Verifying Concurrent Programs by Memory Unwinding*

Ermenegildo Tomasco[1], Omar Inverso[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. We describe a new sequentialization-based approach to the symbolic verification of multithreaded programs with shared memory and dynamic thread creation. Its main novelty is the idea of *memory unwinding* (MU), i.e., a sequence of write operations into the shared memory. For the verification, we nondeterministically guess an MU and then simulate the behavior of the program according to any scheduling that respects it. This approach is complementary to other sequentializations and explores an orthogonal dimension, i.e., the number of write operations. It also simplifies the implementation of several important optimizations, in particular the targeted exposure of individual writes. We implemented this approach as a code-to-code transformation from multithreaded into nondeterministic sequential programs, which allows the reuse of sequential verification tools. Experiments show that our approach is effective: it found all errors in the concurrency category of SV-COMP15.

1 Introduction

Concurrent programming is becoming more important as concurrent computer architectures such as multi-core processors are becoming more common. However, the automated verification of concurrent programs remains a difficult problem. The main cause of the difficulties is the large number of possible ways in which the different elements of a concurrent program can interact with each other, e.g., the number of different interleavings of a program's threads. In practice, however, we fortunately do not need to consider all possible interactions. For example, it is well known that many concurrency errors manifest themselves already after only a few context switches [20]; this observation gives rise to a variety of *context-bounded analysis* methods [16, 19, 13, 5, 8, 6, 9, 11].

Recent empirical studies have pointed out other common features for concurrency errors, but these have not yet been exploited for practical verification algorithms. In particular, Lu et al. [17] observed that "almost all [...] concurrency bugs are guaranteed to manifest if certain partial order among *no more than 4 memory accesses* is enforced."

* Partially supported by EPSRC grant no. EP/M008991/1, INDAM-GNCS 2014 grant and MIUR-FARB 2012-2014 grants.

In this paper we follow up on their observation that only a few memory accesses are relevant, and propose a corresponding new approach to the automated verification of concurrent programs, more specifically multithreaded programs with shared memory. Our approach simulates the executions of a multithreaded program but bounds the total number of write operations into the shared memory that can be read by threads other than the one performing the writing. It is related to context-bounded analyses [19, 16, 13] but the bounding parameter is different, which allows an orthogonal exploration of the search space.

The central concept in our approach is called *memory unwinding* (MU). This is a an explicit representation of the write operations as a sequence that contains for each write the writing thread, the variable or lock, and the written value. Our approach can then be seen as an *eager sequentialization* of the original concurrent program over the unwound memory. We first guess an MU and then simulate all program runs that are compatible with this guess. For the simulation, each thread is translated into a simulation function where write and read accesses over the shared memory are replaced by operations over the unwound memory. The simulation functions are executed sequentially; each thread creation is translated into a call to the corresponding simulation function. All context switches are implicitly simulated through the MU.

The approach allows us to vary which write operations are represented and thus exposed to the other threads. This leads to different strategies with different performance characteristics. In a *fine-grained* MU *every* write operation is represented explicitly and individually while *coarse-grained* MUs only represent a subset of the writes, but group together multiple writes. In an *intra-thread* MU the writes in one group are all executed by one thread; the writes not represented can thus be seen as having been superseded by subsequent writes in the same context. In an *inter-thread* MU the writes in one group can come from different threads, thus summarizing the effect of multiple context switches.

We have implemented in our MU-CSeq tool these strategies as code-to-code transformations for ANSI-C programs that use the POSIX threads API. We have evaluated MU-CSeq over the SV-COMP15 [3] concurrency benchmarks. It has found all errors and shown itself to be competitive with state-of-the-art tools for concurrent programs, in particular CBMC [7] and Lazy-CSeq [12].

In summary, in this paper we make the following main contributions:

- We describe in Section 3 a new sequentialization-based symbolic verification approach for multithreaded programs with shared memory based on the novel idea of memory unwinding.
- We describe in Section 4 different strategies to implement our approach as code-to-code translations that can be used with arbitrary sequential verification backends.
- We evaluate in Section 5 these implementations over the SV-COMP15 benchmark suite; the results are in line with those of the current best tools for concurrency handling.

In addition, we formalize in Section 2 the language we use to illustrate our approach, while we discuss related work in Section 6 and conclude in Section 7.

2 Concurrent programs

We use a simple imperative language for multithreaded programs to illustrate our approach. It features shared variables, dynamic thread creation, and thread join and mutex locking and unlocking for thread synchronization. We adopt a C-like syntax, which is given in Fig. 1; here, terminal symbols are set in typewriter font, and $\langle n\ \texttt{t}\rangle^*$ denotes a possibly empty list of non-terminals n that are separated by terminals `t`. We further denote with x a local variable, y a shared variable, m a mutex, t a thread variable and p a procedure name.

$$
\begin{array}{lcl}
P & ::= & (dec\texttt{;})^*\ (type\ p\ \texttt{(}\langle dec\texttt{,}\rangle^*\texttt{)\{}(dec\texttt{;})^*\, stm\texttt{\}})^* \\
dec & ::= & type\ z \\
type & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
stm & ::= & seq \mid conc \mid \texttt{\{}(stm\texttt{;})^*\texttt{\}} \\
seq & ::= & \texttt{assume}(b) \mid \texttt{assert}(b) \mid x := e \mid p\texttt{(}\langle e\texttt{,}\rangle^*\texttt{)} \mid \texttt{return}\ e \\
 & & \mid \texttt{if}(b)\,\texttt{then}\ stm\ \texttt{else}\ stm \mid \texttt{while}(b)\ \texttt{do}\ stm \\
conc & ::= & x := y \mid y := x \mid t := \texttt{create}\ p(\langle e\texttt{,}\rangle^*) \mid \texttt{join}\ t \\
 & & \mid \texttt{lock}\ m \mid \texttt{unlock}\ m \mid \texttt{atomic}\ stm
\end{array}
$$

Fig. 1. Syntax of concurrent programs.

