j)$:

> There exists an initial configuration $\widehat{c}_I$ and an execution
>
> $$\widehat{\pi}_j = \widehat{c}_I \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_0 \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_1 \rightsquigarrow_{\widehat{P}_{k,\rho}} \cdots \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$$
>
> of $\widehat{P}_{k,\rho}$ such that for $i \in [0,j]$, (a) $c_i \equiv_{round_\pi(i)} \widehat{c}_i$, (b) $\widehat{sh}_i(\texttt{cs}) = cs_\pi(i)$, and (c) $\widehat{sh}_i(\texttt{pc[}en_i\texttt{]}) \leq \ell_P(pc_i)$.

Note that $\mathcal{P}(m)$ is stronger than the lemma's statement because it requires that the valuation of the auxiliary variable cs in $\widehat{\pi}$ at configuration $\widehat{c}_i$ must coincide with the numerical label at which the thread enabled at c_i will context switch out next in π, and moreover, the maximum numerical label that precedes the current program counter must be at least as large as the label stored in the array pc.

Base case: We prove that $\mathcal{P}(0)$ holds by showing the existence of two configurations $\widehat{c}_I$ and $\widehat{c}_0$ such that $\widehat{\pi}_0 = \widehat{c}_I \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_0$, $c_0 \equiv_{round_\pi(0)} \widehat{c}_0$, $\widehat{sh}_0(\mathtt{cs}) = cs_\pi(0)$, and $\widehat{sh}_0(\mathtt{pc[}en_0\mathtt{]}) \leq \ell_P(pc_0)$.

Since $\widehat{P}_{k,\rho}$ contains no local variables, it has a unique initial configuration $\widehat{c}_I$ with the following properties: by construction (1) the valuations in $\widehat{c}_I$ of the shared variables V of P coincide with those in c_0; (2) the valuation of active[0] is true, and the valuation of active[i] is false for any $i \in [1, n]$ (reflecting the fact that the only active thread is the one corresponding to the main procedure of P); (3) the valuation of variable ct (used to keep track of the identifier of the thread under simulation) is 0; and (4) the valuation of all the elements of the array pc is also 0 (which is the numerical label of the first statement of each thread function). Thus, for c_0 and $\widehat{c}_I$ parts (1.a) and (2)-(4) of Definition 4.2 hold.

We also observe that for c_0 and $\widehat{c}_I$ parts (1.b) and (5) of Definition 4.2 may not hold instead. In fact, for part (1.b), the local variables of P have nondeterministic values in c_0 but evaluate to 0 in $\widehat{c}_I$ (since they are mapped to global variables in $\widehat{P}_{k,\rho}$). For part (5), the program counter of $\widehat{P}_{k,\rho}$ is positioned at the beginning of the main driver rather than at the beginning of the sequentialized main procedure (i.e., the first thread simulation function).

We now describe an execution of $\widehat{P}_{k,\rho}$ from $\widehat{c}_I$ to a configuration $\widehat{c}_0$ where parts (1.b) and (5) also hold. Recall that at the beginning of the main driver of $\widehat{P}_{k,\rho}$ the variables corresponding to the local variables of P get assigned with nondeterministic values. Here we choose the same c_0 values for these variables, which allows us to establish part (1.b) in an intermediate configuration $\widehat{c}'_0$. From $\widehat{c}'_0$, we then execute the first thread simulation block corresponding to the first if-statement in the first loop iteration of the main driver of $[\![P]\!]_{k,\rho}$, where ct is 0 (see Figure 5 again), and pick the transition that sets cs to $cs_\pi(0)$, which is always possible as we can choose any nondeterministic value in the range of the thread labels of the main procedure of P. After that, the code of the inlined sequentialized main function gets executed, and the control reaches the macro J(0,1) guarding the first statement. Here the condition of the if-statement is evaluated to false and the control moves to the first statement. We take this configuration as $\widehat{c}_0$. Since the original variables (those from P) are not affected by the execution just described and since $round_\pi(0) = 1$, we get that $c_0 \equiv_{round_\pi(0)} \widehat{c}_0$ and $\widehat{sh}_0(\mathtt{cs}) = cs_\pi(0)$ hold. Finally, $\widehat{sh}_0(\mathtt{pc[}en_0\mathtt{]}) \leq \ell_P(pc_0)$ holds true because $\widehat{sh}_0(\mathtt{pc[}en_0\mathtt{]}) = 0$, which is the minimal numerical label within any tread function.

Inductive step. For $j \in [1, m]$, assume that $\mathcal{P}(j-1)$ holds. To prove that $\mathcal{P}(j)$ also holds, we show the existence of a configuration $\widehat{c}_j$ such that $\widehat{c}_{j-1} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$, and (a) $c_j \equiv_{round_\pi(j)} \widehat{c}_j$, (b) $\widehat{sh}_j(\mathtt{cs}) = cs_\pi(j)$, and (c) $\widehat{sh}_i(\mathtt{pc[}en_j\mathtt{]}) \leq \ell_P(pc_j)$.

We first consider the cases when the statement executed along $c_{j-1} \rightarrow_P c_j$, i.e., $P(pc_{j-1}^{en_{j-1}})$, is not a return-statement. We break the execution of $\widehat{P}_{k,\rho}$ into two parts $\widehat{c}_{j-1} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}$ and $\widehat{c} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$. The first part of the simulation handles the update of P's variables according to the execution of statement $P(pc_{j-1}^{en_{j-1}})$. This establishes parts (1) and (2) of Definition 4.2. The second part of the simulation takes the control to the next relevant configuration of $\widehat{P}_{k,\rho}$, if this is not already the case. Here we possibly update auxiliary variables by the driver and execute the J- and G-macros. This establishes the remaining parts of $\mathcal{P}(j)$. For the second part, we proceed by distinguishing on whether the transition $c_{j-1} \rightarrow_P c_j$ involves a context switch (i.e., whether $en_{j-1} \neq en_j$) or not.

Henceforth pc and pc' denote respectively the values of the program counter of thread $t_{en_{j-1}}$ of P in configurations c_{j-1} and c_j, i.e., $pc = pc_{j-1}^{en_{j-1}}$ and $pc' = pc_j^{en_{j-1}}$ (recall that $t_{en_{j-1}}$ is the enabled thread in c_{j-1}). Moreover, pc_{cs} denotes the value of the program counter of the same thread when the next context switch actually occurs. Note that $pc < pc' \leq pc_{cs}$

must hold since the control in bounded programs only moves forward. Similarly, we define $\widehat{pc}$, $\widehat{pc}'$, and $\widehat{pc}_{cs}$ as the corresponding program counters in $\widehat{P}_{k,\rho}$, i.e., $\widehat{pc} = pc\_map_{P,k}(pc, round_\pi(j-1))$, $\widehat{pc}' = pc\_map_{P,k}(pc', round_\pi(j-1))$, and $\widehat{pc}_{cs} = pc\_map_{P,k}(pc_{cs}, round_\pi(j-1))$. Note that $\widehat{pc} < \widehat{pc}' \leq \widehat{pc}_{cs}$ also holds, in fact our translation preserves the program order within the same inlined copy of each thread function. Note further that $\widehat{pc} = \widehat{pc}_{j-1}$ by the inductive hypothesis.

Proof of the first part: $(\widehat{c}_{j-1} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c})$: Let $stmt = P(pc)$, i.e., the statement executed in the transition $c_{j-1} \rightarrow_P c_j$. We proceed by case inspection over the syntactic type of *stmt*.

Thread creation and joining. If *stmt* is a `create`-statement, i.e., a statement of the form $t := \texttt{create}\ f_i(e)$, the initial configuration of the newly created thread, which has identifier i, is added to the pool of active threads in c_{j-1}. By construction, in $\widehat{P}_{k,\rho}$ the `create`-statement is transformed into the call `seq_create(`e, i`)`, followed by the assignment $t := i$ (see Figure 8). Note that the call updates `active[`i`]` to `true` (see Figure 7), and since `create` $f_i(e)$ evaluates to i, we get that parts (1) and (2) of Definition 4.2 hold for $\widehat{c}$ and c_j w.r.t. thread i.

If *stmt* is a `join`-statement with the thread identifier *tid* as argument, the thread identified by *tid* must already have terminated its execution—otherwise, P could not have made its transition into c_j. Hence, the configuration for the terminated thread must already be removed from c_{j-1}, and by the inductive hypothesis `active[`tid`]` must be `false`. Therefore, $\widehat{P}_{k,\rho}$ transitions silently through the `assume`-statement in `seq_join` (see Figure 7), thus complying with the requirement of part (2) of Definition 4.2 for the thread *tid*.

Since all the components of the configuration that are not explicitly mentioned in the two cases above stay unchanged along both $\widehat{c}_{j-1} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}$ and $c_{j-1} \rightarrow_P c_j$, by the given arguments and the inductive hypothesis we get that parts (1) and (2) of Definition 4.2 hold for $\widehat{c}$ and c_j.

Lock operations. If *stmt* is a `lock`-statement with a mutex m as argument, P can make a transition only if m is available (i.e., not held by any other thread), and thus m evaluates to en_{j-1} in configuration c_j. By the inductive hypothesis, m must be free in c_{j-1} and have the special value `FREE` in $\widehat{c}_{j-1}$, and $\widehat{sh}_{j-1}(\texttt{ct}) = en_{j-1}$. Thus, when $\widehat{P}_{k,\rho}$ executes `seq_lock`, which simulates `lock` (see Figure 7), it transitions through the `assert`-and then the `assume`-statement, and finally sets m to the value of en_{j-1}.

If *stmt* is an `unlock`-statement with a mutex m as argument, then m is free in configuration c_j. The equivalent operation is done in $\widehat{P}_{k,\rho}$ by the corresponding `seq_unlock` function (see Figure 7) that assigns `FREE` to m. If *stmt* is an `init`- or `destroy`-statement, a similar argument holds as above.

In all cases, the transition of $\widehat{P}_{k,\rho}$ performs the same memory update as P does and thus since everything else, except the program counter, is unchanged, by the inductive hypothesis again parts (1) and (2) of Definition 4.2 hold for c_j and $\widehat{c}$.

Remaining statements. In all the other cases, *stmt* and $\widehat{P}_{k,\rho}(\widehat{pc})$ are *identical*, because by the inductive hypothesis $\widehat{pc}$ points to the relevant statement of $[\![stmt]\!]_{k,\rho}$, i.e., past any inserted `J`-macros (see Figure 8). Since c_{j-1} and $\widehat{c}_{j-1}$ are equivalent, also by the inductive hypothesis, the common variables of P and $\widehat{P}_{k,\rho}$ will thus hold the same values in P and $\widehat{P}_{k,\rho}$ after executing *stmt*.[3] Since everything else except the program counter is unchanged, by the inductive hypothesis parts (1) and (2) of Definition 4.2 hold for for c_j and $\widehat{c}$.

[3]Nondeterministic transitions of P due to the occurrence of the $*$ operator can be matched by transitions in $\widehat{P}_{k,\rho}$ where $*$ yields the same value as in P.

Proof of the second part ($\widehat{c} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$): In the second part of the simulation, we need to show that the control moves to the next relevant configuration $\widehat{c}_j$ of $\widehat{P}_{k,\rho}$, if it is not already there, and that $\mathcal{P}(j)$ holds. For this, we recall that for $\widehat{c}$ and c_j, parts (1) and (2) of Definition 4.2 hold, and since the variables involved in these parts are not updated along $\widehat{c} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$, they continue to hold up to $\widehat{c}_j$. We proceed by distinguishing on whether the transition $c_{j-1} \rightarrow_P c_j$ involves a context switch or not. Before this, we state two key properties:

A. The original code of the P thread functions, modulo the thread routines, can be obtained from the corresponding thread simulation functions by removing the injected J- and G-macros (see Figure 8). This can be easily shown by induction on the grammar given in Figure 3.

B. All the G-macros possibly executed along $\widehat{c} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$ are indeed immaterial. In fact, any such G-macro consists of a single `assume`-statement checking `cs` $> \ell$, where ℓ is $\ell_P(pc') - 1$ if the statement at pc' is a concurrent statement (in this case, there would be a numerical label between the G-macro and the statement), and $\ell_P(pc')$ otherwise (see Figure 8). Note that the numerical label corresponding to the next context switch happening along π, identified by $cs_\pi(j-1)$, appears in the code after the occurrence of the G-macro. Thus, ℓ must be strictly less than $cs_\pi(j-1)$. Since $\widehat{sh}_{j-1}(\texttt{cs}) = cs_\pi(j-1)$ by the inductive hypothesis, and $\widehat{sh}(\texttt{cs}) = \widehat{sh}_{j-1}(\texttt{cs})$, we get that `cs` $> \ell$ must hold at $\widehat{c}$.

We denote with $\widehat{c}'$ the configuration reached along $\widehat{c} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$ after the execution of all the possibly injected G-macros. Since such G-macros are immaterial (Property B), and along $\widehat{c}_{j-1} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}$ none of the auxiliary variables `ct`, `cs`, and `pc` are updated, we get that their valuations in $\widehat{c}_{j-1}$ and $\widehat{c}'$ are the same. In the remaining part of the proof, we distinguish on whether a context switch occurs when moving from c_{j-1} to c_j along π.

