UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

A Pragmatic Verification Approach for Concurrent Programs

by

Truc Lam Nguyen

Thesis for the degree of Doctor of Philosophy

May 2017

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

A PRAGMATIC VERIFICATION APPROACH FOR CONCURRENT PROGRAMS

by Truc Lam Nguyen

Developing correct concurrent software is a difficult task, due to the inherently non-deterministic nature of thread interactions. Traditional testing techniques typically perform an explicit exploration of the possible program executions, and are thus inadequate for concurrent software. Symbolic verification techniques for concurrent programs are therefore desirable.

Sequentialization has become one of the most promising symbolic approach for the verification of concurrent programs in recent years. However, current efficient implementations still struggle with concurrent programs that contain rare bugs, and their purposes is restricted to bug-finding. In this thesis, we advance sequentialization to provide pragmatic and scalable verification approaches for concurrent programs, aiming at finding bugs and proving correctness.

Concerning finding rare bugs in concurrent programs, we present our work on optimising Lazy-CSeq sequentialization using abstract interpretation. We empirically demonstrate that this procedure, which is implemented in the tool called Lazy-CSeq+ABS, can lead to significant performance gain for very hard verification problem.

Furthermore, we propose a "swarm" verification approach that can enable existing tools to find rare concurrency bugs which were previously out of reach. We implement the approach in VeriSmart, as a extension of Lazy-CSeq, and empirically demonstrate that VeriSmart can spot rare bugs considerably faster than Lazy-CSeq tool can.

With regard to proving correctness, we develop a novel lazy sequentialization for unbounded concurrent programs and implement the corresponding schema in a tool named UL-CSeq based on the CSeq framework. Empirical experiments show that our new schema is efficient in both proving correctness and finding bugs on concurrency benchmarks in comparison with state-of-the-art approaches.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Truc Lam Nguyen , declare that the thesis entitled *A Pragmatic Verification Approach for Concurrent Programs* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [NFLP15], [NFLP16a], [TNI+16b], [INF+15], [TNI+16a], [NIF+17], [NFLP17], [TNF+17]

Signed:..

Date:..

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Dr Gennaro Parlato for the continuous support, guidance, and encouragement throughout the course of my studies. His guidance helped me throughout the research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank Omar Inverso and Ermenegildo Tomasco for their friendship and collaboration. I thank all my fellow doctoral candidates and friends in ECS, with whom I had the pleasure of spending some time during the past few years. Thanks to Dr Corina Cirstea and Dr Constantin Enea for having agreed to examine my work. Thanks to ECS for funding my doctoral studies.

Last but not least, I would like to thank my family and friends for supporting me spiritually throughout the course of my studies and my life in general.

Chapter 1

Introduction

Concurrent software has become an essential component in modern computing technology. Nowadays, the applications of concurrent software have been deployed everywhere; for example, in embedded automotive hardware in modern cars, or in personal computers with multi-core hardware architecture, or in web servers that are subjected to millions of requests from multiple users in various locations simultaneously. These software applications can exploit the power of multiple processing units efficiently. As a result, concurrent software development becomes more and more important for now and for the foreseeable future.

Developing correct and efficient concurrent software is a difficult task. The source of complexity is not primarily the non-determinism of thread interactions; there are also additional sources in different levels: (1) the level of multi-threaded applications using high-level synchronisation primitives, (2) the level of the software layer that implements the concurrency libraries and synchronisation mechanisms, and (3) the level of the memory models adopted by compilers and modern multi-core architectures in order to optimise performance. Consequently, implementing concurrent software requires developers to take into account all the complexities and ensure the correctness of each individual thread at the same time. As a result, concurrent programs often contain bugs that are difficult to find, reproduce, and fix [BBdH$^+$09].

Several approaches have been proposed to mitigate the above problem, one of them is the development of concurrent data structure libraries. In particular, these libraries encapsulate all complex reasoning about low-level concurrency, and provide only a set of simple operations (API) on the data structures so that these operations are expected to perform correctly under concurrent settings. Moreover, the data structures are often designed with non-blocking (e.g., lock-free) features to be more efficient and allow high-throughput concurrent accesses. Therefore, such libraries can simplify the reasoning about concurrency for developers and thus facilitate the implementation of concurrent software. However, this also moves the difficult task of developing reliable concurrent software from developers to library designers. Unfortunately, designing correct concurrent data structures is also very hard, due to the high level of concurrent interactions in these structures. In fact, concurrent programs using these libraries may contain concurrency bugs which are rarer and more subtle than those in other concurrent programs so that even expert designers overlook them. For instance, the two most resistant known bugs occur in lock-free data structure benchmarks: `safestack` [Vyu10] and `eliminationstack` [HSY04]. Therefore, automatic techniques and tools for the analysis of concurrent software, in general, are essential.

