Bounded Model Checking of Multi-Threaded C Programs via Lazy Sequentialization

Omar Inverso[1], Ermenegildo Tomasco[1], Bernd Fischer[2], Salvatore La Torre[3], and Gennaro Parlato[1]

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

Abstract. Bounded model checking (BMC) has successfully been used for many practical program verification problems, but concurrency still poses a challenge. Here we describe a new approach to BMC of sequentially consistent C programs using POSIX threads. Our approach first translates a multi-threaded C program into a nondeterministic sequential C program that preserves reachability for all round-robin schedules with a given bound on the number of rounds. It then re-uses 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 prototype won the concurrency category of SV-COMP14. It solved all verification tasks successfully and was 30x faster than the best tool with native concurrency handling.

1 Introduction

Bounded model checking (BMC) has successfully been used to verify sequential software and to discover subtle errors in applications [11]. However, attempts to apply BMC directly to the analysis of multi-threaded programs (e.g., [18]) face problems as the number of possible interleavings grows exponentially with the number of threads and statements. Context-bounded analysis (CBA) methods [42, 35, 48] limit the number of context switches they explore and so fit well into the general BMC framework. They are empirically justified by work that has shown that errors manifest themselves within few context switches [44].

In this paper, we develop and evaluate a new technique for context-bounded BMC of multi-threaded C programs. It is based on *sequentialization*, an idea proposed by Qadeer and Wu [49] to reuse without any changes verification tools that were originally developed for sequential programs. Sequentializations can be implemented as a code-to-code translation of the input program into a corresponding nondeterministic sequential program. However, such translations alter 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.

The first sequentialization for an arbitrary but bounded number of context switches was given by Lal and Reps [42] (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 (*i*) each thread is run to completion, and (*ii*) each simulated round works on its own copy of the shared global memory. The initial values of all memory copies are nondeterministically guessed in the beginning (*eager* exploration), 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 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 [40].

Since the set of states reachable by a concurrent program can be much smaller than the whole state space explored by LR, *lazy* techniques that explore only the reachable states can be much more efficient. The first lazy sequentialization schema was given by La Torre, Madhusudan, and Parlato [35] (LMP). It also uses several copies of the shared memory, but in contrast to LR these copies are always computed and not guessed. 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 [35, 36] since they can re-use the summaries from previous rounds. However, it is a serious drawback for applying LMP in connection with BMC because it leads to exponentially growing formula sizes [29]. It is thus an open question whether it is possible to design an effective lazy sequentialization for BMC-based backends.

In this paper, we answer this question and design a new, surprisingly simple but effective lazy sequentialization schema that aggressively exploits the structure of bounded programs and works well with BMC-based backends. The resulting 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 function for each thread, where function calls and loops of the input program are inlined and unrolled, respectively [17]. In each round, the main driver calls each such thread simulation function; however, their execution does not repeat all the steps done in the previous rounds but instead jumps (in multiple hops) back to the stored program location where the previous round has finished. We keep the values of the thread-local variables between the different function activations (by turning them into `static` variables), which avoids their recomputation and thus the exponentially growing formula sizes observed by Ghafari et al. [29]. The size of the formulas is instead proportional to the product of the size of the original program, the number of threads and the number of rounds. 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. In contrast to LR, only reachable states of the input program are explored, and thus the translation requires no built-in error checks nor any special dynamic memory allocation handling, but can rely on the backend for these.

We have implemented this sequentialization in a prototype tool Lazy-CSeq that handles (*i*) the main parts of the POSIX thread API [31], such as dynamic thread creation and deletion, and synchronization via thread join, locks, and condition variables; (*ii*) the full C language with all its peculiarities such as different data types, dynamic memory allocation, and low-level programming features such as pointer arithmetics. Lazy-CSeq implements both bounding and sequentialization as source-to-source translations. The resulting sequential C program can be given to *any* existing verification tool for sequential C programs. We have tested Lazy-CSeq with BLITZ [16], CBMC [4], ESBMC [19], and LLBMC [24].

We have evaluated our approach and tool over the SV-COMP benchmark suite [9]. Lazy-CSeq [30] won the concurrency category of SV-COMP14, where it significantly outperformed both Threader [46], the previous winner in the concurrency category [8], and CBMC v4.5 [4], a mature BMC tool with recently added native concurrency support. The results thus justify the general sequentialization approach, and in contrast to the findings by Ghafari et al. [29], also demonstrate that a lazy translation can be more suitable for use in BMC than the more commonly applied LR translation [42, 22], as Lazy-CSeq also outperforms by orders of magnitude our own LR-based CSeq tool [25].

2 Bounded Multi-threaded C Programs

`pthread_t`: type of thread identifiers

`pthread_create(&t,&f,&arg)`:
creates a thread with unique identifier `t`, by calling function `f` with argument `arg`

`pthread_join(t)`:
suspends current thread until `t` terminates

`pthread_mutex_t`: type of mutex variables

`pthread_mutex_init(&m)`: creates an unlocked mutex `m`

`pthread_mutex_lock(&m)`: blocks until `m` is unlocked, then acquires and locks it

`pthread_mutex_unlock(&m)`:
unlocks `m` if called by the owning thread, returns an error otherwise

`pthread_mutex_destroy(&m)`: frees `m`

Fig. 1. Example Pthreads routines.