A *concurrent* program consists of a list of *shared variable* declarations, followed by a list of procedures. Each procedure has a list of typed parameters, and its body has a declaration of *local* variables followed by a statement. A statement is either a sequential or concurrent statement, or a sequence of statements enclosed in braces. A *sequential statement* can be an `assume`- or `assert`-statement, an assignment, a procedure call with a call-by-value parameter passing semantics a `return`-statement, a conditional statement, or a `while`-loop. All variables involved in a sequential statement must be local. Note that we leave expressions e and Boolean expressions b undefined; we assume the usual constants and operations. We also use $*$ to denote the nondeterministic choice of any possible value of the corresponding type. A *concurrent statement* can be a concurrent assignment, a thread creation, a thread join, a mutex lock or unlock operation, or an atomic block. A concurrent assignment assigns a shared (resp. local) variable to a local (resp. shared) one. A thread creation statement $t :=$ `create` $p(e_1, \ldots, e_n)$ creates a new thread by calling the starting procedure p with the expressions $e_1, \ldots, e_n$ as arguments and assigning the thread identifier to t. A thread join statement `join` t pauses the current thread until the thread identified by t terminates. Lock and unlock statements respectively acquire and release a mutex. If the mutex is already acquired, the lock operation is blocking for the thread, i.e., the thread waits until the mutex is released. Context switches to other threads are disallowed when a thread's control flow is within an atomic statement or block.

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

The semantics is the obvious one: a configuration is a tuple of configurations of each thread that has been created and has not yet terminated, along with a valuation of the shared variables. A thread configuration consists of a *stack* which stores the history of positions at which calls were made, along with valuations for

local variables, and the top of the stack contains the local and global valuations, and a pointer to the current statement being executed.

The behavioral semantics of a program P is obtained by interleaving the behaviors of its threads. At the beginning of any computation only the main thread is *available*. At any point of a computation, only one of the available threads is *active*. A step is either the execution of a step of the active thread or a context-switch that replaces the active thread with one of the available threads that thus becomes the active thread at the next step. A thread may become temporarily unavailable if it is waiting to acquire a mutex or waiting to join another thread. A thread will no longer be available when its execution is terminated, i.e., there are no more steps that it can take.

Fibonacci-example. Fig. 2(a) shows a multithreaded implementation of a nondeterministic Fibonacci-function. This example from the SV-COMP15 benchmark suite uses two threads $f1$ and $f2$ to repeatedly increment the shared variables i and j by j and i, respectively. With a round-robin schedule with context switches after each assignment the variables i and j take on the consecutive values from the Fibonacci-series, and the program terminates with $i = \mathit{fib}(11) = 89$ and $j = \mathit{fib}(12) = 144$ if this interleaving starts with $f1$. Any other schedule will lead to smaller values for i or j.

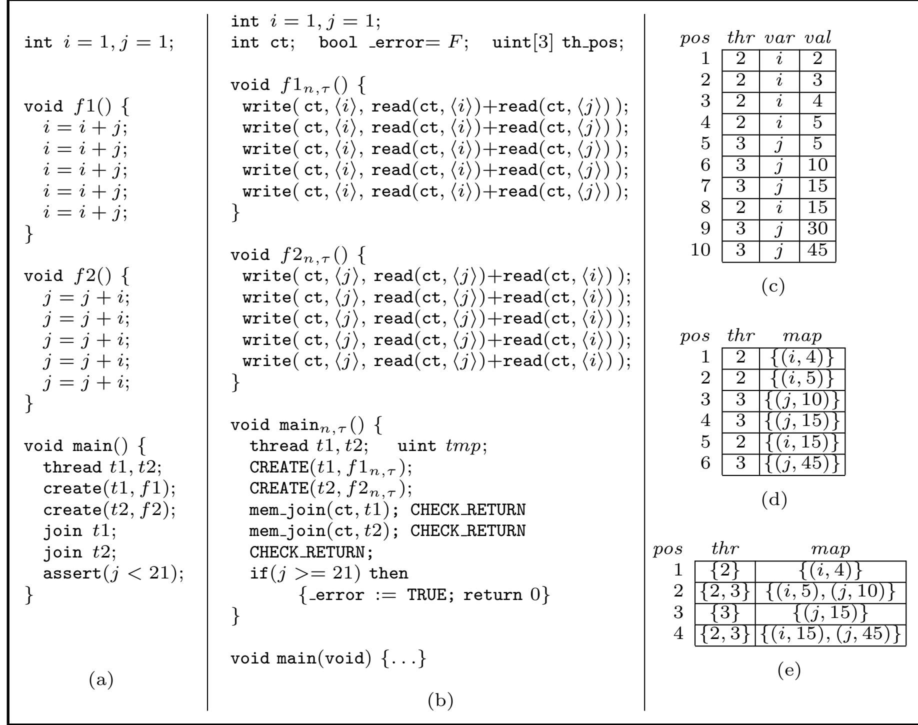
The `main` function first creates the two threads $f1$ and $f2$, then uses two join statements to ensure that both threads run to completion, and finally checks the outcome of the choosen interleaving. Note that each of the assignments contains three sequence points; our core language makes these explicit and thus "$i = i{+}j;$" becomes "$x = i;\ y = j;\ z = i + j;\ i = z;$" for local variables x, y, and z.