Without context switch. In this case $en_{j-1} = en_j$, which entails that $cs_\pi(j) = cs_\pi(j-1)$, and $\ell_P(pc) \leq \ell_P(pc') < cs_\pi(j)$.

If the statement at pc' is not concurrent, we set $\widehat{c}_j = \widehat{c}'$. Since the valuations of `ct`, `cs`, and `pc` are the same in $\widehat{c}_{j-1}$ and $\widehat{c}'$, by the inductive hypothesis we also get that parts (3)-(4) of Definition 4.2 hold for c_j and $\widehat{c}_j$. Moreover, by Property A and the inductive hypothesis, we have that also part (5) of Definition 4.2 holds for c_j and $\widehat{c}_j$.

If the statement at pc' is concurrent, from $\widehat{c}'$ we execute the injected J-macro, which we now show to be immaterial as well: the first disjunct of the `if`-condition, i.e., `pc[ct]` $> \ell_P(pc')$, does not hold since (i) by part (c) of the inductive hypothesis $\widehat{sh}_{j-1}(\texttt{pc}[en_{j-1}]) \leq \ell_P(pc_{j-1})$ and $\widehat{sh}_{j-1}(\texttt{ct}) = en_{j-1}$ both hold, (ii) `pc` and `ct` are not updated, and (iii) in the textual order the numerical labels can only increase and the control can only move forward; the second disjunct of the `if`-condition, i.e., $\ell_P(pc') \geq$ `cs`, also holds `false`, since we assumed that no context switch occurs at this time, and similarly to as observed in Property B, the numerical label corresponding to the concurrent statement must be smaller than $\widehat{sh}_j(\texttt{cs})$.

Regardless of the type of statement addressed by pc', the arguments above also entail $\widehat{sh}_j(\texttt{pc}[en_j]) \leq \ell_P(pc_j)$. Moreover, $\widehat{sh}_j(\texttt{cs}) = cs_\pi(j-1)$ holds by inductive hypothesis, and $cs_\pi(j-1) = cs_\pi(j)$ since no context switch happens. Thus, we get $\widehat{sh}_j(\texttt{cs}) = cs_\pi(j)$, which concludes the proof in this case.

With context switch. As argued in Section 2.2, we only consider runs π where context switches happen at concurrent statements, or when a thread terminates. Note that the latter case will be handled further below.

Let $stmt'$ be the concurrent statement addressed by pc'. From Figure 8, we can see that $stmt'$ is preceded verbatim by a numerical label and an injected J-macro. The second disjunct of the if-condition in this J-macro, i.e., $\ell_P(pc') \geq$ cs, holds because $en_{j-1} \neq en_j$ (since a context switch occurs) and thus $\ell_P(pc') = cs_\pi(j-1)$. This causes the control to jump to the next numerical label, and for the same reason, all subsequent J-macros jump to the next numerical label, until the control moves back to the main driver code (see Figure 4), right after the inlined call to the thread simulation function of thread $t_{en_{j-1}}$ at round $round_\pi(j-1)$. Let $\widehat{c}''$ denote the reached configuration. Observe that along the run from $\widehat{c}'$ to $\widehat{c}''$ no variable is updated.

From $\widehat{c}''$, variable pc[en_{j-1}] is set to cs and the control moves to the next block of the main driver corresponding to the simulation of the thread enabled in configuration c_j. We now argue that in the intermediate thread simulation blocks, if any, (i) none of their relevant statements is executed, and (ii) the values of pc will not be modified up to configuration $\widehat{c}_j$ except pc[en_j]. Note that since $en_j \neq en_{j-1}$, the inductive hypothesis and part (ii) of the claim above ensure that part (4) of Definition 4.2 also holds for c_j and $\widehat{c}_j$. To prove the claim we note the following. The block corresponding to inactive threads are simply skipped. For a block corresponding to an active thread t_i, cs is (nondeterministically) set to the value of pc[i], and this causes the control to jump in multiple hops over the J-macros of the inlined thread simulation function: the first disjunct holds up to the numerical label pc[i] and the second disjunct holds in the remaining part of the code. Note that for all these blocks the values of all variables except for ct remain unchanged.

When the control reaches the block corresponding to thread t_{en_j}, i.e., the thread enabled in configuration c_j, ct is set to en_j. Then, cs is set to $cs_\pi(j)$; this is possible because pc[en_j] evaluates to $\ell_P(pc_{j-1}^{en_j})$ by the inductive hypothesis (specifically part (4) of Definition 4.2), and by definition $cs_\pi(j) > \ell_P(pc_{j-1}^{en_j})$ since in the run π starting from c_j at least one concurrent statement of thread t_{en_j} is executed. After that, similarly to the above, the control jumps over the J-macros in the inlined copy of the simulation function of thread t_{en_j} at round $round_\pi(j)$, up to the J-macro at label $\ell_P(pc_{j-1}^{en_j})$. Here the if-condition evaluates to false (recall that pc[en_j] $= \ell_P(pc_{j-1}^{en_j})$ and cs $> \ell_P(pc_{j-1}^{en_j})$), hence the control moves to the associated relevant statement. We pick the reached configuration as $\widehat{c}_j$.

We now argue that $\mathcal{P}(j)$ holds. First, as already observed parts (1), (2) and (4) of Definition 4.2 hold. Part (3) is ensured by the above assignment to ct. By the inductive hypothesis and the reasoning above, we get that the program counter of $\widehat{c}_j$ is exactly $pc\_map_{P,k}(pc_j^{en_j}, round_\pi(j))$, and thus also part (5) holds. Therefore, Definition 4.2 holds for c_j and $\widehat{c}_j$, and thus part (a) of $\mathcal{P}(j)$ is shown. From the above assignment to cs, we immediately get that part (b) of $\mathcal{P}(j)$ also holds and thus in $\widehat{P}_{k,\rho}$ the simulation of a context switch starts from the J-macro at the numerical label stored in cs. Moreover, since $en_{j-1} \neq en_j$ we get $pc_{j-1}^{en_j} = pc_j^{en_j}$, and as noted above pc[$en_j$] $= \ell_P(pc_{j-1}^{en_j})$, thus also part (c) of $\mathcal{P}(j)$ holds and we are done with the general case.

To conclude the proof it remains to show that $\mathcal{P}(j)$ also holds when pc points to a return-statement. When this is the case, the control in $\widehat{c}_{j-1}$ is at the end of the inlined code of a thread simulation function. Since the inlining replaces the return-statement with a skip-statement, the control immediately moves back to the code of the main driver (see Figure 4), right after the inlined call to the thread simulation function of thread $t_{en_{j-1}}$ at round $round_\pi(j-1)$. From this point on, the behaviour of $\widehat{P}_{k,\rho}$ is as from configuration $\widehat{c}''$ in the "*with context switch*" case above, except that active[en_{j-1}] is set to false but this is required to restore part (2) of Definition 4.2 (thread $t_{en_{j-1}}$ has terminated). Thus, $\mathcal{P}(j)$ holds in this case as well. □

We now show the forward direction of Theorem 4.1. For a configuration c of P and a configuration $\widehat{c}$ of $\widehat{P}_{k,\rho}$ such that $c \equiv_r \widehat{c}$, we have that every `assert`-statement of P fails from c if and only if the same `assert`-statement fails from $\widehat{c}$. Note that `assert`-statements are left unchanged by our translation (see Figure 8). Therefore from Lemma 4.3, we get that the forward direction of Theorem 4.1 holds.

Corollary 4.4. *Let P be a bounded multi-threaded program, ρ be a schedule of P, and k be a positive integer. If P fails an assertion through a k-round-robin execution w.r.t. ρ then $[\![P]\!]_{k,\rho}$ fails an assertion.*

4.2 Simulation of $\widehat{P}_{k,\rho}$ executions by P

Here we show that each execution $\widehat{\pi}$ of $\widehat{P}_{k,\rho}$ ending with a relevant configuration can be simulated by a k-round-robin execution of P that matches all the relevant configurations of $\widehat{\pi}$ with equivalent configurations according to Definition 4.2. Using this result, we then show the completeness of our approach.

Lemma 4.5. *Let P be a bounded multi-threaded program, $\rho = \langle 0, \dots, n\rangle$, and k be a positive integer. For every execution*

$$\widehat{\pi} = \widehat{c}_I \leadsto_{\widehat{P}_{k,\rho}} \widehat{c}_0 \leadsto_{\widehat{P}_{k,\rho}} \widehat{c}_1 \leadsto_{\widehat{P}_{k,\rho}} \cdots \leadsto_{\widehat{P}_{k,\rho}} \widehat{c}_m$$

of $\widehat{P}_{k,\rho}$ such that $\widehat{c}_I$ is initial and $\widehat{c}_0, \widehat{c}_1, \dots, \widehat{c}_m$ are all the relevant configurations of $\widehat{\pi}$, there is a k-round-robin execution of P w.r.t. ρ

$$\pi = c_0 \longrightarrow_P c_1 \longrightarrow_P \cdots \longrightarrow_P c_m$$

such that c_0 is initial, and $c_j \equiv_{round_\pi(j)} \widehat{c}_j$, for every $j \in [0, m]$.

Proof. For $j \in [0, m]$, let $\widehat{c}_j = \langle \widehat{sh}_j, \widehat{pc}_j\rangle$, and $c_j = \langle sh_j, en_j, A_j\rangle$ with $A_j = \{\langle locals_j^h, pc_j^h\rangle\}_{h \in I_j}$.

The proof is by induction over $j \in [0, m]$ for the following stronger property $Q(j)$:

> There is an execution $\pi_j = c_0 \longrightarrow_P c_1 \longrightarrow_P \cdots \longrightarrow_P c_j$ with c_0 an initial configuration such that for every $i \in [0, j]$, denoting $r_i = \widehat{sh}_i(\mathtt{r})$, (a) $c_i \equiv_{r_i} \widehat{c}_i$, and (b) $r_i \geq round_{\pi_j}(i)$.

Base case: We choose the initial configuration c_0 such that the values of the common variables coincide with those of $\widehat{c}_0$. This is possible because by construction *all* variables in P become global in $\widehat{P}_{k,\rho}$ and are initialized as in P: those corresponding to the shared variables are assigned as in P while those corresponding to the local variables are initialized with nondeterministic values at beginning of the main driver before reaching the first relevant configuration.

We now show that $Q(0)$ holds for c_0 and $\widehat{c}_0$.

First, we recall that the auxiliary control variables of an initial configuration of $\widehat{P}_{k,\rho}$ are by construction (see Figure 8) initialized as follows: (1) the valuation of `active[0]` is `true` while that of `active[`i`]` is `false`, for every $i \in [1, n]$, (2) variable `ct` is set to 0, and (3) all the elements of array `pc` are set to 0, which is the numerical label of the first statement in any thread function. Thus, parts (1)-(4) of Definition 4.2 hold for configurations c_0 and $\widehat{c}_I$.

Now, we argue that the first relevant statement simulated along $\widehat{\pi}$ must correspond to the first statement in the main thread:

(1) The first relevant statement along $\widehat{\pi}$ must be in the thread simulation function of the main thread. This is because `active[`i`]` is `false` for $i \neq 0$ in $\widehat{c}_I$, and can be set to `true` only by simulating a thread creation statement but this is not possible being $\widehat{c}_0$ the first relevant configuration.

(2) Moreover, this first statement corresponds to the first statement of the `main` thread in P. To see this, observe that in $\widehat{P}_{k,\rho}$ the simulation of the `main` thread starts in the first simulation block of this thread where `cs > pc[0]` holds.

In fact, if `cs` = `pc[0]`, the `if`-conditions of the `J`-macros in the corresponding simulation function are always true, which causes the control to return to the main driver without simulating any relevant statement. Also note that when this is the case, the value of `pc[0]` stays unchanged. As soon as `cs` > `pc[0]` holds, the `if`-condition of the first `J`-macro evaluates to `false` (since each thread simulation function starts with a statement labeled with the numerical label 0) and the control moves to the first relevant statement which, by construction, must correspond to the first statement of the `main` thread in P.

From the observations above, part (5) of Definition 4.2 holds for c_0 and $\widehat{c}_0$, and our choice of r_0. Moreover, since `ct` is set to 0 before entering a block corresponding to the main thread (see Figure 4) and the other variables except for `cs` stay unchanged along $\widehat{c}_I \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_0$, the other parts of Definition 4.2 hold as well and thus part (a) of $Q(0)$ holds. Also, $r_0 \geq 0$ by definition, thus part (b) of $Q(0)$ holds, which concludes the base case.

Inductive step. For $j \in [1, m]$, assume that $Q(j-1)$ holds. The rest of the proof has many arguments in common with that of Lemma 4.3. From now on we will focus on the additional arguments while recalling the rest from the other proof.

Let *stmt* be the current statement of the enabled thread in c_{j-1}, i.e., $stmt = P(pc_{j-1}^{en_{j-1}})$.

We choose c_j as follows. By the inductive hypothesis, we get $c_{j-1} \equiv_{r_{j-1}} \widehat{c}_{j-1}$ and thus $\widehat{pc}_{j-1} = pc\_map_{P,k}(pc_{j-1}^{en_{j-1}}, r_{j-1})$. As detailed in the first part of the proof of Lemma 4.3, the block of code corresponding to *stmt* in $\widehat{P}_{k,\rho}$ performs a simulation of *stmt*. According to our translation, see Figure 8, this code must be executed along $\widehat{c}_{j-1} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$, which leads to the conclusion that *stmt* is executable from c_{j-1}. Denote c_j the configuration the configuration obtained from c_{j-1} after the execution of *stmt* that captures a possible change of the enabled thread according to $\widehat{c}_j$, i.e., $c_{j-1} \longrightarrow_P c_j$ where $en_j = \widehat{sh}_j(\texttt{ct})$.

