Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs*

Truc L. Nguyen[1], Bernd Fischer[2], Salvatore La Torre[3], and Gennaro Parlato[1]

[1] Electronics and Computer Science, University of Southampton, UK
[2] Computer Science, Stellenbosch University, South Africa
[3] Informatica, Università degli Studi di Salerno, Italy

Abstract. Lazy sequentialization has emerged as one of the most promising approaches for concurrent program analysis but the only efficient implementation given so far works just for bounded programs. This restricts the approach to bug-finding purposes. In this paper, we describe and evaluate a new lazy sequentialization translation that does not unwind loops and thus allows to analyze unbounded computations, even with an unbounded number of context switches. In connection with an appropriate sequential backend verification tool it can thus also be used for the safety verification of concurrent programs, rather than just for bug-finding. The main technical novelty of our translation 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 for C99 programs that use the pthreads API. We evaluate UL-CSeq on several benchmarks, using different sequential verification backends on the sequentialized program, and show that it is more effective than previous approaches in proving the correctness of the safe benchmarks, and still remains competitive with state-of-the-art approaches for finding bugs in the unsafe benchmarks.

1 Introduction

Concurrent programming is becoming more important as concurrent computer architectures such as multi-core processors are becoming more common. However, the automated verification of concurrent programs remains a difficult problem. The main cause of the difficulties is the large number of possible ways in which the different elements of a concurrent program can interact with each other, e.g., the number of different thread schedules. This in turn makes it difficult and time-consuming to build effective concurrent program verification tools, either from scratch or by extending existing sequential program verification tools.

An alternative approach is to translate the concurrent program into a non-deterministic sequential program that *simulates* the original program, and then to reuse an existing sequential program verification tool as a black-box backend to verify this simulation program. This approach is also known as *sequentialization* [23, 19, 15]. It has been used successfully both for bug-finding purposes [3, 12, 25] and for the verification of

* Partially supported by EPSRC grant no. EP/M008991/1, and MIUR-FARB 2013-2016 grants.

reachability properties [7, 16, 17]. Its main advantage is that it separates the concurrency aspects from the rest of the verification tool design and implementation. This has several benefits. First, it simplifies the concurrency handling, which can be reduced to one (usually simple) source-to-source translation. Second, it makes it thus also easier to experiment with different concurrency handling techniques; for example, we have already implemented a number of different translations such as [5, 12, 25] within our CSeq framework [11]. Third, it makes it easier to integrate different sequential backends. Finally, it reduces the overall development effort, because the sequential program aspects and tools can be reused.

The most widely used sequentialization (implemented in Corral [18], Smack [24], and LR-CSeq [5]) by Lal and Reps [19] uses additional copies of the shared variables for the simulation and guesses their values (*eager* sequentialization). This makes the schema unsuitable to be extended for proof finding: it can handle only a bounded number of context switches, and the unconstrained variable guesses lead to over-approximations that are too coarse and make proofs infeasible in practice. *Lazy* sequentializations [15], on the other hand, do not over-approximate the data, and thus maintain the concurrent program's invariants and simulate only feasible computations. They are therefore in principle more amenable to be extended for correctness proofs although efficient implementations exist only for bounded programs [16, 17].

Here, we develop and implement a lazy sequentialization that can handle programs with unbounded loops and an unbounded number of context switches, and is therefore suitable for program verification (both for correctness and bug-finding). The main technical novelty of our translation is the simulation of the thread resumption in a way that does not require that each statement is executed at most once and does (unlike Lazy-CSeq [12, 11, 13]) not rely on gotos to reposition the execution. Instead, we maintain a single scalar variable that determines whether the simulation needs to skip over a statement or needs to execute it. Our first contribution in this paper is the description of the corresponding source-to-source translation in Section 3. As a second contribution, we have implemented this sequentialization in the UL-CSeq tool (within our CSeq framework) for C99 programs that use the pthreads API (see Section 4). We have evaluated, as a third contribution, UL-CSeq on a large set of benchmarks from the literature and the concurrency category of the software verification competition SV-COMP, using different sequential verification backends on the sequentialized program. We empirically demonstrate, also in Section 4, that our approach is surprisingly efficient in proving the correctness of the safe benchmarks and improves on existing techniques that are specifically developed for concurrent programs. Furthermore, we show that our solution is competitive with state-of-the-art approaches for finding bugs in the unsafe benchmarks. We present related work in Section 5 and conclude in Section 6.

2 Multi-threaded programs

In this paper, we use a simple multi-threaded imperative language to illustrate our approach. It includes dynamic thread creation and join, and mutex locking and unlocking operations for thread synchronization. However, our approach can easily be extended to full-fledged programming languages, and our implementation can handle full C99.

$$
\begin{array}{lll}
P & ::= & (dec\texttt{;})^*\ (typ\ p\ \texttt{(}\langle dec\texttt{,}\rangle^*\texttt{)}\ \ \texttt{\{}(dec\texttt{;})^*\ stm\texttt{\}})^* \\
dec & ::= & typ\ z \\
typ & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{mutex} \mid \texttt{void} \\
stm & ::= & seq \mid con \mid \texttt{\{}\langle stm\texttt{;}\rangle^*\texttt{\}} \\
seq & ::= & \texttt{assume(}b\texttt{)} \mid \texttt{assert(}b\texttt{)} \mid x\texttt{=}e \mid p\,\texttt{(}\langle e\texttt{,}\rangle^*\texttt{)} \mid \texttt{return}\ e \\
 & & \mid \texttt{if(}b\texttt{)}\ stm\ [\texttt{else}\ stm] \mid \texttt{while(}b\texttt{)}\ \texttt{do}\ stm \mid l\texttt{:}\ seq \mid \texttt{goto}\ l \\
con & ::= & x\texttt{=}y \mid y\texttt{=}x \mid t\texttt{=}\texttt{create}\ p\,\texttt{(}\langle e\texttt{,}\rangle^*\texttt{)} \mid \texttt{join}\ t \\
 & & \mid \texttt{init}\ m \mid \ \texttt{lock}\ m \mid \texttt{unlock}\ m \mid \texttt{destroy}\ m \mid l\texttt{:}\ con
\end{array}
$$