3 Sequentialization by Memory Unwinding

In this section we first give a general overview of the main concepts of our approach, namely the memory unwinding and the simulation of the read and write operations, before we describe the sequentialization translation. We use the Fibonacci-example above to illustrate the concepts. We describe different implementation alternatives for the concepts in Section 4.

High-level description. Our approach is based on a code-to-code translation of any concurrent program P into a corresponding sequential program $P_{n,\tau}$ that captures all the executions of P involving at most n write operations in the shared memory and τ threads. We show that an assertion fails for such an execution of P if and only if a corresponding assertion fails in $P_{n,\tau}$.

A core concept in this translation is the *memory unwinding*. An *n-memory unwinding* M of P is a sequence of writes $w_1 \ldots w_n$ of P's shared variables; each w_i is a triple (t_i, var_i, val_i) where t_i is the identifier of the thread that has performed the write operation, var_i is the name of the written variable and val_i is the new value of var_i. A *position* in an n-memory unwinding M is an index in the interval $[1, n]$. An execution of P *conforms to* a memory unwinding M if the sequence of its writes in the shared memory exactly matches M. Fig. 2(c) gives a 10-memory unwinding. The following is an execution of the multithreaded



```
int i = 1, j = 1;

void f1() {
  i = i + j;
  i = i + j;
  i = i + j;
  i = i + j;
  i = i + j;
}

void f2() {
  j = j + i;
  j = j + i;
  j = j + i;
  j = j + i;
  j = j + i;
}

void main() {
  thread t1, t2;
  create(t1, f1);
  create(t2, f2);
  join t1;
  join t2;
  assert(j < 21);
}
```

(a)

```
int i = 1, j = 1;
int ct;   bool _error= F;   uint[3] th_pos;

void f1_{n,τ}() {
  write( ct, ⟨i⟩, read(ct, ⟨i⟩)+read(ct, ⟨j⟩) );
  write( ct, ⟨i⟩, read(ct, ⟨i⟩)+read(ct, ⟨j⟩) );
  write( ct, ⟨i⟩, read(ct, ⟨i⟩)+read(ct, ⟨j⟩) );
  write( ct, ⟨i⟩, read(ct, ⟨i⟩)+read(ct, ⟨j⟩) );
  write( ct, ⟨i⟩, read(ct, ⟨i⟩)+read(ct, ⟨j⟩) );
}

void f2_{n,τ}() {
  write( ct, ⟨j⟩, read(ct, ⟨j⟩)+read(ct, ⟨i⟩) );
  write( ct, ⟨j⟩, read(ct, ⟨j⟩)+read(ct, ⟨i⟩) );
  write( ct, ⟨j⟩, read(ct, ⟨j⟩)+read(ct, ⟨i⟩) );
  write( ct, ⟨j⟩, read(ct, ⟨j⟩)+read(ct, ⟨i⟩) );
  write( ct, ⟨j⟩, read(ct, ⟨j⟩)+read(ct, ⟨i⟩) );
}

void main_{n,τ}() {
  thread t1, t2;   uint tmp;
  CREATE(t1, f1_{n,τ});
  CREATE(t2, f2_{n,τ});
  mem_join(ct, t1); CHECK_RETURN
  mem_join(ct, t2); CHECK_RETURN
  CHECK_RETURN;
  if(j >= 21) then
      {_error := TRUE; return 0}
}

void main(void) {...}
```

(b)

pos	*thr*	*var*	*val*
1	2	i	2
2	2	i	3
3	2	i	4
4	2	i	5
5	3	j	5
6	3	j	10
7	3	j	15
8	2	i	15
9	3	j	30
10	3	j	45

(c)

pos	*thr*	*map*
1	2	$\{(i,4)\}$
2	2	$\{(i,5)\}$
3	3	$\{(j,10)\}$
4	3	$\{(j,15)\}$
5	2	$\{(i,15)\}$
6	3	$\{(j,45)\}$

(d)

pos	*thr*	*map*
1	$\{2\}$	$\{(i,4)\}$
2	$\{2,3\}$	$\{(i,5),(j,10)\}$
3	$\{3\}$	$\{(j,15)\}$
4	$\{2,3\}$	$\{(i,15),(j,45)\}$

(e)

Fig. 2. Multithreaded Fibonacci: (a) code for 5 iterations, (b) translated code, and sample memory unwindings: (c) fine grained, (d) intra-thread and (e) inter-thread.

program given in Fig. 2(a) that conforms to it (we omit the main thread): the first three assignments of $f1$, followed by a read of i by $f2$, then the fourth assignment of $f1$, the completion of the first two assignments of $f2$, the read of j by $f1$, the third assignment of $f2$, the last assignment of $f1$ and the remaining assignments of $f2$. Note that a memory unwinding can be *unfeasible* for a program, in the sense that no execution of the program conforms to it. Conversely, multiple executions can also conform to one memory unwinding, although this is not the case for the Fibonacci-example.

We use a memory unwinding M to explore the runs of P that conform to it by running each thread t separately. The idea is to use the MU for the concurrent statements (which involve the shared memory) and execute the sequential statements directly. In particular, when we execute a *write* of t in the shared memory, we check that it matches the next write of t in M. However, a *read* of t in the shared memory is more involved, since the sequence of reads is not explicitly stored in the MU. We therefore need to nondeterministically guess the position in the MU from which we read. Admissible values are all the positions that are in the range from the current position (determined by previous operations on the shared memory) to the position of t's next write in M. The nondeterminis-

tic guess ensures that we are accounting for all possible interleavings of thread executions that conform to the MU.

