# Concurrent Program Verification with Lazy Sequentialization and Interval Analysis*

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] Division of Computer Science, Stellenbosch University, South Africa
[3] Dipartimento di Informatica, Università degli Studi di Salerno, Italy

**Abstract.** Lazy sequentialization has proven to be one of the most effective techniques for concurrent program verification. The Lazy-CSeq sequentialization tool performs a "lazy" code-to-code translation from a concurrent program into an equivalent non-deterministic sequential program, i.e., it preserves the valuations of the program variables along its executions. The obtained program is then analyzed using sequential bounded model checking tools. However, the sizes of the individual states still pose problems for further scaling. We therefore use abstract interpretation to minimize the representation of the concurrent program's (shared global and thread-local) state variables. More specifically, we run the Frama-C abstract interpretation tool over the programs constructed by Lazy-CSeq to compute overapproximating intervals for all (original) state variables and then exploit CBMC's bitvector support to reduce the number of bits required to represent these in the sequentialized program. We have implemented this approach in the last release of Lazy-CSeq and demonstrate the effectiveness of this approach; in particular, we show that it leads to large performance gains for very hard verification problems.

## 1 Introduction

Concurrent programming is becoming more important as concurrent computer architectures such as multi-core processors are becoming more common. However, concurrent program verification remains a stubbornly hard problem, due to the large number of interleavings that a verifier must analyze. Techniques such as testing that analyze interleavings individually struggle to find "rare" concurrency bugs, i.e., bugs that manifest themselves only in a few of the interleavings. Techniques that use symbolic representations to analyze all interleavings collectively typically fare better, especially for rare concurrency bugs.

Sequentialization has proven to be one of the most effective symbolic techniques for concurrent program verification, shown for example by the fact that most concurrency medals in the recent SV-COMP program verification competitions were won by various sequentialization-based tools [31, 17, 32, 35]. It is based on the idea of translating concurrent programs into non-deterministic sequential programs that (under certain

* Partially supported by EPSRC EP/M008991/1, INDAM-GNCS 2016, and MIUR-FARB 2014-2016 grants.

assumptions) behave equivalently, so that the different interleavings do not need to be treated explicitly during verification and, consequently, sequential program verification methods can be reused. Eager sequentialization approaches [24, 10, 33] guess the different values of the shared memory before the verification and then simulate (under this guess) each thread in turn. They can thus explore infeasible computations that need to be pruned away afterwards, which requires a second copy of the shared memory, and so increases the state space. Lazy sequentialization approaches [20] instead guess the context switch points and (re-) compute the memory contents, and thus explore only feasible computations. They also preserve the sequential ordering of the interleaved thread executions and thus the local invariants of the original program. Lazy approaches, such as Lazy-CSeq, are thus typically more efficient than eager approaches.

Lazy-CSeq [15, 17] is implemented as a source-to-source transformation in the CSeq framework [9]: it reads a multi-threaded C program that uses the Pthreads API [18], applies the translation sketched in Section 2 and described in more detail in [16], and outputs the resulting non-deterministic sequential C program. This allows us to use any off-the-shelf sequential verification tool for C as backend, although we have achieved the best results with CBMC [6].

Lazy-CSeq's translation is carefully designed to introduce very small memory overheads and very few sources of nondeterminism, so that it produces simple formulas. It also aggressively exploits the structure of bounded programs and works well with backends based on bounded model checking (BMC). It is very effective in practice, and scales well to larger and harder problems. Currently, Lazy-CSeq is the only tool able to find bugs in the two hardest known concurrency benchmarks, `safestack` [37] and `eliminationstack` [12]. However, for such hard benchmarks the computational effort remains high, and for `eliminationstack` Lazy-CSeq requires close to six hours on a standard machine.

A detailed analysis of these benchmarks shows that a large fraction of the overall effort is not spent on finding the right interleavings that expose the bugs, but on finding the right values of the original (concurrent) programs' shared global and individual thread-local variables. We found that this is caused by the unnecessarily large number of propositional variables (reflecting the default bit-widths of the variables in C) that CBMC uses. In an experiment, we manually reduced this to the minimum required to find the bug (three bits in the case of `safestack`), which leads to a 20x speed-up. This clearly indicates the potential benefits of such a reduction.

In this paper, we describe an automated method based on abstract interpretation to reduce the size of the concurrent programs' shared global and thread-local state variables. More specifically, we run the Frama-C abstract interpretation tool [2] over the sequentialized programs constructed by Lazy-CSeq to compute overapproximating intervals for these variables. We use the intervals to minimize the representation of the (original) state variables, exploiting CBMC's bitvector support to reduce the number of bits required to represent these in the sequentialized program, and, hence, ultimately in the formula fed into the SAT solver. Note that this approach relies on two crucial aspects of Lazy-CSeq's design. On the theoretical side, we rely on the fact that lazy sequentializations only explore feasibly computations to infer "useful" invariants that actually speed up the verification; our approach would not work with eager sequentializations

because they leave the original state variables unconstrained, leading to invariants that are too weak. On the practical side, we rely on the source-to-source approach implemented in Lazy-CSeq, in order to re-use an existing abstract interpretation tool.

We have implemented this approach in the last release of Lazy-CSeq and demonstrate its effectiveness. We show that the effort for the abstract interpretation phase is relatively small, and that the inferred intervals are tight enough to be useful in practice and lead to large performance gains for very hard verification problems. In particular, we demonstrate a 5x speed-up for `eliminationstack`.