To prove that $Q(j)$ holds for such a choice of c_j, we need to show that $c_j \equiv_{r_j} \widehat{c}_j$ and $r_j \geq round_{\pi_j}(j)$.

We start with the former but we postpone the case in which *stmt* is a `return` statement to later in the proof. The fact that $c_{j-1} \equiv_{r_{j-1}} \widehat{c}_{j-1}$ and *stmt* can be executed puts us into the same condition as in the first part of the proof of Lemma 4.3. Thus, we can immediately see that parts (1) and (2) of Definition 4.2 hold for c_j and the configuration $\widehat{c}$ reached after the simulation of *stmt*. However, the execution from $\widehat{c}$ to $\widehat{c}_j$ will not update any of the variables involved in those two parts. Therefore parts (1) and (2) of Definition 4.2 also hold for c_j and $\widehat{c}_j$.

The block of code executed along $\widehat{c} \rightsquigarrow_{\widehat{P}_{k,\rho}} \widehat{c}_j$ takes care of positioning the control to the next relevant statement at $\widehat{c}_j$. According to our translation schema as well as the description given in the second part of the proof of Lemma 4.3, while positioning the control, $\widehat{P}_{k,\rho}$ may or may not go through the main driver.

We consider first the case when the control stays within the same inlined copy of the thread simulation function. All the cases of how the execution can evolve are exactly the same as those detailed in the second part of the proof of Lemma 4.3. This is because Property A holds, and the `G`-macros are immaterial (otherwise $\widehat{c}_j$ could not be reached from $\widehat{c}_{j-1}$). Thus, part (5) of Definition 4.2 also holds for c_j and $\widehat{c}_j$. Moreover, the variable `ct` and the array `pc` are not modified since they are updated only in the main driver, and therefore have the same valuation as in $\widehat{c}_{j-1}$. This is consistent with the fact that there is no change of context along $c_{j-1} \longrightarrow_P c_j$ (by our choice for en_j) and the program counters of the threads, other than the enabled thread, stay unchanged. Hence, the remaining parts of Definition 4.2 hold as well for c_j and $\widehat{c}_j$, and therefore, $c_j \equiv_{r_j} \widehat{c}_j$ holds in this case.

We handle the remaining cases, i.e., when the control passes through the main driver before positioning at the next relevant statement, mainly as in the second part of the proof of Lemma 4.3. However there are two main differences that we will discuss in detail here.

The first difference is the following. In the proof of Lemma 4.3, when defining the execution of $\widehat{P}_{k,\rho}$ we choose to set variable `cs` to $cs_\pi(j-1)$ before entering the inlined code of the thread simulation function, and prove that when the control goes back to the main driver, `cs` has the value of the label of the concurrent statement where the context switch occurred. Here we cannot make any assumptions about the value of `cs` as it is assigned with a nondeterministic value. However, we now show that this property indeed holds along any run of $\widehat{P}_{k,\rho}$ regardless of the value of `cs`. Let us consider again when the control in $\widehat{P}_{k,\rho}$ is right after the execution of the simulation code of *stmt*. As detailed in the proof of Lemma 4.3, before exiting the inlined thread simulation function, we possibly execute a sequence of `G`-macros before the first `J`-macro is executed and the `if`-condition of this `J`-macro evaluates to true because the associated label ℓ is not smaller than the value of `cs`. Also, according to our translation (see Figure 8), such `G`-macros ensure that when the first `J`-macro is executed the value of `cs` must not be strictly less than ℓ. Thus, `cs` must evaluate exactly to ℓ.

The second difference is that the control after leaving a thread simulation block may also enter a following block (corresponding to a lager value of `r`) containing the code of the *same* thread simulation function. When this happens, we use the same arguments given in the second part of the proof of Lemma 4.3, in relation to entering and exiting the blocks of code of the thread simulation functions without executing any relevant statement. Thus, we have that when the control reaches the next relevant statement, all the control variables except for `r` (which is increased) stay unchanged, and in all the cases we conclude that parts (3) and (4) of Definition 4.2 also hold for c_j and $\widehat{c}_j$.

The case when *stmt* is `return` can be argued as in the proof of Lemma 4.3 with addition of the above considerations, which concludes the proof that $c_j \equiv_{r_j} \widehat{c}_j$.

We now show part (b) of $Q(j)$, i.e., $r_j \geq round_{\pi_j}(j)$. We first recall that along any execution of $\widehat{P}_{k,\rho}$ (see the main driver code in Figure 5):

- `r` is first initialized to 0 and then is only incremented at each iteration of the `while`-loop;
- `ct` is assigned with the thread identifier before entering the code of the corresponding thread function;
- in each loop iteration thread simulation functions are called by increasing thread identifiers.

As first consequence, in general $r_j \geq r_{j-1}$ holds. Further, if $en_j < en_{j-1}$, since by definition $en_j = \widehat{sh}_j(\texttt{ct})$ and by the inductive hypothesis $en_{j-1} = \widehat{sh}_{j-1}(\texttt{ct})$, we also get $r_j \geq r_{j-1} + 1$. We now prove that $r_j \geq round_{\pi_j}(j)$. For this, we distinguish the following two cases:

- if $en_j \geq en_{j-1}$, by definition we get that $round_{\pi_j}(j) = round_{\pi_j}(j-1)$ and $round_{\pi_j}(j-1) = round_{\pi_{j-1}}(j-1)$; since by the inductive hypothesis $r_{j-1} \geq round_{\pi_{j-1}}(j-1)$, we get that $r_{j-1} \geq round_{\pi_j}(j)$; thus from $r_j \geq r_{j-1}$, we get that $r_j \geq round_{\pi_j}(j)$ must also hold;
- if $en_j < en_{j-1}$, by definition we get that $round_{\pi_j}(j) = round_{\pi_j}(j-1)+1$ and again $round_{\pi_j}(j-1) = round_{\pi_{j-1}}(j-1)$; since by the inductive hypothesis $r_{j-1} \geq round_{\pi_{j-1}}(j-1)$, from $r_j \geq r_{j-1} + 1$ we get $r_j \geq round_{\pi_j}(j-1) + 1$; by $round_{\pi_j}(j) = round_{\pi_j}(j-1) + 1$, we must thus have that $r_j \geq round_{\pi_j}(j)$ holds also in this case.

This concludes the proof. □

We now show the backward direction of Theorem 4.1. Note that all the assertions in $[\![P]\!]_{k,\rho}$ are relevant statements. Furthermore, any failing assertion leads to an error configuration that ends the execution. Therefore, we can conclude that for each such run, by Lemma 4.5, there is a corresponding run of P that fails the same assertion (and thus also reaches an error configuration).

Corollary 4.6. *Let P be a bounded multi-threaded program, ρ be a schedule of P, and k be a positive integer. If $[\![P]\!]_{k,\rho}$ fails an assertion then P fails an assertion through a k-round-robin execution w.r.t. ρ.*

5 IMPLEMENTATION

We have implemented our sequentialization in the Lazy-CSeq tool for multi-threaded C programs, based on our CSeq framework [27], and have used this implementation for an exhaustive evaluation. In this section, we first sketch the underlying framework (see Section 5.1) before we give more details on the implementation of the tool itself (see Section 5.2).

5.1 The CSeq Framework

Lazy-CSeq is developed within the CSeq framework [27]. The framework builds on ideas from the original CSeq tool [34] but has been improved and fully re-engineered. It provides support for quickly developing new sequentialization-based verification tools. It has also been used to implement the MU-CSeq [87–89, 92] and UL-CSeq [69] tools.

The framework comprises several modules; modules are either *translators* that implement source-to-source transformations of C programs (such as function inlining, loop unrolling, or the rewrite rules given in Figure 8), or *wrappers* that are used for general-purpose tasks that do not produce source code. Each tool built within CSeq is identified by a *configuration* that corresponds to a sequence of translators followed by a sequence of wrappers. It takes as input the file containing the source code of the multi-threaded C program to analyze and the list of verification parameters. The input parameters are passed to the appropriate modules, with the input source file passed to the first module. The output of each module is passed as input to the following module, and the output of the last module in this sequence is returned as the analysis result.

In a typical configuration, the first translator is 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 subsequent 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. Figure 10 sketches the configuration for Lazy-CSeq.

Translators run in two steps: first, the input code is parsed in order to build the abstract syntax tree (AST), the symbol table, and other data structures; second, 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` [7], a parser for C that uses `PLY`, an implementation of Lex and Yacc. We override `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. We found that string-based source transformations are more intuitive and easier to learn than source-to-source translation tools based on term rewrite rules (e.g., DMS [6]). Combined with Python's flexibility it is thus relatively easy to implement complex code transformations quickly.

In addition, the CSeq framework contains reusable program transformation modules that simplify the implementation of analysis tools even further. In particular, it provides a number of generic program optimizations such as constant propagation and constant folding, and loop and control flow transformations that simplify the syntactic structure (e.g., normalize loop constructs); other modules optimize the handling of some concurrent programming idioms (e.g., spin locks). These modules can be used to progressively simplify the input syntax and thus the complex transformations typically occurring later in a configuration. Further modules such as variable renaming, function inlining and duplication of thread functions, and loop unrolling can be used to produce the bounded programs.

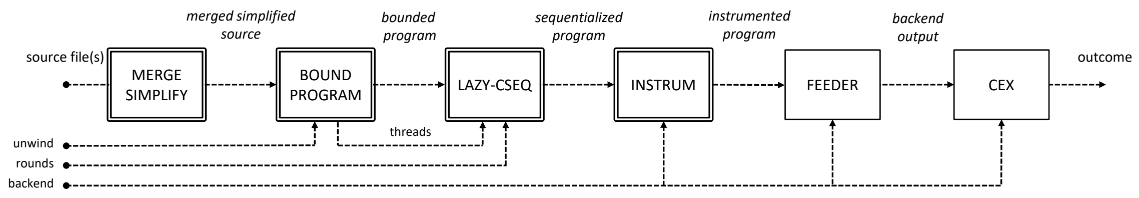


Fig. 10. Configuration sequence of Lazy-CSeq. Double framed boxes denote modules composed of multiple submodules.

Instrumenting the code for a specific backend is in itself a quite simple standalone transformation undertaken by the instrumentation module. It replaces the primitives for handling nondeterminism (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 nondeterminism using `assume`, (3) explicit condition checks using `assert`. The transformation 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.

5.2 The Lazy-CSeq tool

The Lazy-CSeq tool is a CSeq configuration of nineteen modules that can be conceptually grouped into the following categories (see Figure 10 and Inverso et al. [52] for more details):

(1) the source merging module;
(2) eight simple transformation modules that rewrite the input program in steps into a simplified syntax;
(3) six translators for program flattening to produce a bounded program;
(4) two modules implementing the sequentialization algorithm (as a direct translation of the rewrite rules shown in Figure 8) that produce a backend-independent sequentialized file;
(5) standard program instrumentation to instrument the sequentialized file for a specific backend;
(6) two wrappers for backend invocation and user report generation or counterexample translation.

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. Counterexample generation is disabled by default but can be enabled when using the default backend with `--cex`.

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

- bounded model-checkers: `cbmc` [23], `esbmc` [26], `llbmc` [65], `smack` [81];
- abstraction-based tools: `automizer` [46], `cpachecker` [9], `satabs` [24];
- symbolic testing tools: `klee` [15].

Availability and Installation. Lazy-CSeq is available as open source software under BSD license. The version used in the experimental evaluation can be accessed via the CSeq project homepage https://github.com/CSeq/Overview (which also contains pointers to other tools developed within the CSeq framework). More information is available in the `README` file of the installation package.

6 EVALUATION

In this section, we describe the experimental evaluation of our technique. We state our research questions (see Section 6.1) before we describe the experimental setup (see Section 6.2). We then give the detailed results of our evaluation. In particular, we evaluate Lazy-CSeq's ability to find reachable error locations, by comparing its performance with different sequential verification backends to a number of verification tools with built-in concurrency handling. In Section 6.3, we focus on the widely used SV-COMP benchmark set, while we give a more detailed analysis of Lazy-CSeq's performance on several complex benchmarks in Section 6.4.

6.1 Research Questions

With our evaluation we aim to answer the following research questions:

RQ1 How well does our lazy sequentialization technique (as implemented in the Lazy-CSeq tool) perform, compared to other concurrency verification approaches?

RQ2 (a) How is its performance affected by the choice of a specific sequential verification backend? (b) How well do BMC- and CEGAR-based backends, respectively, perform?

RQ3 How well does it scale up to complex verification problems?

We focus on Lazy-CSeq's *performance* here because the software engineering benefits (e.g., quick prototyping) of using the sequentialization approach in general and the CSeq framework in particular can only be evaluated by comparing a number of different sequentialization schemas, which is outside the scope of this paper. More specifically, we focus on Lazy-CSeq's "bug hunting" performance, i.e., its ability to find reachable error locations. This is justified by the fact that our schema is defined for bounded programs only, where the failure to find a reachable error location can be interpreted as a correctness proof.