In this thesis, we investigate automatic verification approaches for concurrent programs, aiming at both finding bugs and proving the absence of errors.

Concerning finding bugs in software, a traditional approach is *testing*. It is very useful and remains the most used paradigm in the software industry as it is scalable on large programs. Testing takes place by in executing a program under its intended environment with the purpose of finding bugs (i.e., errors or other defects). Therefore, it is often effective on concurrent software containing *shallow* bugs (i.e., bugs that occur frequently in different invocations of programs), as the interleavings containing bugs can be explored explicitly. However, testing is not efficient on concurrent programs that contain rare bugs as these bugs have very low probabilities of occurrence [BBdH$^+$09, McK17]. Moreover, it is well known that even recent testing developments for concurrent software do not work well in practice, as all possible executions in the programs have to be explored explicitly. Hence, techniques that analyse all thread executions collectively using symbolic methods are highly desirable, and, thus, can complement testing.

One of the most promising symbolic approaches for finding bugs in software is bounded model checking (BMC) [BCCZ99]. BMC involves unrolling loops and inlining functions in programs to obtain bounded programs, and encoding the resulted programs into formulas which are ultimately fed into a SAT/SMT solver. The high performance of modern SAT/SMT solvers has made BMC effective in finding bugs in sequential programs with many mature high-performance tools [CKL04, CF11, LQ13, LQL12, MFS12]. Moreover, there have been several efficient approaches to make BMC work on concurrent programs, and even allow for finding rare concurrency bugs which testing fails to spot, evidenced, for instance, in the two aforementioned lock-free data structure benchmarks [Vyu10, HSY04]. However, these approaches still have scaling-up difficulties when it comes to large concurrent programs with rare bugs. Below, we briefly outline the two prominent approaches and how we address scaling-up problems.

One approach is to model executions of concurrent programs using partial orders [SW11]. The idea is to build a formula for each thread following the standard approach in sequential programs but leaving global variables unconstrained; after that, the individual formulas are put in conjunction with an additional formula that encodes the computations as a partial order. The separation of threads and global shared memory also allows for encoding complex weak memory models (WMMs) [AKT13, AKNP14]. Nevertheless, the large amount of non-determinism in concurrent programs can result in huge formulas, which can cause scaling-up difficulty for this approach.

Another approach is to use *lazy sequentialization* [LR09, LMP09a] targeting BMC backends. Sequentialization is an approach to translate concurrent programs into equivalent non-deterministic sequential programs so that off-the-shelf sequential techniques and tools can be reused without any alteration. Sequentialization has several advantages: (1) a code-to-code translation is typically much easier to implement than a full-fledged analysis tool; (2) it allows designers to focus only on the concurrency aspects of programs, delegating all sequential reasoning to an existing target analysis tool; and (3) sequentialization can be designed to target multiple backends for sequential program analysis.

In fact, Lazy-CSeq [INF$^+$15, ITF$^+$14b, ITF$^+$14a], a well-tuned implementation of *lazy sequentialization* [LMP09a, LMP10] specifically designed for BMC backends, has proven itself as one of the most effective symbolic techniques for finding bugs in concurrent programs. The translation schema in the tool is carefully designed to introduce very small memory overheads and very few sources of non-determinism to produce simple formulas. It also aggressively exploits the structure of bounded programs and works well with BMC backends. As a result, Lazy-CSeq sequentialization is more efficient than the aforementioned partial order encoding approach for BMC. It is also extended to handle weak memory models [TNI$^+$16a]. To the best of our knowledge, Lazy-CSeq is the only tool that is able to detect bugs in the two hardest lock-free data structure benchmarks mentioned above.

Although Lazy-CSeq is very effective in practice, it still struggles on large concurrent programs with *rare* concurrency bugs, i.e., programs with a large number of interleavings where only few of those interleavings lead to a violation of the program specification. For example, even though the tool can find bugs in the two hardest lock-free benchmarks mentioned above, the produced SAT formulas are too complex and require huge amount of computational resources. In fact, Lazy-CSeq spends many hours to analyse them on a normal machine. Nevertheless, Lazy-CSeq sequentialization remains one of the most promising approach for finding bugs in concurrent programs, and it is worth investigating the sources of complexity in the schema for further improvement.

