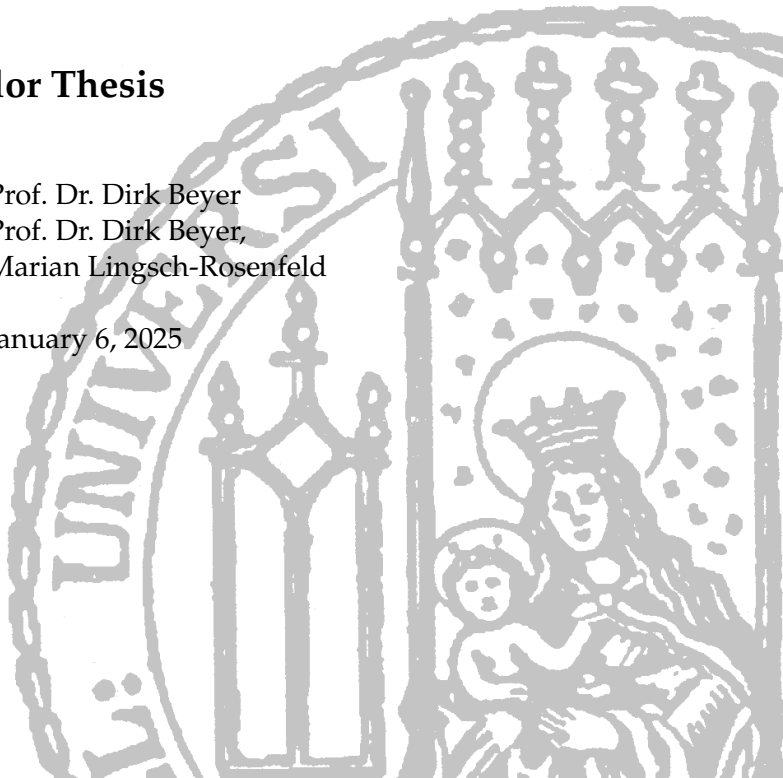
INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

MODULAR PARTIAL ORDER REDUCTION IN SOFTWARE VERIFICATION

Noah König



Bachelor Thesis

Supervisor	Prof. Dr. Dirk Beyer
Advisor	Prof. Dr. Dirk Beyer, Marian Lingsch-Rosenfeld

Submission Date January 6, 2025

Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, January 6, 2025 Noah König

Abstract

The verification of concurrent programs faces the state explosion problem due to non-deterministic context switches at every control location. Partial order reductions mitigate this problem by exploiting the commutativity of statements from different threads. Transforming concurrent programs into a sequential form (i.e. sequentializations) allow the use of optimized sequential reachability algorithms. In this thesis, partial order reductions and sequentializations are combined by creating assumptions over allowed transitions between thread simulations *inside* non-deterministic sequentializations.
Our experiments show that our approach accepts roughly half of the 667 tested input programs. The sequentialized input program induced a verification overhead and decreased analysis effectiveness and efficiency in a vast majority of cases. However, we identified great potential for optimization, especially reductions in the amount of assumptions that verifiers have to evaluate. Increasing the acceptance rate of input programs and addressing the performance issues is subject to future work.

Contents

List of Figures

CHAPTER 1

Introduction

Advancements in computational power allowed humanity to approach problems such as climate modeling, protein folding or big data analysis. From the 1970s to the early 2000s, the increase in computational power was induced by an increase in single-processor performance. This increase automatically made sequential programs faster with the next generation of processors. Since the early 2000s, the average per year performance increase of single-microprocessors has significantly slowed down due to physical constraints. In the mid 2000s, most semiconductor manufacturers changed their processor architectures with a focus on *concurrency*. They started designing single integrated circuits with multiple processors [4, 5].
To take advantage of today's advancements in computational power, it is essential to leverage the benefits of multiple processors. Software developers have to build concurrency into their programs. Programming languages such as C, C++, and Java have extensions that explicitly allow creating concurrent[1] programs running on multiple threads [4].
A thread is a stream of control within a process that allows the execution of multiple tasks concurrently, yielding better responsiveness and throughput [7]. Although concurrent programs offer significant performance benefits, the complexity of software has expanded along with it. The growing potential for errors has made it challenging to maintain confidence in software systems. Software errors in essential sectors such as transport, communication, healthcare and energy can cause economic or safety disasters [8].
Software verification aims to prevent these errors by using mathematical methods to ensure that a program behaves as intended, i.e. aligns with its specification. Verification is part of formal methods that include logic-based techniques, such as software model checking [8]. Model checking is an automated technique used to verify whether a hardware or software system meets certain specifications by

[1] Concurrent and parallel may be used synonymous in computer science literature [6]. The definitions of the two terms differ [6, 7]. In this thesis, we will consistently use the term concurrent when referring to programs running on different threads.

exploring its possible states, i.e. the state space. With a growing number of state variables, the systems state space increases at an exponential rate known as the *state explosion problem*. This problem is especially present in concurrent programs running on multiple threads. Non-deterministic *context switches*, i.e. switching from one thread to another in the execution, can occur at every program location, further increasing the state space [1].

Goals. This thesis aims to reduce the state space when verifying a concurrent program with two techniques:

1. Output an equivalent sequential program, i.e. a sequentialization, for a concurrent input program. A sequentialization can be given to any verifier capable of analyzing sequential programs, making it *modular*. This enables the use of highly-tuned sequential reachability algorithms for concurrent programs [9].

2. Reduce the state space using a *partial order reduction* technique with assumptions over allowed transitions within the sequentialization to mitigate the state explosion problem for reachability verifiers.

Roadmap. Chapter 2 outlines the foundations of the verification of concurrent programs: reductions in the state space and sequentializations. Chapter 3 details a partial order reduction technique, namely that of Ultimate GemCutter [1], and the non-deterministic sequentialization approach of Lazy-CSeq [3]. Chapter 4 highlights our contribution i.e. a sequentialization with partial order reduction assumptions. Chapter 5 compares the analysis performance of our sequentializations (with and without partial order reduction) compared to their corresponding input programs in CBMC, CPAchecker and UAutomizer. Chapter 6 provides an overview of the key findings and the future direction of our implementation. Chapter 7 features instructions on how to use our implementation and explains supported functions, error messages and abbreviations used in this thesis.

CHAPTER 2

Background

2.1 pthreads

Portable Operating System Interface (POSIX) threads or *pthreads* is a C extension that allows software developers to create concurrent C programs. The library contains management functions for the creation, suspension, cancellation, scheduling and synchronization of threads [7]. Our implementation currently supports the pthread functions `pthread_create`, `pthread_mutex_{lock, unlock}` and `pthread_join`.
In order to create a thread with pthreads, the function `pthread_create` is used. Its function declaration is given as [10]:

```
1 int pthread_create(pthread_t *thread,
2                    const pthread_attr_t *attr,
3                    void *(*start_routine)(void*),
4                    void *arg);
```

If `pthread_create` completes successfully, the thread specified by its identifier `pthread_t` immediately starts executing the function `start_routine` with `arg` as its single function parameter. Before any statement in `start_routine` can be executed, the main thread has to execute at least the respective call to `pthread_create` first [10].
Concurrent programs may contain critical sections where threads should not interfere with each other. pthreads allows the usage of mutual exclusion locks (*mutexes*) to prevent thread interference. Objects of the type `pthread_mutex_t` are initialized with the function `pthread_mutex_init` and can be used to prevent critical sections of different threads from interfering with each other. `pthread_mutex_t` objects can be used in the functions `pthread_mutex_lock` and `pthread_mutex_unlock` whose declarations are given as [10]:

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);
2 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```c
// thread 1
extern int x;
int i;
void *t1(void *arg) {
    x = 1;
    if (y == 1) { i = 42; }
    return ((void *)0);
}
```

```c
// thread 2
extern int y;
int j;
void *t2(void *arg) {
    y = 1;
    if (x == 1) { j = 42; }
    return ((void *)0);
}
```

Figure 2.1: A simple concurrent C program running on two threads. It contains global variables `x`, `y` and local variables `i`, `j`.

If `pthread_mutex_lock` completes successfully, the mutex object `pthread_mutex_t` is locked until the thread that locked it unlocks it via `pthread_mutex_unlock`. If another thread tries to call `pthread_mutex_lock` to a locked mutex object, it blocks (i.e. waits) until the mutex is unlocked. Thus it is guaranteed that the thread that locked the mutex leaves the critical section before any other thread can interfere [10].
If a thread *a* works with the computational result of another thread *b* then *a* can wait for the termination of *b* with the function `pthread_join` whose declaration is given as [10]:

```c
int pthread_join(pthread_t tid, void **status);
```

If `pthread_join` is called, the calling or waiting thread blocks until the thread with the `pthread_t` object (the target thread) has terminated [10].

2.2 Interleaving

The usage of libraries such as pthreads, i.e. creating concurrent programs induces enormous complexity that stems from non-deterministic context-switches. When performing a context switch, thread statements are *interleaved*. In a program with $n \geq 2$ threads, there can be as few as $n - 1$ context switches (i.e. all threads execute sequentially). There can also be a context switch at every control location. An *interleaving* thus forms a global sequence and reflects a possible execution of a concurrent program. When verifying a concurrent program, all interleavings have to be considered [1].
Bounded Model Checking (BMC) is a verification technique that addresses the state explosion problem by searching for counterexamples within a bounded number of transitions in a system. BMC is often used in the verification of concurrent programs [3, 11]. In addition, the number of threads is often bounded when verifying concurrent programs. This excludes e.g. a (seemingly) infinite loop creating new threads from verification making the number of statements, permutations of statements and context-switches finite [1], though research for an unbounded number of threads exists [12].

Take for example the simple concurrent program in Fig. 2.1 and their Control Flow Automata (CFA) representations in Fig. 2.2. There are 2 threads with 5 locations each, giving us a total of $5^2 = 25$ possible states in a concurrent execution [11]. Note that the number of states should not be confused with the number of interleavings, where an exact number is more difficult to obtain if the CFA contains multiple leaving edges for at least one node. Assuming that each CFA node has exactly one leaving edge except the exit node which has no leaving edge, the multinomial coefficient [13] can be used to calculate the number of interleavings:

$$\binom{n}{k_1,\ldots,k_m} = \frac{n!}{k_1!\ldots k_m!}$$

Where n is the sum of edges for all thread CFAs and $k_{1 \leq i \leq m}$ is the number of edges in the CFA of thread i. Given that adding additional edges to a CFA increases the number of interleavings, we can assume the CFAs in Fig. 2.2 to have at least 70 interleavings:

$$\frac{n!}{k_1!\ldots k_m!} = \frac{(4+4)!}{4!4!} = \frac{8!}{4!4!} = 70$$

Despite being a very simple program, it demonstrates the complexity of a concurrent execution.

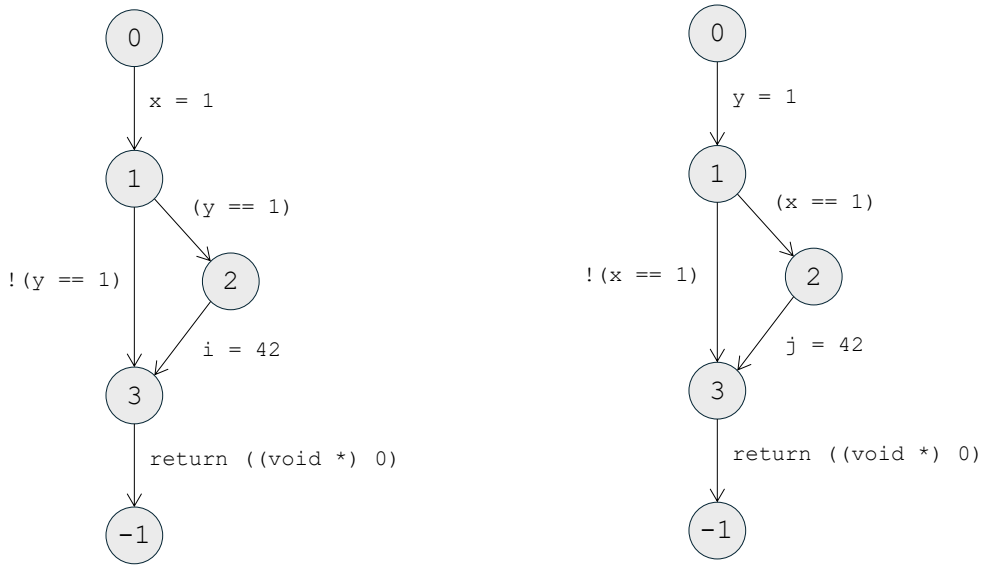


Figure 2.2: Exemplary CFAs for each thread from the concurrent program in Fig. 2.1. The directed transitions or CFA *edges* represent program statements. The CFA *nodes* represent program locations.

2.3 Commutativity

For verification purposes, the amount of interleavings can be reduced by factoring in *commutativity*. In an interleaving, two statements from different threads commute

if their order of execution does not matter (i.e. the assumptions over program variables are equivalent) [1].

In our example from Fig. 2.1, take the first statements of both threads `x = 1` and `y = 1` which we define as *a* and *b* respectively. If both threads are at location `1`, denoted as the sequence $\langle$ `1, 1` $\rangle$, then *a* and *b* were the first two statements to be executed. Let *I* be the set of possible interleavings from the state $\langle$ `1, 1` $\rangle$. Given that *a* and *b* are independent events, it holds that for all $i \in I$ executing *ab* or *ba* is equivalent and reaches the variable values denoted as:

$$\{\texttt{x} \longmapsto 1, \texttt{y} \longmapsto 1, \texttt{i} \longmapsto 0, \texttt{j} \longmapsto 0\}$$

Considering only the commutativity of *a* and *b*, the amount of interleavings from the state $\langle$ `1, 1` $\rangle$ is effectively halved.
For non-commutativity, let us consider the state $\langle$ `1, 0` $\rangle$, i.e. thread 0 has executed `x = 1` which we defined as *a* earlier. The next statement of thread 1 is `y = 1` which we defined as *b*. Let *c* be the assumption on `y` in thread 0, i.e. `y == 1`. Executing *abc* results in `y == 1` evaluating to `true` and `i` being assigned `42`. The reached variable values are:

$$\{\texttt{x} \longmapsto 1, \texttt{y} \longmapsto 1, \texttt{i} \longmapsto 42, \texttt{j} \longmapsto 0\}$$

Conversely, *acb* results in `y` having the initial default `int` value of `0` and `y == 1` evaluating to `false`. In this case, `i` also has the initial value of `0` when both threads have terminated. The reached variable values are:

$$\{\texttt{x} \longmapsto 1, \texttt{y} \longmapsto 1, \texttt{i} \longmapsto 0, \texttt{j} \longmapsto 0\}$$

Therefore, the ordering of *b* and *c* in the execution matters for the resulting variable values and they do not commute.

2.4 Total Strict Order

Another example of non-commutativity are Total Strict Orders (TSO), i.e. an order in which statements must be executed. In a concurrent program, a specific combination of thread locations may induce a TSO [1].
A simple example is a TSO induced by the creation of a thread. Let us expand the example from Fig. 2.1 with main thread that creates the other two threads in Fig. 2.3. The start routine function of a thread (`t1` and `t2`) can only be executed once the respective call to `pthread_create` successfully returns [10].
This induces a clear TSO over the set of statements before and including the call to `pthread_create` (*preceding*) and the set of statements executed by the `start_routine` function (*subsequent*). This order reduces the amount of statement interleavings that must be considered by a verifier [1].