## 2 Verification approach

In this section we illustrate the verification approach we propose in this paper. We recall multi-threaded programs and context-bounded analysis before we give some details on the two pillars of our approach: the lazy sequentialization performed by the tool Lazy-CSeq [16] and the value analysis performed by the tool Frama-C [2].

### 2.1 The general scheme

Verification by sequentialization is based on a translation of the input multi-threaded program into a corresponding sequential program which is then analysed by an off-the-shelf backend verification tool for sequential programs. We improve on this by applying value analysis to the sequentialized program to derive overapproximating intervals for the original program variables and using these intervals to reduce the number of bits used to represent each variable in the backend verification tool. In particular, our approach works in four steps:

1. We compute a sequential program that preserves the reachable states of the input program up to a given number of thread context-switches (*sequentialization*).
2. We compute the bounds on the values that the variables can store along any computation of the sequential program (*value analysis*).
3. We transform the sequentialized program by changing the program variables of numerical type (i.e., `integer` and `double`) to bitvector types of sizes determined by the results of the value analysis (*model refinement*).
4. We verify the resulting sequential program (*verification*).

In sequentializations the control nondeterminism of the original program is replaced by data nondeterminism and thread invocations are replaced by function calls. Lazy sequentialization methods also preserve the sequential ordering of the interleaved thread executions, and thus also the local invariants of the original program. This property ensures that the value analysis can produce good overapproximations of the variable ranges (i.e., tight intervals). We instantiate our approach with the lazy sequentialization implemented in Lazy-CSeq, and the value analysis given by Frama-C.

### 2.2 Multithreaded programs

We consider standard multi-threaded programs with shared variables, dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. We omit the formal definition of the syntax and the semantics of multi-threaded programs which is standard [16]. We adopt a C-like syntax in our examples.

We assume that each multi-threaded program contains a function `main`, which is the starting function of the only thread that exists in the beginning. We call this the *main thread*. As usual, there are no calls to `main` and that no other thread can be created that uses `main` as starting function.

We assume a sequentially consistent semantics by interleaving, thus only one of the *executable* threads can be *active* (i.e., running) at any given time. Initially, only the *main thread* is active; new threads can be spawned from any thread by invoking `create`. Once created, a thread is added to the pool of the executable threads. At a *context switch* the currently active thread is suspended (but remains executable), and one of the executable threads is resumed and becomes the active thread. When a thread becomes active it resumes from the point where it was suspended (or from the beginning, if it becomes active for the first time). For ease of presentation, we assume that each statement is executed atomically.

Each *thread configuration* is a triple $\langle locals, pc, stack \rangle$, where *locals* is a valuation of the local variables, *pc* is the *program counter* that tracks the currently executing statement, and $stack$ is a stack of function calls that works as usual. A *configuration* of a multithreaded program is a tuple of thread configurations along with valuation of the global variables that are shared by all threads.

A *context* is a possibly empty sequence of statements that consecutively executed by a thread in a computation. We underapproximate the behavior of a concurrent program by allowing computations up to a given round of a round-robin schedule (*bounded round-robin computations*). In such computations, each executable thread executes exactly one context for each round and in all considered rounds threads are always scheduled according to a same schedule (note that this is not a real restriction since a thread can execute zero statements in a round).

As an example consider the multithreaded program in Fig. 1. It encodes a producer/consumer system. The program 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.

```
mutex m; int c=0;

void P(void *b) {
  int tmp=(*b);
  lock(&m);
  if(c>0)
    c++;
  else {
    c=0;
    while(tmp>0) {
      c++; tmp--;
    }
  }
  unlock(&m);
}
```

```
void C() {
  assume(c>0);
  c--;
  assert(c>=0);
}

int main(void) {
  int x=1,y=5;
  thread p0,p1,c0,c1;
  mutex_init(&m);
  create(&p0,P,&x);
  create(&p1,P,&y);
  create(&c0,C,0);
  create(&c1,C,0);
  return 0;
}
```