Multi-threaded C programs with Pthreads. Pthreads is a POSIX standard [31] for threads that defines a set of C functions, types and constants. In the following, we will refer to C programs that use the Pthreads API simply as multi-threaded C programs. In Fig. 1 we show the part of the API we use in our running example, in particular thread creation and join, and mutex primitives for thread synchronization; for simplicity, we omit the attribute and status arguments that various routines use. We also handle condition variables but omit them here. During the execution of a multi-threaded C program, we can assume that only one thread is *active* at any given time. Initially, only the `main` thread is active; new threads can be spawned from any thread by a call to `pthread_create`. Once created, a thread is added to the pool of inactive threads. At a *context switch* the current thread

is suspended and becomes inactive, and one of the inactive 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. Since threads can allocate memory dynamically using `malloc`, different threads can simultaneously access and alter shared dynamic data structures. We assume the *sequential consistency* memory model: when a shared variable is updated its new valuation is immediately visible to all the other threads [43]. We further assume that each statement is atomic. This is not guaranteed in general, but we can always rewrite each statement in a way that it involves only one operation on a shared variable by possibly using fresh temporary local variables, so that different interleavings always yield the same result as the original program with atomic executions. We say a statement is *visible* if its execution involves either a read or a write operation of a shared variable, and *invisible* otherwise.

```
pthread_mutex_t m; int c=0;
void P(void *b) {
  int tmp=(*b);
  pthread_mutex_lock(&m);
  if(c>0)
    c++;
  else {
    c=0;
    while(tmp>0) {
      c++; tmp--;
    }
  }
  pthread_mutex_unlock(&m);
}
void C() {
  assume(c>0);
  c--;
  assert(c>=0);
}
int main(void) {
  int x=1,y=5;
  pthread_t p0,p1,c0,c1;
  pthread_mutex_init(&m);
  pthread_create(&p0,P,&x);
  pthread_create(&p1,P,&y);
  pthread_create(&c0,C,0);
  pthread_create(&c1,C,0);
  return 0;
}
```

Fig. 2. Running Example

Running example. We use a producer/consumer system (see Fig. 2) as running example to illustrate our approach. It has 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` (*producer*) and two threads executing `C` (*consumer*). Each producer acquires `m`, increments `c`, and terminates by releasing `m`. Each consumer first checks 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.

Note that 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.

Bounded multi-threaded programs. Given a program, an assertion, and a depth bound, BMC translates the program into a formula that is satisfiable if and only if the assertion has a counterexample of the given depth or less. The resulting formula thus gives a static view of the bounded computations of the program. Since BMC only explores bounded computations, we can simplify the program before translating it; in particular, we can replace or *unwind* loops and function calls by appropriately guarded repeated copies of the corresponding loop and

```
bool active[T]={1,0,0,0,0};
int cs,ct,pc[T],size[T]={5,8,8,2,2};
#define G(L) assume(cs>=L);
#define J(A,B) if(pc[ct]>A||A>=cs) goto B;
pthread_mutex_t m; int c=0;
void P0(void *b) {
 0:J(0,1) static int tmp=(*b);
 1:J(1,2) pthread_mutex_lock(&m);
 2:J(2,3) if(c>0)
 3:J(3,4)   c++;
          else { G(4)
 4:J(4,5)   c=0;
            if(!(tmp>0)) goto _l1;
 5:J(5,6)   c++; tmp--;
            if(!(tmp>0)) goto _l1;
 6:J(6,7)   c++; tmp--;
            assume(!(tmp>0));
            _l1: G(7);
          } G(7)
 7:J(7,8) pthread_mutex_unlock(&m);
          goto _P0; _P0: G(8)
 8:       return;
}
void P1(void *b) {...}
void C0() {
 0:J(0,1) assume(c>0);
 1:J(1,2) c--;
          assert(c>=0);
          goto _C0; _C0: G(2)
 2:       return;
}
void C1() {...}
int main0() {
          static int x=1,y=5;
          static pthread_t p0,p1,c0,c1;
 0:J(0,1) pthread_mutex_init(&m);
 1:J(1,2) pthread_create(&p0,P0,&x,1);
 1:J(2,3) pthread_create(&p1,P1,&y,2);
 2:J(3,4) pthread_create(&c0,C0,0,3);
 3:J(4,5) pthread_create(&c1,C1,0,4);
          goto _main; _main: G(4)
 5:       return 0;
}
int main() {...see Fig. 4...}
```

Fig. 3. Translation of running example with unwinding bound of 2.

function bodies to yield a *bounded program*. Unwinding has one important property that is exploited by our approach: in the resulting bounded program, all jumps are forwards, and each statement is executed at most once in a run.

We implement unwinding with a few modifications for multi-threaded C programs. We do not unwind calls to any Pthreads routines and convert the program's `main` function into a thread. We also create and unwind a fresh copy of each function that appears as an argument in a call to `pthread_create` in the unwound program; these copies will be used to simulate the threads. If the original program can spawn multiple threads with the same start function we thus get multiple copies of that function. We assume the second argument of `pthread_create` is statically determined. We denote any programs with this structure as *bounded multi-threaded C programs*.

Fig. 3 shows the transformation result for the producer/consumer example (cf. Fig. 2), obtained using an unwind bound of 2. The black parts are the unwound original program, while the light gray parts are the instrumentation added by the sequentialization proper, as described in Section 3. Note that we get two separate unwound copies of each of the functions P and C, since the original program spawns two producer and two consumer threads.

3 Lazy Sequentialization for Bounded Programs

We now describe our code-to-code translation from a bounded multi-threaded program P (which can for example be obtained by the unwinding process sketched in the previous section) to a sequential program P_K^{seq} that simulates all round-robin executions with $K > 0$ rounds of P.