```
1 // thread 0 (main thread)
2 int main() {
3     pthread_t id1, id2;
4     pthread_create(&id1, ((void *)0), t1, ((void *)0));
5     pthread_create(&id2, ((void *)0), t2, ((void *)0));
6     return 0;
7 }
```

Figure 2.3: Main thread creating threads 1 and 2 from Fig. 2.1. The threads respective start routines `t1` and `t2` are passed on as the third parameters.

2.5 Partial Order Reduction

In contrast to a TSO, a partial order does not enforce a strict order. The underlying idea of a Partial Order Reduction (POR) is that commuting statements can be executed in any order because they result in equivalent assumptions over program variables. POR techniques are one of the most widespread methods when verifying concurrent programs [11].
The commutativity of statements allows the creation of equivalence classes of interleavings. In order to prove program correctness, one representative from each equivalence class has to be verified. A POR algorithm may extract the smallest interleaving which introduces the least verification overhead [1].
Usually, it is not feasible to construct a CFA containing *all* interleavings and then prune it. Instead, POR algorithms work on the fly: at every control location it is decided if an edge can be soundly excluded from execution [1, 11]. POR algorithms can reduce the state space complexity from exponential to polynomial in the number of program statements [1].

POR can be demonstrated by considering the example program in Fig. 2.1 from state $\langle$ `1,1` $\rangle$. Given that no pair of statements read or write the same variables (or more specifically, the same memory locations) from $\langle$ `1,1` $\rangle$ and onwards, all statements are independent events and commute. Generally, local variable accesses always commute because only one thread is involved and other threads cannot interfere. Only global variable accesses have to be considered for commutativity. The set I defined earlier thus forms an equivalence class and choosing any $i \in I$ is sufficient to show program correctness. This effectively reduces the amount of interleavings to consider from state $\langle$ `1,1` $\rangle$ to 1.
Similarly, the same logic can be applied to the states $\langle$ `2,0` $\rangle$ and $\langle$ `0,2` $\rangle$. Going forward from both states, all pairs of statements commute. Thus, in theory, we can reduce the number of interleavings in our simple example program in Fig. 2.1 from at least 70 to 3 for verification purposes.
Thread 0 from Fig. 2.3 introduces additional complexity and, neglecting the TSOs induced by thread creations, the number of interleavings grows to at least 34650:

$$\frac{n!}{k_1! \dots k_m!} = \frac{(4+4+4)!}{4!4!4!} = \frac{12!}{4!4!4!} = 34650$$

Given that the main thread does not read or write the global variables `x` and `y`, all pairs of statements between threads 0, 1 and threads 0, 2 commute. Thus, thread 0 can terminate before being interleaved with thread 1 or 2 and the execution of thread 0 does not introduce additional complexity during verification. The number of interleavings to verify remains 3.

2.6 Sequentialization

A *sequentialization* is the transformation of a concurrent program into an equivalent sequential program [14]. A sequentialization may be a source-to-source transformation or a transformation based on an automaton of a given input program. A verifier takes as input a concurrent program, processes it and outputs it as a new sequential and non-deterministic program. This pre-processing step alters the concurrent input program by adding control code that simulates context switches between threads. The outputted sequential transformation can be given to a verifier capable of analyzing sequential programs. Sequentializations can introduce assumptions over the program, e.g. if a thread is currently active, to reduce the state space [3, 15, 16].

CHAPTER 3

Related Work

To achieve the goals of this thesis introduced in Chapter 1, we extract ideas from the verification tools UGemCutter [1] and (Lazy-)CSeq [3, 17]. Both UGemCutter and (Lazy-)CSeq took part in editions of the Competition on Software Verification (SV-COMP) that compares verification tools from both industry and research around the world. It is hosted yearly since its first edition in 2012. Different competition categories such as *ConcurrencySafety* for concurrent programs emphasize the strengths and weaknesses of a tool [18, 19].

3.1 Ultimate GemCutter

UGemCutter is a tool analyzing concurrent C programs with a POR approach. UGemCutter placed 2nd in the ConcurrencySafety category in the SV-COMP 2024 [19] and uses automata similar to the CFAs introduced in Fig. 2.2 to internally represent a program with locations and transitions. Before detailing UGemCutters POR technique, let us first introduce the terms and symbols necessary for its understanding [1]:

- $l_i \rightsquigarrow l_j$ is the conflict relation between the locations l_i, l_j of two threads with $i \neq j$. A conflict occurs if an outgoing edge a from l_i and an outgoing edge b from some location l_j' that is reachable from l_j do not commute.
- $enabled_{T_i}(l_i)$ is the set of outgoing edges for the current location of thread T_i.
- $a <_q b$ is the notation of the positional TSO over a and b in program location q.
- Σ is the alphabet i.e. set of all program statements for all threads in the concurrent program automaton P.
- *enabled(q)* is the union of sets of outgoing edges for all threads $T_1, .., T_n$ in the program location $q = \langle\, l_1 \,, \ldots \,, l_n \,\rangle$ i.e. $enabled_{T_1}(l_1) \cup \ldots \cup enabled_{T_n}(l_n)$ with $n \geq 2$.

Preprocessing step: *Compute the conflict relation, i.e., the set* $\{(\ell_i, \ell_j) \mid \ell_i \rightsquigarrow \ell_j\}$

Procedure `CompatiblePersistentSet(`q`)`**:**

Input: state $q = \langle \ell_1, \ldots, \ell_n \rangle$ of program P
Output: set M such that M is weakly persistent at q
$active \leftarrow \left\{ i \in \{1, \ldots, n\} \mid enabled_{T_i}(\ell_i) \neq \emptyset \right\}$
$conflicts \leftarrow \{ (i, j) \in active^2 \mid \ell_i \rightsquigarrow \ell_j$
$\quad \vee \exists a \in enabled_{T_j}(\ell_j), b \in enabled_{T_i}(\ell_i) \,.\, a <_q b \}$
$SCCs \leftarrow$ strongly connected components of graph ($active$, $conflicts$)
$E \leftarrow$ topologically maximal SCC in $SCCs$
return $M := \bigcup_{i \in E} enabled_{T_i}(\ell_i)$

Figure 3.1: Computation of Weakly Persistent Sets [1]. Strongly connected components (SCCs) of directed graphs are subgraphs where all pair of nodes have a path to each other [2].

- $a \circlearrowright_\varphi b$ is the symmetric commutativity relation under condition φ. a and b commute from a program location satisfying the assertion φ.

(Weakly) Persistent Sets. UGemCutter uses the commutativity of statements in a concurrent program to reduce the state space in its POR technique. Firstly, a *persistent set* K is calculated which is a subset of the outgoing edges *enabled(q)* of the program location q. On the other hand, K' with $K \cap K' = \emptyset$ is the set of edges that are either enabled in q or reachable from q through edges $k' \notin K$. If for all $k \in K$, $k' \in K'$ it holds that k and k' are independent (i.e. commute) then $K \subseteq$ *enabled(q)* forms a persistent set in program location q [20].
Persistent sets take into consideration all runs. This makes *weakly* persistent sets (WPS) less restrictive because they only consider sequences of edges that reach a final location with no outgoing edges in the program automaton P [1].

Algorithm Fig. 3.1 is based on the idea of persistent sets. It constructs a directed graph of threads that are in conflict. Threads that have terminated are not taken into consideration, only *active* threads represent nodes in the graph. *Conflicts* between thread locations represent the directed edges. If two thread locations l_i, l_j are in conflict or if there is a TSO on any pair of their outgoing edges, a conflict edge from i to j exists.
The algorithm continues by computing the SCCs of the directed graph and extracting an SCC where no node is connected to another SCC (we call E the *topologically maximal* SCC). E is conflict-closed i.e. if for a node $i \in E$ it holds that $l_i \rightsquigarrow l_j$ (meaning an edge from i to j exists) then $j \in E$. Additionally, a path but not necessarily a direct edge from j to i exists as per the definition of SCCs. Next, the union of enabled edges for all threads in E forms the elements in the WPS at location q [1].

A priori: $V \leftarrow \emptyset$

Procedure `CheckProof`(q, φ, S)**:**

Input: state q of program P, sleep set $S \subseteq \Sigma$,
assertion φ of Floyd/Hoare automaton $\mathcal{A}$

if $\langle q, \varphi, S \rangle \in V$ **then return**

else if $q \in F$ *and* $\varphi \not\models$ *post* **then** "ctex. found"

$V \leftarrow V \cup \{ \langle q, \varphi, S \rangle \}$

for $a \in$ `CompatiblePersistentSet`$(q) \setminus S$ **do**

$S' \leftarrow \{b \in \mathit{enabled}(q) \mid (b \in S \vee b <_q a) \wedge a \leftrightarrows_\varphi b\}$

`CheckProof`$(\delta_P(q, a), \delta_{\mathcal{A}}(\varphi, a), S')$

end

Figure 3.2: Recursive procedure used by the proof check with proof-sensitive sequentialization on the fly [1].

The SCC definition implies that if E has a single element, the thread has no conflicts with another thread. If E has multiple elements, the threads only have conflicts with each other and no conflicts with the threads in E' with $E \cap E' = \emptyset$. Thus, the WPS calculated by the algorithm Fig. 3.1 prefers leaving edges of location q that have minimal conflicts.

(Positional) Lexicographic Preference Orders. The concept of Lexicographic Preference Orders (LPO) works closely with TSOs. In location q, a *positional* LPO adjusts the order of continuing statements based on the past sequence of statements (the *context*). It allows for dynamic reordering of commuting statements that adapts as the system transitions through different locations. Positional LPOs ensure that for a context c that reached location q, appending a suffix s to it respects the TSO of q and that for $a <_q b$ the accepted sequence of statements i.e. interleaving *cas* is *preferred* over *cbs*. The set of interleavings accepted by the automaton P is reduced by positional LPOs. Each equivalence class of interleavings has one unique representative in the positional LPO reduction with only the minimal interleaving with the least verification overhead being accepted. This forms the basis of UGemCutters POR algorithm [1].

Sound Sequentialization with Sleep Set Pruning. Persistent sets work closely with *sleep sets*, the second technique used by UGemCutters POR approach. A sleep set is a set of program statements or edges that can also be soundly pruned from execution. But in contrast to persistent sets that consider commutativity (i.e. the syntax of program statements), sleep sets consider only the previous state exploration information. In some cases, persistent sets cannot prevent the execution of commuting edges in a location. Using persistent and sleep sets in tandem can avoid the exploration of various interleavings of these commuting edges [20].

The recursive sequentialization algorithm of UGemCutter is given in Fig. 3.2. It uses separate automatons for the concurrent program statements and locations (P) as well as its assertions ($\mathcal{A}$). S is the sleep set and initially empty. V keeps track of already verified sequences of locations q, assertions φ and sleep sets S. F is the set of final locations that have no outgoing edges in P. A counterexample is found if the assertion φ violates the programs post condition i.e. does not meet its specification. A selective search on outgoing edges of the program location q is performed by computing the WPS (cf. Sect. 3.1). Edges that are both in the WPS and the current sleep set S are soundly pruned from execution. For every edge a in the WPS that is not pruned, a new state is visited via the transition functions δ of P and $\mathcal{A}$ respectively. Before that, the sleep set S is updated to S' by selecting edges $b \in$ *enabled(q)*. If $a \circlearrowright_{\varphi} b$ holds and b was in the sleep set S before or there is a positional TSO $a <_q b$, then b is added to the udpated sleep set S'.

The commutativity of a and b is used to maximize the sleep set by executing a before b despite the positional TSO $b <_q a$. Thus b does not have to be considered in subsequent locations q' and is pruned i.e. added to the sleep set.
Additionally, the WPS calculated beforehand focuses on conflict-minimal edges described in Sect. 3.1. Thus, the amount of edges $b \in$ *enabled(q)* that are in conflict with the edges a in the WPS and for which $b <_q a$ holds is maximized.
The algorithm in Fig. 3.2 considers exactly the minimal representative from each equivalence class of interleavings and thus sound, forming the basis of its POR technique. The authors showed that their algorithm has at best polynomial complexity in the sum of program statements for all threads [1].

3.2 CSeq

CSeq is a family of verification tools that transform concurrent programs into equivalent sequential programs i.e. sequentializations. It simulates context switches by non-deterministically increasing a round counter `k` up to the bound `K-1`. In each round `k`, a context switch may occur. If the counter reaches the bound, an early exit can be triggered to simulate thread termination. This simulation code is inserted into the original program to represent the behavior of the concurrent input program. The simulation begins with the first thread accessing a copy of the memory and subsequent threads proceed with the memory state left by the previous thread. At the end of the simulation, CSeq asserts that each threads final memory state aligns with the initial guesses so that the simulation corresponds to a possible execution path [17].

pthread control methods are replaced with custom functions simulating their behavior. E.g. `pthread_create` is replaced by `cseq_create` which stores a pointer to the newly created threads function in an array. This array keeps track of the order in which threads are created and their corresponding round. The main function is treated as the first thread, starting in round 0. Thread termination (`pthread_exit`)