6.2 Experimental Setup

Benchmarks. We have evaluated Lazy-CSeq primarily over the benchmark set from the concurrency category of the SV-COMP20 software verification competition [8]. The SV-COMP benchmarks are widely used in the literature to evaluate and compare verification systems because they cover the C language well, and because they provide a good mix of easy and hard problems. The concurrency category is further arranged into several sub-categories, which we use to report summaries. The benchmarks cover a wide range of domains, e.g., simple litmus tests for weak memory models, illustrative examples for the `pthread` API, mutual exclusion algorithms, concurrent data structures, or device drivers. Note that we dropped the benchmarks in the ldv-linux-3.14-races sub-category because they contain embedded assembly code and cannot be processed by any of the tools we were using. The remaining set consists of 1075 concurrent C programs, using POSIX threads as concurrency model, with a total size of about 157k lines of code, not counting `#included` library files. We used the 814 unsafe programs that each contain a single reachable error location. For each benchmark, we individually chose the minimal loop unwinding, function inlining, and context switch bounds, respectively, that are required to expose the error.

Our older LR-CSeq and MU-CSeq tools cannot handle the SV-COMP20 benchmarks, due to some of the preprocessing steps introduced for the competition. For the comparison with these tools, we thus use the SV-COMP17 benchmark set, which is slightly smaller.

In order to answer RQ3, we used three additional complex benchmarks from the domain of lock-free concurrent data structures. These are "real world" benchmarks in the sense that they were not specifically designed as concurrency benchmarks, and that the errors were not deliberately injected into the code. We describe these benchmarks in more detail in Section 6.4.

Hardware. For the experiments over the SV-COMP20 benchmarks we used a dedicated machine equipped with 128GB of physical memory and a dual Xeon E5-2687W 8-core CPU clocked at 3.10GHz with hyper-threading disabled, running 64-bit GNU/Linux with kernel 4.9.95; we also report again the results of some of our earlier experiments over the SV-COMP17 benchmarks, which were run on a machine with 16GB of physical memory and a Xeon E5-2670 2.6GHz CPU, running 64-bit GNU/Linux with kernel 2.6.32. We set a 15GB memory limit and a 1000s timeout for each benchmark; however, for the concurrent data structure benchmarks we used more memory and larger individual timeouts, as reported in Section 6.4.

Verification Tools. We first compared the performance of different sequential verification backends over the sequentialized programs generated by Lazy-CSeq. In particular, we used two versions of CBMC [23] (v5.4, with which we achieved the best results, and v5.28, the most recent version),[4] a mature BMC tool for C that we used in conjunction with both the default MiniSat [30] (v2.2.1) and the external Kissat [12] (vSC-2020) SAT solvers; ESBMC [26] (v6.4.0),[5] a similar BMC tool we used in conjunction with the Boolector [74] (v3.2.0) SMT solver; SMACK [81] (v2.4.0),[6] a BMC tool that translates LLVM bitcode into the Boogie intermediate verification language and then uses the Z3-based (v4.8.3) Corral verifier; CPAchecker [9] (CPA-Seq v1.9),[7] a tool for configurable software verification that supports a wide range of techniques, including BMC, predicate abstraction, and shape and value analysis, and relies on the MathSAT5 [21] (v5.5.4) SMT solver; here, we used both the BMC and the default predicate abstraction configurations; Ultimate Automizer [46] (v0.1.25),[8] an automata-based software model checker that is implemented in the Ultimate software analysis framework, and which uses an internal development version of the SMTInterpol [19] SMT solver; and KLEE [15] (v2.2),[9] a dynamic symbolic execution engine that is built on top of the LLVM compiler infrastructure, which we used in conjunction with the STP [39] (v2.3.3) solver. Note that this set of sequential verification backends includes SV-COMP20 medal winners in the Overall (CPAchecker, Ultimate Automizer) and FalsificationOverall (CPAchecker, ESBMC) categories, respectively, and thus represents the state-of-the-art in (sequential) C program verification.

We then compared Lazy-CSeq against verification tools with native concurrency handling. In particular, we (re-) used CBMC, CPAchecker, ESBMC, and SMACK, but relied on their own concurrency handling. CBMC uses a partial order representation of concurrent programs [2]. CPAchecker supports the analysis of concurrent programs with a limited number of threads by using value analysis or BDD-based analysis. ESBMC analyzes the individual interleavings sequentially [25]. SMACK's concurrency handling is based on a combination of Corral [62], which implements the LR-schema, and Whoop [29], which uses a symbolic pairwise lockset analysis to detect all potential data races and

[4]http://www.cprover.org/cbmc/
[5]https://github.com/esbmc/esbmc/releases/tag/v6.4
[6]https://github.com/smackers/smack
[7]https://cpachecker.sosy-lab.org/CPAchecker-1.9-unix.tar.bz2
[8]https://github.com/ultimate-pa/ultimate/releases/tag/v0.1.25
[9]https://klee.github.io/releases

thus to reduce the number of context switch points that Corral needs to simulate. In addition, we also used two other well-performing tools that participated in SV-COMP20, Yogar-CBMC [95, 96], which uses a graph-based CEGAR method that performs abstraction refinement on the scheduling constraint, and Divine 4 [4], an explicit-state model checker, which uses τ-reduction [83], a combination of path reduction and partial order reduction, to handle concurrency. Finally, we also used the three other sequentializations that we implemented with the CSeq framework,[10] LR-CSeq (v0.5, using ESBMC v1.20 with Z3 v3.2 as sequential backend) [34], which implements the LR sequentialization schema; MU-CSeq (v0.4, using CBMC v5.6 with MiniSAT 2.2.1 as sequential backend) [92], which implements an eager sequentialization based on individual memory location unwindings; and UL-CSeq [69], which implements a lazy sequentialization for unbounded programs [70] and uses SeaHorn (v0.1.0) [44] with Z3 (v4.4.0) as backend tool. Similar to the case of the sequential verification backends, these tools represent the state-of-the-art in (concurrent) C program verification.

We passed to each tool the minimal loop unwinding, function inlining, and context switch bounds, respectively, for each individual benchmark, although not all of the tools can take advantage of this information. We explicitly specified a 32-bit size for integers and set the time and memory limits as above, but all other options and control parameters were taken from the respective tool's SV-COMP20 or default configuration.

6.3 SV-COMP Benchmarks

Tables 1 and 2 summarize the results of our experiments over the SV-COMP20 benchmarks. Since all benchmarks contain errors that can be reached with the given unwind bounds, misses indicate an unsound handling of some aspect of the C language by the backend tool. The errors stem from internal errors, such as parsing errors and assertion violations, and form resource limitations other than timeouts, both in the model checker and the SAT/SMT solver. Overall, the large and complex sequentialized programs appear to uncover a small number of corner cases in the sequential backends.

We can see in Table 1 that Lazy-CSeq performs very well if it is combined with the right backends, solving between 805 (using ESBMC) and 812 (using CBMC 5.4) of the 814 benchmarks. CBMC 5.4 as backend produces only two timeouts while CBMC 5.28 also produces two internal errors. ESBMC produces three misses, internal errors, and timeouts, respectively. However, with other backends, Lazy-CSeq's performance is not as good. With the other two BMC backends it still perform reasonably well, with the BMC version of CPAchecker performing almost on par with CBMC and ESBMC (solving 794 benchmarks with one internal error and 19 timeouts) and outperforming SMACK, which solves only 773 benchmarks and produces 32 misses, one internal error, and eight timeouts. This is in contrast to its performance with non-BMC backends. The default CPAchecker configuration in particular struggles with the complex sequentialized programs and times out on 329 benchmarks, solving only 483; this is unusual, since the default configuration typically outperforms the BMC configuration. Both UAutomizer and KLEE perform better than CPAchecker, solving 627 resp. 592 benchmarks, with 18 internal errors each.

A comparison with the results in Table 2 shows that Lazy-CSeq performs better than even the best tools with built-in concurrency handling. CBMC and ESBMC solve between 793 and 800 benchmarks, with several misses and internal errors (CBMC 5.4 twelve misses, two errors, CBMC 5.28 sixteen errors, and ESBMC three misses and five errors). Moreover, Lazy-CSeq is typically faster (up to 5x) than these tools, even though it incurs a noticeable overhead (typically less than 5 sec., depending on the size of the benchmark, although the *pthread-driver-races* benchmarks that require very large loop unwinding bounds incur larger overheads) for the different translation stages; this particularly

[10]https://github.com/CSeq/Overview

Table 1. Lazy-CSeq results on SV-COMP20 concurrency benchmarks with reachable error label. Each block shows the results for a different backend tool, each row corresponds to a sub-category. Tool details are as above; CBMC 5.28 (K) denotes a version using Kissat instead of the default Minisat solver, CPAchecker 1.9 (B) denotes the BMC configuration of CPAchecker. We report the number of files and the total number of lines of code. *pass* denotes the number of correctly analyzed benchmarks (i.e., error locations found), *miss*, *error* and *t.o.* the number of benchmarks where the tool missed the error location (i.e., proclaimed the benchmark to be safe), returned an unknown result (e.g., unsupported feature or internal error), or exceeded the time limit, respectively, and *time* is the average CPU time in seconds.

			Lazy-CSeq + CBMC 5.4					Lazy-CSeq + CBMC 5.28					Lazy-CSeq + CBMC 5.28 (K)				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	8	-	-	-	11.7	8	-	-	-	11.0	8	-	-	-	10.9
pthread	20	1906	20	-	-	-	40.2	20	-	-	-	29.0	20	-	-	-	12.5
pthread-atomic	2	182	2	-	-	-	4.7	1	-	1	-	4.1	1	-	1	-	4.2
pthread-c-dac	1	1347	1	-	-	-	11.7	-	-	1	-	10.4	-	-	1	-	9.9
pthread-complex	4	663	2	-	-	2	553.8	2	-	-	2	516.6	2	-	-	2	568.0
pthread-divine	7	440	7	-	-	-	30.2	7	-	-	-	14.6	7	-	-	-	8.9
pthread-driver-races	4	1216	4	-	-	-	121.1	4	-	-	-	117.8	4	-	-	-	117.2
pthread-ext	8	253	8	-	-	-	10.2	8	-	-	-	10.2	8	-	-	-	4.4
pthread-lit	3	111	3	-	-	-	5.1	3	-	-	-	4.7	3	-	-	-	4.9
pthread-nondet	3	83	3	-	-	-	6.3	3	-	-	-	5.3	3	-	-	-	4.7
pthread-wmm	754	150270	754	-	-	-	5.5	754	-	-	-	5.2	754	-	-	-	5.2
Totals	814	157602	812	-	-	2	10.0	810	-	2	2	9.1	810	-	2	2	8.8

			Lazy-CSeq + ESBMC 6.4					Lazy-CSeq + SMACK 2.4.0					Lazy-CSeq+CPAchecker 1.9 (B)				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	8	-	-	-	17.2	8	-	-	-	14.3	5	-	-	3	583.6
pthread	20	1906	18	-	1	1	58.0	6	10	1	3	202.7	14	-	-	6	414.4
pthread-atomic	2	182	2	-	-	-	5.2	-	2	-	-	21.8	2	-	-	-	195.1
pthread-c-dac	1	1347	-	-	-	1	1000.2	-	-	-	1	1000.2	-	-	-	1	1000.2
pthread-complex	4	663	2	-	1	1	463.9	1	-	-	3	763.3	-	-	-	4	1000.2
pthread-divine	7	440	6	-	1	-	32.4	4	3	-	-	23.9	5	-	1	1	200.5
pthread-driver-races	4	1216	1	3	-	-	122.2	-	4	-	-	348.7	2	-	-	2	712.6
pthread-ext	8	253	8	-	-	-	4.4	-	8	-	-	41.7	8	-	-	-	10.9
pthread-lit	3	111	3	-	-	-	12.0	-	2	-	1	340.9	2	-	-	1	339.2
pthread-nondet	3	83	3	-	-	-	11.5	-	3	-	-	253.4	2	-	-	1	636.0
pthread-wmm	754	150270	754	-	-	-	5.8	754	-	-	-	9.1	754	-	-	-	71.5
Totals	814	157602	805	3	3	3	11.5	773	32	1	8	23.1	794	-	1	19	96.6

			Lazy-CSeq + CPAchecker 1.9					Lazy-CSeq + UAutomizer					Lazy-CSeq + KLEE 2.2				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	5	-	-	3	566.4	8	-	-	-	385.8	3	-	5	-	1.5
pthread	20	1906	10	-	-	10	554.6	5	-	8	7	403.2	10	-	3	7	367.2
pthread-atomic	2	182	-	-	-	2	1000.1	1	-	1	-	453.2	1	-	-	1	527.2
pthread-c-dac	1	1347	-	-	-	1	1000.2	-	-	-	1	1000.2	-	-	1	-	1.7
pthread-complex	4	663	-	-	-	4	1000.3	-	-	1	3	754.8	-	-	-	4	1000.1
pthread-divine	7	440	3	-	2	2	297.5	3	-	1	3	526.1	2	-	3	2	287.2
pthread-driver-races	4	1216	1	-	-	3	789.6	-	-	-	4	1001.5	-	-	4	-	0.7
pthread-ext	8	253	8	-	-	-	21.3	4	-	4	-	76.3	7	-	1	-	3.6
pthread-lit	3	111	2	-	-	1	341.3	2	-	-	1	355.2	2	-	-	1	334.8
pthread-nondet	3	83	1	-	-	2	902.1	-	-	2	1	500.7	-	-	1	2	667.2
pthread-wmm	754	150270	453	-	-	301	612.4	604	-	1	149	537.6	567	-	-	187	375.6
Totals	814	157602	483	-	2	329	602.0	627	-	18	169	527.3	592	-	18	204	366.8

pronounced for the large but relatively simple *pthread-wmm* benchmarks. However, the overheads could be reduced with an optimized implementation.