The complexity of concurrent programs, in general, consists of two elements: sequential and concurrent. Typically, the source of *sequential complexity* includes programs' states, e.g., shared global and individual thread-local variables, while *concurrent complexity* involves the non-deterministic thread interleavings. The produced formulas from Lazy-CSeq, therefore, inherit the complexities from both, where each of them can severely affect the overall performance of the tool. Therefore, it is essential to reduce at least one of the elements to achieve more scalable analysis.

With regard to reducing the *sequential complexity*, we first carry out detailed analyses on the SAT formulas generated by Lazy-CSeq's BMC backend on some hard benchmarks. The analyses indicate that a large fraction of the overall effort is not spent on finding the right interleavings that expose the bugs but on finding the right values of the original concurrent programs' shared global and individual thread-local variables. Further investigation also reveals that there are unnecessarily large number of propositional variables resulted from the default bit-widths of the variables in C language, for example, an `int` variable can be represented with 32 bits on `x86_64` platform. Moreover, in an experiment, we manually reduced this to the minimum required to find the bug (three bits in the case of `safestack`), which leads to an order of magnitude speed-up [TNI+16a]. Therefore, safely reducing the domains of the concurrent programs' state variables can lead to more effective and scalable analysis. A suitable approach for this reduction is abstract interpretation [CC77b, CC79], which has been known to be very effective in determining the domains of programs' variables.

Therefore, we apply an automated method based on abstract interpretation in Frama-C [CCM09] over the sequentialized programs constructed by Lazy-CSeq to compute over-approximating intervals for these variables. Then we use the intervals to minimise the representation of the (original) state variables, exploiting the BMC backend's bitvector support to reduce the number of bits required to represent these in the sequentialized program, and, hence ultimately in the formula fed into the SAT solver.

We implement this approach in a tool called Lazy-CSeq+ABS, on top of Lazy-CSeq, which supports C99 programs with POSIX thread library. Detailed empirical experiments show that the effort required for the abstract interpretation phase is relatively low, and that the inferred intervals are tight enough to be useful in practice and lead to large performance gains for very hard verification problems.

Concerning reducing the *concurrent complexity* in concurrent programs, our initial solution is to use partial order reduction (POR) [DHRR04, God97, VHB+03]. POR is a technique for reducing the search space of programs by partitioning the programs' executions into equivalent classes, such that only one representative execution for each equivalent class is considered during the analysis. We speculate that applying POR to the sequentialization schema in Lazy-CSeq may reduce redundant executions in concurrent programs. This can lead to smaller formulas fed into the SAT solver, and ultimately generate better performance. However, preliminary experiments have suggested that adding POR instrumentation to the sequentialized programs does not lead to large performance gain, while the additional code brings considerable overheads, which diminishes the overall analysis. An explanation is that the Lazy-CSeq schema is sufficiently optimised so that it can capture virtually minimal number of thread interleavings containing bugs, thereby making further reduction redundant.

Therefore, we need a more efficient approach to reduce the source of non-determinism in thread interleavings. Fortunately, concurrency can offer this opportunity. Inspired by the idea of *swarm verification* [HJG08, HJG11], we design an approach that is based on the divide-and-conquer paradigm, and can make full use of the available multi-core hardware. The approach is called *task competition*: we run the *same* algorithm on multiple processing units, without information exchange, on *different* tasks derived from the original problem. The first variant that produces a definitive answer (i.e., counterexample or proof) "wins" and aborts the others.

More specifically, we generate several program variants that each captures a subset of the original program's interleavings, in a way that each of the original program's interleavings is captured by at least one of these variants. The interleavings are evenly distributed across all generated variants, which means that for programs with rare concurrency bugs, most of these variants do not contain a bug. However, in variants that do contain such bugs, the bugs are generally more frequent (i.e., manifest in a higher fraction of the interleavings) than in the original program. Consequently, while the overall effort can still be considerable, bugs can be found faster and with fewer resources, because the individual variants are simpler and can be analysed in parallel, each with a shorter time-out and smaller memory consumption. Therefore, this approach can inherit the strengths of both testing and symbolic techniques, and mitigate the weaknesses of the two approaches.