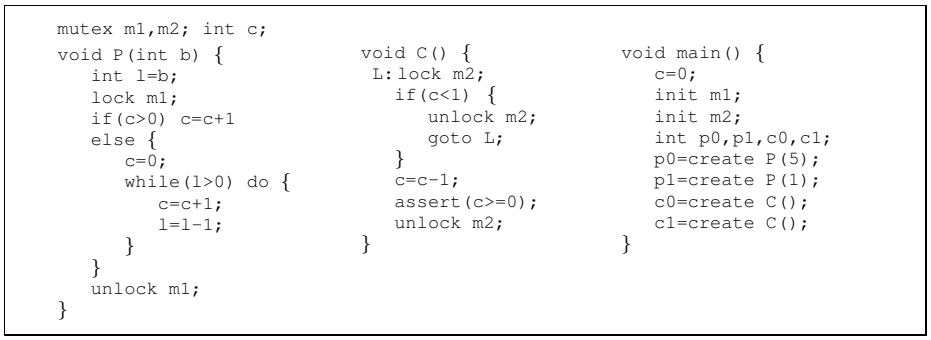
Fig. 1. Syntax of multi-threaded programs.

Syntax. The syntax of multi-threaded programs is defined by the grammar shown in Figure 1. x denotes a local variable, y a shared variable, m a mutex, t a thread variable and p a procedure name. All variables involved in a sequential statement are local. We assume expressions e to be local variables, constants, that can be combined using mathematical operators. Boolean expressions b can be `true` or `false`, or Boolean variables, which can be combined using standard Boolean operations.

A *multi-threaded* program P consists of a list of *global* variable declarations (i.e., *shared* variables), followed by a list of procedures. 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 statement stm is either a sequential, or a concurrent statement, or a sequence of statements enclosed in braces.

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 labelled sequential statement, or a jump to a label. Local variables are considered uninitialised right after their declaration, which means that they can take any value from their respective domains. Therefore, until not explicitly set by an appropriate assignment statement, they can non-deterministically assume any value allowed by their type. We also use the symbol $\star$ to denote the expression that non-deterministically evaluates to any possible value; for example, with `x = ⋆` we mean that `x` is assigned any possible value of its type domain.

A *concurrent statement* con can be a concurrent assignment, a call to a thread routine, such as a thread creation, a join, or a mutex operation (i.e., init, lock, unlock, and destroy), or a labelled concurrent statement. A concurrent assignment assigns a shared (resp. local) variable to a local (resp. shared) one. Unlike local variables, global variables are always assumed to be initialised to a default value. 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, pauses the current thread until the thread identified by t *terminates* its execution. Lock and unlock statements respectively acquire and release a mutex. If the mutex is already acquired, the lock op-



```
mutex m1,m2; int c;
void P(int b) {
   int l=b;
   lock m1;
   if(c>0) c=c+1
   else {
      c=0;
      while(l>0) do {
         c=c+1;
         l=l-1;
      }
   }
   unlock m1;
}

void C() {
 L: lock m2;
   if(c<1) {
      unlock m2;
      goto L;
   }
   c=c-1;
   assert(c>=0);
   unlock m2;
}

void main() {
   c=0;
   init m1;
   init m2;
   int p0,p1,c0,c1;
   p0=create P(5);
   p1=create P(1);
   c0=create C();
   c1=create C();
}
```

Fig. 2. Producer-Consumer multi-threaded program containing a reachable assertion failure. In the `main` thread, functions `P` and `C` are both used twice to spawn a thread.

eration 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 does not contain direct or indirect recursive function calls but 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 no other thread can be created that uses `main` as starting procedure. Finally, our programs are not *parameterized*, in the sense that we allow only for a bounded number of thread creations.

Semantics. We assume a C-like semantics for each thread execution and a standard semantics by interleaving for the concurrent executions. At any given time of a computation, only one thread is executing (*active*). In the beginning only the main thread is active and no other thread exists; new threads can be spawned by a thread creation statement and are added to the pool of *enabled* threads. At a *context switch* the currently active thread is suspended and becomes enabled, and one of the enabled threads is resumed and becomes the new active thread. When a thread is resumed its execution continues either from the point where it was suspended or, if it becomes active for the first time, from the beginning.

All threads share the same address space: they can write to or read from global (*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 [20]. We further assume that each statement is atomic. This is not a severe restriction, as it is always possible to decompose a statement into a sequence of statements, each involving at most one shared variable.

Example. The program shown in Figure 2 models a producer-consumer system, with two shared variables, two mutexes `m1` and `m2`, 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` (*producer*) and two threads executing `C` (*consumer*). Each producer acquires `m1`, increments

c if it is positive or copies over the initival value "one-by-one", and terminates by releasing m1. Each consumer first acquires m2, then checks whether all the elements have been consumed; if so, it releases m2 and restarts from the beginning (goto-statement); otherwise, it decrements c, checks the assertion $\mathtt{c} \geq 0$, releases m2 and terminates.

At any point of the computation, mutex m1 ensures that at most one producer is operating and mutex m2 ensures that only one consumer is attempting to decrement c. Therefore the assertion cannot be violated (*safe instance* of the Producer-Consumer program). However, by removing the consumers' synchronization on mutex m2, the assertion could be violated since the behavior of the two consumer threads now can be freely interleaved: with $\mathtt{c} = 1$, both consumers can decrement c and one of them will write the value -1 back to c, and thus violate the assertion (*unsafe instance* of the Producer-Consumer program). □

3 Unlimited Lazy Sequentialization

In this section we present a code-to-code translation from a multi-threaded program P to a sequential program P^{seq} that simulates all executions of P.