The results in Table 2 also emphasize the difficulty of building a complete, correct, and efficient concurrency handling into a verification tool. SMACK, CPAchecker and especially Divine miss the error locations in a large number of benchmarks or produce errors, and SMACK and CPAchecker time out on a large number of the *pthread-wmm* benchmarks. Note in particular that we were unable to get Yogar-CBMC to work properly outside the SV-COMP benchmarking environment, resulting in 779 failures.

Lazy-CSeq also outperforms the other sequentializations we implemented within the CSeq framework. UL-CSeq performs surprisingly well, considering that it is not based on a bounded analysis. It finds the error location in 787 benchmarks, and despite the nine internal errors outperforms CPAchecker, the only other tool that can provide proofs rather than only counterexamples, confirming the observation from our earlier work [69].

Table 2. Results on SV-COMP20 benchmarks with reachable error label (other concurrency verification tools). See Table 1 for explanations.

			CBMC 5.4					CBMC 5.28					CBMC 5.28 (K)				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	3	5	-	-	0.8	8	-	-	-	0.4	8	-	-	-	0.6
pthread	20	1906	17	-	2	1	142.4	8	-	11	1	159.6	8	-	11	1	103.5
pthread-atomic	2	182	2	-	-	-	0.7	2	-	-	-	0.8	2	-	-	-	0.7
pthread-c-dac	1	1347	1	-	-	-	6.4	-	-	1	-	0.1	-	-	1	-	0.1
pthread-complex	4	663	-	1	-	3	818.8	-	-	3	1	250.1	-	-	3	1	250.2
pthread-divine	7	440	1	5	-	1	144.6	6	-	1	-	0.7	6	-	1	-	1.4
pthread-driver-races	4	1216	3	1	-	-	1.2	4	-	-	-	16.2	4	-	-	-	18.8
pthread-ext	8	253	7	-	-	1	276.6	7	-	-	1	275.0	8	-	-	-	18.4
pthread-lit	3	111	2	-	-	1	333.6	2	-	-	1	333.4	2	-	-	1	333.5
pthread-nondet	3	83	3	-	-	-	172.9	3	-	-	-	204.8	3	-	-	-	281.4
pthread-wmm	754	150270	754	-	-	-	0.6	754	-	-	-	0.3	754	-	-	-	0.6
Totals	814	157602	793	12	2	7	19.9	794	-	16	4	10.2	795	-	16	3	6.9
			ESBMC 6.4					SMACK 2.4.0					Yogar-CBMC				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	8	-	-	-	0.3	7	1	-	-	35.6	4	-	4	-	2.1
pthread	20	1906	12	3	2	3	230.5	8	6	-	6	382.2	13	-	7	-	52.1
pthread-atomic	2	182	2	-	-	-	48.7	2	-	-	-	48.1	1	-	1	-	1.5
pthread-c-dac	1	1347	-	-	-	1	1004.5	-	1	-	-	3.6	1	-	-	-	3.5
pthread-complex	4	663	1	-	2	1	606.9	1	2	-	1	261.3	-	-	4	-	5.1
pthread-divine	7	440	6	-	1	-	101.3	4	2	-	1	150.9	1	-	6	-	2.8
pthread-driver-races	4	1216	4	-	-	-	4.3	-	4	-	-	22.8	4	-	-	-	0.8
pthread-ext	8	253	8	-	-	-	0.1	1	7	-	-	4.9	5	-	3	-	0.4
pthread-lit	3	111	2	-	-	1	333.5	1	1	-	1	338.8	3	-	-	-	26.0
pthread-nondet	3	83	3	-	-	-	0.2	1	1	-	1	337.0	3	-	-	-	4.5
pthread-wmm	754	150270	754	-	-	-	42.3	254	8	-	492	795.2	-	-	754	-	0.7
Totals	814	157602	800	3	5	6	52.8	279	33	-	502	746.2	35	-	779	-	4.9
			CPAchecker 1.9 (CPA-Seq)					Divine 4.4.0					UL-CSeq				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	3	-	5	-	4.3	5	3	-	-	2.9	8	-	-	-	146.1
pthread	20	1906	9	-	11	-	121.8	11	1	8	-	139.9	10	-	1	9	487.2
pthread-atomic	2	182	2	-	-	-	20.0	1	-	1	-	48.2	2	-	-	-	39.0
pthread-c-dac	1	1347	-	-	1	-	915.6	-	-	-	1	1000.2	-	-	-	1	1000.0
pthread-complex	4	663	-	-	4	-	332.8	1	-	1	2	502.1	-	-	-	4	1000.0
pthread-divine	7	440	2	-	5	-	7.2	4	-	3	-	3.4	3	-	4	-	14.6
pthread-driver-races	4	1216	-	-	4	-	112.0	-	-	3	1	251.5	-	-	4	-	0.5
pthread-ext	8	253	-	-	8	-	4.0	2	-	5	1	127.5	8	-	-	-	28.7
pthread-lit	3	111	1	-	2	-	3.6	1	-	-	1	335.2	2	-	-	1	334.8
pthread-nondet	3	83	-	-	3	-	3.5	-	1	-	-	2.4	-	3	-	-	340.5
pthread-wmm	754	150270	626	-	46	82	173.1	548	3	-	4	41.6	754	-	-	-	35.7
Totals	814	157602	643	-	89	82	168.6	573	210	21	10	49.3	787	3	9	15	55.3

Table 3. Results on SV-COMP17 benchmarks with reachable error label. See Table 1 for explanations.

			Lazy-CSeq + CBMC 5.4					LR-CSeq 0.5 + ESBMC 1.20					MU-CSeq 0.4 + CBMC 5.6				
sub-category	files	l.o.c.	pass	miss	unkn.	t.o.	time	pass	miss	unkn.	t.o.	time	pass	miss	unkn.	t.o.	time
ldv-races	8	662	8	-	-	-	2.4	-	-	8	-	1.9	6	1	1	-	46.2
pthread	17	1736	17	-	-	-	25.6	6	-	8	3	208.6	17	-	-	-	8.3
pthread-c-dac	1	63	1	-	-	-	2.0	-	-	1	-	1.8	-	1	-	-	1.7
pthread-atomic	2	180	2	-	-	-	2.2	2	-	-	-	5.6	2	-	-	-	3.5
pthread-driver-races	4	1212	4	-	-	-	4.5	-	-	4	-	2.1	-	-	4	-	2.1
pthread-ext	8	253	8	-	-	-	7.5	2	-	6	-	2.9	8	-	-	-	5.3
pthread-lit	3	105	3	-	-	-	1.9	2	-	-	1	667.4	2	1	-	-	1.3
pthread-wmm	754	150273	754	-	-	-	2.0	754	-	-	-	11.6	754	-	-	-	2.2
Totals	797	154484	797	-	-	-	2.5	766	-	27	4	18.0	789	3	5	-	2.8

Both LR-CSeq and MU-CSeq are no longer actively maintained and cannot handle the SV-COMP20 benchmarks, due to some of the recently introduced preprocessing steps. In Table 3 we therefore report the results over the earlier and slightly smaller SV-COMP17 benchmark set. Lazy-CSeq finds the error location in all 797 benchmarks. LR-CSeq does not support the constructs used in 25 of these benchmarks and produces time-out or memory-out in six cases. However, it also requires much longer in the successful cases than Lazy-CSeq (see for example its 11.6sec average analysis time for the *pthread-wmm* benchmarks, compared to 2.0sec for Lazy-CSeq). This reflects the much bigger search spaces introduced by the LR schema. MU-CSeq produces three misses and five errors, but it never times out and its runtimes are comparable to those of Lazy-CSeq.

Given these results, we can formulate a strong answer to our first research question:

RQ1 Lazy-CSeq outperforms all other tools used in our evaluation, including those implementing other sequentialization schemas.

Table 1 also reveals some differences between the different backends we used for Lazy-CSeq. It performs well with all BMC-based backends: while it achieves the best results (812 successes, two timeouts) with CBMC 5.4, its performance with CBMC 5.28, ESBMC, and even the BMC configuration of CPAchecker is not far behind (810, 805, and 794 successes, respectively). Using CBMC generally produces results slightly faster than using ESBMC (9.2 sec. vs. 11.5 sec.), while the BMC configuration of CPAchecker is substantially slower (96.6 sec.). Using SMACK as backend leads to 40 failures (32 misses, one error and 8 timeouts), and substantially longer runtimes across all benchmark categories, even in the successful cases.

Surprisingly, each "sequential" BMC tool outperforms its "concurrent" version, i.e., each tool finds more bugs in the sequentialized program versions than in original concurrent ones, with comparable or better runtimes. This is particularly pronounced if we disregard the 754 benchmarks from the *pthread-wmm* sub-category: for CBMC 5.4, we get 58 vs. 39 successes, for CBMC 5.28 56 vs. 40 resp. 41, depending on the SAT solver, and for ESBMC still 51 vs. 46. This indicates that the control code injected by our lazy sequentialization schema does indeed incur only minimal overheads, and that the schema by itself is an efficient representation of the nondeterminism inherent to concurrency.

The situation is less clear for the CEGAR-based backends. Both CPAchecker and Ultimate Automizer perform substantially worse as sequential backends than any of the BMC-based tools, finding only 483 and 627, respectively, of the bugs. Moreover, CPAchecker performs much worse on the sequentialized benchmark versions than on the original concurrent ones. The majority of the failures (301 out of 331 errors and timeouts) are in the *pthread-wmm* sub-category. These benchmarks have a very simple structure, i.e., they comprise a small number of threads which each consist of a sequence of conditional assignments. On the sequentialized versions, CPAchecker typically fails with a refinement error; more precisely, it finds spurious counterexamples that it cannot remove from the abstract reachability trees. We conjecture that this is a consequence of the structure of the benchmarks in this category: since almost all assignments in the original versions involve shared variables, the sequentialized versions have complex control flow graphs where each control flow predicate depends on a nondeterministic variable (i.e., the guessed context switch point). Hence, the refinement fails. If we disregard this sub-category, CPAchecker solves 30 out of 60 benchmarks, compared to the 17 it solves on the concurrent versions.

RQ2 (a) The choice of a specific sequential verification backend can affect Lazy-CSeq's performance considerably. (b) BMC-based backends tend to perform uniformly well, to the level of outperforming the built-in concurrency handling of the BMC tool. Backends based on CEGAR or symbolic execution techniques tend to perform worse than BMC-based backends; their performance varies much more with the benchmark category.

6.4 Concurrent Data Structure Benchmarks

Table 1 shows that Lazy-CSeq times out on two benchmarks from the *pthread-complex* sub-category. These are derived from concurrent data structure implementations. Such data structures often induce very hard verification problems, and

Table 4. Results of concurrent data structure benchmarks. The entries report runtime and maximal memory consumption (as measured by the `time` command) for each system; *t.o.* denotes a time-out after 24 hours, *m.o.* denotes that the system ran out of the allocated 32GB memory. Failures are shown in *cursive*; † denotes an internal CBMC error "pointer handling for concurrency is unsound".

	Lazy-CSeq +							
	CBMC 5.4		CBMC 5.28		CBMC 5.28 (K)		ESBMC 6.4	
safestack	3h:19m:21s	2.63GB	9h:02m:54s	5.62GB	7h:44m:08s	0.94GB	*t.o.*	5.38GB
elimstack	6h:55m:10s	2.24GB	4h:20m:47s	1.87GB	29m:06s	1.05GB	10h:31m:04s	5.98GB
DCAS	2h:51m:31s	4.67GB	2m:26s	0.84GB	34m:08s	0.65GB	2h:22m:38s	2.17GB
	CBMC 5.4		CBMC 5.28		CBMC 5.28 (K)		ESBMC 6.4	
safestack	*17h:18m:35s*	*m.o.*	*8h:37m:40s*	*m.o.*	*8h:36m:38s*	*m.o.*	*4h:05m:04s*	*m.o.*
elimstack	2h:22m:09s	20.46GB	*<1s*†	-	*<1s*†	-	13h:12m:55s	11.62GB
DCAS	3h:26m:34s	17.63GB	*<1s*†	-	*<1s*†	-	*2m:51s*	*m.o.*

we investigate here whether Lazy-CSeq scales to such problems. Note that these implementations were not specifically designed as concurrency benchmarks, and that the errors were not deliberately injected into the code.