For example, consider again the 10-memory unwinding of Fig. 2(c). The execution of $f1$ is simulated over this as follows. The first four writes are matched with the first four positions of the MU; moreover, the related reads are positioned at the current index since they are each followed by the write which is at the next position in the MU. The fifth write is matched with position 8. The corresponding read operations can be assigned nondeterministically to any position from 4 to 7. However, in order to match the value 15 with the write, the read of j must be positioned at 6. Note that the read of i can be positioned anywhere in this range since it was written last time at position 4.

We stress that when simulating one thread we assume that the writes executed by the other threads, and stored in the memory unwinding, indeed all occur and, moreover, in the ordering shown in M. Thus, for the correctness of the simulation, for each thread t we must ensure not only that each of its writes involving the shared variables conforms to the write sequence in M, but also that all the writes claimed in the MU are actually executed. Further, t should not contribute to the computation before the statement that creates it has been simulated. This can be easily enforced by making the starting position of the child thread to coincide with the current position in M of the parent thread when its creation is simulated.

Construction of $P_{n,\tau}$. The program $P_{n,\tau}$ first guesses an n-memory unwinding M and then simulates a run of P that conforms to M. The simulation starts from the main thread (which is the only active thread when P starts) and then calls the other threads one by one, as soon as their thread creation statements are reached. Thus, the execution of the parent thread is suspended and then resumed after that the simulation of the child thread has completed. Essentially, dynamic thread creation in P is modeled by function calls in $P_{n,\tau}$.

$P_{n,\tau}$ is formed by a main, and a new procedure $p_{n,\tau}$ for each procedure p of P. It uses some additional global variables: `_error` is initialized to false and stores whether an assertion failure in P has occurred; `ct` stores the identifier of the current thread; the array `th_pos` stores the current position in the memory unwinding for each thread.

```
void main(void) {
  mem_init(V, n, τ);
  ct := mem_thread_create(0);
  main_{n,τ}(x_1, ..., x_k);
  mem_thread_terminate(ct);
  mem_allthreads_executed();
  assert(_error ≠ 1)   }
```

Fig. 3. $P_{n,\tau}$: `main()`.

The main procedure of $P_{n,\tau}$ is given in Fig. 3. First, we call `mem_init`(V, n, τ) that guesses an n-memory unwinding with V variables and τ threads, and then `mem_thread_create`(0) that registers the main thread and returns its id. Note that we encode each of P's shared variables y with a different integer $\langle\langle y \rangle\rangle$ in the interval $[1, V]$ and each thread with a different integer in $[1, \tau]$; once τ threads are created `mem_thread_create` returns -1 (an invalid id) that causes the thread not to be simulated. The parameter passed to `mem_thread_create` is the id of the thread that is invoking the thread creation. For the creation of the main thread we thus pass 0 to denote that this is the first created thread.

```
1. [type p (par*){dec* stm}]  ::=  type p_{n,τ} (par*){dec*; uint tmp; [stm]}
2. [{(stm;)*}]  ::=  {([stm];)*}
Sequential statements:
3. [assume(b)]  ::=  CHECK_RETURN; assume(b)
4. [assert(b)]  ::=  CHECK_RETURN; if(¬b) then {_error:= TRUE; return 0}
5. [p(e_1, ..., e_n)]  ::=   p_{n,τ}(e_1, ..., e_n); CHECK_RETURN
6. [return e]  ::=  return e
7. [x := e]  ::=   x := e
8. [x := p(e_1, ..., e_n)]  ::=   x := p_{n,τ}(e_1, ..., e_n); CHECK_RETURN
9. [while(b) do stm]  ::=  while(b) do {CHECK_RETURN; [stm]}
10. [if (b) then stm else stm]   ::=   if (b) then [stm] else [stm]
Concurrent statements:   (x is local, y is shared, ct contains the current tread id)
11. [y := x]  ::=   write(ct, ⟨⟨y⟩⟩, x); CHECK_RETURN
12. [x := y]  ::=   x :=read(ct, ⟨⟨y⟩⟩); CHECK_RETURN
13. [t := create p(e_1, ..., e_n)]  ::=   tmp := ct;    t := mem_thread_create(ct);
                                        if(t ≠ −1) then {
                                            ct := t;  p_{n,τ}(e_1, ..., e_n);
                                            mem_thread_terminate(t);   };
                                        ct := tmp;
14. [join t]  ::=   mem_join(ct, t); CHECK_RETURN
15. [lock v]  ::=   var_lock(ct, ⟨⟨v⟩⟩); CHECK_RETURN
    [unlock v]  ::=   var_unlock(ct, ⟨⟨v⟩⟩); CHECK_RETURN
16. [atomic stm] ::=   call mem_lock(ct); CHECK_RETURN;
                       [stm]; mem_unlock(ct); CHECK_RETURN
```

Fig. 4. Rewriting rules.

The call to $\texttt{main}_{n,\tau}$ starts the simulation of the main thread. Then, we check that all the write operations guessed for the main thread have been executed (by `mem_thread_terminate`), and all the threads involved in the guessed writes have indeed been simulated (by `mem_allthreads_executed`). If either one of the above checks fails, the simulation is infeasible and thus aborted. The global variable `_error` is used to check whether an assertion has been violated. It is set to `TRUE` in the simulation of the threads of P whenever an assertion gets violated and is never reset.