We assume that P consists of $n + 1$ functions $f_0, \ldots, f_n$, where f_0 is the main function, and that there are no function calls and each create statement (1) is executed at most once in any execution and (2) is associated with a distinct start function f_i. Consequently, the number of threads is bounded, and threads and functions can be identified. For ease of presentation, we also assume that thread functions have no arguments. We adopt the convention that each statement in P is annotated with a (unique) numerical label: the first statement of each function is labelled by 0, while its following statements are labelled with consecutive numbers increasing in the text order. This ordering on the numerical labels is used by our translation for controlling the simulation of the starting program in the resulting sequential program. These restrictions are used only to simplify the presentation.

P^{seq} simulates P in a *round-robin* fashion. Each computation of P is split into rounds. Each *round* is an execution of zero or more statements from each thread in the order $f_0, \ldots, f_n$. Note that this suffices to capture any possible execution since we allow for unboundedly many rounds and we can arbitrarily skip the execution of a thread in any round (i.e., execute zero statements). The main of P^{seq} is a driver formed by an infinite while-loop that simulates one round of P in each iteration, by repeatedly calling the thread simulation function f_i^{seq} of each thread f_i.

Each simulation function f_i^{seq} can non-deterministically exit at any statement to simulate a context switch. Thus, for each thread f_i, P^{seq} maintains in a global variable $\mathtt{pc}_i$ the numerical label at which the context switch was simulated in the previous round and where the computation must thus resume from in the next round. The local variables of f_i are made persistent in f_i^{seq} (i.e., changed to static) such that we do not need to recompute them on resuming suspended executions. Each f_i^{seq} is essentially f_i with few lines of injected control code for each statement that guard its execution, and the thread routines (i.e., create, join, init, lock, unlock, destroy) are replaced with calls to corresponding simulation functions. The execution of each call to a function f_i^{seq} goes through the following modes:

RESUME: the control is stepping through the lines of code without executing any actual statements of f_i until the label stored in $\texttt{pc}_i$ is reached; this mode is entered every time the function f_i^{seq} is called.

EXECUTE: the execution of f_i has been resumed (i.e., the label stored in $\texttt{pc}_i$ has been reached) and the actual statements of f_i are now executing.

SUSPEND: the execution has been blocked and the control returns to the main function; hence, no actual statements of f_i are executed in this mode. It is entered nondeterministically from the `EXECUTE` mode; on entering it, the numerical label of the current f_i statement (the one to be executed next) is stored in $\texttt{pc}_i$.

Code-to-code translation

We now describe our translation in a top-down fashion and convey an informal correctness argument as we go along. The entire translation is formally described by the recursive code-to-code translation function $[\![\cdot]\!]$ defined by the rewrite rules given in Figure 3. Rule 1 gives the outer structure of P^{seq}: it adds the declarations of the global auxiliary variables, replaces each thread function f_i with the corresponding simulation function f_i^{seq}, adds the code stubs for the thread routines, and then the main function. The remaining rules give the transformation for all statement types in our grammar; we will return to this in the description of the translation of each thread function f_i into the corresponding simulation function f_i^{seq}.

We start by describing the global auxiliary variables used in the translation. Then, we give the details of function `main` of P^{seq}, and illustrate the translation from f_i into f_i^{seq}. Finally, we discuss how the thread routines are simulated.

Auxiliary variables. Let `N` denote the maximal number of threads in the program other than the main thread. We statically assign a distinct identifier to each thread of P from the interval $[\texttt{0}, \texttt{N}]$; the identifier assigned to `main` is `0`. During the simulation of P, P^{seq} maintains the following auxiliary variables, for $i \in [\texttt{0}, \texttt{N}]$:

- $\texttt{bool created}_i$ tracks whether the thread with identifier i has ever been created. Initially, only $\texttt{created}_0$ is set to `true` since f_0^{seq} simulates the `main` function of P.
- $\texttt{int pc}_i$ stores the numerical label of the last context switch point for thread i. All the variables $\texttt{pc}_i$ are initialized to `0` that is the numerical label of the first statement of all thread functions.
- `int s` tracks the simulation mode as described above. It can only assume the values `RESUME`, `EXECUTE`, or `SUSPEND`.

Main driver. The new `main` of P^{seq} (see Figure 4) consists of an infinite loop that calls at each iteration the thread functions of the active threads.

Thread simulation functions. Each function f_i representing a thread in P is translated into the thread simulation function f_i^{seq} in P^{seq} as follows. First, the local variables of f_i are declared as `static` in f_i^{seq} to make them *persistent* between consecutive invocations of f_i^{seq}. Then, $[\![\cdot]\!]_i$ is applied recursively to the statements in the body of f_i^{seq} (see Rule 1 of Figure 3).