```
1  int main() {
2      for(r=1; r<=K; r++) {
3          ct=0;
4          if(active[ct]) {                  // only active threads
5              cs=pc[ct]+nondet_uint();      // next context switch
6              assume(cs<=size[ct]);         // appropriate value?
7              main_thread();                // thread simulation
8              pc[ct]=cs;                    // store context switch
9          }
10         // ...
11         ct=2;
12         if(active[ct]) {
13         // ...
14         }
15     }
16 }
```

Figure 3.3: A `main` function produced by Lazy-CSeq featuring non-deterministic context switches [3].

and joining (`pthread_join`) are handled with CSeq-specific code. The `cseq_exit` function updates the threads status upon termination, while `cseq_join` uses an `assume` statement to ensure that a thread only continues once its target thread has finished executing. Similarly, mutex operations such as locking and unlocking are replaced by CSeq-specific code that tracks the status of locks using integer variables [17].

Lazy-CSeq. Lazy-CSeq [3] is a verification tool for concurrent C programs which won the ConcurrencySafety category in SV-COMP 2015 [21], 2020 [22] and 2021 [23]. Lazy-CSeq injects a new `main`[1] method into a sequentialization that allows for non-deterministic context switching (cf. Fig. 3.3) with the help of numerous injected variables:

- `r`: An `int` value tracking the current round, bounded to K.
- `ct`: An `int` thread identifier.
- `active`: A `bool` array tracking if threads have terminated yet.
- `cs`: The `pc` at which the next context switch occurs.
- `pc`: The `int` program counters for all threads as per Fig. 2.2.
- `size`: An `int` array tracking how many statements a thread contains in total.

A `for` loop with a bounded number of iterations (K) handles thread simulations i.e. continue executing a thread. Program counters are injected into the source code to identify and continue thread simulation from a certain point in the program via parametrized `goto` statements. One thread simulation run executes n statements

[1]The original `main` method is renamed to `main_thread` as per Fig. 3.3 line 7.

where n is 0 or a positive `int` (i.e. an unsigned `int` from `nondet_uint()`) before a context switch occurs, covering all possible interleavings. pthread methods are replaced with custom implementations, e.g. `pthread_create_2` sets the `active` index of the created thread to `true` [3].

Differences to our Approach. Our non-deterministic sequentialization introduced in the next Chapter 4 uses a variant of the context non-determinism method introduced by Lazy-CSeq where thread simulations are cycled through in a loop. At the loop head, we inject assumptions over allowed transitions between thread simulations to reduce the state space. We also introduce our own variable management and replace pthread control methods with variables to simulate their behavior via assumptions.
In contrast to (Lazy-)CSeq, our approach does not re-implement pthread methods or features injected assertions, the latter of which will be handled by the tool analyzing the sequential transformation. In our approach, a thread simulation run executes a single statement. Furthermore, we do not simulate the entire memory state, but reuse exactly the variables from the input program.

CHAPTER 4

Approach

Our algorithm aims to combine ideas of both UGemCutters and (Lazy-)Cseq introduced in Chapter 3. In this chapter, the details and limitations of our implementation are discussed. The basis for our analysis and transformation of concurrent programs is CPAchecker [24], a framework for the configurable verification of C programs, based on Configurable Program Analysis (CPA). CPAchecker uses a CFA with nodes and egdes (cf. Fig. 2.2) to internally represent an input program. CPAcheckers CFA implementation is highly sophisticated and a well suited starting point for our code transformation compared to building a source-to-source transformation from scratch.

4.1 Automata Transformation

Our implementation derives thread-specific CFAs which are subsets of the input CFAs nodes and edges that are reachable by a thread. The set of threads contains the main thread and all threads created in `pthread_create` calls, i.e. pthreads. The input CFA is searched for edges that are calls to `pthread_create`. If an edge is found, it is treated as a single created thread.
The `start_routine` function, i.e. the third parameter of `pthread_create`, is extracted from the function call. The input CFA is searched for the `start_routines` entry node which serves as the starting node of the threads CFA while the exit node marks the threads termination.
The `pthread_t` object, i.e. the first parameter of `pthread_create`, is also extracted from the function call and serves as a unique thread identifier. These identifiers are important when dealing with the semantics of pthread functions that use `pthread_t` objects [10].

Despite not being used, the `pthread_t` variables are still declared inside the sequentialization. Generally, all variables that are declared in the input program

are also declared in the sequentialization. Each variable is given a new name in the sequentialization, i.e. it is *substituted*. In the sequentialization, all local variables from the input program are global. To avoid naming collisions, a unique unsigned `int` identifier is created for all variable names. In this thesis however, we use substitute names of the form `G_{var_name}` for global and `L{thread_id}_{var_name}` for thread-local variables to improve readability. The appendix features a Table 7.3 with shortened variable and function names in this thesis and their actual names in the sequentialization.

The `main` function of the sequentialization is used to simulate context-switches and interleavings. The `main` function non-deterministically switches between thread simulations in a loop. Each thread is simulated with a switch case that contains all possible control locations of a thread simulation. One execution of the loop executes at most one case statement which may represent multiple statements of a thread simulation.
Furthermore, the sequentialization introduces an `assume` function:

```
1 assume(const int cond) {
2     if (cond == 0) { abort(); }
3 }
```

It is a helper function to create assumptions over variables and allowed transitions. If `cond` evaluates to `0`, the function calls `abort()`, terminating the execution. This ensures that only interleavings satisfying `cond` are considered by a verifier, reducing the amount of program states to explore.
An assumption may be an implication of the form $a \Rightarrow b \iff \neg a \vee b$ which can be encoded into the sequentialization via `assume(!a || b)`. To improve readability in this thesis, we use the (non-existent) C expression `a ==> b` for implications which we define as equivalent to `!a || b`.

Example. Let us analyze the example concurrent program from Fig. 2.1 and Fig. 2.3 introduced in Chapter 2. Recall that the number of interleavings for threads 0, 1 and 2 is at least 34650 without factoring in TSOs. Our sequentialization approach preserves this complexity while still being human readable. This is demonstrated in the sequentialization in Fig. 4.1 which is equivalent to the concurrent program. We inject various variables that allow thread simulations:

- The `next_thread` variable is assigned non-deterministically to a value between `0` and `2`.
- The `int` array `pc` tracks the current location of all thread simulations.
- `Ti_ACTIVE` variables are introduced to track whether a thread simulation `i` is currently active. The `ACTIVE` variable of the main thread is initialized to `1`, all other to `0`.
- The assumption `assume(pc[next_thread] != -1)` implies that if all thread simulations have reached an exit location, the analysis has finished.