Each $p_{n,\tau}$ is obtained from p according to the transformation function $[\cdot]$ defined inductively over the program syntax by the rules given in Fig. 4. For example, Fig. 2(b) gives the transformations for the functions of the Fibonacci program from Fig. 2(a). There we use the macro `CREATE` as a shorthand for the code given in the translation rules for the create statement. Also, we have omitted the declaration of *tmp* in the functions $f1_{n,\tau}$ and $f2_{n,\tau}$ since it is not used there, and reported the translation of the assignments in a compact form.

The transformation adds a local variable *tmp* that is used to store the current thread id when a newly created thread is simulated. The sequential statements are left unchanged except for the injection of the macro `CHECK_RETURN` that is defined as "`if(is_th_terminated()) then return 0;`", where `is_th_terminated` is a function that checks if the simulation of the current thread is terminated. The macro is injected after each function call, as a first statement of a loop, and before any `assume`- and `assert`-statement; in this way, when the simulation of the current thread has finished or is aborted, we resume the simulation of the parent thread.

The concurrent statements are transformed as follows. A *write* of v into a shared variable x in thread t is simulated by a call to `write` that checks that the next write operation of t in the guessed MU (starting from the current position) writes x and that the guessed value coincides with v. Otherwise, the simulation of all threads must be aborted as the current execution does not conform to the MU. If t has already terminated its writes in the MU, we return immediately; otherwise we update t's current position to the index of its next write operation. A *read* of a shared variable x in thread t is simulated by a call to `read`. The read value is determined by nondeterministically guessing a position i between the current position for `ct` and the position prior to the next write operation of t in the memory unwinding. Thus, we return the value of the write operation involving x that is at the largest position $j \leq i$ and then update the stored position of t to i.

As above, we use `mem_thread_create` and `mem_thread_terminate` for the translation of thread creations but we check whether the thread creation was successful before calling the simulation function. Also, we save the current thread id in a temporary variable during the execution of the newly created thread.

The remaining concurrent statements are simulated by corresponding functions. A call `mem_join(ct,` t`)` returns if and only if the simulation of t is terminated at the current position of thread `ct` in the memory unwinding. Otherwise, the simulation of `ct` is aborted. A call `mem_lock(ct)` gives exclusive usage of the shared memory to thread `ct`. If the memory is already locked, the whole simulation is aborted. The unlocking is done by calling `mem_unlock`. Similarly, `var_lock` and `var_unlock` respectively lock and unlock an individual variable. Note that for join, lock, and unlock operations we choose to abort also computations that are still feasible, i.e., the lock could still be acquired later or we can wait for another thread to terminate. This is indeed correct for our purposes, in the sense that we are not missing bugs for this. In fact, we can capture those computations by scheduling the request to acquire the lock or to join exactly at the time when this will be possible, and by maintaining the rest of the computation unchanged. Due to space limitations we do not discuss the simulation of the locking mechanism in any detail, and omit it from the code fragments shown in the following section.

The correctness of our construction is quite straightforward to demonstrate. For *soundness*, assume any execution of P that does at most n writes in the shared memory, creates at most τ threads, and violates an assertion statement. We guess the exact sequence of writes in the MU, and simulate for each thread exactly the same steps as in P. This will allow us to execute all the writes for each thread and eventually reach the if-statement corresponding to the violated assertion of P. This will be properly propagated back to the main procedure of $P_{n,\tau}$; and since all threads have done all their writes, all the invocations of `mem_thread_terminate` will successfully return and thus the only assertion of $P_{n,\tau}$ will fail. For *completeness*, assume that there is an execution ρ of $P_{n,\tau}$ that violates the assertion in its main. This means that, along ρ we guess an n-memory unwinding M and simulate step by step a run ρ' of P that conforms

to M and reaches an assertion failure. In fact, when on ρ we set `_error` to `TRUE`, ρ' reaches the corresponding if-statement that violates an assertion of P. Before reaching the assertion in the main of $P_{n,\tau}$, we have already checked that all the invocations of `mem_thread_terminate` in each thread and the only invocation of `mem_allthreads_executed` in the main thread have successfully returned; therefore all the writes of M have been simulated. Therefore, we get:

Theorem 1. *A concurrent program P violates an assertion in at least one of its executions with at most n writes in the shared memory and τ thread creations if and only if $P_{n,\tau}$ violates its only assertion.*

4 Memory Unwinding Implementations

In this section, we discuss different implementation strategies of the memory unwinding approach that are characterized by orthogonal choices. The first choice we make is either to store in the MU all the writes of the shared variables in a run (*fine-grained* MU) or to expose only some of them (*coarse-grained* MU). In either case, depending on how we store the values of the variables that are not written at a position of the MU, we have two implementation alternatives that we call *read-explicit* (where all the shared variables are duplicated to each position, not only those that are changed in the writes) and *read-implicit* (where only the modified variables are duplicated at each position).

In a coarse-grained MU we store at each position a partial mapping from the shared variables to values, with the meaning that the variables in the domain of the mapping are modified from the previous position and the value given by the mapping is their value at this position. A variable that is modified at position $i+1$ could also be modified between positions i and $i+1$ by other writes that are not exposed in the MU. We distinguish the implementations according to whether only one (*intra-thread coarse-grained* MU) or multiple (*inter-thread coarse-grained* MU) threads are entitled to modify the variables.