For the experiments described in this section we used the sequential backends that performed best over the SV-COMP20 benchmarks, i.e., CBMC 5.4, CBMC 5.28 (with both Minisat and Kissat as SAT solvers), and ESBMC 6.4. We also compare Lazy-CSeq's performance with each backend against the performance of the respective backend's own built-in concurrency handling, using the same loop unwinding and round bounds as for Lazy-CSeq. Each benchmark is run on the same machine as the SV-COMP20 benchmarks, but with a 32GB memory limit and a 24 hour timeout. Since the errors in the data structures are independent of the payload data type, we used a 32-bit integer representation; runtimes and memory consumption are likely to increase substantially for a 64-bit representation, without changing the structure of the verification problem. Table 4 summarizes the results of these experiments; we discuss details in the following. Note that the runtimes are dominated by the SAT/SMT solver; the sequentialization and the backend's own parsing process take only between 5 and 30 seconds, depending on the benchmark.

safestack. This is a real-world implemention of a lock-free stack designed for weak memory models. It was posted to the CHESS forum by Dmitry Vyukov.[11] It is a unique benchmark in the sense that it contains a very rare bug that requires at least three threads and five context switches to be exposed when running under the SC semantics. In the verification literature, it was shown that real-world bugs typically require at most three context switches to manifest themselves [86]. The *safestack* benchmark, for this reason, presents a non-trivial challenge for concurrency testing and symbolic tools. For this benchmark, we used 4 rounds of computation and a loop unwinding bound of 3 to expose the bug; for Lazy-CSeq, we also optimized the inlining of functions with parameters that are passed by value (`--simplify-args`).

With CBMC 5.4 as backend, Lazy-CSeq requires 3h:19m:21s (of which about three hours is spent in the Minisat solver) and 2.69 GB memory to find the bug. Surprisingly, it becomes much slower with the most recent version CBMC 5.28 as backend. Here, Kissat outperforms Minisat, in particular in terms of memory consumption. With ESBMC as backend, Lazy-CSeq is unable to find the bug, and times out after 24 hours with a memory peak of 5.38GB.

Lazy-CSeq is the only tool we are aware of that can automatically find the concurrency bug in *safestack*. On the concurrent original version, the two CBMC versions run out of memory after more than 17 resp. 8 hours, before even starting the SAT solver, while ESBMC runs out of memory after about 4 hours.

[11] https://social.msdn.microsoft.com/Forums/

elimstack. This is a C implementation [13] of the elimination stack by Hendler et al. [47] that follows the original pseudocode presentation. It augments Treiber's stack with a "collision array", used when an optimistic push or pop detects a conflicting operation; the collision array pairs together concurrent push and pop operations to "eliminate" them without affecting the underlying data structure. This implementation is incorrect if memory is freed in pop operations. In particular, if memory is freed only during the elimination phase, then exhibiting a violation (an instance of the infamous ABA problem) requires a seven thread client where three push operations are concurrently executed with four pops. To witness the violation, the implementation is annotated with several assertions that manipulate counters as described by Chebaro et al. [17]. For this benchmark, we used 2 rounds of computation and a loop unwinding bound of 1 to expose the bug.

Lazy-CSeq can again automatically find the bug. Here, CBMC 5.28 improves over CBMC 5.4, in particular, when we switch from Minisat to Kissat—this gives us almost an order magnitude of speed-up and requires only 29m:06 with a memory peak of 1.05 GB. ESBMC can also find the bug in the sequentialized program but requires more time and memory than either of the CBMC versions.

In this benchmark, CBMC 5.4 finds the bug faster (2h:22m:09s) on its own than as Lazy-CSeq backend (6h:55m:10s) but requires substantially more memory (20.46GB vs. 2.24GB). CBMC 5.28, in contrast, rejects the input because it cannot guarantee soundness of its pointer handling in this case. ESBMC also finds the bug, but requires substantially more time (12h:31m:43s) and memory (12.11GB) than as Lazy-CSeq backend. Note that both tools on their own are still substantially slower than Lazy-CSeq with CBMC 5.28 and Kissat; this shows the software engineering benefits of sequentialization as verification approach: we get the performance gains from the ongoing CBMC development for free.

DCAS. This benchmark implements a non-blocking algorithm for two-sided queues presented by Agesen et al. [1]. The algorithm has a subtle bug that was discovered in an attempt to prove its correctness with the help of the PVS theorem prover. The discovery of the bug took several months of human effort. Although the bug has been automatically discovered using the model checker SPIN (see [49] and http://spinroot.com/dcas/), a generalized version of the benchmark remains a challenge for explicit exploration approach. In fact, after 138h of CPU-time (using 1000 cores), and an exploration of 10^{11} states the error was still undetected [50]. We have translated this benchmark from Promela to C using the Pthread library; we also consider a more complex version that has 10 threads while the version of Holzmann [50] only considers 8 threads. In order to expose the bug, we use three rounds of computation, coupled with a mixed loop unwinding schema: all loops with a statically determined bound are completely unwound, while all other loops were unwound only for a single round.

Lazy-CSeq can detect the bug in this more complex variant using all four backends, but the best results are achieved with CBMC 5.28. Here, Minisat outperforms Kissat by an order of magnitude, and detects the bug in 2m:26s, using less than 1GB of memory. Both CBMC 5.4 and ESBMC require more than two hours.

On the original version, only CBMC 5.4 can find the bug, succeeding after 3h:26m:34s, but using 17.63GB of memory. CBMC 5.28 rejects the program with the same warning about pointer handling as in the case of *elimstack*, and ESBMC runs out of memory after 2m:51s.

Conclusions. Given the results shown above, we can formulate a strong answer to our final research question.

RQ3 Lazy-CSeq scales to complex real-world concurrency verification problems. Moreover, it is the only tool that we are aware of that solves all three problems we have investigated, and it substantially outperforms both CBMC's and ESBMC's built-in concurrency handling. However, the different sequential backends perform less uniformly over these complex problems than over the simpler SV-COMP20 benchmarks and no backend is dominant.

6.5 Threats to Validity

Since our evaluation does not involve human subjects, many of the common threats to validity do not apply. Nevertheless, some remain, and in this section we discuss the threats to internal and external validity that we have identified, and the mitigation measures we have put into place.

Threats to Internal Validity. We identified errors in the measurement process, in the benchmarks, in the tools, and in the tool usage as the main threats to the internal validity of our conclusions.

We used standard Linux tools to measure and control the resource consumption of the different verification tools. These are known to introduce measurement errors if multiple processes share a computational node, but we mitigated against this threat by running the individual verification attempt on an otherwise idle processor. We report the runtimes provided by the individual verification tools, which could in principle introduce some bias. However, these tools have been used widely and over a long time, and no such bias has been reported in the literature.

The C programming language has a complex and sometimes unclear semantics; the benchmarks may contain errors, and their predicted outcome (i.e., the *oracle*) may thus be wrong. We partially mitigated against this threat by using the SV-COMP benchmark set, an established and community-curated benchmark set that has been used in numerous studies and competitions. The errors in the complex benchmarks (see Section 6.4) have been identified and published in other studies.

We mitigated against the threat introduced by systemic tool errors, which could invalidate their results, by using mature tools that have successfully participated in the SV-COMP, and have thus been thoroughly examined. Lazy-CSeq is also available as open-source software and can thus be inspected by third parties. We did not check the counterexample witnesses returned by the tools (if any), because there is no comprehensive and stable checker for concurrent counterexamples.

The final threat we identified is wrong tool usage, in particular, sub-optimal parameter settings that could negatively affect a tool's performance. We mitigated against this threat by using the minimal values for the loop unwinding and context switch bounds for each benchmark (which we know from exhaustive evaluation of different tools) for all context bounded tools that can take advantage of this, and the settings provided by tool developers for SV-COMP20 in all other cases. We also consulted with the tool developers and incorporated any feedback that they provided.

Threats to External Validity. The only threat to external validity that we identified is *selection bias*, both for the benchmarks and the tools used in our evaluation.

In particular, the SV-COMP20 benchmark set contains a large number of small and relatively simple problems (especially in the *pthread-wmm* sub-category), and only very few truly hard problems, and might thus not be representative to allow a generalization of our claims. However, the SV-COMP20 benchmark set provides a good coverage of the different concurrency constructs, and it is widely used in other studies. We further mitigate against these threats by using the full set of benchmarks that contain reachable error locations, rather than sub-sampling. We also report results for each sub-category, to prevent in particular the large *pthread-wmm* sub-category to overshadow the other sub-categories. We use additional hard benchmarks from the literature to complement the SV-COMP20 benchmark set.

Furthermore, we selected only *unsafe* benchmarks with reachable error locations. This is in line with our focus on evaluating Lazy-Cseq's bug-finding capabilities, but the results reported in Sections 6.3 and 6.4 may not generalize to *safe* benchmarks without reachable error locations. However, results from preliminary experiments over the safe *pthread-wmm* benchmarks mitigate against this threat: under a round bound of 2, it took Lazy-CSeq on average 4.3 seconds to claim the benchmarks as safe when using CBMC 5.28 as backend, while it took 80.0 seconds when using CPAchecker 1.9.

We addressed a possible tool selection bias by including the publicly available top-performing tools from the SV-COMP competition, which represent the state-of-the-art in software verification (at least for the C programming language). We used tools representing different technologies both as sequential verification backends for Lazy-CSeq and for comparison. However, all tools we use in our evaluation have been trained on (subsets of) the SV-COMP benchmark set, which may have lead to overfitting of the tools and may introduce a residual threat to validity.

7 RELATED WORK

7.1 Sequentialization

Sequentialization was originally developed by Qadeer and Wu [79] by simulating the context switches as procedure calls and returns, which allowed reusing without any changes verification tools that were originally developed for sequential programs. The sequentialized program only simulates a subset of all executions of the original concurrent program, so the approach is sound only for programs with two threads making only two context switches. Further work lifted these limitations, following two different routes.

Eager sequentialization. Eager sequentialization schemas start with a nondeterministic guess of the shared memory contents that are visible to the different threads at the different access times, and then simulate all thread schedules that are compatible with this guess.

The first (but still widely used) eager schema for an arbitrary but bounded number of context switches was given by Lal and Reps [63] (LR). Its basic idea is to simulate in the sequential program all round-robin schedules of the threads in the concurrent program, in such a way that (1) each thread is run to completion, and (2) each simulated round works on its own copy of the shared global memory. The initial values of all memory copies are guessed eagerly in the beginning, while the context switch points are guessed during the simulation of each thread. At the end a checker prunes away all infeasible runs where the initial values guessed for one round do not match the values computed at the end of the previous round. This requires a second set of memory copies. LR thus uses a large number of extra variables; the number of assignments involved in handling these variables, the high degree of (data) nondeterminism, and the late pruning of infeasible runs can all cause performance problems for the backend tool. Moreover, due to the eager exploration, LR cannot rely on error checks built into the backend and also requires specific techniques to handle programs with heap-allocated memory [61].

Extensions of the LR schema allow more liberal scheduling policies than LR's round-robin scheduling [31], modelling of unbounded, dynamic thread creation [14, 31], and dynamically linked data structures allocated on the heap [3].

The LR schema has been implemented in LR-CSeq for C programs with bounded thread creation [34, 35], and in STORM that also handles dynamic memory allocation [61]. Poirot [31, 77] and Corral [62] are successors of STORM. Rek implements a variant of LR that targets real-time systems [16].

Another eager sequentialization schema is based on the concept of *memory unwinding* (MU), an explicit representation of the program's write operations as a sequence that contains for each write the writing thread, the variable or lock, and the written value. For the analysis, the MU is guessed and each thread is translated into a simulation function where write and read accesses over the shared memory are replaced by operations over the unwound memory. The simulation functions are executed sequentially; all context switches are implicitly simulated through the MU [89]. This is implemented in the MU-CSeq tool [87, 88].

The most recent version of MU-CSeq [92] implements a MU variant called IMU that uses a separate MU for each individual shared memory location corresponding to a scalar type or a pointer. This requires a timestamp (i.e., a distinct natural number) with each write in each individual MU to recreate a global total order over the shared memory writes. 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). IMU gives a simple and effective way to support dynamic memory allocation and pointer arithmetics.

Lazy sequentialization. Lazy sequentialization schemas only guess the context switch points, but compute the memory contents precisely, and thus explore only reachable states, which means that they can be more efficient than eager schemas. The first lazy sequentialization schema was given by La Torre, Madhusudan, and Parlato [58] (LMP). However, since the local state of a thread is not stored on context switches, the values of the thread-local variables must be recomputed from scratch when a thread is resumed. This recomputation poses no problem for tools that compute function summaries [58, 59] since they can re-use the summaries from previous rounds. Consequently the LMP schema was empirically shown to be more effective than the LR schema in analyzing multi-threaded Boolean programs [57, 58]. However, the recomputation is a serious drawback for applying LMP in connection with BMC because it can lead to exponentially growing formula sizes [42].

Our schema is carefully designed to address this issue, and aggressively exploits the structure of bounded programs. It avoids the recomputation of the local memory by turning the thread-local variables into `static` variables, so that their values are persisted between the different invocations of the thread simulation functions. It repositions the program counter after a context switch using a sequence of jumps, which leads to more compact control-flow graphs than a large multiplexer at the start of the thread simulation functions. Its only source of nondeterminism is the guess of the context switch point in the main driver, so that it produces compact formulas and is very effective in practice.

Herdt et al. [48] have described a small modification of our schema that prunes empty rounds. This has a small positive effect on most unsafe benchmarks, but can substantially speed up the full search space exploration of safe programs. We also combined our translation with an abstract interpretation in order to reduce the number of bits required to represent the variables of the sequentialized program, and thus to improve scalability even further [71, 72].