```
1  // sequentialization control variables
2  const int NUM_THREADS = 3;
3  int pcs[] = { 0, 0, 0 };
4  int next_thread = -1;
5
6  // thread simulation variables
7  int T0_ACTIVE = 1, T1_ACTIVE = 0, T2_ACTIVE = 0;
8
9  // global variables
10 extern int G_x;
11 extern int G_y;
12
13 // local variables of thread 0, 1, 2 respectively
14 pthread_t L0_id1, L0_id2;
15 int L1_i;
16 int L2_j;
17
18 while (1) {
19    // assign next_thread non-deterministically and bound to [0,2]
20    next_thread = __VERIFIER_nondet_int();
21    assume(0 <= next_thread && next_thread < NUM_THREADS);
22
23    // iterate loop until all pcs are -1
24    assume(pc[next_thread] != -1);
25
26    // enforce TSOs: only active threads can execute
27    assume(!T0_ACTIVE ==> next_thread != 0);
28    assume(!T1_ACTIVE ==> next_thread != 1);
29    assume(!T2_ACTIVE ==> next_thread != 2);
30
31    // represent each thread simulation with a switch statement
32    if (next_thread == 0) {
33       switch (pc[0]) {
34          case 0: T1_ACTIVE = 1; pc[0] = 2; continue;
35          case 2: T2_ACTIVE = 1; T0_ACTIVE = 0; pc[0] = -1; continue;
36       }
37    } else if (next_thread == 1) {
38       switch (pc[1]) {
39          case 0: G_x = 1; pc[1] = 1; continue;
40          case 1: if (G_y == 1) { pc[1] = 2; }
41                  else { pc[1] = 3; } continue;
42          case 2: L1_i = 42; T1_ACTIVE = 0; pc[1] = -1; continue;
43       }
44    } else if (next_thread == 2) {
45       switch (pc[2]) {
46          case 0: G_y = 1; pc[2] = 1; continue;
47          case 1: if (G_x == 1) { pc[2] = 2; }
48                  else { pc[2] = 3; } continue;
49          case 2: L2_j = 42; T2_ACTIVE = 0; pc[2] = -1; continue;
50       }
51    }
52 }
```

Figure 4.1: The body of the `main` method in the sequentialization handling non-deterministic context-switches.

- The `assume` statements in lines 27 to 29 enforce the TSOs induced by the calls to `pthread_create`. The assumptions exclude non-active threads from executing.
- Switch statements are derived from the CFA representations of each thread in the input program. The case labels `case n:` represent a single CFA node `n`. The case blocks, that is, the statements executed in a case, represent one or multiple leaving CFA edges of `n`.
- Statements like `pthread_t` declarations and `return` statements from the input program (cf. Fig. 2.1 and Fig. 2.3) are deemed unnecessary and pruned in the sequentialization, reducing the state space.

Though correct, the sequentialization only enforces the necessary TSOs over thread creations to reduce the amount of possible interleavings. The commutativity of statements and the POR technique are not factored in via `assume` statements. This is one of the two goals of this thesis introduced in Chapter 1 and elaborated on further in this chapter.

4.2 Inlining Functions

The sequentializations of input programs produced by our implementation only contain function definitions (i.e. functions with a body) for `reach_error` (elaborated on later in Sect. 4.5), `assume` and `main`. All other functions from the input program are inlined, i.e. included inside the `main` function. Thus, all statements from different threads and different functions can be interleaved individually. The inlining requires us to store as well as retrieve original calling contexts, simulate parameter values and replace return statements.

For function parameters, we introduce variables with names of the form `P{thread_id}_{var_name}`. The values passed to the function are assigned to the corresponding parameter variables. These parameter variables are thread-unique because each thread-specific CFA has its own CFA representation of the functions called by the thread. We introduce thread-specific nodes and edges that store the original nodes and edges from the input program. Thus, e.g. two thread nodes in two different threads CFAs may represent the same original node of a function CFA. This allows us to interleave the same function called by different threads.

If a function is defined in the input program, we create a CFA representation for the functions body that allows us to identify the original calling context of a function call. A function may be called numerous times by a thread but we only inline functions once for each thread. Thus we introduce `RETURN_PC` variables that store the original context in the sequentialization. `RETURN_PC` variables are unique for each thread and function. The name for a `RETURN_PC` has the form `T{thread_id}_RETURN_PC_{func_name}`.
The edges returning to the calling context are initially blank i.e. they contain no statements in the input CFA [24]. In the sequentialization, we add a statement to

these return edges where we assign the stored `RETURN_PC` to the `pc` of the current thread.

If the return value of a function is used, the sequentialization replaces both the function call and the return statements of a function. Otherwise, return statements present in a threads case statements would stop the `main` function from executing or prevent compiling the C program if the return data types do not match.
The input program may store the return value of a function explicitly, e.g. `int var = func()`, or implicitly, e.g. `if (func()) { ... }` or `return func()`. In both cases, the expression of the return statement is extracted and assigned to variable storing the return value. These assignments are context-sensitive i.e. we only update the variable storing the return value that is linked to the current value of the `RETURN_PC` in a switch statement.

Example. Let us consider the example program in Fig. 4.2 and its CFA. Even though it contains only the main thread, it is sufficient to showcase our approach to representing a thread and a function call in the sequentialization with a switch case. The step by step procedure of our algorithm is as follows:

1. Declare the global variables `int G_x`, `int L0_y` and `int L0_CPA_TMP` as substitutes for the global variable `x` and the local variables `y` and `CPA_TMP`, respectively.

2. Declare a global variable `int P0_x` for the parameter declaration `x` of the function `is_even`.

3. Declare a global variable `T0_RETURN_PC_is_even` to store the calling context when calling the function `is_even`.

4. For each node in the threads CFA, create a case statement in the threads switch case in the `main` functions loop.

5. Prune unnecessary cases to reduce the state space.

The resulting switch case representation of the CFA is shown in Fig. 4.3.
Case `0` and `1` highlight the separation of variable declarations and assignments. All declarations are made global and put outside the `main` function in the sequentialization, which is why they are not featured in Fig. 4.3. The switch statements only feature variable assignments and not declarations, otherwise the variables would be out of scope in other cases.
Case `2` showcases the assignments of the parameter variable `P0_z` and the `RETURN_PC` to the successor node of the storing the calling context, i.e. `3`. In addition, case `2` jumps into the CFA of the `is_even` function, i.e. case `7`.
In case `7`, the original return statement of `is_even` is replaced with the context-sensitive assignment of the variable storing the return value (in this case `L0_CPA_TMP`) which is extracted from the edge storing the calling context. The assigned value is the expression of the return statement, i.e. `P0_z % 2 == 0`. Note

```c
int x = 21;
int is_even(int);
int main();
int is_even(int z) { return z % 2 == 0; }
int main(void) {
    int y = 21;
    if (is_even(x + y)) { printf("even"); }
    else { printf("odd"); }
    return 0;
}
```

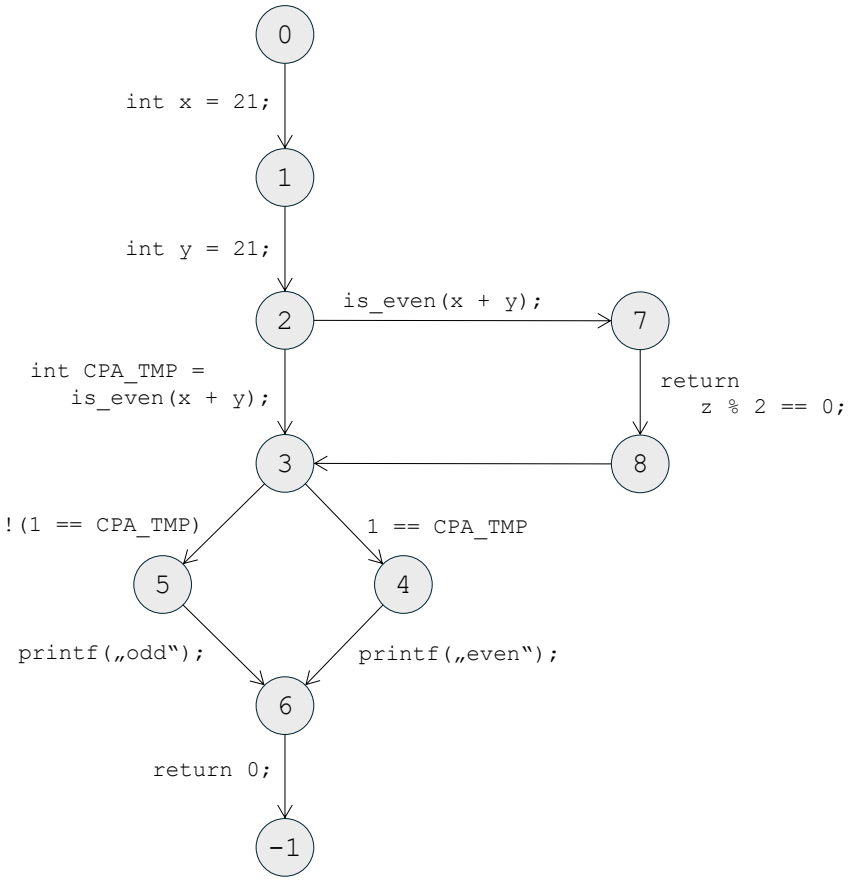


Figure 4.2: A sequential C program and its simplified CFA. CPAchecker introduces `__CPAchecker_TMP` variables (abbreviated to `CPA_TMP` in this example) that store the return values of a function if they are implicitly used.

how `z` was substituted with its parameter variable `P0_z`. Case `8` has initially no statement i.e. is blank, but we assign the value of the corresponding `RETURN_PC` (in this case `3`) to the `pc` of thread 0 to jump back to the original calling context.

Case `3` represents the handling of control flow statements that are all replaced with `if` and `else if` statements in the switch cases, meaning that loops do not require explicit handling in the sequentialization. All loop statements from different threads can thus be interleaved individually.

```
switch (pc[0]) {

    // case 0 to 5 from main()
    case 0: G_x = 21;
            pc[0] = 1; continue;
    case 1: L0_y = 21;
            pc[0] = 2; continue;
    case 2: P0_x = G_x + L0_y;
            T0_RETURN_PC_is_even = 3;
            pc[0] = 7; continue;
    case 3: if (1 == L0_CPA_TMP) { pc[0] = 4; }
            else if (!(1 == L0_CPA_TMP)) { pc[0] = 5; }
            continue;
    case 4: printf("even");
            pc[0] = 6; continue;
    case 5: printf("odd");
            pc[0] = 6; continue;
    case 6: T0_ACTIVE = 0;
            pc[0] = -1; continue;

    // case 7 to 8 from is_even(int x)
    case 7:
        switch (RETURN_PC_T0_is_even) {
            case 3: L0_CPA_TMP = P0_x % 2 == 0;
        }
        pc[0] = 8; continue;
    case 8: pc[0] = RETURN_PC_T0_is_even; continue;
}
```

Figure 4.3: The switch case of the main thread in Fig. 4.2 derived from its CFA.

4.3 Thread Simulation

Not only thread objects such as `pthread_t` but also thread control methods are removed in the sequentialization. We inject thread simulation variables into the sequentialization to correctly reproduce the behavior of a thread control method. Input programs usually do not define thread control methods but include the header file or declare the methods themselves. In this case, the thread control methods have no function CFA representation in contrast to the function calls in Sect. 4.2.
We distinguish between thread control methods that are:

1. explicitly handled
2. supported but need no explicit handling
3. unsupported, resulting in an input program rejection

The appendix features a non-exhaustive list of supported thread control methods Table 7.1.
Our algorithm identifies explicitly handled thread control methods and takes care of their semantics, i.e. their induced TSOs. To enforce a TSO in the sequentialization, we include assumptions over allowed program transitions (i.e. context-switches) to reduce the state space and to realistically simulate threads. We support several

methods from the pthread standard introduced in Sect. 2.1 as well as methods used in the SV-COMP to instruct verifiers handling concurrency, all of which are elaborated on in this section.

pthread_create. The sequentialization replaces `pthread_create` calls with updates to `int` variables of the form `T{thread_id}_ACTIVE`. A threads respective variable is set to `1` if the corresponding `pthread_create` is encountered and to `0` if the `pc` of the thread is set to `-1`, i.e. if the threads simulation is terminated.
An `ACTIVE` variable for the main thread is also introduced and initialized to `1`. If the `pc` of the main thread is set to `-1`, the main threads `ACTIVE` variable is set to `0`.

An assumption is created over the `ACTIVE` variable of a thread `i` in the sequentialization loop:

```
1 assume(!Ti_ACTIVE ==> next_thread != i);
```

This enforces that if a thread `i` is not active, it cannot be the next thread executing a case statement.

pthread_mutex_{lock, unlock}. For each `pthread_mutex_t` object in the sequentialization, we create a global `int` variable of the form `{mutex_name}_LOCKED`. In addition, for all mutexes locked by a thread, we introduce a global `int` variable of the form `T{thread_id}_LOCKS_{mutex_name}`. The `LOCKED` and `LOCKS` variables work in tandem to replace calls to `pthread_mutex_lock` in a case statement:

```
1 if (m_LOCKED) { Ti_LOCKS_m = 1; }
2 else { m_LOCKED = 1; Ti_LOCKS_m = 0; pc[i] = x; } continue;
```

If a mutex `m` is locked, then thread `i` is in a waiting state (the `LOCKS` variable set to `1`) and its `pc` is not updated. If `m` is not locked, the waiting state of the thread is reset (the `LOCKS` variable set to `0`) and the mutex `m` marked as `LOCKED`. Thread `i` can then continue executing case statements because its `pc` is updated.
Following this logic, the `pthread_mutex_unlock` method is replaced by a reset of the `LOCKED` variable:

```
1 m_LOCKED = 0;
```

An assumption is created over the `LOCKED` and `LOCKS` variables of a mutex and added to the sequentialization loop:

```
1 assume((m_LOCKED && Ti_LOCKS_m) ==> next_thread != i);
```

This enforces that if a mutex `m` is locked and a thread `i` waits for `m` to be unlocked, then `i` cannot be the next thread executing a case statement.
This assumption is sufficient to include mutex lock TSOs in the sequentialization because the pthreads standard defines that relocking a mutex causes a deadlock. In addition, a thread trying to unlock a mutex it did not lock or that is already unlocked results in undefined behavior [10].