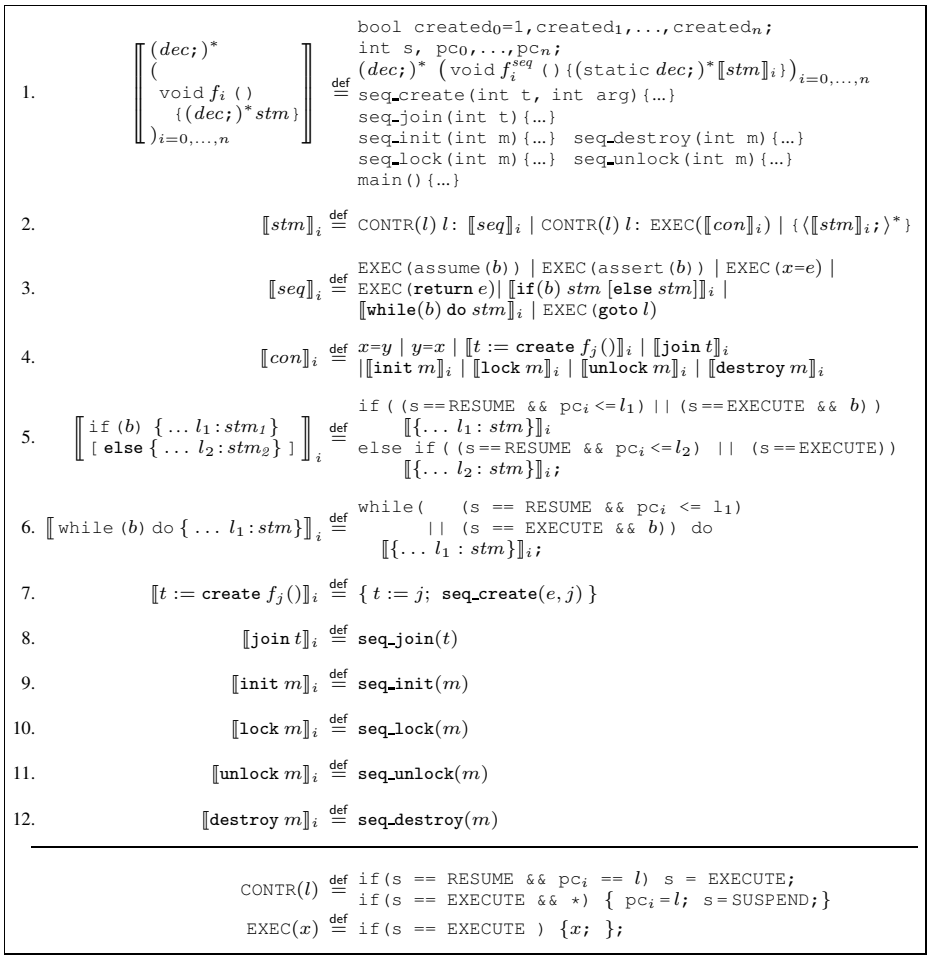


Fig. 3. Rewriting rules for the lazy sequentialization.

For each statement we inject a few lines of code that implement the control of the simulation, i.e., make decisions on mode transitions in the simulation and, depending on the current mode, execute or skip the guarded statement. Specifically, every original statement is preceded by the code of the macro CONTR defined in Figure 3 that takes as input the label l of the statement (see Rule 2). The injected code allows to set the mode to EXECUTE if the simulation is in RESUME mode and the old context switch point is reached. After that, if the simulation is in EXECUTE mode, it can non-deterministically transit into SUSPEND, and if so the label l is stored into $\mathtt{pc}_i$. Note that, to skip the execution of a thread in a round, we need first to switch from RESUME to EXECUTE and then to SUSPEND before the simulation of the original statement. Furthermore, except for if- and while-statements, all the other statements are guarded by an if-

```
int main(void){
  while(true) do{
    s = RESUME; /* set mode to RESUME before thread simulation */
    f0 (); /* main thread simulation */

    s = RESUME;
    if (created1) f1 (); /* simulation of thread with id 1 */
    ...
    s = RESUME;
    if (createdn) fn (); /* simulation of thread with id n */
  }
}
```

Fig. 4. The main function of P^{seq}.

statement injected by the macro EXEC that prevents their simulation unless the mode of the simulation is EXECUTE.

We need to (partially) simulate the if- and while-statements even if we are in RESUME mode, in order to position the execution back to the resumption point stored in $\texttt{pc}_i$. We achieve this by modifying their respective control flow guards. For the if-statement (see Rule 3), we check whether $\texttt{pc}_i$ is in either of the then- or else-branch (note that if $\texttt{pc}_i$ was less then the label of the current if-statement, we must already be in the EXECUTE mode and so we need to compare only against l_1 and l_2 which are respectively the labels of the last statements in the then- and else branches). If so, we go into the corresponding branch, independent of the *current* valuation of the condition b; we do this because we are only repositioning, and our resumption point reflects the *previous* valuation of the condition that held when the context switch occurred. Of course, if we are in EXECUTE mode, we need to check the condition. We follow a similar approach for while-statements. Note that here we only need one iteration over the loop's body to find the resumption point, so we do not need to check the condition in the RESUME mode. Finally, each call to a thread routine is also translated into a call to the corresponding simulation function (Rules 7–12).

Figure 5 shows the thread simulation function resulting from sequentializing the thread P shown in Figure 2.

Simulation of the thread routines. For each thread routine we provide a verification stub, i.e., a simple standard C function that replaces the original implementation for verification purposes. The verification stubs are identical to those used by Lazy-CSeq. Below, we informally describe how they work; full details are given in [12]. In seq_create we simply set the thread's created flag. Note that we do not need to store the thread start function, as the main driver calls all thread simulation functions explicitly and seq_create uses an additional integer argument that serves as thread identifier that is statically determined in the call.