7.2 Shared memory concurrency handling in SAT-based program verification

Biere et al. [11] introduced BMC to capitalize on the rapidly improving performance of modern SAT/SMT solvers (see Biere [10] for a survey on BMC), but many abstraction- and automata-based approaches also rely on SAT-based methods. Shared memory concurrency is handled either by explicitly exploring all possible interleavings (typically up to a given bound on the number of context switches) or by using a symbolic representation of the interleavings or the effects of shared memory writes by the different threads.

Explicit interleaving exploration. Cimatti et al. [20, 22] describe a predicate abstraction approach to verify SystemC that combines the explicit exploration of the different possible interleavings with symbolic model checking (i.e., the symbolic representation and updates of the state). This approach encodes the semantics of the non-preempting SystemC scheduler. Cordeiro and Fischer [25] describe a similar approach in the context of a BMC-based model checker for

C, which allows pre-emptions at any visible instruction. They also exploit the SMT techniques on large problems by encoding all possible interleavings into a single formula rather than starting the solver for each individual interleaving.

Symbolic and concolic (i.e., mixed concrete-symbolic) execution techniques explore individual program paths, and rely on a SAT/SMT solver to prune infeasible paths and interleavings. Farzan et al. [32] extend concolic testing to concurrent programs. They use the notion of thread interference scenario, which is a representation of a set of bounded interferences [82] among the threads, which define the scheduling constraints for a concurrent program run. These interference scenarios are then explored in a systematic way by generating a schedule and input vectors that conform with the scenario. However, this approach requires modifying the core of a concolic testing tool in order to handle concurrent programs and thus it cannot flexibly leverage the increasing power of existing sequential checkers. Guo et al. [43] describe an incremental symbolic execution engine for concurrent software. However, this relies on the existence of two program versions, a base line and an update, and uses a multi-threaded change impact analysis to minimize the effort. It remains unclear how well this performs for a full analysis of the base line.

All of these explicit methods face scalability problems, even under context bounds, as the number of possible interleavings grows exponentially with the number of threads and statements.

Symbolic interleaving representation. Rabinovitz and Grumberg [80] describe an approach where program executions are encoded as partial orders; more specifically, each thread is modelled an SSA program and operations on the shared memory are constrained by a global conjunct modeling the memory model. Their implementation exploits CBMC to handle shared memory, but works directly on the SSA program and can handle only two threads. Several other approaches [2, 38, 84, 85] follow similar ideas. In particular, CBMC's native concurrency handling is also based on partial orders [2].

7.3 Correctness

Here, we have concentrated on reachability problems; we have sketched an extension to detect deadlocks elsewhere [51] and currently working on detecting data races.

However, sequentialization has also been used to prove correctness of programs. Garg and Madhusadan [41] describe an approach that is closely related to rely-guarantee proofs and is aimed to avoid the cross-product of the thread-local states. Only the valuation of some local variables of the other threads (forming the abstraction for the assume-guarantee relation) is retained when simulating a thread. For this, frequent recomputations of the thread-local states are required (in particular, whenever a context switch needs to be simulated in the construction of the rely-guarantee relations) which introduces control nondeterminism and recursive function calls even if the original program does not contain any recursive calls. Moreover, the resulting sequentialization yields an over-approximation of the original program and thus cannot be used for bug-finding. The LMP schema has also been extended to parameterized programs [59, 60] and then used to prove correctness of (over-) abstractions of several Linux device drivers, but this can again not be used for bug-finding.

We have also given a lazy schema that does not unwind loops and thus allows to analyze unbounded computations, even with an unbounded number of context switches [70]. Its main technical novelty is the simulation of the thread resumption in a way that does not use gotos and thus does not require that each statement is executed at most once. We have implemented this translation in the UL-CSeq tool [69] and shown that (in connection with an appropriate sequential backend verification tools) it performs well for proving the correctness of safe benchmarks, but still remains competitive with state-of-the-art approaches for finding bugs in unsafe benchmarks.

7.4 Weak memory models

In this paper, we assume the *sequential consistency* (SC) memory model [64]. However, our approach is not tied to this memory model, and can be extended to handle weak memory models (WMMs) by means of *shared memory abstractions* (SMAs) [90, 91]. More specifically, all accesses to shared memory items (i.e., reads from and writes to shared memory locations, and synchronization primitives like lock and unlock) are replaced by explicit calls to the SMA's API. The SMA can thus be seen as an abstract data type that encapsulates the semantics of the underlying WMM and implements it under the simpler SC model, which isolates the WMM from the remaining concurrency aspects. SMAs can be defined both for eager [90] and for lazy [91] schemas, but can also be used for other concurrency analysis techniques.

8 ONGOING AND FUTURE WORK

Our lazy sequentialization schema and its implementation in Lazy-CSeq have shown to be an efficient tool for the analysis of concurrent C programs. Nevertheless, we see several avenues for future work; some of these we already pursue.

Deadlock and data race detection. In this paper we have focused on reachability problems; we are currently extending our approach by adding typical concurrency checks such as deadlocks and data races. We believe that deadlock checks can be encoded efficiently in the verification stubs, similar to the way ESBMC handles deadlock checks. For data races, we are currently investigating an optimized encoding that relies on an extended SAT solver API to directly set propositional variables; however, these API calls can still be generated as part of the source-to-source translation.

Weak memory models. Complex WMMs such as the Power model allow the processor to re-order memory accesses. We believe that such re-orderings can be encapsulated in appropriate SMA implementations. We also believe that this also opens up ways to extend our approach to other communication primitives such as MPI [37].

Unbounded Verification. UL-CSeq [70] uses a variant of the lazy schema that does not unwind loops and thus allows to analyze unbounded computations, even with an unbounded number of context switches. This still performs well, both for finding bugs in unsafe programs (see Table 2) and for proving the correctness of safe programs. Moreover, the schema maintains invariants, so invariant synthesis techniques [33, 40] could be applied on the sequentialized program. However, the UL-CSeq schema is not fully unbounded, as it assumes a bounded number of thread creations, and we are working to lift this restriction.

Concolic testing and Partial order reduction. The combination of Lazy-CSeq with a sequential symbolic execution backend such as KLEE [15] immediately gives us a *concolic testing* tool for concurrent programs. In particular, if we make the input fully concrete, we can search through all schedules symbolically, and so gain more information from a given test suite. Conversely, we can also fix a particular schedule, by choosing specific values for the context switch points, and leave all inputs symbolic; if no errors are found, we could automatically rewrite the program to adhere to the given schedule, or use a deterministic scheduler, to get verified executions.

However, this style of concolic testing subsumes the data-driven exploration of the symbolic schedule representation under the control-driven normal path exploration, which proceeds path by path. It could thus benefit from partial order reduction techniques that have been shown to improve the performance [93] of symbolic model checking. The approach of Kahlon et al. [55] for SAT-based analysis fits particularly well in our sequentialization schema, and we currently explore its implementation as code-to-code translation on top of Lazy-CSeq.

Swarm-based verification. We have recently used source-to-source translation as a method to parallelize Lazy-CSeq [36, 73]. More specifically, we developed a parametrizable translation that generates a set of simpler program instances, each capturing a reduced set of the original program's interleavings. These instances can then be checked independently in parallel by Lazy-CSeq (or indeed any tool that can handle multi-threaded C programs). This can reduce the wall-clock run times for difficult verification problems (see Section 6.4) by orders of magnitude. We currently investigate the use of abstract interpretation to quickly identify bug-free program instances, and so to prevent the BMC from being overloaded.

Parallel context bounded analysis. We have recently also proposed a parallelization technique [53] based on Lazy-CSeq using CBMC v5.4 and MiniSat v2.2.1 that allows to parallelize the analysis over multiple cores (or machines) up to a given number of execution contexts. The idea is to partition the search space examined by the SAT solver, and then to analyse it with multiple solvers running in isolation. To that end, we slightly alter the main driver of the sequentialized program, and override the heuristic decisions of CBMC's builtin SAT solver (i.e., MiniSat) at the very beginning of the propositional analysis so as to run the analysis under assumptions, where each assumption identifies a different subset of the execution of the program under analysis. Experiments on complex instances (including some of the complex programs considered in this paper) have shown consistent speedups when increasing the number of available computational units, within both safe and unsafe context bounds.

Embedded systems. Embedded systems software is typically characterized by a relatively simple program structure (e.g., bounded loops, simple data structures, no heap-allocated memory), which makes it very suitable for BMC approaches [26]. However, embedded systems typically employ more complicated concurrency models, based on interrupts, time-triggered events, and specific schedulers. Such models can still be handled by sequentialization [16, 56], and we will investigate how our lazy schema can be modified. The main difficulty is to correctly encode the limited (compared to general threads) nature of the allowed context switches. We will also try to exploit restrictions of the model (e.g., rate-monotonic scheduling) to achieve better performance.

Other types of transition systems. We believe that our method can also be applied to other types of (concurrent) transition systems, such as Petri nets [76] or Statecharts [45], either directly, or by translating them into C and then sequentializing the resulting code. The latter solution is particularly attractive for notations where code generators are already available.

9 CONCLUSIONS

We have presented a new and efficient lazy sequentialization schema for bounded multi-threaded C programs that has been carefully designed to take advantage of BMC tools developed for sequential programs. A core feature of our schema that significantly impacts its effectiveness is that it just injects light-weight, non-invasive control code into the input program. The control code is composed of few lines of guarded `goto`-statements and, within the added function `main`, also very few assignments. It does not use the program variables and it is clearly separated from the program code. This is in sharp contrast to existing sequentializations, where multiple copies of the shared variables are used and assigned in the control code.

We have implemented our approach in the Lazy-CSeq tool as a code-to-code translation. Lazy-CSeq can also be used as a stand-alone model checker that supports different verification tools as backends. We validated our approach experimentally on the SV-COMP20 [8] concurrency benchmarks suite, as well as several hard benchmarks from concurrent data structures where most other approaches fail. The results show that:

- Lazy-CSeq can reach the error locations in all but two of the unsafe SV-COMP20 benchmarks, and outperforms state-of-the art BMC tools that natively handle concurrency.
- Lazy-CSeq scales well to hard problems and substantially outperforms other tools on these.
- Lazy-CSeq is generic in the sense that it works uniformly well with different BMC-based backends; it also works with other backends, but the performance is more variable.

Laziness allows us to avoid handling all spurious errors that can occur in an eager exploration. Thus, we can inherit from the backend tool all checks for sequential programs such as array-bounds-check, division-by-zero, pointer-checks, overflow-checks, reachability of error labels and assertion failures, etc.

Our approach offers three general benefits that set it apart from previous approaches, and that simplify the development of full-fledged, robust model-checking tools based on sequentialization. First, the translation only needs to handle concurrency—all other features of the programming language remain opaque, and the backend tool can take care of them. This is in contrast to, for example, LR where dynamic allocation of the memory is handled by using maps [61]. Second, the original motivation for sequentializations was to reuse for concurrent programs the technology built for sequential program verification, and in principle, a sequentialization could work as a generic concurrency preprocessor for such tools. However, previous implementations needed specific tuning and optimizations for the different tools (see [34]). In contrast, Lazy-CSeq works well with a variety of backends (including BMC- and CEGAR-based model checkers, and symbolic testing tools), and the only required tuning is to comply with the actual program syntax supported by them. Finally, the clean separation between control code and program code makes it simple to generate a counter-example starting from the one generated by the backend tool.

Acknowledgments

This work was partially supported by EPSRC grants EP/M008991/1 and EP/P022413/1, by FARB 2019-2020 grants from Università degli Studi di Salerno, by INDAM-GNCS 2020-2021 and 2021 Visiting Professor grants, by University of Southampton Scholarship Schemes, and by AWS 2021 Amazon Research Awards.

REFERENCES

[1] Ole Agesen, David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. 2000. DCAS-based concurrent deques. In *Proceedings of the Twelfth annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2000, Bar Harbor, Maine, USA, July 9-13, 2000*, Gary L. Miller and Shang-Hua Teng (Eds.). ACM, 137–146. https://doi.org/10.1145/341800.341817

[2] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

[3] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2011. Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads. *Log. Methods Comput. Sci.* 7, 4 (2011). https://doi.org/10.2168/LMCS-7(4:4)2011

[4] Zuzana Baranová, Jiri Barnat, Katarína Kejstová, Tadeás Kucera, Henrich Lauko, Jan Mrázek, Petr Rockai, and Vladimír Still. 2017. Model Checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10482)*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer, 201–207. https://doi.org/10.1007/978-3-319-68167-2_14

[5] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. 2013. DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 863–868. https://doi.org/10.1007/978-3-642-39799-8_60

[6] Ira D. Baxter, Christopher W. Pidgeon, and Michael Mehlich. 2004. DMS®: Program Transformations for Practical Scalable Software Evolution. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 625–634. https://doi.org/10.1109/ICSE.2004.1317484

[7] Eli Bendersky. 2015. pycparser, v2.11. https://github.com/eliben/pycparser

[8] Dirk Beyer. 2020. Advances in Automatic Software Verification: SV-COMP 2020. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12079)*, Armin Biere and David Parker (Eds.). Springer, 347–367. https://doi.org/10.1007/978-3-030-45237-7_21

[9] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[10] Armin Biere. 2009. Bounded Model Checking. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 457–481. https://doi.org/10.3233/978-1-58603-929-5-457

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1579)*, Rance Cleaveland (Ed.). Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[12] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions (Department of Computer Science Report Series B, Vol. B-2020-1)*, Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.). University of Helsinki, 51–53.