```
1  pthread_t id1;
2  pthread_mutex_t m;
3  int x = 0;
4  void *t1(void *arg) {
5    pthread_mutex_lock(&m);
6    x++;
7    pthread_mutex_unlock(&m);
8    return ((void *)0);
9  }
10 int main() {
11     pthread_mutex_init(&m, 0);
12     pthread_create(&id1, ((void *)0), t1, ((void *)0));
13     pthread_mutex_lock(&m);
14     x *= 2;
15     pthread_mutex_unlock(&m);
16     pthread_join(id1, ((void *)0));
17     return 0;
18 }
```

Figure 4.4: A concurrent C program running on 2 threads featuring the core methods `pthread_create`, `pthread_mutex_(un)lock` and `pthread_join`.

pthread_join. For all threads joined by a thread, we introduce a global `int` variable of the form `T{waiting_thread_id}_JOINS_T{target_thread_id}`, initialized to `0`. The `ACTIVE` and `JOINS` variables work in tandem to replace calls to `pthread_join` in a case statement:

```
1 if (Ti_ACTIVE) { Tj_JOINS_Ti = 1; }
2 else { Tj_JOINS_Ti = 0; pc[j] = x; } continue;
```

If a thread `i` is active, then thread `j` is in a waiting state (the `JOINS` variable set to `1`) and its `pc` is not updated. If `i` is not active, the waiting state of the thread is reset (the `JOINS` variable set to `0`). Thread `j` can then continue executing case statements because its `pc` is updated. Note that a thread cannot join itself i.e. `i == j` cannot hold.

An assumption is created over the `JOINS` variable in the sequentialization loop:

```
1 assume((Ti_ACTIVE && Tj_JOINS_Ti) ==> next_thread != j);
```

This enforces that if a thread `i` is active and a thread `j` wants to join `i`, then `j` cannot be the next thread executing a case statement.

__VERIFIER_atomic_{begin, end}. The function declarations of `__VERIFIER_atomic_begin` and `__VERIFIER_atomic_end` are given as:

```
1 extern int __VERIFIER_atomic_begin(void);
2 extern int __VERIFIER_atomic_end(void);
```

`__VERIFIER_atomic_begin` is used together with `__VERIFIER_atomic_end` in the SV-COMP to mark an atomic block. Calling `__VERIFIER_atomic_begin` results in the calling thread not being interrupted in its execution until

__VERIFIER_atomic_end is called [25].

If the input program contains at least one call to __VERIFIER_atomic_begin, an int variable with the name ATOMIC_IN_USE is added to the sequentialization. In addition, for each thread calling __VERIFIER_atomic_begin in the input program, we create a global int variable of the form T{thread_id}_BEGINS_ATOMIC. The ATOMIC_IN_USE and BEGINS variables work in tandem to replace calls to __VERIFIER_atomic_begin in a case statement:

```
1 if (ATOMIC_IN_USE) { Ti_BEGINS_ATOMIC = 1; }
2 else { ATOMIC_IN_USE = 1; Ti_BEGINS_ATOMIC = 0; pc[i] = x; } continue;
```

If the atomic section is currently in use i.e. any other thread is currently in an atomic section then thread i waits (the BEGINS variable set to 1) and its pc not updated. If the atomic section is not in use, the waiting state of thread is reset (the BEGINS variable set to 0) and the atomic section marked as in use. Thread i then can execute further case statements because the pc is updated.
To ensure that ATOMIC_IN_USE is reset we replace calls to __VERIFIER_atomic_end with:

```
1 ATOMIC_IN_USE = 0;
```

Assumption are created over the ATOMIC_IN_USE and BEGINS variables in the sequentialization loop:

```
1 assume((ATOMIC_IN_USE && Ti_BEGINS_ATOMIC) ==> next_thread != i);
```

This enforces that if the atomic section is currently in use and thread i is in a waiting state, then i cannot be the next thread executing a case statement.

TSO Example. To demonstrate the handling of thread control methods in our implementation, let us consider another example program in Fig. 4.4. The approach also follows the procedures introduced in Sect. 4.1 and Sect. 4.2, i.e. handling function calls, substituting variable names and creating a switch statement for each thread. We now focus on handling thread control methods, where each...

- thread has its own ACTIVE variable.
- mutex has its own LOCKED variable.
- thread locking a mutex has its own LOCKS variable.
- thread joining a thread has its own JOINS variable.
- program has an ATOMIC_IN_USE variable.
- thread beginning an atomic section has its own BEGINS variable.

All of these variables are initialized to 0 except the ACTIVE variable for the main thread which is initialized to 1.

A sequentialized version of the example program Fig. 4.4 can be found in Fig. 4.5. Note that the call to `pthread_mutex_init` needs no explicit handling and can be soundly pruned from the sequentialization. These prunings result in gaps between `pc`, e.g. the main thread `0` has no `case 1`. Calls to `pthread_create` for a thread `i` can be replaced by the assignment `ti_active = 1` as seen in `case 2` in thread `0`. The locking and unlocking of the mutex `m` are demonstrated in cases `2` and `4` in thread `0` respectively. The `pthread_join` method call is replaced with the `if-else` statement in `case 5` in thread `0`.
The `return` statements of each thread are removed entirely because we inline all functions in the `main` function of the sequentialization. The `return` statements of `main` and `t1` (i.e. the start routine) mark the termination of a thread. The respective `ACTIVE` variables are set to `0`.

4.4 Modular Partial Order Reduction

To further reduce the state space, we inject a simple form of POR in the sequentialization via assumptions over allowed transitions. The underlying idea of our POR is the commutativity of statements similar to the POR of UGemCutter (cf. Sect. 3.1). However, the sequentialization may not choose exactly the minimal representative of each equivalence class of interleavings because it does not check if statements commute on the fly (i.e. factoring in the past sequence of statements). Instead, the sequentialization checks whether a case statement accesses local or global variables. This approach is feasible based on just the syntax of the input program.
Per definition, statements that only access thread-local variables commute because they cannot interfere with other threads. We introduce assumptions over `pc` that do not access a global variable. This goal is achieved by associating a case statement and its `pc` with the input CFA edges it represents. CPAchecker features global access checking for a given CFA edge, based on if it reads or writes a variable that has a global declaration. If all edges of a given case statement are local, the previous thread continues executing until a case statement with a global access is encountered.
Additionally, we identify case statements where the `pc` of a thread is not guaranteed to be updated e.g. when simulating calls to `pthread_join` as described in Sect. 4.3. Assumptions are created to correctly simulate the behavior of `pthread_join`. These assumptions could be contradictory to the POR assumptions resulting in preemptive and incorrect thread simulation termination. This is why POR assumptions for these case statements are not included, regardless of whether they are local or global.

```
// input program variables
pthread_t G_id1;
pthread_mutex_t G_m;
int G_x = 0;

// thread simulation variables
int T0_ACTIVE = 1, T1_ACTIVE = 0;
int G_m_LOCKED = 0;
int T0_LOCKS_G_m = 0, T1_LOCKS_G_m = 0;
int T0_JOINS_T1 = 0;

/* function declarations and assume function here ... */
int main() {
  /* declare and initialize NUM_THREADS, pc[] and next_thread ... */
  while (1) {
    /* assign and bound next_thread ... */

    // TSO assumptions over thread simulation variables
    assume(!T0_ACTIVE ==> next_thread != 0);
    assume(!T1_ACTIVE ==> next_thread != 1);
    assume((G_m_LOCKED && T0_LOCKS_G_m) ==> next_thread != 0);
    assume((G_m_LOCKED && T1_LOCKS_G_m) ==> next_thread != 1);
    assume((T1_ACTIVE && T0_JOINS_T1) ==> next_thread != 0);

    // switch statements for each thread simulation
    if (next_thread == 0) {
      switch (pc[0]) {
        case 0: T1_ACTIVE = 1; pc[0] = 2; continue;
        // simulate pthread_mutex_lock
        case 2: if (G_m_LOCKED) { T0_LOCKS_G_m = 1; }
                else { T0_LOCKS_G_m = 0; G_m_LOCKED = 1;
                pc[0] = 3; } continue;
        case 3: G_x *= 2; pc[0] = 4; continue;
        // simulate pthread_mutex_unlock
        case 4: G_m_LOCKED = 0; pc[0] = 5; continue;
        // simulate pthread_join
        case 5: if (T1_ACTIVE) { T0_JOINS_T1 = 1; }
                else { T0_JOINS_T1 = 0; pc[0] = 6; } continue;
        case 6: T0_ACTIVE = 0; pc[0] = -1; continue;
      }
    } else if (next_thread == 1) {
      switch (pc[1]) {
        // simulate pthread_mutex_lock
        case 0: if (G_m_LOCKED) { T1_LOCKS_G_m = 1; }
                else { T1_LOCKS_G_m = 0; G_m_LOCKED = 1;
                pc[1] = 1; } continue;
        case 1: G_x++; pc[1] = 2; continue;
        // simulate pthread_mutex_unlock
        case 2: G_m_LOCKED = 0; pc[1] = 3; continue;
        case 3: T1_ACTIVE = 0; pc[1] = -1; continue;
      }
    }
  }
  return 0;
}
```

Figure 4.5: An exemplary sequentialization of the concurrent program in Fig. 4.4.

```
1  // assign -1 to prev_thread (no assumption holds in first iteration)
2  int prev_thread = -1;
3  int next_thread = -1;
4
5  while (1) {
6     next_thread = __VERIFIER_nondet_int();
7     /* other assumptions as before here ... */
8
9     // POR assumptions over local accesses:
10    assume((prev_thread == 0 && pc[0] == 0) ==> next_thread == 0);
11    assume((prev_thread == 0 && pc[0] == 2) ==> next_thread == 0);
12    assume((prev_thread == 1 && pc[1] == 2) ==> next_thread == 1);
13    assume((prev_thread == 2 && pc[2] == 2) ==> next_thread == 2);
14
15    // reassign prev_thread after all reads in this loop iteration
16    prev_thread = next_thread;
17
18    if (next_thread == 0) {
19       /* switch cases as before here ... */
20    }
21 }
```

Figure 4.6: The sequentialization in Fig. 4.1 with POR assumptions over local accesses injected into it.

For POR, we inject a `prev_thread` variable (initialized to `-1`) into the sequentialization that stores the previous value of `next_thread`, i.e. the thread that executed a case statement in the previous loop iteration. The assignment `prev_thread = next_thread` is handled after all reads of `prev_thread` in the sequentialization loop. Assumptions are created over the `prev_thread` variable in the sequentialization loop:

```
1 assume((prev_thread == i && pc[i] == n) ==> next_thread == i);
```

This enforces that if a thread `i` executed a case statement in the previous loop iteration and is now at location `n`, a `pc` which has no global access and that is guaranteed to update the threads `pc`, then `i` also executes the next case statement. This reduces the amount of locations where a context-switch can occur.

POR Example. To showcase the POR approach via assumptions, let us again consider the concurrent program in Fig. 2.1 and Fig. 2.3 from Chapter 2. It is a suitable example because it features both local and global variables. An exemplary sequentialization loop was given in Fig. 4.1. The sequentialization featured assumptions over thread simulation variables in lines 24 to 29.
Given that `T0_ACTIVE` is initialized to `1` and both `T1_ACTIVE`, `T2_ACTIVE` to `0`, the sequentialization ensures that we always start in thread simulation `0`.
The additional assumptions our algorithm injects into the sequentialization are demonstrated in Fig. 4.6. Note that the POR assumptions are created over the original program statements. In the sequentialization, `T1_ACTIVE = 1` is a global access because the thread simulation variable `T1_ACTIVE` is global, while the origi-

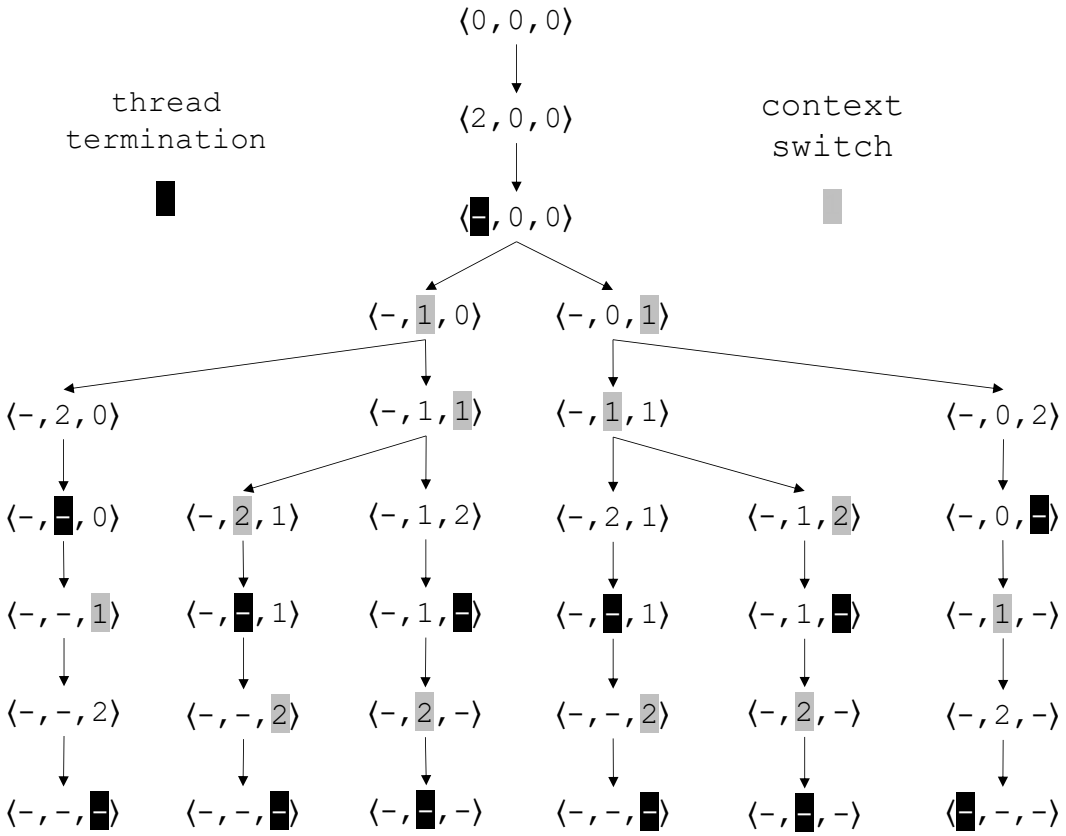


Figure 4.7: Visualization of the possible interleavings in the sequentialization in Fig. 4.6. Each location represents the state of the `pc` array. To improve readability, `-1` is replaced with `-`.

nal statement, `pthread_create(&id1, ((void *)0), t1, ((void *)0))` does not modify the global state and always commutes.

In Sect. 4.1, we showed that the amount of interleavings for the parallel program in Fig. 2.1 and Fig. 2.3 is at least 34650 when neglecting TSOs. For verification purposes where only one interleaving from each equivalence class has to be considered, the amount of execution paths can be reduced to at most 3 as shown in Sect. 2.5. In this simple example, our POR technique reduces the amount of interleavings to consider from at least 34650 (neglecting TSOs) to 6, effectively. The interleaving model for this sequentialization is visualized in Fig. 4.7, showing that thread `0` always terminates before a context-switch occurs.

4.5 Sequentialization Errors

Given that we feed the sequentializations back to the sequential reachability analysis of verifiers, it is fundamental to preserve the correctness characteristic in the code transformation for a correct verdict. To increase the reliability of our sequentializa-

tions, we inject assertion failures for program locations that are only reachable when the output program is erroneous.

The SV-COMPs `ConcurrencySafety-Main` category checks if calls to the function `reach_error` are reachable. This is why we use `reach_error` for assertion failures marking erroneous sequentialization program locations:

```
// function declaration
void reach_error(
    const char *__file, unsigned int __line, const char *__function) {
    __assert_fail("0", __file, __line, __function);
}
// exemplary function call
reach_error("__MPOR_SEQ__{input_file}", 1,
    "__SEQUENTIALIZATION_ERROR__");
```

Where `"0"` is a string representation of the expression expected to be true. It evaluates to `0`, i.e. `false`. The following parameters `"__MPOR_SEQ__{input_file}"`, `1` and `"__SEQUENTIALIZATION_ERROR__"` are the filename, line of code and function name in which the assertion failure occurred, respectively. The tested input programs usually used `"__PRETTY_FUNCTION__"`, i.e. the function name together with its signature, making it distinct from our `"__SEQUENTIALIZATION_ERROR__"`, except the unlikely case that the original function also had this name. This, together with the file name, allows us to distinguish erroneous sequentializations from faulty input programs.

All switch statements in the sequentialization feature a call to `reach_error` as the `default` case. Whenever a `pc` or `RETURN_PC` cannot be matched to a case, the `reach_error` indicates that an invalid value was assigned.

Furthermore, invariants are injected to the start of the sequentialization loop if enabled (cf. Chapter 7). We identify invalid combinations of `LOCKS`, `JOINS` and `ACTIVE` thread simulation values:

(i) If a thread `i` locks a mutex `m`, then `i` must be active:

```
if (Ti_LOCKS_m && !Ti_ACTIVE) {
    reach_error(...); }
```

(ii) If a thread `i` wants to join a thread `j`, then `i` must be active:

```
if (Ti_JOINS_Tj && !Ti_ACTIVE) {
    reach_error(...); }
```

These assumptions hold if pre-emptive thread terminations are not allowed. Our algorithm does not currently support functions that allow the termination of a waiting thread, e.g. `pthread_cancel`, `pthread_kill` or `pthread_cond_timedwait`, making this approach sound.

An example of assertion failures in a sequentialization can be found in Fig. 4.8.