We implement the above "swarm" approach in a tool named VeriSmart based on the CSeq framework [INF$^+$15], where we use the code-to-code translation of Lazy-CSeq to derive the tasks. In general, the approach can also be used with other symbolic analysis techniques, explicit-state space exploration techniques, or even testing. We evaluate VeriSmart on the two hardest known concurrency benchmarks, `safestack` [Vyu10] and `eliminationstack` [HSY04] for three different memory models[1]: SC, TSO, PSO. The experiments are conducted in concurrent settings[2], where we launch multiple tasks in parallel and examine each task with different parameters. Detailed experimental results show that (i) although the number of instances to generate can be extremely high, we only have to consider a few instances (selected randomly out of millions or billions) of the original benchmarks to find bugs with high probability; and (ii) the approach is particularly effective for symbolic methods: it reduces the memory consumption and runtime of each individual verification task and also leads to a considerable reduction in wall-clock time for the global verification. In other words, our approach can enable existing tools to find rare bugs that were previously out of reach.

While finding bugs in concurrent programs is essential, proving the absence of errors also plays an important role, apparently when programs do not contain bugs. We

[1]see Section 2.1.2
[2]Iridis cluster, http://cmg.soton.ac.uk/iridis

have observed that although efficient implementations are restricted to bug-finding purpose [INF$^+$15, ITF$^+$14b, ITF$^+$14a], lazy sequentialization [LMP10, LMP12] has the potential to be extended for correctness proofs because it can maintain the concurrent program's invariants and discover only feasible computations. Therefore, we develop a new lazy sequentialization that can handle programs with unbounded loops and an unbounded number of context switches, and is therefore suitable for program verification (both for correctness and bug-finding). The novelty of the translation is the simulation of the thread resumption mechanism in a way that does not require that each statement is executed at most once and does not rely on unconditional jumps (i.e., `goto`) to reposition the execution, as opposed to Lazy-CSeq sequentialization [INF$^+$15, ITF$^+$14b, ITF$^+$14a]. Instead, a single scalar variable is maintained to determine whether the simulation needs to skip over a statement or execute it.

We implement the corresponding schema in a tool named UL-CSeq for C99 programs with POSIX thread library, based on the CSeq framework [INF$^+$15]. We also evaluate UL-CSeq on a large set of benchmarks from the literature, including the concurrency category of the Software Verification Competition (SV-COMP)[3], using different sequential verification backends on the sequentialized programs. Empirical evaluation demonstrates that our lazy sequentialization is efficient in proving the correctness of the safe benchmarks and improves on existing techniques that are specifically developed for concurrent programs. Furthermore, our solution is also competitive with state-of-the-art approaches for finding bugs in the unsafe benchmarks.

To summarise, in this thesis, we make the following contributions[4]:

- we amplify Lazy-CSeq sequentialization using abstract interpretation to provide a more scalable tool for finding bugs in hard concurrency benchmarks (Chapter 3);
- we propose a novel "swarm" verification approach for finding rare concurrency bugs; this approach can leverage sequential verification engines on parallel environments (Chapter 4);
- we develop, implement and evaluate a novel sequentialization aiming at proving the correctness of concurrent programs; this schema is also suitable for bug-finding purposes (Chapter 5).

[3] `https://sv-comp.sosy-lab.org/`

[4] During my PhD studies, I have also been the main developer of VAC (verifier of access control), a state-of-the-art tool for security analysis of administrative RBAC systems [FMNP14], `http://users.ecs.soton.ac.uk/gp4/VAC.html`. VAC implements the algorithms described in [FMP12, FMP13]. Moreover, VAC can be effectively adapted for reasoning about administrative models with temporal constraints [UAS$^+$12, UAV$^+$14] and group-based user-attribute administrative policies in Azure-like access control systems [FSLN17].

Structure of the Thesis

This thesis is organised as follows. In Chapter 2, we provide the formal definition of concurrent programs we used throughout this thesis; we also give a short overview of sequential verification techniques, sequentialization, and our CSeq sequentialization framework. Our first contribution of optimising Lazy-CSeq sequentialization using abstract interpretation is illustrated in Chapter 3. In Chapter 4, we present our second and main contribution, a novel "swarm" verification approach for finding rare bugs in concurrent programs. Chapter 5 demonstrates our first contribution, a lazy sequentialization schema for safety verification of concurrent programs. Finally, we conclude in Chapter 6 with our considerations and possible directions for future work.