**Fig. 1.** Producer/Consumer program.

### 2.3 Lazy sequentialization schema

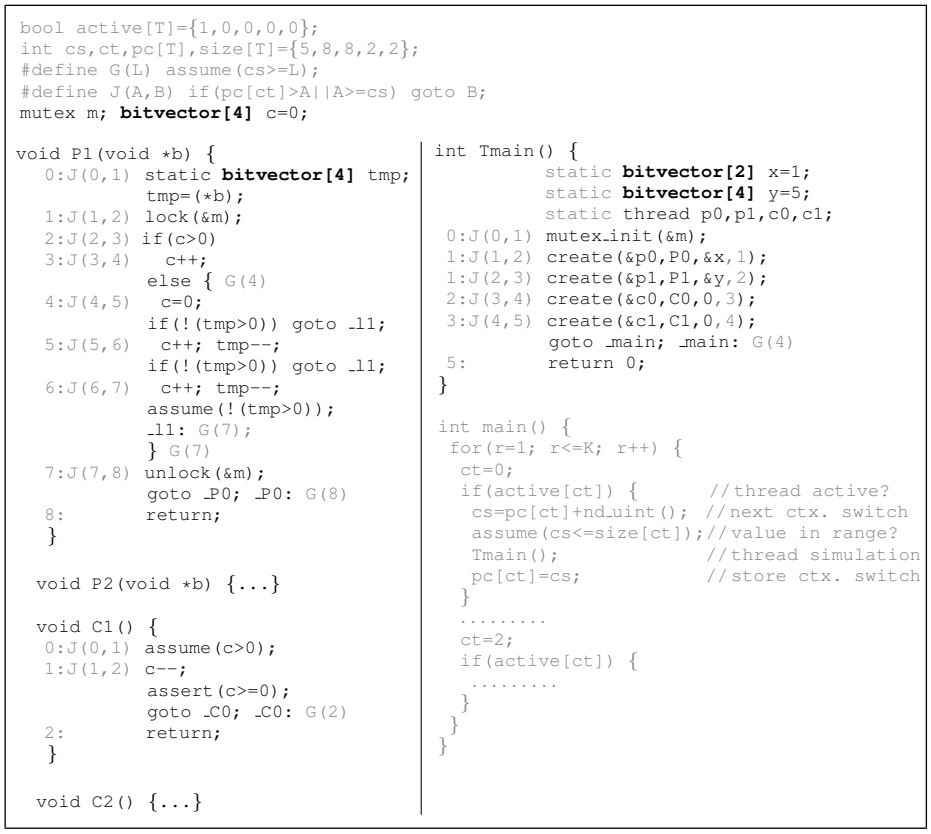
In this section, we briefly recall the lazy sequentialization encoding that we use in our approach. This is implemented in our Lazy-CSeq tool [16, 15]. We assume that a concurrent program $P$ consists of $n + 1$ functions $f_0, \ldots, f_n$, where $f_0$ denotes the `main` function, and that $P$ creates at most $n$ threads, with the respective start functions $f_1, \ldots, f_n$. Moreover, no function $f_i$ contains loops. Note that these assumptions can easily be enforced by bounding the programs in BMC fashion and cloning the start functions, if necessary (*bounded multi-threaded program*). Since each start function is thus associated with at most one thread, we can identify threads and (start) functions.

Consider a bounded multithreaded program $P$ as described above. In our analysis of bounded round-robin computations, we fix a number of rounds $K$ and an arbitrary schedule $\rho$ by permuting the functions $f_0, \ldots, f_n$ that form the starting program. Thus, the lazy sequentialization of $P$ yields a sequential program $P'$ such that $P$ fails an assertion in $K$ rounds if and only if $P'$ fails the same assertion. $P'$ is composed of a new function `main` and a thread simulation function $T_i$ for each thread $f_i$ in $P$. The lazy sequentialization of the Producer/Consumer program given in Fig. 1 generated by Lazy-CSeq (with two loop unwindings) is the code shown in Fig. 2 with the bitvector type in bold replaced by the integer type. In the figure, we emphasize the code injected by Lazy-CSeq showing in black the original code and in gray the injected code.

Note that the sequential verification of $P'$ relies on stubs provided by Lazy-CSeq. $P'$ thus uses a slightly modified version of the Pthreads API. For example, the `create` stub takes an additional argument for the (statically known) id of the calling thread; see [16] for details.

The new `main` of $P'$ is a driver that calls, in the order given by $\rho$, the functions $T_i$ for $K$ complete rounds. For each thread it maintains the label at which the context switch was simulated in the previous round and where the computation must thus resume in the current round. Moreover, before each call of $T_i$, the label at which the control will context-switch out is nondeterministically guessed.

Each $T_i$ is essentially $f_i$ with few lines of injected control code and with labels to denote the relevant context-switch points in the original code. When executed, each $T_i$



```
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;
mutex m; bitvector[4] c=0;

void P1(void *b) {
  0:J(0,1) static bitvector[4] tmp;
           tmp=(*b);
  1:J(1,2) 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) unlock(&m);
           goto _P0; _P0: G(8)
  8:       return;
  }

 void P2(void *b) {...}

 void C1() {
  0:J(0,1) assume(c>0);
  1:J(1,2) c--;
           assert(c>=0);
           goto _C0; _C0: G(2)
  2:       return;
  }

 void C2() {...}

int Tmain() {
          static bitvector[2] x=1;
          static bitvector[4] y=5;
          static thread p0,p1,c0,c1;
 0:J(0,1) mutex_init(&m);
 1:J(1,2) create(&p0,P0,&x,1);
 1:J(2,3) create(&p1,P1,&y,2);
 2:J(3,4) create(&c0,C0,0,3);
 3:J(4,5) create(&c1,C1,0,4);
          goto _main; _main: G(4)
 5:       return 0;
}

int main() {
 for(r=1; r<=K; r++) {
  ct=0;
  if(active[ct]) {      //thread active?
   cs=pc[ct]+nd_uint(); //next ctx. switch
   assume(cs<=size[ct]);//value in range?
   Tmain();             //thread simulation
   pc[ct]=cs;           //store ctx. switch
  }
  .........
  ct=2;
  if(active[ct]) {
   .........
  }
 }
}
```

**Fig. 2.** Lazy-CSeq sequentialized code of the Consumer/Producer program modified according to the value analysis by Frama-C.

jumps (in multiple hops) to the saved position in the code and then restarts its execution until the label of the next context switch is reached. This is achieved by the `J`-macro. Context-switching at branching statements requires some extra care; see [16] for details. We also make the local variables persistent (i.e., `static`) such that we do not need to re-compute them when resuming suspended executions.