According to the semantics of the join-statement, a thread executing join t should be blocked until thread t is terminated (i.e., the corresponding pc variable is set to LAST_LABEL that is a statically defined constant larger than any other label in P). We choose to not implement in P^{seq} any notion of blocking or unblocking a thread; instead seq_join uses an assume-statement with the condition pc_t == LAST_LABEL

```
void P (int b){ static int l;
   if (s == RESUME && pc == 0) s = EXECUTE;
   if (s == EXECUTE && *) {pc = 0; s = SUSPEND;}
   if (s == EXECUTE) { l = b; }
   if (s == RESUME && pc == 1) s = EXECUTE;
   if (s == EXECUTE && *) {pc = 1; s = SUSPEND;}
   if (s == EXECUTE) { seq_lock(m1); }
   if (s == RESUME && pc == 2) s = EXECUTE;
   if (s == EXECUTE && *) {pc = 2; s = SUSPEND;}
   if ((s == RESUME && pc <= 3) || (s == EXECUTE && (c > 0))){
      if (s == RESUME && pc == 3) s = EXECUTE;
      if (s == EXECUTE && *) {pc = 3; s = SUSPEND;}
      if (s == EXECUTE && LOCKED(m1)) { c = c + 1; }}
   else if ((s == RESUME && pc <= 6) || (s == EXECUTE)) {
      if (s == RESUME && pc == 4) s = EXECUTE;
      if (s == EXECUTE && *) {pc = 4; s = SUSPEND;}
      if (s == EXECUTE && LOCKED(m1)) { c = 0; }
      if (s == RESUME && pc == 5) s = EXECUTE;
      if (s == EXECUTE && *) {pc = 5; s = SUSPEND;}
      while ((s == RESUME && pc <= 6) || ((s == EXECUTE) && (l > 0))) do {
         if (s == RESUME && pc == 6) s = EXECUTE;
         if (s == EXECUTE && *) {pc = 6; s = SUSPEND;}
         if (s == EXECUTE && LOCKED(m1)) { c = c + 1; }
         if (s == EXECUTE && LOCKED(m1)){ l = l - 1; }}}
   if (s == RESUME && pc == 7) s = EXECUTE;
   if (s == EXECUTE && *) {pc = 7; s = SUSPEND;}
   if (s == EXECUTE && LOCKED(m1)) { seq_unlock(m1); }
   if (s == EXECUTE || (s == RESUME && pc == 8)){ pc = 8; s = SUSPEND; }
}
```

Fig. 5. Translation of thread P from Fig. 2.

to prune away any simulation that corresponds to a blocking join. We can then see that this pruning does not alter the thread reachability properties of the original program. Assume that the joining thread `t` terminates after the execution of `join t`. The invoking thread should be unblocked then but the simulation has already been pruned. However, this execution can be captured by another simulation in which a context switch is simulated right before the execution of this `join`-statement, and the invoking thread is scheduled to run only after `t` has terminated, hence avoiding the pruning as above.

For mutexes we need to know whether they are free or already destroyed, or which thread holds them otherwise. For this, in the corresponding functions, we use two constants `FREE` and `DESTROY`. On initializing or destroying a mutex we assign it the appropriate constant. In `seq_lock`, we assert that the mutex is not destroyed and then check whether it is free before assigning it the index of the thread that has invoked the function. As in the case of the `join`-statement we block the simulation if the lock is held by another thread. In `seq_unlock`, we first assert that the lock is held by the invoking thread and then set it to `FREE`. We also support re-entrant mutexes.

Correctness. The correctness of our construction is quite straightforward.

For the *completeness*, assume any non-empty execution ρ of P that creates at most N threads. Let $\rho = \rho_0 \dots \rho_k$ be split into maximal execution contexts (i.e., each ρ_i is non-empty and has statements only from one thread and ρ_i and ρ_{i+1} are from different threads). Clearly, ρ_0 is a context of the main thread of P that is the only one existing in the beginning. P^{seq} starts the execution from the driver `main` and then calls f_0^{seq}

(i.e., the simulation function of the main thread of P). At the first injected control code, since `s` evaluates to `RESUME` and $\texttt{pc}_0$ evaluates to 0 (since `s` is always set to `RESUME` in the driver before calling a simulation function and all the $\texttt{pc}_i$'s are initialized to 0), and since we do not context switch yet, `s` is updated to `EXECUTE` and the original statement of P is executed (see Figure 3). The simulation of the remaining statements in ρ_0 is done similarly. On context-switching from ρ_0 to ρ_1, at the second `if`-statement of the macro `CONTR` injected to control the first statement in ρ_1, since we are in the `EXECUTE` mode, we can select to context-switch and thus $\texttt{pc}_0$ is updated with the label of this statement (that is the next to execute when the thread will be resumed) and change the simulation mode to `SUSPEND`. From this point to the end of f_0^{seq} the control code will skip the execution of all the remaining statements of f_0, and thus the control returns to the main function of P^{seq} after the call to f_0^{seq}. Now, assume that ρ_1 is a context of a thread f_j, $j \neq 0$. Clearly, the thread must have been created in ρ_0, thus $\texttt{created}_j$ must hold true. Thus in the main driver we skip all calls to f_i for $i < j$, either because $\texttt{created}_i$ is false (i.e., the thread has not been created yet) or because we context-switch out immediately when calling f_i^{seq}. Then, we call f_j^{seq} and repeat the same argument as for ρ_0. To complete this part we need just to handle the case when we execute a context ρ_j of thread f_i that is not its first context. In this case, since the simulation mode is set to `RESUME` in the main driver, the control code forces to skip all the statement of P until we reach the label stored in $\texttt{pc}_i$. Since all the local variables are declared static and there are no function calls besides the call to the thread routine stubs, the local state of f_i is exactly as it was when the thread was pre-empted last time. Therefore, we can simulate ρ_j as observed above and we are done.

The *soundness* argument is a direct consequence of the fact that P^{seq} executes statements of P and the injected control code just positions the control for the simulation of context-switching. Thus, from each execution ρ of P^{seq} we can extract an execution of P by simply projecting out the auxiliary variables and the control code statements.

Therefore, we get that P^{seq} violates an assertion if and only if P does and the following theorem holds:

Theorem 1. *A concurrent program P violates an assertion in at least one of its executions with at most N thread creations if and only if P^{seq} violates the same assertion.*

4 Implementation and Experiments

4.1 Implementation

We have implemented in UL-CSeq v0.2[1] the schema discussed in Section 3 as a code-to-code transformation for sequentially-consistent concurrent C programs with POSIX threads (`pthreads`). This implementation is slightly optimized compared to the version that participated (using the CPAchecker backend) in SV-COMP16 [22].