Chapter 2

Backgrounds

In this chapter we outline the context and main concepts developed in this thesis. Section 2.1 gives the overview of the target language that we use. In Section 2.2, we introduce current techniques for the verification of sequential programs. Section 2.3 briefly describes *sequentialization*, including eager and lazy schemas, and the Lazy-CSeq sequentialization schema. Finally, in Section 2.4, we present our CSeq sequentialization framework, which is the basis upon which we develop all the prototypes in this thesis.

The contents of Section 2.1 and Section 2.4 are largely based on our published work [TNI+16a, INF+15].

2.1 Shared-memory Multi-threaded Programs

We use a simple imperative language to describe multi-threaded programs. It includes dynamic thread creation and join, and mutex locking and unlocking operations for thread synchronisation. Thread communication is implemented via shared memory and modelled by global variables. In this section and throughout this thesis, we use the terms *multi-threaded program* and *concurrent program* interchangeably.

2.1.1 Syntax

The syntax of multi-threaded programs is defined by the grammar shown in Figure 2.1. x denotes a local variable, y a shared variable, m a mutex, t a thread variable and p a procedure name. All variables involved in a sequential statement are local. We assume expressions e to be local variables, constants, that can be combined using mathematical operators. Boolean expressions b can be `true` or `false`, or Boolean variables, which can be combined using standard Boolean operations.

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

Figure 2.1: Syntax of multi-threaded programs.

A *multi-threaded* program P consists of a list of *global* variable declarations (i.e., *shared* variables), followed by a list of procedures. Each procedure has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement. A statement *stm* is either a sequential statement or a concurrent one, or a sequence of statements (compound statement) enclosed in braces.

A *sequential statement seq* can be an `assume`- or `assert`-statement, an assignment, a call to a procedure that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional—or branching—statement, a `while`-loop, a labelled sequential statement, or a jump to a label. Local variables are considered uninitialised right after their declaration, which means that they can take any value from their respective domains. Therefore, until not explicitly set by an appropriate assignment statement, they can non-deterministically assume any value allowed by their type. We also use the symbol `*` to denote the expression that non-deterministically evaluates to any possible value; for example, with `x = *` we mean that `x` is assigned any possible value of its type domain.

A *concurrent statement con* can be a concurrent assignment, a call to a thread routine, such as a thread creation, a join, or a mutex operation (i.e., init, lock, unlock, and destroy), or a labelled concurrent statement. A concurrent assignment assigns a shared (resp. local) variable to a local (resp. shared) one. Unlike local variables, global variables are always assumed to be initialised to a default value. A thread creation statement $t\texttt{=}\ \texttt{create}\ p\texttt{(}e_1, \ldots, e_n\texttt{)}$ spawns a new thread from procedure p with expressions $e_1, \ldots, e_n$ as arguments. A thread join statement, $\texttt{join}\ t$, pauses the current thread until the thread identified by t *terminates* its execution. Lock and unlock statements respectively acquire and release a mutex. If the mutex is already acquired, the lock operation is blocking for the thread, i.e., the thread is suspended until the mutex is released and can then be acquired.

We assume that a valid program P satisfies the usual well-formedness and type-correctness conditions. We also assume that P does not contain direct or indirect recursive function calls but contains a procedure `main`, which is the starting procedure of the only thread that exists in the beginning. We call this the *main thread*. We further assume that there are no calls to `main` in P and no other thread can be created that uses `main` as its starting procedure. Finally, our programs are not *parameterised*, in the sense that we only allow for a bounded number of thread creations.

2.1.2 Semantics

A *thread configuration* is a triple $\langle locals, pc, stack\rangle$, where *locals* is a valuation of the local variables, *pc* is the *program counter* that tracks the current statement being executed, and *stack* is a stack of procedure calls that works as follows. At a procedure call, the program counter of the caller and the current valuation of its local variables are pushed onto the stack, and the control moves to the initial location of the callee. At a procedure return, the top element of the stack is popped, and the local variables and the program counter are restored. Any other statement follows the standard *C-like semantics.*

A *thread identifier* is a positive integer. A *multi-threaded program configuration* c consisting of n threads with identifiers $\{i_1, \ldots, i_n\}$, is a tuple of the form $\langle sh, en, th_{i_1}, \ldots, th_{i_n}\rangle$, where: (1) *sh* is a valuation of the shared variables, (2) $en \in \{i_1, \ldots, i_n\}$ is the identifier of the only thread that is enabled to make a transition, or $en \in \{-1, 0\}$, and (3) th_{i_j} is the configuration of the thread with identifier i_j. A configuration c is *initial* if *sh* is the default evaluation of the shared variables, $n = i_1 = 1$ and th_1 is the initial configuration of `main`. An *error* configuration c is such that $en = -1$.