We use some additional data structures and variables to control the context-switching in and out of threads as described above. The data structures are parameterized over $\mathtt{T} \leq n$ which denotes the maximal number of threads activated in $P$'s executions. We keep track of the active threads (`active`), the arguments passed in each thread creation (we omitted it in our example since the considered thread functions have no arguments), the largest label used in each $T_i$ (`size`), the current label of each $T_i$ (`pc`), and for the currently executed thread its index (`ct`) and the context-switch point guessed in the main driver before calling the thread (`cs`).

Note that the control code that is injected in the translation is designed such that each $T_i$ reads but does not write any of the additional data structures. These are updated only in the main driver and in the portions of code simulating the API functions con-

cerning thread creation and termination. This introduces fewer dependencies between the injected code and the original code, which typically leads to a better performance of the backend tool (e.g., for BMC backends this results in smaller formulas).

### 2.4 Value analysis

The value analysis of programs aims at computing supersets of possible values for all the variables at each statement of the analyzed program. All executions of the instruction that are possible starting from the function chosen as the entry-point of the analysis are taken into account.

The value analysis of Frama-C [2] is a plug-in based on abstract interpretation and is capable of handling C programs with pointers, arrays, structs, and type casts. Abstract interpretation links the set of all possible executions of a program (*concrete semantics*) to a more coarse-grained semantics (*abstract semantics*). Frama-C explores symbolic execution of the program, translating all operations into the abstract semantics. For the soundness of the approach, any transformation in the concrete semantics must have an abstract counterpart that captures all possible outcomes of the concrete operation. Thus, when several execution paths are possible, e.g., when analyzing an if-statement, all branches need to be explored and then at the point where the branches join together, e.g., after the if statement, the lattice-theoretic join of the results along each branch is taken. In Frama-C this is implemented as the smallest interval that encloses all intervals computed along the individual branches. For-loops require additional care, since value analysis is not guaranteed to terminate. However, this aspect is not relevant to our approach as the output of Lazy-CSeq does not contain loops but only bounded programs.

As an example, consider the sequentialization of the Producer/Consumer program generated by Lazy-CSeq. On this program, Frama-C computes for the integer shared variable `c` and the integer local variable `tmp` of producer threads the interval of values $[-2, 5]$. Thus, in the verification analysis we can safely reduce the size of these integer variables to $4$ bits (one bit is for the sign) instead of the standard 32 bits used for the type `int`. Therefore, we can transform the sequentialized program accordingly by replacing the type `int` in the declaration of these variables with the bitvector type. The resulting sequentialized code is shown in Fig. 2.

## 3 Implementation

We have implemented our approach in a relatively straightforward way within the CSeq framework, as an extension (Lazy-CSeq+ABS) to the existing Lazy-CSeq implementation. CSeq consists of a number of independent Python modules that provide different program transformations (e.g., function inlining, loop unrolling) as well as parsing and unparsing [15]. These modules can be configured and composed easily to implement different sequentializations as source-to-source transformation tools.

The architecture of Lazy-CSeq+ABS is shown in Fig. 3. We now briefly illustrate the architecture of Lazy-CSeq (shown in Figure 3 in blue), and then incrementally describe how we have extended it. Lazy-CSeq consists of a chain of modules:

- a module that preprocesses the source files merging them into a single file;

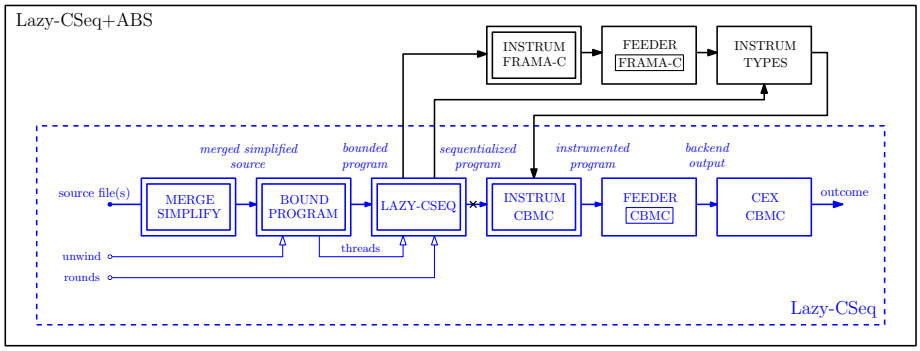



**Fig. 3.** Lazy-CSeq+ABS Architecture

- a module that simplifies the syntax;
- a module for unrolling loops and inlining functions to produce a bounded program;
- a module that implements the Lazy-CSeq sequentialization [16] which produces a backend-independent sequentialized file;
- a module to instrument the sequentialized file for a specific backend (in our case, CBMC);
- two wrappers, one for backend invocation (FEEDER), and another one that generates counterexamples (CEX).

We reuse all these module as follows. The output of the LAZY-CSEQ module, which produces a backend-independent sequentialized file, is now instrumented for Frama-C by replacing the nondeterministic choice, assert, and assume statements with the equivalent Frama-C primitives. The next module consists of a wrapper that invokes Frama-C on the instrumented code. The result of this analysis, which reports for each variable a lower and upper bound on the value that the variable can take along any execution of the bounded program, is used by the INSTRUM TYPES module to compute the minimal number of bits required for each program variable. This module then replaces the original scalar type of each variable, say `x`, in the sequentialized file produced by the LAZY-CSEQ module with the CBMC type `__CPROVER_bitvector[`$i$`]` where $i$ is the number of bits computed for `x`. The resulting program is then passed to the INSTRUM module and the remaining process is the same as Lazy-CSeq. The additional modules of Lazy-CSeq+ABS are implemented in Python as well.