4.1 Fine-grained Memory Unwinding

In this approach all writes are stored individually in the MU as described in Section 3. We use three arrays such that for each position i: `thread`[i] stores the thread id, `var`[i] stores the variable name, and `value`[i] stores the value of the i-th write. For an efficient implementation of the functions used in the translation described in Section 3, we use additional data such as variable `last_wr_pos` that stores the index of the last write performed in the simulation and table `th_nxt_wr`[t, i] that for each thread t and position i stores the position of the next write of t after i in the MU.

We describe the *read-explicit* and the *read-implicit* schemes only for this approach. It is not hard to extend them to the coarse-grained MU approach (discussed later) and thus we will omit this here.

Read-explicit scheme. We use a matrix `mem` to store for each variable v and each position i of the MU the value of v at i. `mem` is logically characterized

as follows: for every memory position $i \in [1, n]$ and variable index $v \in [1, V]$, $\texttt{mem}[i][v]$ is the valuation of variable v after the i-th write operation (assuming the values of arrays `thread`, `var`, and `value`). At memory position 1, all variables in $[1, V] \setminus \{v\}$ with $v = \texttt{var}[1]$ have their initial value, i.e., 0, and $\texttt{mem}[1][v]$ coincides with $\texttt{value}[1]$. For all the other memory positions, `mem` has the same valuation for all variables as in the previous position except for the one written at that position.

In Fig. 5, we give an implementation of `read` for the read-explicit scheme. For the current thread t, `Jump` guesses a position *jump* in the MU from the current position to the next write of t in the MU. If the simulation of t is deemed terminated, then consistently the read is not performed and the control returns (we recall that since we have injected the macro `CHECK_RETURN` at the end of each function call, all the calls to functions of t in the stack will immediately return and the simulation of t will actually end). Otherwise, the current position of t is updated to this value, and the valuation of the variable at position *jump* obtained from `mem` is retuned.

```
int read(uint t, uint v) {
 if (is_th_terminated()) then return 0;
 th_pos[t]=Jump(t);
 return (mem[th_pos[t]][var_name]);  }
```

Fig. 5. Function `read` (explicit-read schema).

Read-implicit scheme. Here, instead of replicating the valuations of the shared variables at each memory position, on reading a variable we get its value from the last relevant write. For this, we use two arrays `var_nxt_wr` and `var_fst_wr` s.t.: for each $i \in [1, n]$, $\texttt{var_nxt_wr}[i]$ is the smallest memory position $j > i$ s.t. $\text{var}[j] = \text{var}[i]$ and for each variable v, $\texttt{var_fst_wr}[v]$ is the position of its first write in the MU.

```
int read(uint t, uint v) {
 uint pos = th_pos[t];
 uint jump = *;
 if (is_th_terminated()) then return 0;
 if (var_fst_wr[v]==0) then return 0;
 assume((jump <= last_wr_pos)
   && (jump < th_nxt_wr[t][pos]));
 assume( var[jump] == var_name );
 if (jump < pos) then
    assume(var_nxt_wr[jump] > pos);
 else { if (jump < var_fst_wr[v]) then
          return 0;
        th_pos[t]=jump;  }
 return (value[jump]);    }
```

Fig. 6. Function `read` (implicit-read schema).

In Fig. 6, we give an implementation of function `read` in the read-implicit scheme. The first if-statement (corresponding to the `CHECK_RETURN` macro) takes care of thread termination as usual. The following if-statement handles the case when variable v is never written in the MU, and thus its value is always the initial one, i.e., 0. The first `assume`-statement constraints *jump* to a range of valid values similarly to as function `Jump` does in the previous scheme. Additionally, the second assume-statement requires that the guessed index indeed coincides with a write of variable v. Now, if *jump* is less than the thread's current position *pos*, we finally ensure that *jump* coincides with the last write operation involving v up to *pos*; otherwise we update the thread's current position to *jump*. In either case, the returned value is that at position *jump* unless *jump* precedes the index of the first write of v in the MU, and in this case the initial value is returned.

Mixing implicit and explicit read operations. We have also implemented a third schema that mixes the ideas of the above two. It uses an explicit representation

for scalar variables and an implicit representation for arrays, in order to balance the translation's memory overhead against the complexity of the sequentialized program.

4.2 Coarse-grained Memory Unwinding

The main idea of this approach is to expose only some of the writes of an MU. This has two main consequences in terms of explored runs of the original multithreaded program. On the one side, we restrict the number of possible runs that can match an MU. In fact, the unexposed writes cannot be read externally, and thus some possible interleavings of the threads are ruled out. On the other side, we can handle larger number of writes by nondeterministically deeming few of them as interesting for the other threads.

Intra-thread coarse-grained MU. We store a sequence of *clusters of writes* where each cluster is formed by a thread that is entitled to write and a partial mapping from the shared variables to values. The intended meaning is as follows. Consider the simulation of a thread t at a position i. If i is a position where t does not write into the shared memory, we are only allowed to read from the shared memory, and we reason similarly as to the approach given in the previous section. If i is a position of t (i.e., t is entitled to write into the shared memory), we ensure that all the writes in the shared memory only involve the variables that are annotated in the cluster at i and that all the writes in the cluster are matched before advancing to the next position in the simulation (some writes on the same variables can be processed before matching the value assigned in the cluster).

As an example, consider the intra-thread MU from Fig. 2(d). It is matched by the same execution as the MU from Fig. 2(c). Note that in this execution, the writes at the positions 1, 2, 5 and 9 are not used by the other thread and thus this is consistent with hiding them.