P consists by definition of $n + 1$ functions $f_0, \ldots, f_n$ (where f_0 denotes the unwound `main` function) and contains n calls to `pthread_create`, which create (at most) n threads with the start functions $f_1, \ldots, f_n$. Each start function is

associated with at most one thread, so that we can identify threads and functions. For round-robin executions, we fix an arbitrary schedule ρ by permuting $f_0, \ldots, f_n$; in each round we execute an arbitrary number of statements from each thread $\rho_0, \ldots, \rho_n$. For any fixed ρ our translation then guarantees that P fails an assertion in K rounds if and only if P_K^{seq} fails the same assertion. The translation thus preserves not only bounded reachability, but allows us to perform on the bounded multi-threaded program all analyses that are supported by the sequential backend tool.

P_K^{seq} is composed of a new function `main` and a thread simulation function f_i^{seq} for each thread f_i in P. The new `main` of P_K^{seq} calls, in the order given by ρ, the functions f_i^{seq} for K complete rounds. For each thread it maintains the label at which the context switch was simulated in the previous round and where the computation must thus resume in the current round. Each f_i^{seq} is essentially f_i with few lines of additional control code and with labels to denote the relevant context switch points in the original code. When executed, each f_i^{seq} jumps (in multiple hops) to the saved position in the code and then restarts its execution until the label of the next context switch is reached. We make the local variables persistent (i.e., of storage class `static`) such that we do not need to re-compute them when resuming suspended executions.

We describe our translation in a top-down fashion. We also convey a correctness proof and provide implementation details as we go along. We start by describing the (global) auxiliary variables used in the translation. Then, we give the details of the function `main` of P_K^{seq} , and illustrate how to construct each f_i^{seq} from f_i. Finally, we discuss how the Pthreads routines are simulated.

Auxiliary Data Structures. While simulating P, the sequentialized program P_K^{seq} maintains the data structures below; here `T` is a symbolic constant denoting the maximal number of threads in the program, i.e., $n + 1$.

- `bool active[T];` tracks whether a thread is active, i.e., has been created but not yet terminated. Initially, only `active[0]` is `true` since f_0^{seq} simulates the `main` function of P.
- `void* arg[T];` stores the argument used for thread creation.
- `int size[T];` stores the largest label used as jump target in the thread simulation functions f_i^{seq}.
- `int pc[T];` stores the label of the last context switch point for each thread simulation function.
- `int ct;` tracks the index of the thread currently under simulation.
- `int cs;` contains the label at which the next context switch will happen.

Note that the thread simulation functions f_i^{seq} read but do not write any of the data structures. `T` and `size[]` are computed by the unwinding phase and remain unchanged during the simulation. `arg[]` is set by (the simulation of) `pthread_create` and remains unchanged once it is set. `active[]` is set by `pthread_create` and unset by `pthread_exit`. `pc[]`, `ct`, and `cs` are updated by the driver.

Main Driver. Fig. 4 shows the new function `main` in P_K^{seq} , which drives the simulation. Each iteration of the loop simulates one entire round of a computation of P. The simulation of each thread $f_{\mathtt{ct}}$ invokes the corresponding simulation function $f_{\mathtt{ct}}^{seq}$ 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, here $0, \ldots, n$. For each active thread the driver thus executes the following steps: (*i*) nondeterministically guess the label for next context switch and store it in `cs`, (*ii*) check that the value is appropriate, (*iii*) simulate the thread from `pc[ct]` through to `cs`, and (*iv*) store `cs` in `pc[ct]`, since in the next round the computation must restart from this label.

```
void main(void) {
 for(r=1; r<=K; r++) {
  ct=0;
  // only active threads
  if(active[ct]) {
   // next context switch
   cs=pc[ct]+nondet_uint();
   // appropriate value?
   assume(cs<=size[ct]);
   // thread simulation
   fseq_0(arg[ct]);
   // store context switch
   pc[ct]=cs;
  }
  .........
  ct=n;
  if(active[ct]) {
   .........
}}}
```

Fig. 4. P_K^{seq} : `main()`.

The choice of an appropriate value for `cs` is simplified by the structure of P, more precisely, by the fact that the control flow always moves forward because 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 label in $f_{\mathtt{ct}}^{seq}$ that is added in the translation (which corresponds to the last possible context switch point in the code of the corresponding thread $f_{\mathtt{ct}}$). We stress that this guess is the only source of nondeterminism introduced by our translation.

Thread Translation. Each function f_i representing a thread in P is converted into a corresponding function f_i^{seq} in P_K^{seq} that is obtained as follows.

Turning local variables into static variables. Each thread f_i in P is simulated in P_K^{seq} by repeated calls to f_i^{seq}; each invocation executes a fragment of the code according to the context switch points that are guessed nondeterministically in the `main` function. Since each thread simulation function is only called once in each round, we can persist the thread-local variables between consecutive invocations (by turning them into `static` variables), and so avoid the inefficient recomputation of their values. However, uninitialized local variables may contain undefined values, while static variables are initialized to 0 by default. Thus, after the declaration of these variables we assign them with a nondeterministic value. For instance, `int tmp;` is turned into `static int tmp=nondet_int();`. This directly applies to all primitive C types. For arrays and structured types, we just do this at the level of the components.

Positioning and returning from a thread. When a function f_i^{seq} is called for the first time (i.e., in the first round), 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 the context switch point. When the control reaches the label guessed for the 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 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.

In contrast, the schema we present here, although at first perhaps counterintuitive, scales well when used together with BMC backends. We use `goto` statements in a way that avoids complex branching in the control flow. We use consecutive natural numbers as labels, starting with 0 for the first statement in each function, and label the other statements with numbers increasing in program order (see Fig. 3). To reduce the nondeterminism, we insert the labels (which are only used to simulate the context switches) only at the first statement, the last statement, and every visible statement. Note that this suffices, as we are only interested in assertion violations and in general properties involving only the shared memory and the local state of one thread.