Lazy-CSeq+ABS is publicly available at: `http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html`.

## 4 Experimental Evaluation

In this section we report on a large number of experiments where we compare Lazy-CSeq v1.0 and Lazy-CSeq+ABS with the aim of demonstrating the effectiveness of the approach proposed in this paper. The results of this empirical study show that Lazy-CSeq+ABS is substantially more efficient on complex benchmarks, i.e., larger programs

| | | | Lazy-CSeq | | | | Lazy-CSeq+Abs | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | CBMC | | | | CBMC | | | | Frama-C | Total |
| Subcategory | #files | LOC | sec. | GB | #vars | #clauses | sec. | GB | #vars | #clauses | sec. | sec. |
| pthread | 17 | 4085 | 34.7 | 84.9 | 89317.7 | 336250.1 | 18.0 | 66.8 | 47961.4 | 184287.8 | 5.5 | 23.5 |
| pthread-atomic | 2 | 204 | 1.7 | 33.3 | 9131.0 | 29186.0 | 1.8 | 46.2 | 6259.5 | 17936.0 | 0.9 | 2.7 |
| pthread-ext | 8 | 780 | 6.5 | 358.4 | 647840.1 | 2654905.9 | 4.5 | 83.1 | 89718.9 | 423391.8 | 1.1 | 5.5 |
| pthread-lit | 3 | 123 | 1.9 | 38.3 | 9993.0 | 31206.7 | 1.9 | 49.3 | 5882.0 | 16421.0 | 1.2 | 3.1 |
| pthread-wmm | 754 | 236496 | 2.0 | 31.4 | 2427.1 | 5668.3 | 2.2 | 46.1 | 2402.2 | 5578.8 | 0.9 | 3.1 |

**Table 1.** Experiments on SV-COMP unsafe benchmarks

that contain rare bugs. Furthermore, for simple benchmarks, which Lazy-CSeq v1.0 already solves quickly, the overhead of running Frama-C on is often negligible.

In our experiments we use CBMC[1] v5.6 as sequential backend for both Lazy-CSeq v1.0 and Lazy-CSeq+ABS. CBMC encodes symbolically the executions of the bounded program into a CNF formula that is then checked by the SAT solver MINISAT v2.2.1. Furthermore, we use Frama-C[2] v13-Aluminium for Lazy-CSeq+ABS. In the remainder of the paper we denote Lazy-CSeq v1.0 simply as Lazy-CSeq.

We have performed the experiments on an otherwise idle machine with a Xeon W3520 2.6GHz processor and 12GB of memory, running a Linux operating system with 64-bit kernel 2.6.32.

Since we use a BMC tool as a backend, we individually set the parameters for the analysis (i.e., loop unwinding, function inlining and rounds of computations) for each unsafe benchmark (i.e., program with a reachable error location) to the minimum values required to expose the corresponding error.

### SV-COMP'16 benchmarks

The first series of experiments is conducted on the benchmark set from the Concurrency category of the Software Verification Competition (SV-COMP'16) held at TACAS. This set consists of 1005 concurrent C files using the Pthread library, with a total size of about 277,000 lines of code. 784 of the files contain a reachable error location. We use this benchmark set because it is widely used and many state-of-the-art analysis tools have been trained on it. Moreover, it offers a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms.

Table 1 reports on the experiments for the unsafe benchmarks and Table 2 on those for the safe ones. Each row of these two tables summarizes the experiments by grouping them into sub-categories. For each sub-category, we report the number of files and the total number of lines of code in that sub-category. The tables also gather the results of the experiments performed using Lazy-CSeq v1.0 and Lazy-CSeq+ABS on these benchmarks. For the CBMC backend analysis, we indicate with *time* the average time in seconds, *mem* the average memory peak usage expressed in MB, and with *#vars* and *#clauses* the average number of variables and clauses of the CNF formula produced by CBMC. Furthermore, only for Lazy-CSeq+ABS, the column *Frama-C* denotes the average time in seconds taken by Frama-C for the value analysis.

[1] CBMC: `http://www.cprover.org/cbmc/`

[2] Frama-C: `http://frama-c.com`

| | | | Lazy-CSeq | | | | Lazy-CSeq+Abs | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | CBMC | | | | CBMC | | | | Frama-C | Total |
| Subcategory | #files | LOC | sec. | GB | #vars | #clauses | sec. | GB | #vars | #clauses | sec. | sec. |
| pthread | 15 | 1285 | 172.4 | 1124.4 | 1732068.1 | 7270420.0 | 98.6 | 945.3 | 1424912.1 | 6004425.3 | 8.4 | 107.0 |
| pthread-atomic | 9 | 1136 | 2.7 | 37.9 | 18947.4 | 67709.0 | 2.9 | 47.7 | 16611.9 | 58334.0 | 2.0 | 4.9 |
| pthread-ext | 45 | 3683 | 71.7 | 876.8 | 1660452.6 | 6949976.3 | 49.4 | 552.6 | 937205.0 | 4036919.5 | 2.2 | 51.6 |
| pthread-lit | 8 | 432 | 5.8 | 43.7 | 15207.3 | 57356.2 | 4.9 | 51.6 | 11094.2 | 42161.0 | 1.0 | 5.9 |
| pthread-wmm | 144 | 29282 | 1.6 | 31.5 | 3154.7 | 9420.2 | 1.6 | 45.7 | 3065.4 | 9084.0 | 0.9 | 2.5 |