A *transition* of a multi-threaded program P from a configuration c to a configuration c', denoted by $c \xrightarrow[P]{j} c'$, corresponds to the execution of a statement by the thread with identifier $j = en$ (no transitions can be taken from configurations where $en \in \{-1, 0\}$). If the statement being executed is sequential, only t_{en}'s configuration is updated as usual. In particular, the execution of an `assert` statement on a condition that does not hold causes the whole program to terminate immediately into an error configuration. In contrast, an `assume` statement will take to a configuration with $en = 0$ and the computation will abort but without entering an error configuration. Concerning concurrent statements, a thread creation statement adds a new thread configuration to the configuration of the multi-threaded program with a fresh identifier $i > 0$. A thread join operation on a thread identifier t will not allow any further transition for the invoking thread t_{en} until the thread identified by t terminates its execution. A thread `lock` statement on a free mutex m (i.e., a mutex not held by any thread) will lead to a new configuration where the value of m is set to t_{en}. If the mutex is not free, an attempt to lock it will prevent t_{en} from making any further transitions. The execution of a thread `unlock` statement on a mutex m, held by t_{en}, allows it to be freed. When a

t_{en} terminates, its configuration is removed from the pool of active threads. The enabled thread in c' is non-deterministically selected from the pool of active threads of c'. We define $\underset{P}{\rightarrow}$ to be the union of all relations $\overset{j}{\underset{P}{\rightarrow}}$.

Let P be a multi-threaded program with configurations c and c'. A *run* or *execution* of P from c to c', denoted $c \underset{P}{\rightsquigarrow} c'$, is any sequence of zero or more transitions $c_0 \underset{P}{\rightarrow} c_1 \underset{P}{\rightarrow} \cdots \underset{P}{\rightarrow} c_n$ where $c = c_0$ and $c' = c_n$. A configuration c' is *reachable* in P, if $c \underset{P}{\rightsquigarrow} c'$ and c is the initial configuration of P.

A *context* of thread t from c to c', denoted $c \overset{t}{\underset{P}{\rightsquigarrow}} c'$, is any run $c_0 \overset{t}{\underset{P}{\rightarrow}} c_1 \overset{t}{\underset{P}{\rightarrow}} \cdots \overset{t}{\underset{P}{\rightarrow}} c_n$ for some n, where $c = c_0$, $c' = c_n$. A run $c \underset{P}{\rightsquigarrow} c'$ is k-*context bounded* if it can be obtained by concatenating at most k contexts of P, i.e., there exists $c_0, c_1, \ldots, c_{k'}$ with $k' \leq k$, such that $c_{i-1} \underset{P}{\rightsquigarrow} c_i$ is a context (of some thread) for any $i \in \{1, \ldots, k'\}$.

For any fixed sequence ρ of thread indices (called *schedule*), a run of P is a *round* w.r.t. ρ, also known as *round-robin execution*, if there exists a run $c_0 \overset{t_1}{\underset{P}{\rightsquigarrow}} c_1 \overset{t_2}{\underset{P}{\rightsquigarrow}} \cdots \overset{t_n}{\underset{P}{\rightsquigarrow}} c_n$ for some n such that $t_1, t_2, \ldots, t_n$ is a subsequence of ρ. A run is k *round-robin* if it can be obtained by concatenating at most k round-robin executions of P.

Memory Models. A *shared memory* is a sequence of memory locations of fixed size. The content of each location can be read or written using an explicit memory operation. The semantics of read and write operations depend on the adopted memory model.

A typical statement that is used in programs that are meant to run under weak memory models is `fence`. A `fence`-statement essentially enforces that all the read and write operations that precede it in the thread code are executed before all those that follow it. Thus we add the following production to the syntax given in Figure 2.1:

$$\mathit{conc} ::= \texttt{fence}.$$

Sequential consistency (SC). SC is the "standard model", where a write into the shared memory is performed directly on the memory location. This has the effect that the newly written value is instantaneously visible to all the other threads [Lam79].