```
1  while (1) {
2
3    // loop invariants identifying invalid value combinations
4    if (T0_LOCKS_G_m && !T0_ACTIVE) { reach_error(...); }
5    if (T1_LOCKS_G_m && !T1_ACTIVE) { reach_error(...); }
6    if (T0_JOINS_T1 && !T0_ACTIVE) { reach_error(...); }
7
8    next_thread = VERIFIER_nondet_int();
9    /* other assumptions as before here ... */
10
11   if (next_thread == 0) {
12     switch (pc[0]) {
13       /* other cases as before here ... */
14       default: reach_error(...); break;
15     }
16
17   } else if (next_thread == 1) {
18     switch (pc[1]) {
19       /* other cases as before here ... */
20       default: reach_error(...); break;
21     }
22   }
23 }
```

Figure 4.8: Excerpts from the sequentialization Fig. 4.5 with shortened assertion failures injected into it.

4.6 Limitations

Our implementation imposes restrictions on the accepted input programs and may terminate without producing an output. The appendix features a non-exhaustive Table 7.2 of error messages. The technical reasons behind the input rejections are:

- Given that edges that are calls to `pthread_create` in the input CFA are treated as a single created thread, `pthread_create` calls inside loops are not supported.
- Extracting `pthread_t` and `pthread_mutex_t` objects from calls to `pthread_create` and `pthread_mutex_lock` is essential when handling their semantics i.e. induced TSOs. This is why arrays of `pthread` objects are not supported.
- Thread control methods are only simulated and never actually return in the sequentialization resulting in return values of thread control being unsupported.
- When handling function calls, the parameter variables are constant for duration of a call. Thus our implementation does not support (in)direct recursion to prevent parameter variables from being overwritten before the function completes execution.
- If a function is guaranteed to abort the program or execute an infinite loop, it contains no exit node in its CFA representation. If a thread has a `start_routine` without an exit node, the input program is rejected.

- Our POR approach assumes that a parameter variable is always local, even if a global variable is assigned to it. Thus, non-commuting statements may be assumed to commute in the sequentialization.

Additionally, the algorithm may produce a sequential output program not equivalent to the input program if any C thread creation library other than pthreads is used (i.e. undefined behavior).

CHAPTER 5

Results and Discussion

SV-Benchmarks[1] is a collection of verification tasks used in the SV-COMP to measure the performance of a verification tool. For the evaluation of our implementation, we use the `Concurrency.set`, a diverse subset of SV-Benchmark C programs that contain concurrency. For all tasks, SV-Benchmarks features a `.c` or `.i` file with C code and a `.yml` file with metadata containing e.g. `properties` with an `expected_verdict` that can be `true` or `false`. From the `Concurrency.set` we extract all tasks that include the property file `unreach-call.prp` i.e. whether calls to the function `reach_error` are reachable. This specification is also used in the `ConcurrencySafety-Main` category in SV-COMP 2024 [19].

For the evaluation, we define 4 questions:

- **Q1.** What is the percentage of input programs our algorithm accepts (cf. 7.2)?
- **Q2.** Do the sequentialized versions improve analysis effectiveness compared to the input program?
- **Q3.** Do the sequentialized versions improve analysis efficiency compared to the input program?
- **Q4.** Do the additional POR assumptions (cf. Sect. 4.4) improve analysis effectiveness and efficiency?

For **Q2**, **Q3** and **Q4**, we use tools capable of analyzing both sequential and concurrent C programs, namely CBMC [26], CPAchecker [24] and UAutomizer [27]. The versions used where the SV-COMP submissions from 2023 for CBMC[2] (which was not updated for SV-COMP since) and from 2025 for CPAchecker[3] and

[1] gitlab.com/sosy-lab/benchmarking/sv-benchmarks
[2] zenodo.org/records/10396159
[3] zenodo.org/records/14205219

UAutomizer[4].

The comparison contains 3 sets of verification tasks:

- input programs `input` as taken from the SV-Benchmarks
- sequentializations `seq` of each input program
- sequentializations with POR assumptions `seq-por` of each input program (cf. Sect. 4.4)

The sequentializations were created with the MPORAlgorithm implemented in the CPAchecker branch `modular-partial-order-reduction`[5]. Additional loop invariants (cf. Sect. 4.5) were not enabled.
The evaluation was performed with the benchmarking framework BenchExec [28] (version 3.28-dev[6]) on a Linux 6.8.0-51-generic machine with an Intel Xeon E3-1230 v5 CPU running at 3800 MHz and 33 GB of RAM. Each verification run was limited to a timeout of 900s, 15 GB of memory usage and 2 CPU cores.

Q1. The task extraction from SV-Benchmarks amounted to a total of 726 input programs. Parsing failed for 59 out of 726 (attributable to CPAchecker), giving us a total of 667 input programs our implementation tried to transform. 319 or about 48 percent of which where accepted by our implementation. 61 programs are correct, i.e. the function `reach_error` is not reachable, whereas 258 programs are false. 304 programs were rejected by our implementation (cf. Table 7.2) and 44 resulted in internal code transformation errors, e.g. assertion failures when substituted variables are expected to be present.

Q2. When it comes to the effectiveness of the analysis, an overview of verdicts produced by CBMC, CPAchecker and UAutomizer can be found in Table 5.1. Of 319 `input` tasks, the native concurrency analysis of CBMC, CPAchecker and UAutomizer were able to correctly analyze 302, 277 and 304 programs, respectively. The sequentialized versions `seq` (`seq-por`) had correct verdicts in 10 (9), 4 (=) and 6 (=) occasions by CBMC, CPAchecker and UAutomizer, respectively. CBMC timed out on `circular_buffer_bad` (cf. Table 5.2) from `seq-por` while it was able to correctly falsify the program in the `seq` tasks. CBMC timed out on the `stack_longest-2` task in the `seq-por` set where it ran out of memory in the `seq` set. The task `queue_longer` reported another error code in the `seq-por` set despite the analysis of CBMC running out of memory, same as in the `seq` set. CPAcheckers verdicts and errors were identical between the `seq` and `seq-por` sets. UAutomizer verdicts were identical between the 2 sequentialized sets, although it tended to run out of memory more often in the `seq-por` set.
CBMC was able to produce correct verdicts for 2 programs in both the `seq` and `seq-por` sets that resulted in errors in the `input` set, namely

[4] zenodo.org/records/14209043
[5] gitlab.com/sosy-lab/software/cpachecker/-/tree/modular-partial-order-reduction, revision 5ed7be831478bcb261e0f721cfa059eeafad65ce
[6] github.com/sosy-lab/benchexec, revision b2cb946ffdb7327c92b530906cb1e4e7d6137c0d

		correct		incorrect		error		
		true	false	true	false	timeout	out of RAM	other
input	CBMC	51	251	0	0	3	0	14
	CPAchecker	50	227	0	0	29	3	10
	UAutomizer	52	252	0	0	2	13	0
seq	CBMC	4	6	0	3	303	3	0
	CPAchecker	2	2	0	0	315	0	0
	UAutomizer	3	3	0	0	9	304	0
seq-por	CBMC	4	5	0	3	305	1	1
	CPAchecker	2	2	0	0	315	0	0
	UAutomizer	3	3	0	0	1	312	0

Table 5.1: Overview of analysis verdicts by CBMC, CPAchecker and UAutomizer for the task sets `input`, `seq` and `seq-por`.

`singleton` and `time_var_mutex` (cf. Table 5.2). CBMC also produced incorrect verdicts for 3 programs in both the `seq` and `seq-por` sets that resulted in errors in the `input` set, namely `bigshot_s`, `bigshot_s2` and `singleton_with-uninit-problems` (cf. Table 5.2). Both `bigshot_s` and `bigshot_s2` were falsified due to internal `POINTER_OBJECT` mismatching in CBMC while `singleton_with-uninit-problems` reached the `reach_error` function. Both CPAchecker and UAutomizer were unable to correctly analyze sequentializations when the input program could not be analyzed. Both also did not produce any incorrect verdict in any run.

Q3. For the comparison of analysis efficiency, we only include programs where the tools were able to produce correct verdicts in all 3 task sets (cf. Table 5.2). CBMC, CPAchecker and UAutomizer were able to correctly solve 7, 4 and 6 tasks, respectively as shown in Fig. 5.1. When considering the 4 tasks that all tools were able to solve (cf. Table 5.2) and using CBMCs analysis times as a baseline normalized to 1, CPAchecker and UAutomizer were 46.3 and 84.6 times slower. Note that the efficiency differences between tools is of only minor concern in this thesis.
We focus on the efficiency differences between the task sets `input`, `seq` and `seq-por` and use relative analysis times to provide a more comparable evaluation. The times in the `input` set are used as a baseline (see Fig. 5.2), with the relative times for the `seq` and `seq-por` sets adjusted accordingly. On average, the `seq` (`seq-por`) verification was 4.54 (4.82), 9.57 (10.92) and 5.57 (14.09) times slower compared to the `input` verification for CBMC, CPAchecker and UAutomizer, respectively. CBMC verified one program in the `seq` task, namely `lazy01`, in 68 percent of the time compared to the corresponding `input` task. No sequentialization from `seq-por` was verified faster compared to the corresponding `input` task.

		input			seq			seq-por		
		result	time	RAM	result	time	RAM	result	time	RAM
CBMC	13-privatized_66-mine-W-init_true	true	.212	8.12	true	.995	18.3	true	1.02	19.4
	36-apron_16-traces-unprot2_true	true	.217	8.40	true	3.67	42.5	true	3.83	49.4
	36-apron_41-threadenter-no-locals_unknown_1_neg	false	.178	8.37	false	.275	8.52	false	.305	10.5
	36-apron_41-threadenter-no-locals_unknown_1_pos	false	.183	8.20	false	.276	8.65	false	.293	10.5
	bigshot_s	error	.169	8.90	false	1.60	58.2	false	1.90	61.7
	bigshot_s2	error	.168	9.18	false	1.37	44.2	false	1.54	57.5
	circular_buffer_bad	false	16.0	251	false	197	728	error	875	746
	lazy01	false	3.58	100	false	2.44	47.2	false	7.15	83.7
	singleton	error	.169	8.15	false	112	627	false	97.4	618
	singleton_with-uninit-problems	error	.162	8.31	false	150	678	false	151	689
	stateful01-1	false	6.57	135	false	31.7	271	false	30.1	286
	stateful01-2	true	34.4	152	true	57.2	305	true	47.6	323
	time_var_mutex	error	.163	8.56	true	91.4	286	true	96.8	290
CPAchecker	13-privatized_66-mine-W-init_true	true	8.81	187	true	50.6	506	true	62.2	586
	36-apron_16-traces-unprot2_true	true	9.25	205	true	252	809	true	279	757
	36-apron_41-threadenter-no-locals_unknown_1_neg	false	9.37	204	false	26.8	321	false	29.6	426
	36-apron_41-threadenter-no-locals_unknown_1_pos	false	9.17	214	false	22.4	325	false	30.1	431
UAutomizer	13-privatized_66-mine-W-init_true	true	18.9	302	true	66.2	734	true	132	1460
	36-apron_16-traces-unprot2_true	true	17.0	296	true	133	1550	true	533	2830
	36-apron_41-threadenter-no-locals_unknown_1_neg	false	15.1	284	false	29.6	356	false	62.9	819
	36-apron_41-threadenter-no-locals_unknown_1_pos	false	15.8	304	false	28.0	370	false	57.3	635
	bigshot_s2	true	30.1	405	true	252	1890	true	480	3430
	stateful01-1	false	17.4	313	false	174	4800	false	391	13600

Table 5.2: Overview of CBMC, CPAchecker and UAutomizer verification effectiveness and efficiency for programs in the task sets `input`, `seq` and `seq-por` where at least one sequentialized version did not produce an error. Verdicts are either correct, incorrect or erroneous. Times are given in seconds and RAM usage in MB.

Q4. When it comes to efficiency factoring in the additional POR assumptions, the analysis was, on average, slower by 6.2, 14.1 and 253.0 percent for CBMC, CPAchecker and UAutomizer, respectively. CBMC verified 2 sequentializations

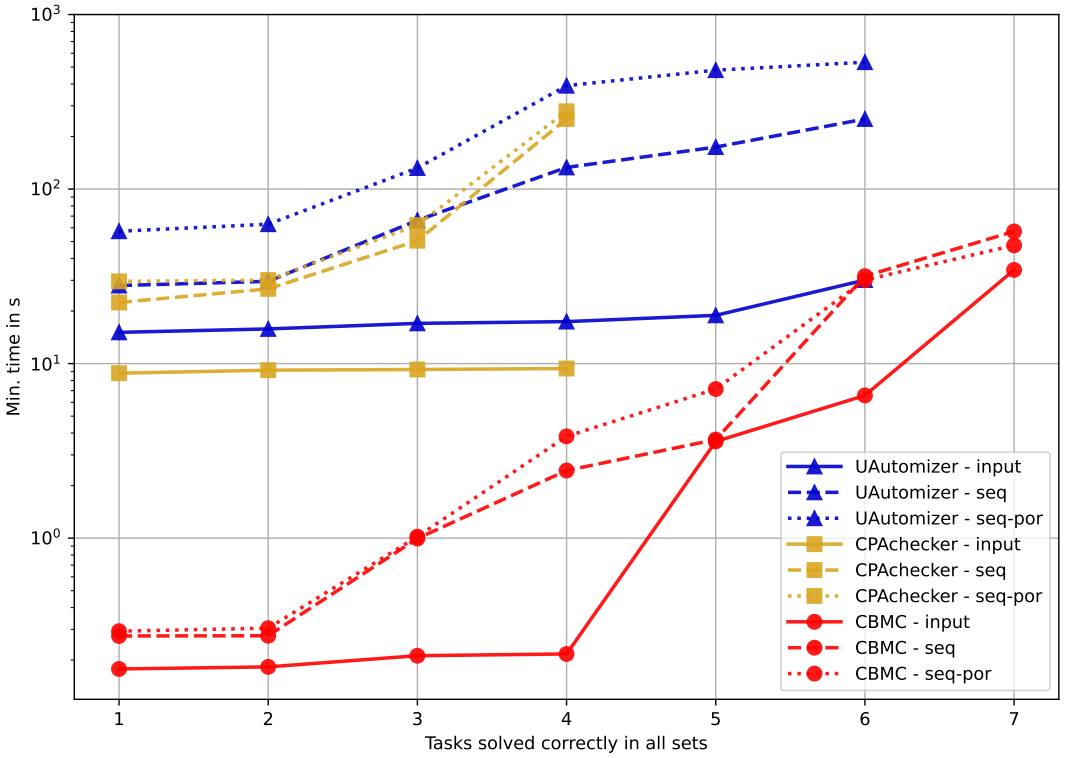


Figure 5.1: Quantile Plot showing the progression of execution times along with the number of correctly solved tasks for CBMC, CPAchecker and UAutomizer. Only tasks with correct verdicts in all task sets (`input`, `seq` and `seq-por`) are included (cf. Table 5.2).

faster with POR assumptions, namely `stateful01-1` and `stateful01-2`, by 5.2 and 20.3 percent, respectively.

Discussion. While the acceptance rate of roughly 48 percent (**Q1**) is a good starting point for our implementation, there is room for improvement. The 44 tasks that produced internal errors resulted e.g. from parameters passed to start routines of threads via the `pthread_create` function which can be supported without much effort. The 300 rejected input programs however will be more difficult to support. We did not obtain exact numbers matching programs to an error message (cf. Table 7.2) but many concurrency SV-Benchmarks use thread creations in loops together with arrays of pthread objects. These rejections can later be supported through loop unrolling. Additionally, further support for more pthread functions can be implemented and function call stacks simulated to support recursion.

For verification effectiveness (**Q2**) we observed that our sequentializations in both the `seq` and `seq-por` task sets prevented a correct verdict for a vast majority of programs (98.4 percent on average) for CBMC, CPAchecker and UAutomizer (cf. Table 5.1). CBMC was able to verify 2 sequentializations that resulted in errors for their corresponding input programs. However, CBMC also produced incorrect verdicts for 3 sequentializations (with `bigshot_s` among them, cf. Table 5.2) that resulted in errors for their corresponding input programs. Given that UAutomizer was able to verify `bigshot_s`, the incorrect falsification of CBMC may be attributed to a tool error and not a faulty sequentialization i.e. when an injected error is reachable (cf. Sect. 4.5). For `singleton_with-uninit-problems`, CBMC did not report the line of code in which `reach_error` was called, leaving the possibility for a sequentialization error open (cf. Sect. 4.5). It cannot be safely derived that the sequentializations enabled an analysis, i.e. improved effectiveness, on any occasion. Further examination of CBMCs 3 incorrect falsifications is required.

The analysis efficiency (**Q3**) saw major differences between CBMC, CPAchecker and UAutomizer for the `input`, `seq` and `seq-por` task sets, emphasizing the distinct analysis approaches of each tool. Given that there was only one occasion where the verification of sequentialized version performed better compared to the corresponding `input` program, we derive that the current form of sequentialized versions generally decrease analysis efficiency.

For POR efficiency (**Q4**), recall that the `seq-por` tasks feature additional assumptions at the loop head compared to their `seq` counterparts. Given that all assumptions are evaluated in each loop iteration and that the `seq-por` set, on average, performed worse compared to the `seq` set, we derive that the overhead introduced by additional POR assumptions weighs heavier than the reduction in the state space. Reducing the amount of assumptions, e.g. grouping POR assumptions with similarities in their left-hand sides together in an `if`-block may improve efficiency. Another option would be to skip the assumptions in the loop head entirely by staying within a thread simulations `case` block when statements commute, e.g. via `goto` statements. These ideas are theoretical and their implementation and evaluation are subject to future work. It remains to be seen whether a reduction of assumptions improves efficiency which may indirectly impact effectiveness.

Threats to Validity. Correct verdicts in all 3 task sets were produced in 7, 4 and 6 out of 319 occasions for CBMC, CPAchecker and UAutomizer, respectively (cf. Table 5.2). This relatively small dataset makes the efficiency evaluation subject to imprecision.
Furthermore, earlier results showed that contradictory assumptions in sequentializations resulted in preemptive thread terminations and thus incorrect `true` verdicts. To prevent contradictory assumptions over the `next_thread` variable, we exclude cases that are not guaranteed to update the `pc` of a thread from POR assumptions (cf. Sect. 4.4), e.g. simulations of the function `pthread_mutex_lock` (cf. Fig. 4.5). Given that no tool produced an incorrect `true` verdict on any `seq` or `seq-por` task,

we infer that contradictory assumptions were not present in our algorithms version used for the evaluation.
The loop invariants introduced in Sect. 4.5 were not included in the evaluation. In earlier experiments however, none of the tested programs reported a `reach_error` from loop invariants during analysis.

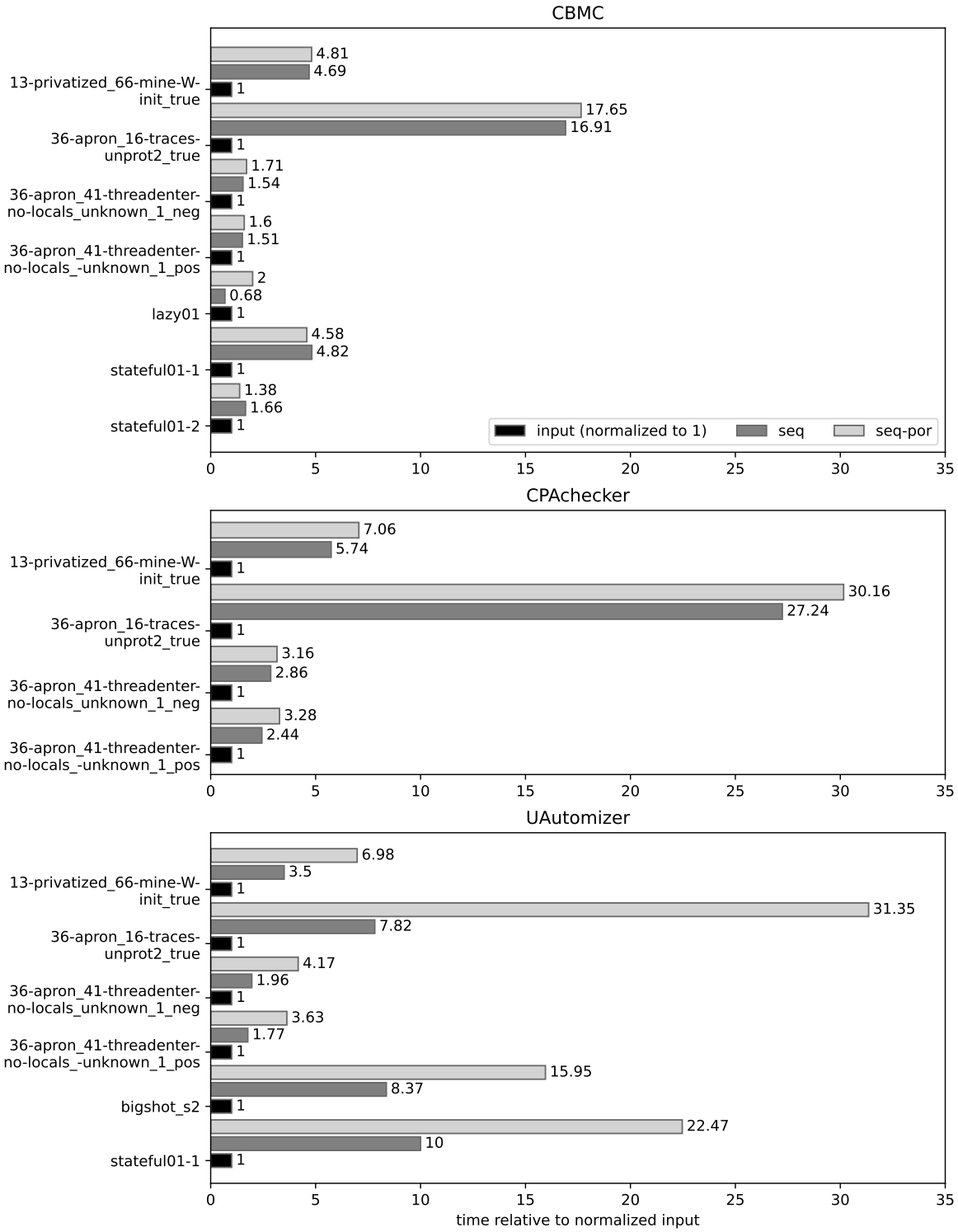


Figure 5.2: Normalized CBMC, CPAchecker and UAutomizer analysis times for the programs in Table 5.2 where all verdicts from `input`, `seq` and `seq-por` were correct.

CHAPTER 6

Conclusion and Future Work

Verifying concurrent programs is difficult due to exponential increases in the state space with a growing number of threads and statements. Refined POR methods like that of UGemCutter (cf. Sect. 3.1) can reduce the complexity from exponential to polynomial by exploiting the commutativity of statements.
Sequentializations like that of Lazy-CSeq (cf. Fig. 3.3) allow verifying concurrent programs with sequential reachability algorithms. Sequentialized programs remain human readable while preserving the complexity of concurrent programs and also allowing a reduction in the state space.
In this thesis, we presented an approach to combine the two methods by creating assumptions over allowed context-switches between thread simulations inside sequentializations. In Chapter 4 we showed that the theoretical basis is solid when it comes to reducing the amount of interleavings.
The experimental evaluation showed that the algorithm accepted a fair amount (roughly half) of the 667 tested programs. Analysis runs with the verification tools CBMC, CPAchecker and UAutomizer showed that our approach introduces an overhead that prevented verification in a vast majority of tested programs. In cases where the analysis produced a correct verdict, the sequentialized programs performed less efficient compared to their input counterparts. POR assumptions worsened efficiency even more (cf. Chapter 5), indicating that despite a few outliers, assumptions create an overhead for verifiers.
This led us to identify great potential for optimization, especially when reducing the amount of assumptions a verifier has to evaluate at each control location.
In future work, we want to add more support to increase the input program acceptance rate of our implementation. Additionally, we plan to reduce the amount of assumptions to evaluate e.g. by grouping assumptions together into control flow statements. Further optimizations can be done by not only checking for global accesses in the POR assumptions but for accesses to the same variable or memory locations.

CHAPTER 7

Appendix

Usage. Our algorithm is currently implemented in the CPAchecker repository[1] branch `modular-partial-order-reduction` and can be run from the root folder of CPAchecker:

```
bin/cpachecker --mpor path/to/input_program.i
```

CPAchecker requires preprocessed input C files without `#include` statements. All declarations from header files must be present in the input program. Preprocess input C files if necessary with the additional `--preprocess` command:

```
bin/cpachecker --mpor path/to/input_program.c --preprocess
```

Configuration. The output sequentialization can be configured with additional `--option` commands. All options are disabled by default i.e. the options only have to be included if their value is `true`:

- Enable POR assumptions over global variable accesses:

  ```
  --option analysis.algorithm.MPOR.includePOR=true
  ```

- Enable loop invariants over thread simulation variables to identify faulty value combinations (slows down verification performance):

  ```
  --option analysis.algorithm.MPOR.includeLoopInvariants=true
  ```

[1] gitlab.com/sosy-lab/software/cpachecker

Function Name	Description	Explicit Handle
`pthread_create`	Creates a new thread.	✓
`pthread_join`	The calling thread waits for the given thread to terminate.	✓
`pthread_mutex_init`	Initializes the given mutex.	
`pthread_mutex_lock`	The calling thread locks the given mutex.	✓
`pthread_mutex_unlock`	The calling thread unlocks the given mutex.	✓
`__VERIFIER_atomic_begin`	The calling threads execution is not interrupted.	✓
`__VERIFIER_atomic_end`	The calling threads execution can be interrupted again.	✓

Table 7.1: Overview of supported pthread functions.

Error Message	Example Trigger
MPOR only supports language C	-
MPOR expects concurrent C program with at least one `pthread_create` call	-
MPOR does not support `pthread_create` calls in loops (or recursive functions)	`while(1) { pthread_create (...); }`
MPOR does not support arrays of pthread objects in line ...	`pthread_t ids[1];`
MPOR does not support the function in line ...	`pthread_cond_wait(...);`
MPOR does not support the pthread method return value assignment in line ...	`x = pthread_create(...);`
MPOR does not support the (in)direct recursive function in line ...	-
MPOR expects the main function and all start routines to contain a `FunctionExitNode`	-

Table 7.2: Overview of input program characteristics rejected by our implementation. Error Messages ending with "..." feature the line in which the characteristic was found and the code of the CFA edge.

Abbreviation	Full Name
`G_varName`	`GLOBAL_varId_varName`
`Li_varName`	`LOCAL_THREADi_varId_varName`
`Pi_paramName`	`PARAM_THREADi_varId_paramName`
`assume(const int cond)`	`__MPOR_SEQ__assume(const int cond)`
`Ti_RETURN_PC_funcName`	`__MPOR_SEQ__THREADi_RETURN_PC_funcName`
`Ti_ACTIVE`	`__MPOR_SEQ__THREADi_ACTIVE`
`mutexName_LOCKED`	`__MPOR_SEQ__mutexName_LOCKED`
`Ti_LOCKS_mutexName`	`__MPOR_SEQ__THREADi_LOCKS_mutexName`
`Ti_JOINS_Tj`	`__MPOR_SEQ__THREADi_JOINS_THREADj`
`ATOMIC_IN_USE`	`__MPOR_SEQ__ATOMIC_IN_USE`
`Ti_BEGINS_ATOMIC`	`__MPOR_SEQ__THREADi_BEGINS_ATOMIC`

Table 7.3: Overview of variable and function name abbreviations in this thesis and their actual names in the output sequentialization. `varId` starts at 0 and is incremented for every substituted variable during the sequentialization process to prevent naming collisions. The prefix `__MPOR_SEQ__` marks ghost elements i.e. they were not present in the input program as variables or functions.

Bibliography

[1] A. Farzan, D. Klumpp, and A. Podelski, "Sound sequentialization for concurrent program verification," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, (New York, NY, USA), p. 506–521, Association for Computing Machinery, 2022.

[2] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[3] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 807–812, 2015.

[4] P. Pacheco and M. Malensek, *An Introduction to Parallel Programming*. Elsevier Science, 2021.

[5] J. Shalf, "The future of computing beyond moore's law," *Phil. Trans. R. Soc.*, 2020.

[6] S. Kumar, *Introduction to Parallel Programming*. Cambridge University Press, 2023.

[7] B. Lewis and D. Berg, *Threads Primer: A Guide to Multithreaded Programming*. SunSoft Press, 1996.

[8] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa, *Rigorous Software Development: An Introduction to Program Verification*. Undergraduate Topics in Computer Science, Springer London, 1 ed., 2011. Published: 04 January 2011, Softcover ISBN: 978-0-85729-017-5, 52 b/w illustrations.

[9] S. La Torre, P. Madhusudan, and G. Parlato, "Reducing context-bounded concurrent reachability to sequential reachability," in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), (Berlin, Heidelberg), pp. 477–492, Springer Berlin Heidelberg, 2009.

[10] Sun Microsystems, Inc., *Multithreaded Programming Guide*, 2005.

[11] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, *Model Checking and the State Explosion Problem*, pp. 1–30. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[12] K. Athanasiou, P. Liu, and T. Wahl, "Unbounded-thread program verification using thread-state equations," in *Automated Reasoning* (N. Olivetti and A. Tiwari, eds.), (Cham), pp. 516–531, Springer International Publishing, 2016.

[13] R. P. Stanley, *Enumerative Combinatorics*. Cambridge University Press, 2011.

[14] S. Qadeer and D. Wu, "Kiss: keep it simple and sequential," *SIGPLAN Not.*, vol. 39, p. 14–24, jun 2004.

[15] B. Fischer, O. Inverso, and G. Parlato, "Cseq: A sequentialization tool for c," in *Tools and Algorithms for the Construction and Analysis of Systems* (N. Piterman and S. A. Smolka, eds.), (Berlin, Heidelberg), pp. 616–618, Springer Berlin Heidelberg, 2013.

[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 *Computer Aided Verification* (A. Biere and R. Bloem, eds.), (Cham), pp. 585–602, Springer International Publishing, 2014.

[17] B. Fischer, O. Inverso, and G. Parlato, "Cseq: A concurrency pre-processor for sequential c verification tools," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 710–713, 2013.

[18] D. Beyer, "Competition on software verification - (sv-comp)," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2012.

[19] D. Beyer, "State of the art in software verification and witness validation: Sv-comp 2024," in *Tools and Algorithms for the Construction and Analysis of Systems* (B. Finkbeiner and L. Kovács, eds.), (Cham), pp. 299–329, Springer Nature Switzerland, 2024.

[20] P. Godefroid, "Partial-order methods for the verification of concurrent systems," in *Lecture Notes in Computer Science*, 1996.

[21] D. Beyer, "Software verification and verifiable witnesses," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Baier and C. Tinelli, eds.), (Berlin, Heidelberg), pp. 401–416, Springer Berlin Heidelberg, 2015.

[22] D. Beyer, "Advances in automatic software verification: Sv-comp 2020," in *Tools and Algorithms for the Construction and Analysis of Systems* (A. Biere and D. Parker, eds.), (Cham), pp. 347–367, Springer International Publishing, 2020.

[23] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," in *Proc. TACAS (2)*, LNCS 12652, pp. 401–422, Springer, 2021.

[24] D. Baier, D. Beyer, P.-C. Chien, M.-C. Jakobs, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, H. Wachowitz, and P. Wendler, "Software verification with cpachecker 3.0: Tutorial and user guide (extended version)," 2024.

[25] J. Erhard, M. Bentele, M. Heizmann, D. Klumpp, S. Saan, F. Schüssele, M. Schwarz, H. Seidl, S. Tilscher, and V. Vojdani, "Correctness witnesses for concurrent programs: Bridging the semantic divide with ghosts (extended version)," 2024.

[26] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (K. Jensen and A. Podelski, eds.), vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176, Springer, 2004.

[27] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski, "Ultimate automizer with smtinterpol," in *Tools and Algorithms for the Construction and Analysis of Systems* (N. Piterman and S. A. Smolka, eds.), (Berlin, Heidelberg), pp. 641–643, Springer Berlin Heidelberg, 2013.

[28] D. Beyer, S. Löwe, and P. Wendler, "Benchmarking and resource measurement," in *Model Checking Software* (B. Fischer and J. Geldenhuys, eds.), (Cham), pp. 160–178, Springer International Publishing, 2015.