**Table 2.** Experiments on SV-COMP safe benchmarks

The two tables paint a relatively clear picture in terms of runtimes. For the larger and more complex benchmark categories pthread (both safe and unsafe instances) and pthread-ext (only safe instances), where Lazy-CSeq takes on average more than 30 seconds, the effort for the abstract interpretation is relatively small (approx. 5%-20% of the original CBMC runtimes) and is easily recouped, so that we see overall performance gains of approx. 25%-40%. For the simpler benchmarks, Frama-C takes almost as much time as Lazy-CSeq on its own, without substantially reducing the size or complexity of the problems. In most cases we thus see some slow-downs, but in absolute terms these are small (approx. 2 seconds) and outweighed by the larger gains on the more complex benchmarks.

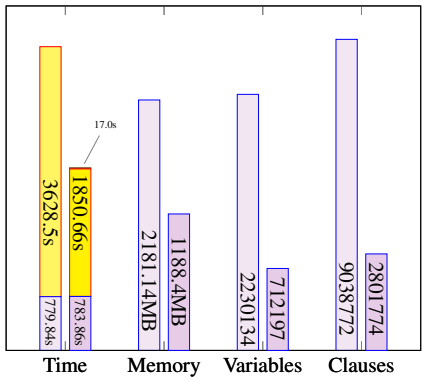
A very similar picture emerges for peak memory consumption—reductions of approx. 15%-75% for the larger benchmarks that outweigh the relatively large but absolutely small increases for the smaller benchmarks.

If we look at the number of variables and clauses, we can see how effective our approach is in reducing the size of the induced SAT problems. In most case we see a reduction of approx. 30% to 50%. These reductions are not necessarily correlated to reductions in either the SAT solver's runtime or peak memory consumption, but this is expected, as the size of a SAT problem is generally not a reliable predictor for its difficulty. However, there are two notable exceptions. For the unsafe pthread-ext benchmarks we see a much larger reduction of approx. 85%, but this is skewed by two benchmarks that involve large arrays that allow these large reductions. Conversely, for the pthread-wmm benchmarks we see almost no reduction in size. This is a consequence of the very simple structure of these benchmarks—they are typically loop-free, which means that the unwound programs only contain a (relatively) small number of assignments. Hence, there is little scope to optimize the representation of the program variables.

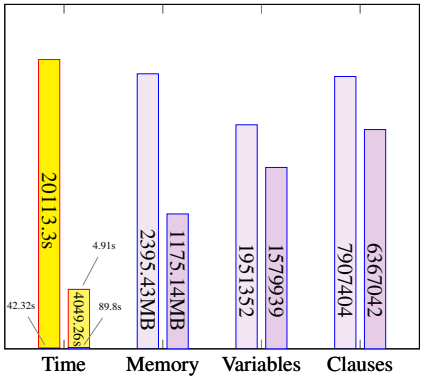
### Complex benchmarks

We now report on the experiments for three unsafe benchmarks that present a non-trivial challenge for bug-finding tools. These benchmarks consist of non-blocking algorithms for shared data-structures. It is hardly surprising that lock-free programming is an important source of benchmarks whose complexity truly stems from the system's concurrent interactions, not its computations. In fact, the focus there is to minimize the amount of synchronization for performance optimization, thus generating a large amount of nondeterminism due to interleaving. Here we demonstrate that Lazy-CSeq is very effective in spotting rare bugs in these programs, and that Lazy-CSeq+Abs allows to amplify its effectiveness both in terms of verification time and memory peak usage.

`safestack`. This is a real world benchmark implementing a lock-free stack designed for weak memory models. It was posted to the CHESS forum by Dmitry Vyukov.[3] It is unique in the sense that it contains a very rare bug that requires at least three threads and five context-switches to be exposed when running under the SC semantics. In the verification literature, it was shown that real-world bugs require at most three context-switches to manifest themselves [30]. `safestack`, for this reason, presents a non-trivial challenge for concurrency testing and symbolic tools. Lazy-CSeq is the only tool we are aware of that can automatically find such concurrency bugs in `safestack`. It requires about 1h:13m:28s (of which about one hour is spent in the SAT solver) to find a bug and has a memory peak of 2.18 GB (by setting the minimal parameters to expose the bug to 4 rounds of computation and 3 loop-unwinding). Lazy-CSeq+ABS, with the same parameters, requires 44m:11s time, where the same time is spend in the symbolic execution, and 17s is the time required for the value analysis by Frama-C, which leads to a 1.7x speed-up. Also, it uses only 1.19 GB of memory, i.e., roughly half of the memory required by Lazy-CSeq. All this is illustrated in the figure on the right where we also report on the number of variables and clauses of the produced CNF formulas.