Inter-thread coarse-grained MU. The notion of cluster is extended with multiple threads assigned to each position. The idea is that all such *writing threads at i* can cooperate to match the writes exposed in the cluster. Thus, the unexposed writes are not local to a thread as in the alternative scheme but they can be exposed to the other writing threads. For this, in our implementation, we use for each position i in the sequence of clusters an additional copy of the shared variables that are modified at i (i.e., that are in the domain of the partial mapping at i). In the simulation of each writing thread at position i we treat them as local variables (thus we do not use the `read` and `write` functions, but we just use the name of the variable and the assignment). The intra-thread MU version of the MUs from Fig. 2(c) and (d) is given in Fig. 2(e).

5 Implementation and Evaluation

Implementation and Architecture. We have implemented in MU-CSeq (v0.3, `http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html`) the different variants

Table 1. Performance comparison among different tools on the unsafe instances of the SV-COMP15 *Concurrency category.*

			CBMC 4.9			LR-CSeq 0.5a			Lazy-CSeq 0.5			MU-CSeq 0.3		
sub-category	files	l.o.c.	pass	fail	time	pass	fail	time	pass	fail	time	pass	fail	time
pthread	14	4381	12	2	33.4	6	8	76.5	11	3	134.2	14	0	19.9
pthread-atomic	2	202	2	0	0.3	2	0	3.0	2	0	6.1	2	0	2.0
pthread-ext	8	763	6	2	153.3	1	7	119.6	8	0	16.6	8	0	2.4
pthread-lit	3	117	2	1	166.8	2	1	166.9	3	0	7.0	3	0	2.7
pthread-wmm-mix	466	146448	466	0	57.2	466	0	8.9	466	0	6.3	466	0	21.8
pthread-wmm-podwr	16	4240	16	0	10.6	16	0	4.2	16	0	6.0	16	0	12.3
pthread-wmm-rfi	76	20981	76	0	9.3	76	0	4.1	76	0	6.2	76	0	12.7
pthread-wmm-safe	184	57391	184	0	27.3	184	0	6.6	184	0	6.2	184	0	18.6
pthread-wmm-thin	12	4008	12	0	9.6	12	0	6.2	12	0	6.1	12	0	29.3

of the MU schema discussed in Section 4 as a code-to-code transformation for sequentially-consistent concurrent C programs with POSIX threads (`pthreads`). The output of MU-CSeq can, in principle, be processed by any analysis tool for sequential programs, but we primarily target BMC tools, in particular CBMC [7]. However, since the schema is very generic and the instrumentation for the different backends only differs in a few lines, backend integration is straightforward and not fundamentally limited to any underlying technology. A wrapper script bundles up the translation and the call to the backend for the actual analysis.

MU-CSeq is implemented as a chain of modules within the CSeq framework [9, 10]. The sequentialized program is obtained from the original program through transformations, which (*i*) insert boilerplate code for simulating the `pthreads` API; (*ii*) automatically generate control code for the bounded memory unwinding layer parameterized on n and τ; (*iii*) map the statements that in the original program correspond to read and write operations on shared variables to corresponding operations on the memory unwinding; (*iv*) insert code for the simulation of the `pthreads` API, concurrency simulation, and finalize the translation by adding backend-specific instrumentation.

Experiments. We have evaluated MU-CSeq with CBMC (v4.9) as a backend on the benchmark set from the Concurrency category of the TACAS Software Verification Competition (SV-COMP15) [3]. These are widespread benchmarks, and many state-of-the-art analysis tools have been trained on them; in addition, they offer a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms.

Since we use a BMC tool as a backend, and BMC can in general not prove correctness, but can only certify that an error is not reachable within the given bounds, we only evaluate our approach on the unsafe files. Our prototype does not fully support dynamic memory allocation of shared memory, so five of the test cases are excluded here. We thus use 781 of the 993 files in the whole benchmark set, with a total of approx. 240,000 lines of code.

We have performed the experiments on an otherwise idle machine with a Xeon W3520 2.6GHz processor and 12GB of memory, running a Linux operating system with 64-bit kernel 3.0.6. We set a 10GB memory limit and a 500s timeout for the analysis of each test case.

The experiments are summarized in Table 1. Each row corresponds to a sub-category of the SV-COMP15 benchmarks, where we report the number of files and the total number of lines of code. The table reports the evaluation of CBMC [7], LR-CSeq [9], Lazy-CSeq [12, 11], and MU-CSeq on these benchmarks. For each tool and sub-category we consider the best parameters (i.e., minimal loop unrolling, number of rounds, etc.). Furthermore, we indicate with *pass* the number of correctly found bugs, with *miss* the number of unsuccessful analyses including tool crashes, backend crashes, memory limit hits, and timeouts, and with *time* the average time in seconds to find the bug.

The table shows that MU-CSeq is competitive with other tools based on BMC, and in particular it is able to find all bugs. However, as in other bounded methods the choice of the bounds (i.e., size of the unwinding and number of simulated threads) also influences MU-CSeq's performance. Here we have simply increased the unwinding bounds until we found all bugs. In the first four sub-categories, a 24-memory unwinding is sufficient; with this, the explicit-read fine-grained implementation has the best performance. For the remaining sub-categories an MU with at least 90 writes is required; here the performance of the fine-grained implementation degrades, and the inter-thread coarse-grained variant performs best. A more refined strategy selection is left for future work.

The MU-Cseq source code, static Linux binaries and benchmarks are available at `http://users.ecs.soton.ac.uk/gp4/cseq/CSeq-MU-TACAS.tar.gz`.