UL-CSeq is implemented as a chain of modules within the CSeq framework [5, 6]. The sequentialized program is obtained from the original program through transformations, which (i) insert boilerplate code for simulating the `pthreads` API; (ii) unwind

[1] http://users.ecs.soton.ac.uk/gp4/cseq/files/ul-cseq-0.2_64bit.tar.gz

any loops that create threads; (iii) create multiple copies of the thread start functions, and inline all other function calls; (iv) implement the translation rules, as shown in Figure 3; and (v) insert code for the main driver, and finalize the translation by adding backend-specific instrumentation.

4.2 Experiments

We experimentally evaluated the capabilities and performance of our UL-CSeq implementation (as sketched above) for both verification and bug-finding purposes. We mainly used the benchmark set from the Concurrency category of the TACAS Software Verification Competition (SV-COMP16) [2]. These are widespread benchmarks, and many state-of-the-art analysis tools have been trained on them. They offer a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms. In addition, we also used two smaller benchmark collections from the literature [27, 7]. For all benchmarks we unwound thread-creating loops twice. Since we executed the verification and the bug-finding experiments on different machines and benchmark subsets, we report on them separately.

Verification. Here, we used UL-CSeq in combination with four different sequential backends (SeaHorn, Ultimate Automizer, CPAchecker, and VVT), and compared it with four different verification tools with built-in concurrency handling (Impara, Satabs, Threader, and VVT). These were chosen to cover a range of different sequential and concurrent verification techniques. Please note that we cannot compare to the top tools of the SV-COMP because all three medal winners are based on bounded model checking and do not produce proofs but simply claim benchmarks to be safe if they do not find a bug with their chosen settings.

Experimental Setup. For the verification experiments, we used the 221 safe benchmarks from the SV-COMP collection as well as the 13 safe benchmarks from [27] and [7]. The total size of the benchmarks was approximately 37K lines of code. We ran the experiments on a large compute cluster of Xeon E5-2670 2.6GHz processors with 16GB of memory each, running a Linux operating system with 64-bit kernel 2.6.32. We set a 15GB memory limit and a 900s timeout for the analysis of each benchmark. We used SeaHorn [9] (v0.1.0),[2] an LLVM-based [21] framework for verification of safety properties of programs using Horn Clause solvers; Ultimate Automizer [10] (SV-COMP16),[3] an automata-based software model checker that is implemented in the Ultimate software analysis framework; CPAchecker (v1.4 with predicate abstraction),[4] a tool for configurable software verification that supports a wide range of techniques, including predicate abstraction, and shape and value anlysis; Impara (v0.2),[5] a tool that implements an algorithm that combines a symbolic form of partial-order reduction and lazy abstraction with interpolants for concurrent programs; Satabs (v3.2),[6] a verification

[2] https://github.com/seahorn/seahorn/releases/download/v0.1.0/SeaHorn-0.1.0-Linux-x86_64.tar.gz

[3] http://ultimate.informatik.uni-freiburg.de/downloads/svcomp2016/UltimateAutomizer.zip

[4] http://cpachecker.sosy-lab.org/CPAchecker-1.4-unix.tar.bz2

[5] http://www.cprover.org/concurrent-impact/impara-linux64-0.2.tgz

[6] http://www.cprover.org/satabs/download/satabs-3-2-linux-32.tgz

Table 1. Performance comparison of different verification tools on safe benchmarks: UL-CSeq with different sequential backends (top); other tools with built-in concurrency handling (bottom). Each row corresponds to a sub-category of the SV-COMP16 benchmarks, or to one of the benchmark sets from the literature; we report the number of files and the total number of lines of code. *pass* denotes the number of correctly verified safe benchmarks (i.e., proofs found), *fail* the number of benchmarks where the tool found a spurious error or crashed (including running out of memory), *t.o.* the number of benchmarks on which the tool exceeded the given time limit, and *time* is the average proof time (i.e., excluding failed attempts).

			UL-CSeq +															
			SeaHorn				Automizer				CPAchecker				VVT			
sub-category	files	l.o.c.	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time
pthread	15	1285	3	2	10	67.3	3	2	10	390.8	2	3	10	204.9	5	3	7	247.3
pthread-atomic	9	1136	6	1	2	167.9	3	1	5	456.7	5	0	4	352.6	5	0	4	171.8
pthread-ext	45	3679	27	0	18	199.1	12	2	31	226.5	15	0	30	214.6	16	5	24	179.7
pthread-lit	8	427	3	0	5	23.3	1	0	7	544.9	3	0	5	164.1	3	2	3	79.8
pthread-wmm	144	29426	144	0	0	32.5	60	0	84	421.6	26	0	118	271.3	141	0	3	275.3
[27]	7	542	5	0	2	51.1	3	1	3	238.6	4	0	3	244.7	4	1	2	133.1
[7]	6	290	6	0	0	5.7	5	0	1	181.8	5	0	1	44.9	6	0	0	17.2
Totals	234	36785	194	3	37	59.9	87	6	141	376.2	60	3	171	235.7	180	11	43	248.2

			Impara				Satabs				Threader				VVT			
sub-category	files	l.o.c.	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time
pthread	15	1285	5	2	8	12.2	3	8	4	308.7	6	8	1	128.4	5	1	9	7.3
pthread-atomic	9	1136	5	0	4	61.8	4	3	2	1.3	7	0	2	24.4	7	1	1	143.7
pthread-ext	45	3679	30	0	15	8.7	15	13	17	34.6	36	1	8	104.8	38	1	6	66.2
pthread-lit	8	427	2	0	6	0.4	2	5	1	8.1	0	7	1	N/A	5	1	2	7.3
pthread-wmm	144	29426	24	0	120	9.0	100	22	22	312.2	0	144	0	N/A	130	0	14	222.2
[27]	7	542	6	0	1	0.5	4	1	2	1.0	5	1	1	27.5	4	3	0	154.7
[7]	6	290	5	1	0	2.7	6	0	0	0.8	3	3	0	58.2	3	3	0	8.8
Totals	234	36785	77	3	154	11.2	134	52	48	244.0	57	164	13	88.2	192	10	30	172.6

tool based on predicate abstraction; and Threader (SV-COMP14),[7] a tool that uses compositional reasoning with regards to the thread structure of concurrent programs based on abstraction refinement. VVT (SV-COMP16),[8] a tool that can both verify programs using IC3 and predicate abstraction also can find bugs using bounded model checking. We ran each tool with its default configuration.