Together with each label i (except for the last one) we also inject a conditional goto of the form `if( pc[ct]>`i `||` i `>=cs ) goto` *i+1*`;` in front of the statement. Note that the fragment *i+1* is evaluated at translation time, and thus simplifies to an integer literal that also occurs as label. When the thread simulation function tries to execute statements before the context switch of the previous round, or after the guessed context switch, the condition becomes true, and the control 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 when the guessed context switch label is reached, the fall-through to the last statement of the thread (which is by assumption always a `return`). Note that, whenever the control is between these two labels, the injected code is immaterial, and the statements of $f^{seq}_{\mathtt{ct}}$ in this part of the code are executed as in the original thread. We use a macro `J` to package up the injected control code (see Fig. 3).

As an example, consider the program in Fig. 3, and assume that `P0` is called (i.e., `ct`=1) with `pc[1]`=2 and `cs`=6. At label 0, the condition of the injected `if` statement holds, thus the `goto` statement is executed and the control jumps to label 1. Again, the condition is true, and then the control jumps to label 2. Now, the condition fails, thus the underlying code is executed, up to label 5. At label 6, the condition of the injected `if`-statement holds again, thus the control jumps to label 7, and then to label 8, thus reaching the `return` statement without executing any other code of the producer thread.

Handling branching statements. Eager schemas such as LR need to prune away guesses for the shared variables that lead to infeasible computations. A similar issue arises in our schema for the guesses of context switches. We remark that this is the only source of nondeterminism introduced by our translation.

Consider for example the `if-then-else` in `P0`, as shown in Fig. 3, and assume that `pc[0]`=2 and `cs`=3, i.e., in this round the sequentialized program is assumed to simulate (feasible) control flows between labels 2 and 3. However, if `c`$\leq$0, then

the program jumps from label 2 directly to label 4 in the `else`-branch; if we ignore the `G(4)` macro, the condition in the `if` statement inserted by `J(4,5)` would be tested, and since it would hold, the control flow would slide through to label 8, and return to the `main` driver, which would then set `pc[0]` to 3. In the next round, the computation would then duly resume from this label—which should be unreachable! Similar problems may occur when the context switch label is in the body of the `else`-branch, and with `goto` statements.

Note that assigning `pc` in the called function rather than in the `main` driver would fix this problem. 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 formulas computed by the backend tools. We therefore simply prune away simulations that would store unreachable labels in `pc`. For this, we use a simple guard of the form `assume(cs>=j);`, where `j` is the next inserted label in the code. 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 label in the actual code of the simulated thread. Again, we package this up in a macro called `G` (see Fig. 3).

This solution prunes away all spurious control flows. 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 `l` is in the code after the planned context switch point. But then the inserted `G` assumption fails, and the simulation is correctly aborted. The argument for `if` statements is more involved but follows the same lines. First consider that the planned context switch is the `then` branch. If the simulation takes the control flow into the `else` branch, then the guard fails because the first label in this branch is guaranteed to be greater than any label in the `then` branch, and the simulation is aborted. In the symmetric scenario, the guard after the `if` statement will do the job because `cs` is guaranteed to be smaller than the next label used as argument in the `G`. Note that the `J` macro at the last context switch point in the `else` branch (in the example `J(6,7)`) jumps over this guard so that it never prunes feasible control flows.

We stress that though the guess of context-switch point is done eagerly and thus we need to prune away infeasible guesses, the simulation of the input program is still done lazily. In fact, even when we halt a simulation at a guard, all the statements of the input program executed until that point correspond to a prefix of a feasible computation of the input program.

Simulation of Pthreads Routines. For each Pthreads routine we provide a verification stub, i.e., a simple standard C function that replaces the original implementation for verification purposes. Fig. 5 shows the stubs for the routines used in this paper. Variables of type `pthread_t` are simply mapped to integers, which serve as unique thread identifiers; all other relevant information is stored in the auxiliary data structures, as described in Sect. 3.

```
typedef pthread_t int;
int pthread_create(pthread_t *t,
      void *(*f)(void*), void *arg, int id)
  { active[id] = ACTIVE;
    arg[id]=arg;
    *t=id;
    return 0; }
int pthread_join(pthread_t t)
  { assume(pc[t]==size[t]); }
int pthread_exit(void *value_ptr)
  { return 0; }
typedef pthread_mutex_t int;
int pthread_mutex_init(pthread_mutex_t *m)
  { *m=FREE; }
int pthread_mutex_destroy(pthread_mutex_t *m)
  { *m=DESTROY; }
int pthread_mutex_lock(pthread_mutex_t *m)
  { assert(*m!=DESTROY);
    assume(*m==FREE); *m=t; }
int pthread_mutex_unlock(pthread_mutex_t *m)
  { assert(*m==t); *m=FREE; }
```

Fig. 5. Pthreads verification stubs

In `pthread_create` we simply set the thread's `active` flag and store the argument to be passed to the thread simulation function. Note that we do not need to store the thread start function, as the `main` driver calls all thread simulation functions explicitly, and that the `pthread_create` stub uses an additional integer argument `id` that serves as thread identifier and is copied into the `pthread_t` argument `t`. The `id` values are added to the `pthread_create` calls by the unwinding phase, corresponding to the order in which the calls occur in the unwound program.