Write-to-read program order relaxation. The total store ordering model (TSO) [SSO+10, ABP11, ABP14] is a *relaxed-consistency* memory model where write and read operations of the same location but by different threads can be reordered. The behaviour of TSO can be described using the simplified architecture shown in Figure 2.2. Each thread t is equipped with a local *store buffer* that is used to cache the write operations performed by t according to a FIFO policy. Updates to the shared memory occur non-deterministically along the computation, by selecting a thread, removing the oldest write operation from its store buffer, and then updating the shared memory valuation accordingly. Before updating, the effect of a cached write is visible only to the thread that has performed it. A read by t of a variable y retrieves the value from the shared memory unless there

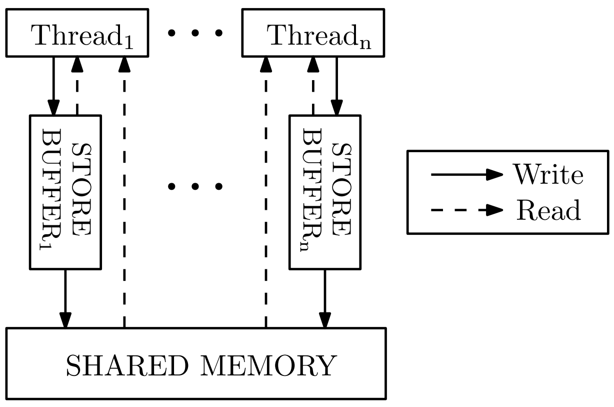


Figure 2.2: TSO architecture.

is a cached write to y pending in its store buffer; in that case, the value of the *most recent* write in t's store buffer is returned. A thread can also execute a *fence*-statement to block its execution until its store buffer has been emptied.

Write-to-write program order relaxation. The partial store ordering model (PSO) is similar to TSO except that the write operations to different locations by the same thread can also be reordered. Its semantics is thus the same as for TSO except that each thread is endowed with a store buffer for each shared memory location.

More general program order relaxation. Other relaxed memory models such as ARM and POWER relax program order for all memory operations (reads and writes) to different locations. The reordering is only constrained by fulfilling some dependency relations that aim to preserve the correct semantics of the sequential parts, such as standard data-flow, address, and control relations (see [SSA+11]). In these models, reordering can be constrained by using `fence`-statements to enforce the commitment of pending writes.

2.1.3 Reachability

Let P be a multi-threaded program and k be a positive integer. The *reachability problem* asks whether there is a reachable error configuration of P. Similarly, the *k-context (respectively, k round-robin) reachability problem* asks whether there exists an error configuration of P which is reachable through a k-context (respectively, k round-robin) execution.