Results. Table 1 summarizes the results. It demonstrates that our approach is (with suitable backends) surprisingly effective: using SeaHorn, we can prove 194 out of the 234 benchmarks, and just edge out victory over VVT, the best-performing tool with built-in concurrency handling. However, note that UL-CSeq's performance varies widely with the applied backend, and using Automizer or CPAchecker produces noticeably worse results. Proof times are difficult to compare in aggregate, but overall UL-CSeq's proof times are within the range of the other tools, indicating that the sequentialization does not introduce too much complexity. This is further corroborated by the fact that the combination of UL-CSeq and VVT (which finds 180 proofs) is only slightly weaker than VVT relying on its built-in concurrency handling (which finds 192 proofs).

Bug-finding. Here, we used UL-CSeq in combination with CBMC as sequential backend, and compared it with four different bug-finding tools, Lazy-CSeq, CBMC, CIVL, and Smack. All four are (ultimately) based on bounded model checking, and have per-

[7] https://www7.in.tum.de/tools/threader/threader.tgz

[8] http://vvt.forsyte.at/releases/vvt-svcomp.tar.xz

Table 2. Performance comparison of different tools on the unsafe instances of the SV-COMP16 *Concurrency category*. Each row corresponds to a sub-category of the SV-COMP16 benchmarks; we report the number of files and the total number of lines of code. *pass* now denotes the number of correctly identified unsafe benchmarks (i.e., counterexamples found) and *t.o.* the number of benchmarks on which the tool exceeded the given time limit, and *time* the average time to find a bug. None of the tools reported any spurious counterexample.

			UL-CSeq + CBMC			Lazy-CSeq + CBMC			CBMC			CIVL			Smack		
sub-category	files	l.o.c.	pass	t.o.	time	pass	t.o.	time	pass	t.o.	time	pass	t.o.	time	pass	t.o.	time
pthread	17	4085	14	3	12.2	17	0	19.4	16	1	63.1	17	0	14.9	8	9	84.2
pthread-atomic	2	204	2	0	1.4	2	0	1.0	2	0	0.4	2	0	3.4	2	0	15.0
pthread-ext	8	780	8	0	1.0	8	0	0.3	7	1	12.0	8	0	0.3	8	0	47.2
pthread-lit	3	148	3	0	1.4	3	0	1.3	2	1	0.2	3	0	2.7	1	2	11.1
pthread-wmm	754	237700	754	0	1.1	754	0	1.2	754	0	0.5	754	0	6.1	753	1	78.1
Total	784	242917	781	3	1.4	784	0	1.6	781	3	2.9	784	0	6.2	772	12	77.6

formed very well in the recent SV-COMP verification competitions: both Lazy-CSeq and CIVL scored full marks. Note that the verfiers we used in the experiments described in the previous section performed noticeably worse.

Experimental Setup. For the bug-finding experiments, we used the 784 unsafe benchmarks from the SV-COMP collection. The total size of the benchmarks was approximately 240K lines of code. We ran the experiments on an otherwise idle machine with an Intel i7-3770 CPU 3.4GHz and 16GB of memory, running a Linux operating system with 64-bit kernel 4.4.0. We also set a 15GB memory limit and a 900s timeout for the analysis of each benchmark.

We used CBMC [4] (v5.4)[9] both as sequential backend (for UL-CSeq and Lazy-CSeq) and stand-alone bug-finding tool. It is a mature SAT-based bounded software model checker that uses a partial-order approach [1] to handle concurrent programs. We further used Lazy-CSeq [12] (v1.0),[10] a lazy sequentialization for bounded programs; CIVL [28] (v1.5),[11] a framework that uses a combination of explicit model checking and symbolic execution for verification; and SMACK [24] (v1.5.2),[12] a bounded software model checker that verifies programs up to a given bound on loop iterations and recursion depth. For all tools we used as loop unwinding and round bounds the (same) minimum values necessary to find all bugs in the given sub-category.

Results. Table 2 summarizes the results. We can see that our *proof*-oriented sequentialization does not actually impact negatively on our tool's *bug-finding* performance. UL-CSeq solves 781 of the 784 benchmarks, only three fewer than Lazy-CSeq (whose sequentialization specifically exploits the structure of bounded programs) or CIVL, and more than Smack. Analysis times are comparable across all tools, with the exception of the noticeably slower Smack. These results indicate that unwinding and lazy sequentialization can effectively be applied in either order.

[9] http://www.cprover.org/cbmc/download/cbmc-5-4-linux-64.tgz
[10] http://users.ecs.soton.ac.uk/gp4/cseq/files/lazy-cseq-1.0.tar.gz
[11] http://vsl.cis.udel.edu/lib/sw/civl/1.5/svcomp16/CIVL-1.5_2739_svcomp16.tgz
[12] http://soarlab.org/smack/smack-1.5.2-64.tgz

The UL-CSeq source code, static Linux binaries and benchmarks are available at http://users.ecs.soton.ac.uk/gp4/cseq/atva16.zip.