In a real Pthreads implementation a thread invoking `pthread_join(t)` should be blocked until `t` is terminated. In the simulation a thread is terminated if it has reached the thread's last label, which corresponds to a `return` but there is no notion of blocking and unblocking. Instead, the stub for `pthread_join` uses an `assume` statement with the condition `pc[t]==size[t]` (which checks that the argument thread `t` has reached its last label) to prune away any simulation that corresponds to a blocking join. We can then see that this pruning does not change the reachability of error states. Assume that the joining thread `t` terminates after the invocation of `pthread_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 the `pthread_join`, and the invoking thread is scheduled to run only after the thread `t` is 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. We thus map the type `pthread_mutex_t` to integers, and define two constants `FREE` and `DESTROY` that have values different from any possible thread index. When we initialize or destroy a mutex we assign it the appropriate constant. If we want to lock a variable 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 `pthread_mutex_lock`. Similarly to the case of `pthread_join` we block the simulation if the lock is held by another thread. If a thread executes `pthread_mutex_unlock`, we first assert that the lock is held by the invoking thread and then set it to `FREE`.

4 Implementation and Evaluation

Implementation. We have implemented the sequentialization described in the previous sections in the prototype tool Lazy-CSeq. It takes as input a multi-threaded C program `p.c` and two parameters `r` and `u` representing the round and

unwind bounds, respectively, and produces the sequentialized program `_cs_p.c`. This is an un-threaded but nondeterministic bounded C program that can be processed by any analysis tool for sequential C (note not just BMC tools).

Lazy-CSeq can also be used as a wrapper around existing sequential verification backends. If the optional parameter `b` to specify the backend is given, it calls the tool to check for the reachability (within the given bounds) of the `ERROR` label in the original program. If the label is reachable, Lazy-CSeq returns in a separate file a counterexample trace (in the backend format) as witness to the error. The current version of Lazy-CSeq supports as backends the bounded model-checkers BLITZ, CBMC, LLBMC and ESBMC. However, since the implemented schema is very generic, the instrumentation for the different backends differs only in a few lines and other backends can be integrated easily.

Lazy-CSeq is implemented in Python on top of `pycparser` [7]. It consists of three main phases, where the input program is first parsed into an abstract syntax tree (AST) then transformed by repeatedly visiting the AST, and finally un-parsed into C program text. The transformation phase is implemented as a chain of several modules, each taking the program at some step in the overall translation process, and producing the program for next step. The modules can be grouped according to the following main phases of the translation, (*i*) preprocessing: merge, introduce workarounds to avoid known backend corner-cases, perform light input program simplifications; (*ii*) program bounding: perform function and loop unwinding, and thread duplication; (*iii*) instrumentation: insert code for simulation of the `pthread` API, concurrency simulation, and finalize the backend specific instrumentation.

Evaluation. We have evaluated our sequentialization approach with Lazy-CSeq on the benchmark set from the concurrency category of the SV-COMP14 [9] software verification competition. This set consists of 76 concurrent C files using the Pthread library, with a total size of about 4,500 lines of code. 20 of the files contain a reachable error location. We chose this benchmark set because it is widely used and all tools (but Corral) we compare against have been trained on this set for the competition.

We ran the experiments on an otherwise idle PC with a Xeon W3520 2.6GHz processor and 12GB of memory, running Linux with a 64-bit kernel (3.0.6). We set a 10GB memory limit and a 750s timeout for each benchmark.

The experiments are split into two parts. The first part concerns only the unsafe programs, where we investigate the effectiveness of several tools at finding errors. The second part concerns the safe programs, where we estimate whether limiting the round bounds to small values allows a more extensive exploration of programs in terms of increased values of loop unwinding bounds. The tools used in the experiments are BLITZ [16] (4.0), CBMC [4] (4.5 and 4.7), Corral [41], CSeq [25] (0.5) ESBMC [19] (1.22), LLBMC [24] (2013.1), and Threader [46].

Unsafe instances. The evaluation on unsafe instances is, again, split into two parts. We first evaluated the performance of Lazy-CSeq with the different sequential backend tools; the results are shown on the left of Table 1. Note that

Table 1. Bug-hunting performance (unsafe instances); $-^1$: timeout (750s); $-^2$: internal error; $-^3$: manual translation not done; $-^4$: test case rejected; $-^5$: unknown failure.