[13] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Tractable Refinement Checking for Concurrent Objects. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 651–662. https://doi.org/10.1145/2676726.2677002

[14] Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato. 2011. On Sequentializing Concurrent Programs. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 129–145. https://doi.org/10.1007/978-3-642-23702-7_13

[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[16] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. 2011. Time-bounded analysis of real-time systems. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, Per Bjesse and Anna Slobodová (Eds.). FMCAD Inc., 72–80. http://dl.acm.org/citation.cfm?id=2157669

[17] Omar Chebaro, Pascal Cuoq, Nikolai Kosmatov, Bruno Marre, Anne Pacalet, Nicky Williams, and Boris Yakobowski. 2014. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng.* 21, 1 (2014), 107–143. https://doi.org/10.1007/s10515-013-0127-x

[18] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. 2013. BLITZ: Compositional bounded model checking for real-world programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 136–146. https://doi.org/10.1109/ASE.2013.6693074

[19] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7385)*, Alastair F. Donaldson and David Parker (Eds.). Springer, 248–254. https://doi.org/10.1007/978-3-642-31759-0_19

[20] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. 2011. Kratos - A Software Model Checker for SystemC. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 310–316. https://doi.org/10.1007/978-3-642-22110-1_24

[21] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

[22] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. 2010. Verifying SystemC: A software model checking approach. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, Roderick Bloem and Natasha Sharygina (Eds.). IEEE, 51–59. http://ieeexplore.ieee.org/document/5770933/

[23] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[24] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. 2005. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 570–574. https://doi.org/10.1007/978-3-540-31980-1_40

[25] Lucas C. Cordeiro and Bernd Fischer. 2011. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 331–340. https://doi.org/10.1145/1985793.1985839

[26] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. 2012. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Trans. Software Eng.* 38, 4 (2012), 957–974. https://doi.org/10.1109/TSE.2011.59

[27] CSeq framework 2015. https://github.com/CSeq/Overview

[28] David W. Currie, Alan J. Hu, and Sreeranga P. Rajan. 2000. Automatic formal verification of DSP software. In *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000*, Giovanni De Micheli (Ed.). ACM, 130–135. https://doi.org/10.1145/337292.337339

[29] Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamaric. 2015. Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 166–177. https://doi.org/10.1109/ASE.2015.30

[30] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers (Lecture Notes in Computer Science, Vol. 2919)*, Enrico Giunchiglia and Armando Tacchella (Eds.). Springer, 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

[31] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422. https://doi.org/10.1145/1926385.1926432

[32] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2colic testing. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 37–47. https://doi.org/10.1145/2491411.2491453

[33] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2019. Quantified Invariants via Syntax-Guided Synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 259–277. https://doi.org/10.1007/978-3-030-25540-4_14

[34] Bernd Fischer, Omar Inverso, and Gennaro Parlato. 2013. CSeq: A concurrency pre-processor for sequential C verification tools. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 710–713. https://doi.org/10.1109/ASE.2013.6693139

[35] Bernd Fischer, Omar Inverso, and Gennaro Parlato. 2013. CSeq: A Sequentialization Tool for C - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 616–618. https://doi.org/10.1007/978-3-642-36742-7_46

[36] Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2019. VeriSmart 2.0: Swarm-Based Bug-Finding for Multi-threaded Programs with Lazy-CSeq. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1150–1153. https://doi.org/10.1109/ASE.2019.00124

[37] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[38] Malay K. Ganai and Aarti Gupta. 2008. Efficient Modeling of Concurrent Systems in BMC. In *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5156)*, Klaus Havelund, Rupak Majumdar, and Jens Palsberg (Eds.). Springer, 114–133. https://doi.org/10.1007/978-3-540-85114-1_10

[39] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 519–531. https://doi.org/10.1007/978-3-540-73368-3_52

[40] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[41] Pranav Garg and P. Madhusudan. 2011. Compositionality Entails Sequentializability. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6605)*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer, 26–40. https://doi.org/10.1007/978-3-642-19835-9_4

[42] Naghmeh Ghafari, Alan J. Hu, and Zvonimir Rakamaric. 2010. Context-Bounded Translations for Concurrent Software: An Empirical Evaluation. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6349)*, Jaco van de Pol and Michael Weber (Eds.). Springer, 227–244. https://doi.org/10.1007/978-3-642-16164-3_17

[43] Shengjian Guo, Markus Kusano, and Chao Wang. 2016. Conc-iSE: incremental symbolic execution of concurrent software. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 531–542. https://doi.org/10.1145/2970276.2970332

[44] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science,*

Vol. 9206), Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[45] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

[46] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[47] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, Phillip B. Gibbons and Micah Adler (Eds.). ACM, 206–215. https://doi.org/10.1145/1007912.1007944

[48] Vladimir Herdt, Hoang Minh Le, Daniel Große, and Rolf Drechsler. 2015. Lazy-CSeq-SP: Boosting Sequentialization-Based Verification of Multi-threaded C Programs via Symbolic Pruning of Redundant Schedules. In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9364)*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.). Springer, 228–233. https://doi.org/10.1007/978-3-319-24953-7_18

[49] Gerard J. Holzmann. 2014. Mars code. *Commun. ACM* 57, 2 (2014), 64–73. https://doi.org/10.1145/2560217.2560218

[50] Gerard J. Holzmann. 2016. Cloud-Based Verification of Concurrent Software. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 311–327. https://doi.org/10.1007/978-3-662-49122-5_15

[51] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 807–812. https://doi.org/10.1109/ASE.2015.108

[52] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 585–602. https://doi.org/10.1007/978-3-319-08867-9_39

[53] Omar Inverso and Catia Trubiani. 2020. Parallel and distributed bounded model checking of multi-threaded programs. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 202–216. https://doi.org/10.1145/3332466.3374529

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

[55] Vineet Kahlon, Aarti Gupta, and Nishant Sinha. 2006. Symbolic Model Checking of Concurrent Programs Using Partial Orders and On-the-Fly Transactions. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4144)*, Thomas Ball and Robert B. Jones (Eds.). Springer, 286–299. https://doi.org/10.1007/11817963_28

[56] Nicholas Kidd, Suresh Jagannathan, and Jan Vitek. 2010. One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6349)*, Jaco van de Pol and Michael Weber (Eds.). Springer, 245–261. https://doi.org/10.1007/978-3-642-16164-3_18

[57] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2009. Analyzing recursive programs using a fixed-point calculus. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 211–222. https://doi.org/10.1145/1542476.1542500

[58] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2009. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 477–492. https://doi.org/10.1007/978-3-642-02658-4_36

[59] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2010. Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 629–644. https://doi.org/10.1007/978-3-642-14295-6_54

[60] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2012. Sequentializing Parameterized Programs. In *Proceedings Fourth Workshop on Foundations of Interface Technologies, FIT 2012, Tallinn, Estonia, 25th March 2012 (EPTCS, Vol. 87)*, Sebastian S. Bauer and Jean-Baptiste Raclet (Eds.). 34–47. https://doi.org/10.4204/EPTCS.87.4

[61] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. 2009. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 509–524. https://doi.org/10.1007/978-3-642-02658-4_38

[62] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. 2012. A Solver for Reachability Modulo Theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 427–443. https://doi.org/10.1007/978-3-642-31424-7_32

[63] Akash Lal and Thomas W. Reps. 2009. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods Syst. Des.* 35, 1 (2009), 73–97. https://doi.org/10.1007/s10703-009-0078-9

[64] Leslie Lamport. 1979. A New Approach to Proving the Correctness of Multiprocess Programs. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 84–97. https://doi.org/10.1145/357062.357068

[65] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7152)*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.). Springer, 146–161. https://doi.org/10.1007/978-3-642-27705-4_12

[66] Markus Müller-Olm. 2006. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs.* Lecture Notes in Computer Science, Vol. 3800. Springer. https://doi.org/10.1007/11871743

[67] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 446–455. https://doi.org/10.1145/1250734.1250785

[68] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 267–280. http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf

[69] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Unbounded Lazy-CSeq: A Lazy Sequentialization Tool for C Programs with Unbounded Context Switches - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 461–463. https://doi.org/10.1007/978-3-662-46681-0_45

[70] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9938)*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). 174–191. https://doi.org/10.1007/978-3-319-46520-3_12

[71] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Concurrent Program Verification with Lazy Sequentialization and Interval Analysis. In *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10299)*, Amr El Abbadi and Benoît Garbinato (Eds.). 255–271. https://doi.org/10.1007/978-3-319-59647-1_20

[72] Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Lazy-CSeq 2.0: Combining Lazy Sequentialization with Abstract Interpretation - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10206)*, Axel Legay and Tiziana Margaria (Eds.). 375–379. https://doi.org/10.1007/978-3-662-54580-5_26

[73] Truc L. Nguyen, Peter Schrammel, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Parallel bug-finding in concurrent programs via reduced interleaving instances. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 753–764. https://doi.org/10.1109/ASE.2017.8115686

[74] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* 9, 1 (2014), 53–58. https://doi.org/10.3233/sat190101

[75] Doron A. Peled. 1993. All from One, One for All: on Model Checking Using Representatives. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 697)*, Costas Courcoubetis (Ed.). Springer, 409–423. https://doi.org/10.1007/3-540-56922-7_34

[76] James L. Peterson. 1977. Petri Nets. *ACM Comput. Surv.* 9, 3 (1977), 223–252. https://doi.org/10.1145/356698.356702

[77] Shaz Qadeer. 2011. Poirot - A Concurrency Sleuth. In *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6991)*, Shengchao Qin and Zongyan Qiu (Eds.). Springer, 15. https://doi.org/10.1007/978-3-642-24559-6_3

[78] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

[79] Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William Pugh and Craig Chambers (Eds.). ACM, 14–24. https://doi.org/10.1145/996841.996845

[80] Ishai Rabinovitz and Orna Grumberg. 2005. Bounded Model Checking of Concurrent Programs. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3576)*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer, 82–97. https://doi.org/10.1007/11513988_9

[81] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[82] Niloofar Razavi, Azadeh Farzan, and Andreas Holzer. 2012. Bounded-Interference Sequentialization for Testing Concurrent Programs. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 7609)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 372–387. https://doi.org/10.1007/978-3-642-34026-0_28

[83] Petr Rockai, Jiri Barnat, and Lubos Brim. 2013. Improved State Space Reductions for LTL Model Checking of C and C++ Programs. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7871)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer, 1–15. https://doi.org/10.1007/978-3-642-38088-4_1

[84] Nishant Sinha and Chao Wang. 2010. Staged concurrent program analysis. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 47–56. https://doi.org/10.1145/1882291.1882301

[85] Nishant Sinha and Chao Wang. 2011. On interference abstractions. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 423–434. https://doi.org/10.1145/1926385.1926433

[86] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: an empirical study. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, José E. Moreira and James R. Larus (Eds.). ACM, 15–28. https://doi.org/10.1145/2555243.2555260

[87] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 402–404. https://doi.org/10.1007/978-3-642-54862-8_30

[88] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. MU-CSeq 0.3: Sequentialization by Read-Implicit and Coarse-Grained Memory Unwindings - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 436–438. https://doi.org/10.1007/978-3-662-46681-0_38

[89] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Verifying Concurrent Programs by Memory Unwinding. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 551–565. https://doi.org/10.1007/978-3-662-46681-0_52

[90] Ermenegildo Tomasco, Truc Lam Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models. In *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10469)*, Alessandro Cimatti and Marjan Sirjani (Eds.). Springer, 185–202. https://doi.org/10.1007/978-3-319-66197-1_12

[91] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 193–200. https://doi.org/10.1109/FMCAD.2016.7886679

[92] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. MU-CSeq 0.4: Individual Memory Location Unwindings - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 938–941. https://doi.org/10.1007/978-3-662-49674-9_65

[93] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. 2009. Symbolic pruning of concurrent program executions. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, Hans van Vliet and Valérie Issarny (Eds.). ACM, 23–32. https://doi.org/10.1145/1595696.1595702

[94] Dexi Wang, Chao Zhang, Guang Chen, Ming Gu, and Jiaguang Sun. 2016. C Code Verification based on the Extended Labeled Transition System Model. In *Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016 (CEUR Workshop Proceedings, Vol. 1725)*, Juan de Lara, Peter J. Clarke, and Mehrdad Sabetzadeh (Eds.). CEUR-WS.org, 48–55. http://ceur-ws.org/Vol-1725/demo7.pdf

[95] Liangze Yin, Wei Dong, Wanwei Liu, Yunchou Li, and Ji Wang. 2018. YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 422–426. https://doi.org/10.1007/978-3-319-89963-3_25

[96] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2020. On Scheduling Constraint Abstraction for Multi-Threaded Program Verification. *IEEE Trans. Software Eng.* 46, 5 (2020), 549–565. https://doi.org/10.1109/TSE.2018.2864122

[97] Manchun Zheng, John G. Edenhofner, Ziqing Luo, Mitchell J. Gerrard, Michael S. Rogers, Matthew B. Dwyer, and Stephen F. Siegel. 2016. CIVL: Applying a General Concurrency Verification Framework to C/Pthreads Programs (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 908–911. https://doi.org/10.1007/978-3-662-49674-9_57