5 Related work

There is a wide range of approaches to verify concurrent programs. However, here we focus on more closely related sequentialization approaches. The idea of sequentialization was originally proposed by Qadeer and Wu [23]. The first scheme for an arbitrary but bounded number of context switches was given in [19]. Since then, several algorithms and implementations have been developed (see [5, 18, 3, 15, 14]).

Lazy sequentialization schemes have played an important role in the development of efficient tools. Their main feature is that they do not guess the original program's data but just its schedules and so induce less non-determinism and often simpler verification conditions. They also only explore reachable states of the original program, thus preserving the local invariants. This last property makes them suitable for static analysis [19]. The first such sequentialization was given in [15] for bounded context switching and extended to unboundedly many threads in [16, 17]. These schemes avoid the cross-product of the local states (since only one thread is tracked at any time of a computation) but require their recomputation at each context-switch. This is a major drawback when such a sequentialization is used in combination with bounded model-checking (see [8]). The scheme Lazy-CSeq [12] avoids such recomputations by flattening the programs and making the locals persistent, and achieves efficiency by handling context-switches with a very lightweight and decentralized control code.

All sequentializations mentioned above yield under-approximations of the multithreaded programs and thus (except for [16] that gives a sufficient condition to test completeness of the reached state space) are designed mainly for bug-finding. The new lazy sequentialization that we have designed in this paper is similar in spirit to Lazy-CSeq in that it injects lightweight control code to reposition the program counter on simulating a thread resumption but the injected control code itself is completely different. The main limitation of Lazy-CSeq's approach is that it assumes that each thread program counter uniquely identify its local state (which can be guaranteed for loop-free bounded programs), whereas our approach can handle a wider class of programs. First, we do not unwind loops and thus we allow for an exact simulation of unbounded loops. Second, we do not bound the number of context-switches in any explored computation. Our experiments show that the new control code is almost as effective as the goto-based control code used in Lazy-CSeq when using UL-CSeq with a bounded model checking backend, and performs very well when used to prove correctness of programs.

The only sequentialization that can be used to prove correctness of multithreaded programs is [7], but its approach is quite different from ours. It is closely related to the rely-guarantee style 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 non-determinism and recursive function calls even

if the original program does not contain any recursive calls. Moreover, the resulting sequentialization yields an overapproximation of the original program and thus cannot be used for bug-finding.

6 Conclusions and Future Work

We have presented a new sequentialization of concurrent programs that does not need to bound the number of context-switches or to unwind the loops. We only bound the number of threads and do not allow unbounded function call recursion. Noticeably, the resulting sequential program preserves all local invariants of the original program. In combination with suitable sequential verification tools it can thus be used both to find bugs (i.e., prove assertion violations) and prove concurrent programs safe.

We have implemented this sequentialization in the tool UL-CSeq within our framework CSeq and provided support for several backends. We have conducted a large set of experiments which have shown that UL-CSeq performs almost as efficiently as the best performing tools for bug-finding, and is very competitive for proving correctness. To the best of our knowledge this is the first approach that works well both as bug finder and to prove correctness for concurrent programs.

UL-CSeq is a first prototype implementation and has wide margins for improvements with fine tuning and optimizations. As future work, we plan to extend the range of programs that UL-CSeq can handle. We will modify the translation to lift some of the restrictions (e.g., the bounded number of thread creations), and will support new language features (e.g., other thread synchronization and communication primitives). We will also integrate further backends. Finally, we are working to extend our approach to support weak memory models implemented in modern architectures [26].

References

1. J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pages 141–157, 2013.
2. D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *TACAS*, pages 887–904, 2016.
3. S. Chaki, A. Gurfinkel, and O. Strichman. Time-bounded analysis of real-time systems. In *FMCAD*, pages 72–80, 2011.
4. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
5. B. Fischer, O. Inverso, and G. Parlato. CSeq: A concurrency pre-processor for sequential C verification tools. In *ASE*, pages 710–713, 2013.
6. B. Fischer, O. Inverso, and G. Parlato. CSeq: A sequentialization tool for C - (competition contribution). In *TACAS*, pages 616–618, 2013.
7. P. Garg and P. Madhusudan. Compositionality entails sequentializability. In *TACAS*, pages 26–40, 2011.
8. N. Ghafari, A. J. Hu, and Z. Rakamaric. Context-Bounded Translations for Concurrent Software: An Empirical Evaluation. In *SPIN*, pages 227–244, 2010.
9. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *CAV*, pages 343–361, 2015.

10. M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol - (competition contribution). In *TACAS*, pages 641–643, 2013.
11. O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. In *ASE*, pages 807–812, 2015.
12. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *CAV*, pages 585–602, 2014.
13. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A lazy sequentialization tool for C - (competition contribution). In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 398–401. Springer, 2014.
14. S. La Torre, P. Madhusudan, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI*, pages 211–222, 2009.
15. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, pages 477–492, 2009.
16. S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, pages 629–644, 2010.
17. S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing parameterized programs. In *FIT*, pages 34–47, 2012.
18. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV*, pages 427–443, 2012.
19. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.
20. L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.
21. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86. IEEE, 2004.
22. T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Unbounded Lazy-CSeq: A lazy sequentialization tool for C programs with unbounded context switches - (competition contribution). In *TACAS*, pages 461–463, 2015.
23. S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
24. Z. Rakamaric and M. Emmi. SMACK: decoupling source language details from verifier implementations. In *CAV*, pages 106–113, 2014.
25. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In *TACAS*, pages 551–565, 2015.
26. E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *FMCAD, to appear, http://eprints.soton.ac.uk/397759/*, 2016.
27. B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with impact. In *FMCAD*, pages 210–217, 2013.
28. M. Zheng, J. G. Edenhofner, Z. Luo, M. J. Gerrard, M. S. Rogers, M. B. Dwyer, and S. F. Siegel. CIVL: applying a general concurrency verification framework to C/pthreads programs (competition contribution). In *TACAS*, pages 908–911, 2016.