			Sequentialized version					Concurrent version					
	unwind	round	BLITZ 4.0	CBMC 4.5	CBMC 4.7	ESBMC 1.22	LLBMC 2013.1	CBMC 4.5	CBMC 4.7	Corral	CSeq 0.5	ESBMC 1.22	Threader
27_boop_simple_v	2	2	0.3	0.3	0.3	0.8	0.4	$-^5$	0.4	1.9	1.0	$-^1$	117.6
28_buggy_simple_loop1	2	1	0.2	0.2	0.2	0.3	0.3	$-^5$	0.3	0.8	0.2	624.7	0.3
32_pthread5_vs	2	2	0.4	0.2	0.3	0.2	0.2	$-^5$	0.8	2.2	$-^2$	$-^1$	$-^1$
40_barrier_v	4	1	0.2	0.3	0.2	0.3	0.3	$-^5$	0.6	0.8	$-^2$	$-^2$	0.7
49_bigshot_p	1	2	0.3	0.4	0.3	0.3	0.6	0.4	0.3	$-^3$	$-^4$	1.7	$-^2$
50_bigshot_s	1	2	0.3	0.4	0.3	0.3	0.6	$-^5$	0.5	$-^3$	$-^4$	4.0	$-^2$
53_fib_bench	5	5	36.6	1.1	1.0	15.2	2.1	0.7	1.8	5.8	6.3	31.1	6.9
55_fib_bench_longer	6	6	155.5	4.1	1.5	402.1	3.1	1.6	3.2	14.4	7.2	150.9	10.4
57_fib_bench_longest	11	11	$-^1$	425.7	214.0	$-^1$	$-^1$	645.9	75.2	$-^1$	$-^2$	$-^1$	54.3
61_lazy01	1	1	0.3	0.2	0.2	0.2	0.4	0.6	0.5	1.3	0.7	398.6	7.1
63_qrcu	1	2	1.4	0.6	0.8	0.7	$-^5$	0.6	0.7	5.8	$-^5$	$-^1$	$-^1$
65_queue	2	2	1.6	8.4	8.8	1.1	$-^1$	18.8	20.9	$-^3$	128.7	$-^1$	$-^2$
67_read_write_lock	1	2	0.5	0.3	0.3	0.4	$-^5$	0.4	0.4	1.8	2.6	$-^1$	38.4
69_reorder_2	2	1	0.3	0.6	0.6	$-^2$	1.3	1.0	0.7	1.3	$-^2$	$-^1$	2.4
70_reorder_5	4	1	0.4	0.8	0.9	$-^2$	3.3	2.1	0.7	1.9	$-^2$	$-^1$	3.5
72_sigma	16	1	1.4	7.6	7.8	$-^2$	73.0	$-^1$	219.1	$-^3$	$-^4$	$-^1$	$-^2$
73_singleton	1	3	0.7	0.6	0.5	0.5	$-^5$	$-^5$	1.6	$-^3$	$-^4$	$-^1$	$-^2$
75_stack	2	1	0.2	0.3	0.3	0.3	1.0	3.2	0.8	2.1	2.1	$-^1$	151.9
77_stateful01	1	1	0.2	0.2	0.2	0.3	0.5	0.7	0.7	2.0	0.7	$-^1$	0.9
82_twostage_3	2	1	0.3	0.7	0.8	$-^2$	8.0	9.1	4.9	3.6	$-^4$	$-^1$	$-^1$

only the backend run-times are given. The additional Lazy-CSeq pre-processing time, which is the same for every backend, is about one second for each file with our current Python prototype implementation. This could easily and substantially be reduced with a more efficient implementation. The results show that the tools were able to process most of the files generated by Lazy-CSeq's generic pre-processing, and found most of the errors. This is in marked contrast to our experience with CSeq, where the integration of a new backend required a substantial development effort, due to the nature of the LR schema. They also show that the different backends generally perform relatively uniformly, except for few cases where the performance gap is noticeably wide, probably due to a different handling of subtle corner-cases in the input from the backends. Both observations gives us further confidence that our approach is general and not bound to a specific verification backend tool.

We then compared the bug-hunting performances of Lazy-CSeq and several tools with different native concurrency handling approaches. CBMC and ESBMC are both BMC tools; CBMC uses partial orders to handle concurrency symbolically while ESBMC explicitly explores the different schedules [18]. CSeq is based on eager sequentialization, implementing a variant of LR, and uses CBMC as sequential backend. Corral uses a dynamic unwinding of function calls and loops, and implements abstractions on variables with the aim of discovering bugs faster. Threader, the winner in the Concurrency category of the SV-COMP13 competition, is based on predicate abstraction. For each tool (except Threader) we adjusted, for each file, all parameters to the minimum needed

to spot the error. The results, given on the right of Table 1, show that Lazy-CSeq is highly competitive. Of the "native" tools only CBMC is able to find all errors, but only with the most recent bug-fix version. All other tools time out, crash, or produce wrong results for several files. This shows how difficult it is to integrate concurrency handling into a verification tool—in contrast to the conceptual and practical simplicity of our approach. Moreover, for simple problems (with verification times around one second), Lazy-CSeq performs comparably with the fastest competitor. On the more demanding instances, Lazy-CSeq is almost always the fastest, except for the Fibonacci tests (53, 55 and 57) that are specifically crafted to force particularly twisted interleavings. In most cases (again except for the Fibonacci tests), Lazy-CSeq successfully finds the errors in all test cases using only three rounds, confirming that few context switches are sufficient to find bugs.

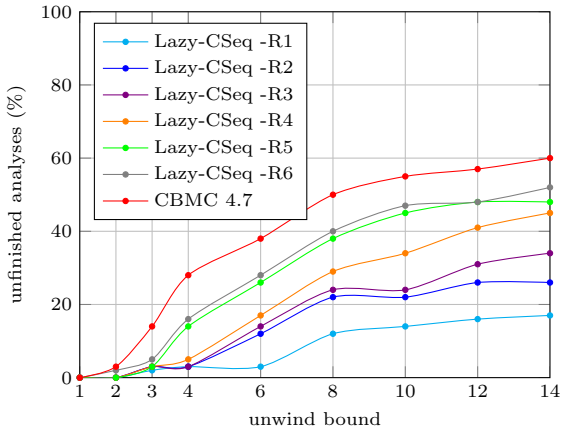

Fig. 6. Evaluation of safe benchmarks for increasing loop unwind bounds.

Safe instances. The evaluation on safe instances consisted in comparing Lazy-CSeq using CBMC v4.7 as backend with the best tool with native concurrency handling. We ran nine sets of experiments for CBMC, with unwinding bound to 1, 2, 3, 4, 6, 8, 10, 12, and 14, respectively. Notice that CBMC considers all possible interleavings. For Lazy-CSeq, we ran six repetitions of the sets, with a bound on the number of rounds from one to six, for each of the above unwinding values, respectively.

As shown in Fig. 6, we observe that CBMC starts performing worse than Lazy-CSeq, in terms of number of instances on which the analysis is completed, as we increase the loop unwinding bound. Overall, with the settings from the SV-COMP, Lazy-CSeq, is about 30x faster than CBMC for safe instances. This points out how the introduction of an extra parameter for BMC, i.e., the bound on the number of rounds, can offer a different, alternative coverage of the state-space. In fact, it allows larger loop unwindings, and therefore a deeper exploration of loops, than feasible with other methods.