6 Related work

The idea of sequentialization was originally proposed by Qadeer and Wu [20] but became popular with the first scheme for an arbitrary but bounded number of context switches given by Lal and Reps [16] (LR). This has been implemented and modified by several authors, e.g., in CSeq [9, 10], and in STORM that also handles dynamic memory allocation [14]. Poirot [18, 8] and Corral [15] are successors of STORM. Rek implements a sequentialization targeted to real-time systems [6].

The basic idea of the LR schemas 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.

Similarly to LR, sequentialization by memory unwinding runs each thread only once and simulates it to completion; however, there are several differences. First, the threads are not scheduled in a fixed ordering and in rounds. Instead, any scheduling that matches the memory unwinding is taken into account, including schedules with unboundedly many context switches (although one can

show that a subset of them using a bounded number of context-switches suffices to expose the same bugs). Second, the consistency check to prune away unfeasible computations is interleaved with the simulation, thus many unfeasible runs can be found earlier and not only at the end of the simulation. This can improve the performance, in particular for BMC backends. Third, it is possible to show that the assertion violations that can be exposed by our sequentialization is equivalent to those that can be exposed with LR, with different parameter values though. For example, for our intra-thread MU implementation, the exectutions that can be captured up to a size n in the memory unwinding can also be captured by LR with at most $2\,n-1$ rounds, and vice-versa all the computations of LR up to k context-switches (note that $k = r\,n - 1$ where n is the number of threads and r is the number of rounds) can be captured with at most k clusters.

MU can also be seen as a hybrid eager/lazy technique. It guesses the thread interactions at the beginning of the simulation, like the eager techniques in the Lal/Reps mould. However, it prunes away unfeasible computations incrementally, like Lazy-CSeq [12, 11], but it calls the thread simulation function only once and runs it to completion, rather then repeatedly traversing it. Unlike the original lazy techniques [13], it also does not need to recompute the values of the local variables.

A parameter related to the memory unwinding bound has been considered in [4] for message passing programs where the bounded analysis is done on the number of "process communication cycles".

7 Conclusions and Future Work

We have presented a new approach to verify concurrent programs based on bounding the number of the shared-memory writes that are exposed in the interaction between threads. At its core it is a new *eager sequentialization* algorithm that uses the notion of *memory unwinding*, i.e., the sequence of the exposed writes, to synchronize the separate simulation of singular threads.

We have designed different strategies and implemented them as code-to-code transformations for ANSI-C programs that use the Pthreads API; our implementations support the full language, but the handling of dynamic memory allocation is still limited. We have evaluated them over the SV-COMP15 [3] concurrency benchmarks, finding all the errors and achieving performance on par with those of the current best BMC tools with built-in concurrency handling as well as other sequentializations.

We have found that in general our fine-grained MU implementations work well for most problem categories, thus confirming the good results we achieved last year with MU-CSeq [21], which is based on an initial version of the work presented here. However, for the problems in the weak memory model category the size of the fine-grained unwindings becomes too big; here, coarse-grained MUs work better.

The main future direction of this research is to extend our approach to weak memory models implemented in modern architectures (see for example [2, 1]),

and to other communication primitives such as MPI. For MPI programs, our memory unwinding approach can be rephrased for the sequence of *send operations* in a computation.

References

1. J. Alglave, D. Kroening, and M. Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. *CAV*, LNCS 8044, pp. 141–157, 2013.
2. M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. *CAV*, LNCS 6806, pp. 00–115, 2011.
3. D. Beyer. SV-COMP home page, http://sv-comp.sosy-lab.org/.
4. A. Bouajjani and M. Emmi. Bounded phase analysis of message-passing programs. *TACAS*, LNCS 7214, pp. 451–465, 2012.
5. A. Bouajjani, M. Emmi, and G. Parlato. On Sequentializing Concurrent Programs. *SAS*, LNCS 6887, pp. 129–145, 2011.
6. S. Chaki, A. Gurfinkel, and O. Strichman. Time-bounded analysis of real-time systems. *FMCAD*, pp. 72–80, 2011.
7. E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. *TACAS*, *LNCS* 2988, pp. 168–176, 2004.
8. M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. *POPL*, pp. 411–422, 2011.
9. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-Processor for Sequential C Verification Tools. *ASE*, pp. 710-713, 2013.
10. B. Fischer, O. Inverso, and G. Parlato. CSeq: A Sequentialization Tool for C (Competition Contribution). *TACAS*, *LNCS* 7795, pp. 616–618, 2013.
11. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. *CAV*, LNCS 8559, pp. 585–602, 2014.
12. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A Lazy Sequentialization Tool for C - (Competition Contribution). *TACAS*, LNCS 8413, pp. 398–401, 2014.
13. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. *CAV*, LNCS 5643, pp. 477–492, 2009.
14. S. K. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. *CAV*, LNCS 5643, pp. 509–524, 2009.
15. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. *CAV*, LNCS 7358, pp. 427–443, 2012.
16. 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.
17. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, Mar. 2008.
18. S. Qadeer. Poirot - a concurrency sleuth. *ICFEM*, LNCS 6991, pp. 15, 2011.
19. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. *TACAS*, LNCS 3440, pp. 93–107, 2005.
20. S. Qadeer and D. Wu. Kiss: keep it simple and sequential. *PLDI*, pp. 14–24, 2004.
21. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings - (Competition Contribution). *TACAS*, LNCS 8413, pp. 402–404, 2014.

View publication stats