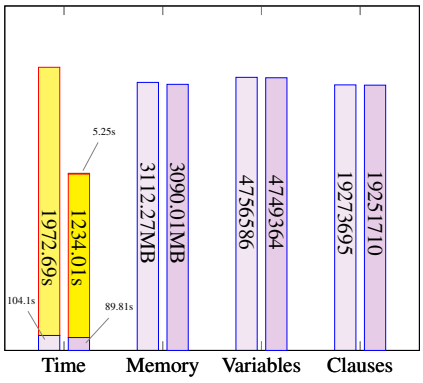



`eliminationstack`. This is a C implementation of Hendler et al.'s Elimination Stack [12] that follows the original pseudocode presentation. It augments Treiber's stack with a "collision array", used when an optimistic push or pop detects a conflicting operation; the collision array pairs together concurrent push and pop operations to "eliminate" them without affecting the underlying data structure. This implementation is incorrect if memory is freed in pop operations. In particular, if memory is freed only during the "elimination" phase, then exhibiting a violation (an instance of the infamous ABA problem) requires a seven thread client where three push operations are concurrently executed with four pops. To witness the violation, the implementation is annotated with several assertions that manipulate counters as described in [4]. Lazy-CSeq is the only tool we are aware of that can automatically find bugs in this benchmark and requires 5h:35m:13s time and 2.39 GB of memory to find a bug. Lazy-CSeq+ABS, with the same parameters, requires 1h:07m:29s time, where 4.9s is the time required for the value analysis by Frama-C, which leads to a 5x




[3] `https://social.msdn.microsoft.com/Forums/`

speed-up. As for the memory usage, it uses only half of the memory required by Lazy-CSeq, namely 1.17 GB. All this is illustrated in the figure on the right where we also report on the number of variables and clauses of the produced CNF formulas.




`DCAS`. This is a non-blocking algorithm for two-sided queues presented in [1]. This algorithm has a subtle bug that was discovered in an attempt to prove its correctness with the help of the PVS theorem prover. The discovery of the bug took several months of human effort. Although the bug has been automatically discovered using the model checker SPIN (see [13] and `http://spinroot.com/dcas/`), a generalized version of the benchmark remains a challenge for explicit exploration approach. In fact, after 138h of CPU-time (using 1000 cores), and an exploration of $10^{11}$ states the error was still undetected [14]. Here, we have translated this benchmark from Promela to C99 with Pthread library considering a more complex version that has 10 threads while the version of [14] only considers 8 threads. Lazy-CSeq can detect the bug within 32m:52s and with a memory peak usage of 3.11GB. Instead, Lazy-CSeq+Abs takes only 20m:34s with a memory peak of 3.09GB. All this is illustrated in the figure on the right where we also report on the number of variables and clauses of the produced CNF formulas.

## 5 Related Work

The idea of sequentialization was originally proposed by Qadeer and Wu [29]. The first scheme for an arbitrary but bounded number of context switches was given in [24]. Since then, several algorithms and implementations have been developed (see [9, 23, 3, 20, 19, 33]). *Lazy* sequentialization schemes have played an important role in the development of efficient tools. The first such sequentialization was given in [20] for bounded context switching and extended to unboundedly many threads in [21, 22]. These schemes require frequent recomputations and are not suitable for use in combination with bounded model-checking (see [11]). Lazy-CSeq [16] avoids such recomputations and achieves efficiency by handling context-switches with a very lightweight and decentralized control code. Lazy-CSeq has been recently extended to handle relaxed memory models [34] and to prove correctness [25].

Abstract interpretation [7] is a widely used static analysis technique which has been scaled up to large industrial systems [8]. However, since the abstraction functions typically overapproximate the values a program variable can take on, abstract interpretation is prone to false alarms, and considerable effort went into designing suitable abstractions (e.g., [27, 36]).

An alternative approach combines abstract interpretation with a post-processing phase based on a more precise analysis to either confirm or filter out warnings. Post

et al. [28] describe a semi-automatic process in which they use CBMC repeatedly on larger and larger code slices around potential error locations identified by Polyspace.[4] They report a reduction of false alarms by 25% to 75%, depending on the amount of manual intervention. Chebaro et al. [5, 4] describe the SANTE tool, which uses dynamic symbolic execution or concolic testing to try and construct concrete test inputs that confirm the warnings. The main difference to our work is that such approaches use abstract interpretation only to "guide" the more precise post-processing phase towards possible error locations but do not inject information from the abstractions into the post-processing in the same way as in our work.

Wu et al. [38] also combine sequentialization and abstract interpretation, but in a different context and with different goals. More specifically, they consider interrupt-driven programs (IPDs) for which they devise a specific lazy sequentialization schema; they then run a specialized abstract interpretation, which takes into account some properties of the IPDs such as schedulability, in order to prove the absence of some numerical runtime errors. In contrast, we consider general C programs over the more general Pthreads API, and use a generic sequentialization schema but a simpler abstract interpretation. However, the main difference is that we use the abstract interpretation only to produce hints for a more precise analysis (i.e., BMC), and not to produce the ultimate analysis result.

## 6 Conclusions and Future Work

Concurrent program verification remains a stubbornly hard problem, but lazy sequentialization has proven to be one of the most effective techniques, and has, in combination with a SAT-based BMC tool as sequential verification backend, been used successfully to find errors in hard benchmarks on which all other tools failed. However, the sizes of the individual states (which are determined by concurrent program's shared global and thread-local variables) still pose problems for further scaling. We have therefore proposed an approach where we use abstract interpretation to minimize the representation of these variables. More specifically, we run the Frama-C abstract interpretation tool over the programs constructed by Lazy-CSeq to compute overapproximating intervals for all (original) program variables and then exploit CBMC's bitvector support to reduce the number of bits required to represent these in the sequentialized program. We have implemented this approach on top of Lazy-CSeq and have demonstrated the effectiveness of this approach; it has performed very well in SV-COMP'17 competition, where it solved all tasks [26]. In this paper, in particular, we have further shown that it leads to large performance gains for very hard verification problems.