5 Related Work

We already discussed the main sequentialization approaches [49, 42, 35] in the introduction. The lazy schema LMP was empirically shown to be more effective than LR in analyzing multithreaded Boolean programs [35, 34]. This work has

been extended to parametrized programs [36, 37] and used to prove correctness of abstractions of several Linux device drivers. Other sequentializations cope with the problem of handling thread creation [22, 12] and use different bounding parameters [39, 38, 53]. Ghafari et al. [29] observed that LMP is inefficient with BMC backends. LR has been implemented in CSeq for Pthreads C programs with bounded thread creation [25, 26], and in STORM that also handles dynamic memory allocation [40]. Poirot [47, 22] and Corral [41] are successors of STORM. Rek implements a sequentialization targeted to real-time systems [15]. None of the tools specifically targets BMC backends, though.

Biere *et al.* [11] introduced BMC to capitalize on the capacity of modern SAT/SMT solvers; see [10, 21] for a survey on BMC. The idea of loop unwinding in BMC of software was inspired by Currie *et al.* [20]. Several industrial-strength BMC tools have been implemented for the C language, including CBMC [17], ESBMC [18], EXE [14], F-SOFT [32], LLMBC [24], and SATURN [55].

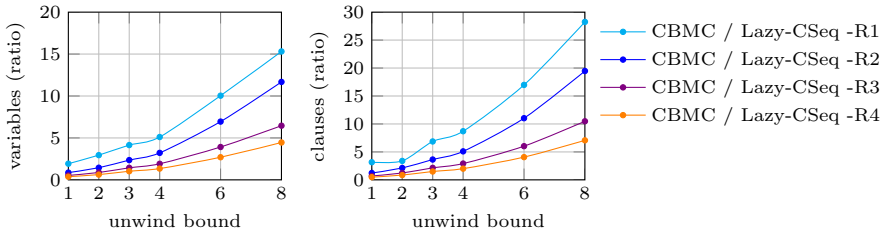

Fig. 7. Ratio between the number of variables on the original files and on the sequentialized files (left), and between the number of clauses on the original files and on the sequentialized files (right) (safe instances).

Several approaches [50, 28, 52, 51, 4] encode program executions as partial orders, in which each thread is an SSA program and operations on the shared memory are constrained by a global conjunct modeling the memory model. In [4] the authors argued that the formula size of their encodings on the considered benchmarks (among which are 36 from SV-COMP14) is smaller than those of [50, 28, 52, 51]. In our work, we have empirically evaluated the formula size of our encoding against CBMC (see Fig. 7). The main result is that our approach yields smaller formulas already for small unwind bounds, even for four rounds; with increasing unwind bounds (e.g., $n = 8$), CBMC's formulas contain 5x to 15x more variables and 5x to 25x more clauses, depending on the number of rounds.

6 Conclusions

We have presented a novel lazy sequentialization schema for bounded multi-threaded C programs that has been carefully designed to take advantage of BMC tools developed for sequential programs. We have implemented our approach in the prototype tool Lazy-CSeq as a code-to-code translation. Lazy-CSeq can also be used as a stand-alone model checker that currently supports four different BMC tools as backends. We validated our approach experimentally on the SV-COMP14 [9] concurrency benchmarks suite. The results show that:

– Lazy-CSeq can detect all the errors in the unsafe files, and is competitive with or even outperforms state-of-the art BMC tools that natively handle concurrency;
– it allows a more extensive analysis of safe programs with a higher number of loop unwindings by imposing small bounds on the number of rounds;
– it is generic in the sense that works well with different backends.

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 C programs such as array-bounds-check, division-by-zero, pointer-checks, overflow-checks, reachability of error labels and assertion failures, etc.

A core feature of our code-to-code translation 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 with the existing sequentializations (LR, LMP and the like, which can handle also unbounded programs) where multiple copies of the shared variables are used and assigned in the control code.

As consequence, we get three general benefits that set our work 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 [40]. 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 [25]). In contrast, Lazy-CSeq works well with different backends (currently BLITZ, CBMC, ESBMC, and LLBMC), and the only required tuning was 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.

Future work. We see two main future directions. One is to investigate optimizations to improve the performance of our approach. Partial order reduction techniques combined with symbolic model checking can improve the performance [54], and the approach of [33] for SAT-based analysis fits well in our sequentialisation schema. Also, a tuning of the backends on the class of programs generated in our translations could boost performance. It is well known that static code optimizations such as constant propagation are essential for performance gain in BMC. The other direction is to extend our approach to weak memory models implemented in modern architectures (see for example [5, 3]), and to other communication primitives such as MPI [27].

Lazy-CSeq homepage: http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html.

References

1. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE, 2013.
2. E. Ábrahám and K. Havelund, editors. *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*, volume 8413 of *LNCS*. Springer, 2014.
3. J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software Verification for Weak Memory via Program Transformation. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 512–532. Springer, 2013.
4. J. Alglave, D. Kroening, and M. Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
5. M. F. Atig, A. Bouajjani, and G. Parlato. Getting Rid of Store-Buffers in TSO Analysis. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 99–115. Springer, 2011.
6. T. Ball and M. Sagiv, editors. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011.
7. E. Bendersky. code.google.com/p/pycparser/.
8. D. Beyer. Second Competition on Software Verification - (Summary of SV-COMP 2013). In Piterman and Smolka [45], pages 594–609.
9. D. Beyer. Status report on software verification - (competition summary sv-comp 2014). In Ábrahám and Havelund [2], pages 373–388.
10. A. Biere. Bounded Model Checking. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.
11. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
12. A. Bouajjani, M. Emmi, and G. Parlato. On Sequentializing Concurrent Programs. In E. Yahav, editor, *SAS*, volume 6887 of *LNCS*, pages 129–145. Springer, 2011.
13. A. Bouajjani and O. Maler, editors. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *LNCS*. Springer, 2009.
14. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.
15. S. Chaki, A. Gurfinkel, and O. Strichman. Time-bounded Analysis of Real-time Systems. In P. Bjesse and A. Slobodová, editors, *FMCAD*, pages 72–80. FMCAD Inc., 2011.
16. C. Y. Cho, V. D'Silva, and D. Song. BLITZ: Compositional Bounded Model Checking for Real-world Programs. In *ASE* [1], pages 136–146.
17. E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

18. L. Cordeiro and B. Fischer. Verifying Multi-threaded Software using SMT-based Context-bounded Model Checking. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 331–340. ACM, 2011.
19. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.
20. D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic Formal Verification of DSP software. In *DAC*, pages 130–135, 2000.
21. V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
22. M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded Scheduling. In Ball and Sagiv [6], pages 411–422.
23. K. Etessami and S. K. Rajamani, editors. *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *LNCS*. Springer, 2005.
24. S. Falke, F. Merz, and C. Sinz. The Bounded Model Checker LLBMC. In *ASE* [1], pages 706–709.
25. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-processor for Sequential C Verification Tools. In *ASE* [1], pages 710–713.
26. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Sequentialization Tool for C - (Competition Contribution). In Piterman and Smolka [45], pages 616–618.
27. M. P. I. Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.
28. M. K. Ganai and A. Gupta. Efficient Modeling of Concurrent Systems in BMC. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *SPIN*, volume 5156 of *LNCS*, pages 114–133. Springer, 2008.
29. N. Ghafari, A. J. Hu, and Z. Rakamaric. Context-Bounded Translations for Concurrent Software: An Empirical Evaluation. In J. van de Pol and M. Weber, editors, *SPIN*, volume 6349 of *LNCS*, pages 227–244. Springer, 2010.
30. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A Lazy Sequentialization Tool for C - (Competition Contribution). In Ábrahám and Havelund [2], pages 398–401.
31. ISO/IEC. *Information technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, ISO/IEC/IEEE 9945:2009.* 2009.
32. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software Verification Platform. In Etessami and Rajamani [23], pages 301–306.
33. V. Kahlon, A. Gupta, and N. Sinha. Symbolic Model Checking of Concurrent Programs Using Partial Orders and On-the-Fly Transactions. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 286–299. Springer, 2006.
34. S. La Torre, P. Madhusudan, and G. Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. In M. Hind and A. Diwan, editors, *PLDI*, pages 211–222. ACM, 2009.
35. S. La Torre, P. Madhusudan, and G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In Bouajjani and Maler [13], pages 477–492.
36. S. La Torre, P. Madhusudan, and G. Parlato. Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

37. S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing Parameterized Programs. In S. S. Bauer and J.-B. Raclet, editors, *FIT*, volume 87 of *EPTCS*, pages 34–47, 2012.
38. S. La Torre, M. Napoli, and G. Parlato. Scope-Bounded Pushdown Languages. In A. Shur and M. Volkov, editors, *DLT*, LNCS. Springer, 2014.
39. S. La Torre and G. Parlato. Scope-bounded Multistack Pushdown Systems: Fixed-Point, Sequentialization, and Tree-Width. In D. D'Souza, T. Kavitha, and J. Radhakrishnan, editors, *FSTTCS*, volume 18 of *LIPIcs*, pages 173–184. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
40. S. K. Lahiri, S. Qadeer, and Z. Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In Bouajjani and Maler [13], pages 509–524.
41. A. Lal, S. Qadeer, and S. K. Lahiri. A Solver for Reachability Modulo Theories. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.
42. 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.
43. L. Lamport. A New Approach to Proving the Correctness of Multiprocess Programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.
44. M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In J. Ferrante and K. S. McKinley, editors, *PLDI*, pages 446–455. ACM, 2007.
45. N. Piterman and S. A. Smolka, editors. *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*, volume 7795 of *LNCS*. Springer, 2013.
46. C. Popeea and A. Rybalchenko. Threader: A Verifier for Multi-threaded Programs - (Competition Contribution). In Piterman and Smolka [45], pages 633–636.
47. S. Qadeer. Poirot - A Concurrency Sleuth. In S. Qin and Z. Qiu, editors, *ICFEM*, volume 6991 of *LNCS*, page 15. Springer, 2011.
48. S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
49. S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. In W. Pugh and C. Chambers, editors, *PLDI*, pages 14–24. ACM, 2004.
50. I. Rabinovitz and O. Grumberg. Bounded Model Checking of Concurrent Programs. In Etessami and Rajamani [23], pages 82–97.
51. N. Sinha and C. Wang. Staged Concurrent Program Analysis. In G.-C. Roman and K. J. Sullivan, editors, *SIGSOFT FSE*, pages 47–56. ACM, 2010.
52. N. Sinha and C. Wang. On Interference Abstractions. In Ball and Sagiv [6], pages 423–434.
53. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings - (Competition Contribution). In Ábrahám and Havelund [2], pages 402–404.
54. C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic Pruning of Concurrent Program Executions. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 23–32. ACM, 2009.
55. Y. Xie and A. Aiken. Saturn: A SAT-Based Tool for Bug Detection. In Etessami and Rajamani [23], pages 139–143.