Our approach is easy to implement and effective because of the confluence of four different strands. First, we use a source-to-source transformation tool for the sequentialization. This makes it easy to re-use an off-the-shelf tool (i.e., Frama-C) for the interval analysis. Second, we use a backend verification tool (i.e., CBMC) that can effectively exploit the information provided by Frama-C, by means of a specialized bitvector type. Third, we are using a lazy sequentialization, which ensures that the interval analysis can

[4] `https://www.mathworks.com/products/polyspace.html`

compute tight intervals; our approach would not work with an eager sequentialization where the state variables remain unconstrained. Fourth, the interval analysis strikes the right balance between analysis efforts and results—that is, it runs fast enough, and the computed intervals are tight enough, so that the overheads are easily recouped, and we actually improve the overall performance. Other, more elaborate, abstract interpretations have in fact proven to be counter-productive.

In this paper, we have demonstrated our approach for sequentially consistent concurrent programs that use the Pthreads API. However, all specific aspects of the concurrency model are actually encapsulated in the sequentialization. Our approach is therefore also applicable to other concurrency models, as long as we have (or can design) a corresponding lazy sequentialization, and we plan to extend our work to weak memory models, based on our previous work [34].

Another avenue for future work is to investigate other "cheap" analyses that can be run over sequentialized program; specifically, we plan to use a points-to analysis to reduce the amount of possible sharing that the BMC backend needs to encode into the SAT formula.

## References


1. O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. A. Martin, N. Shavit, and G. L. S. Jr. Dcas-based concurrent deques. In *SPAA*, pages 137–146, 2000.
2. G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *SPAA*, pages 123–124, 2009.
3. S. Chaki, A. Gurfinkel, and O. Strichman. Time-bounded Analysis of Real-time Systems. In *FMCAD*, pages 72–80, 2011.
4. O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng.*, 21(1):107–143, 2014.
5. O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *TAP*, pages 78–83, 2011.
6. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
8. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.
9. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-processor for Sequential C Verification Tools. In *ASE*, pages 710–713, 2013.
10. B. Fischer, O. Inverso, and G. Parlato. Cseq: A sequentialization tool for C - (competition contribution). In *TACAS*, pages 616–618, 2013.
11. N. Ghafari, A. J. Hu, and Z. Rakamaric. Context-Bounded Translations for Concurrent Software: An Empirical Evaluation. In *SPIN*, pages 227–244, 2010.
12. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, pages 206–215. ACM, 2004.
13. G. J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, 2014.
14. G. J. Holzmann. Cloud-based verification of concurrent software. In *VMCAI*, pages 311–327, 2016.

15. 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.
16. 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.
17. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A lazy sequentialization tool for C - (competition contribution). In *TACAS*, pages 398–401, 2014.
18. ISO/IEC. *Information technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, ISO/IEC/IEEE 9945:2009.* 2009.
19. S. La Torre, P. Madhusudan, and G. Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. In *PLDI*, pages 211–222, 2009.
20. S. La Torre, P. Madhusudan, and G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *CAV*, pages 477–492, 2009.
21. S. La Torre, P. Madhusudan, and G. Parlato. Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In *CAV*, pages 629–644, 2010.
22. S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing Parameterized Programs. In *FIT*, pages 34–47, 2012.
23. A. Lal, S. Qadeer, and S. K. Lahiri. A Solver for Reachability Modulo Theories. In *CAV*, pages 427–443, 2012.
24. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Meth. in Sys. Des.*, (1):73–97, 2009.
25. T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *ATVA*, pages 174–191, 2016.
26. T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq 2.0: Combining lazy sequentialization with abstract interpretation - (competition contribution). In *TACAS*, 2017. To appear.
27. M. Oulamara and A. J. Venet. Abstract interpretation with higher-dimensional ellipsoids and conic extrapolation. In *CAV*, pages 415–430, 2015.
28. H. Post, C. Sinz, A. Kaiser, and T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *ASE*, pages 188–197, 2008.
29. S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
30. P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: an empirical study. In *PPoPP*, pages 15–28, 2014.
31. 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 *TACAS*, pages 402–404, 2014.
32. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Mu-cseq 0.3: Sequentialization by read-implicit and coarse-grained memory unwindings - (competition contribution). In *TACAS*, pages 436–438, 2015.
33. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In *TACAS*, pages 551–565, 2015.
34. 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*, pages 193–200, 2016.
35. E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Mu-cseq 0.4: Individual memory location unwindings - (competition contribution). In *TACAS*, pages 938–941, 2016.
36. A. Venet. The gauge domain: Scalable analysis of linear inequality invariants. In *CAV*, pages 139–154, 2012.
37. D. Vyukov. Bug with a context switch bound 5, 2010.
38. X. Wu, L. Chen, A. Miné, W. Dong, and J. Wang. Numerical static analysis of interrupt-driven programs via sequentialization. In *EMSOFT*, pages 55–64, 2015.