The *reachability problem for concurrent programs* asks whether a particular statement in the program marked using a special label `goal` is reachable. The *reachability problem for concurrent programs under a context-switch bound k*, for $k \geq 1$, asks whether `goal` is reachable within k context-switches.

```
mutex m1,m2; int c;

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

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

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

Figure 2.3: A multi-threaded program containing a reachable assertion failure. In the `main` thread, functions `P` and `C` are both used twice to spawn a thread.

Example. The program shown in Figure 2.3 models a producer-consumer system, with two shared variables, two mutexes `m1` and `m2`, and an integer `c` that stores the number of items that have been produced but not yet consumed.

The `main` function initialises the mutex and spawns two threads executing `P` (*producer*) and two threads executing `C` (*consumer*). Each producer acquires `m1`, increments `c` if it is positive or copies over the initial value "one-by-one", and terminates by releasing `m1`. Each consumer first acquires `m2`, then checks whether all the elements have been consumed; if so, it releases `m2` and restarts from the beginning (`goto`-statement); otherwise, it decrements `c`, checks the assertion `c` ≥ 0, releases `m2`, and terminates.

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

2.2 Sequential Verification Techniques

In this section, we give an overview of verification techniques for sequential programs, including bounded model checking, abstract interpretation and predicate abstraction.

2.2.1 Bounded Model Checking

Bounded model checking (BMC) [BCCZ99] is a symbolic technique in program verification where only subsets of feasible program behaviours are explored. It checks, given a program, a property, and a bound k, if the property can be violated within k execution steps.

More specifically, BMC efficiently reduces program analysis to propositional satisfiability, often SAT problem [Coo71], to check the negation of a given property (or a counterexample) up to a given execution depth k (or bound). The input program is first transformed into a bounded program, then it is simplified into an intermediate representation (IR); after that, the IR is compiled into a propositional formula, or verification condition; finally, the verification condition is analysed by a SAT solver. The formula is satisfiable if and only if there exists an execution of the program that violates the property within, at most, k steps. The satisfiability of the formula implies the existence of an error in the initial program; however, the absence of detected errors is indecisive, because an error might still occur beyond the given execution bound.

The transformation from sequential programs into bounded programs requires the removal of loops and function calls. Conceptually, any loop structure, including recursive calls, can be removed by replicating its body k times. A function call can also be removed by *inlining* its code in the calling function, transforming the return statements into assignments to a newly introduced variable that stores the return value (if any) followed by a jump to the end of the inlined function. In practice, BMC-based tools may not simplify the programs as outlined above; however, the bounded programs all share one important feature: all jumps are forward and each statement is executed, at most, for one run. Additionally, the static single assignment (SSA) form [CFR+89, LA04] is often used in the IR for bounded programs as SSA typically enables more efficient translation into propositional formulas.

BMC has several advantages. One is that BMC can exploit the considerable performance gains achieved by modern SAT solvers; another lies in its ability to provide a counterexample, or error trace[1]; third, BMC is also fully automatic. The considered strengths of BMC has made it attractive for industrial applications. However, as stated above, there is one major drawback of BMC; it cannot prove correctness.

Therefore, extending BMC to a complete analysis method is also of fundamental importance [BCCZ99, CES09]. One approach is to pre-compute a *completeness threshold* [Bie09] before the actual analysis, in order to determine a bound k that is sufficient to cover the entire program's state space. However, this approach is not effective as searching for the exact completeness threshold can be as hard as the model-checking

[1] A satisfying assignment of variables in the verification condition can be converted into the exact sequence of steps to follow in the input program to reproduce any detected error.

problem itself [CKOS05]. Alternatively, one can use specific characteristics of the program's transition system, such as the *diameter* or the *recurrence diameter* (the longest shortest path and the longest simple path between any two states, respectively) to establish the threshold; this is implemented in several methods [BKA02, KS03, MS03, CKOS04, BK04, Kro06, KOS$^+$11, BOW12]. However, these methods still depend on the program and the kind of property to prove; therefore, determining reasonably accurate completeness thresholds in the general case remains challenging.

Another approach is k-induction, where the idea is to use invariants to construct a k-step inductive proof [SSS00, ES03, Bra11]. This technique is theoretically an induction. Several variations have been proposed [DKR10, DHKR11] and in general require auxiliary invariants to be provided externally, usually through manual source-code annotations. Nevertheless, techniques to automatically generate invariants [AS06, BHMR07, BM08] often require additional effort and are not guaranteed to provide invariants powerful enough to imply the correctness of the property.

2.2.2 Abstract Interpretation

Abstract interpretation is a theory that defines the abstraction or constructive approximation, or over-approximation, of the formal description of programs, which can be complex or infinite, in order to infer or verify the properties of the programs [CC77a]. Invented in the late seventies, abstract interpretation has seen many advancements and become widespread in many aspects of computer science, such as type inference, abstraction/refinement, termination inference, model checking, and, specifically, static analysis [CC77b, CC79]. Production-quality verification tools based on abstract interpretation are available and used in advanced software, hardware, transportation, communication, and medical industries [CC14]. Among those, Frama-C [CKK$^+$12] and Astree [CCF$^+$05] are the exemplary tools, as they can scale on complex programs with millions of lines of code [CCF$^+$09].

Informally, abstract interpretation is to give several semantics of a program (i.e., concrete semantics) by relations of abstraction. Typically, *collecting semantics* is used to define the concrete semantics of a program—it is also the strongest semantics of a program. For example, the collecting semantics of an imperative program associates the set of execution traces it may produce to the program concerned, where an execution trace is the sequence of possible states of the execution of the program, and a state typically consists of the values of the program counter and the memory locations. Abstract interpretation derives the concrete semantics to more abstract ones which allows for computing the semantics interpretation at some point of the program's actual execution. For instance, one can choose to represent the states of a program manipulating integer variables by forgetting the actual values and only retaining their signs. We can conclude the sign of the result from some operations on variables such as multiplication if we know the signs of the operands. However, as the results of abstraction, precision on other operations such as subtraction or addition may be lost. In general, there is always a trade-off between the precision of the analysis and its computability. The more precise the abstract semantics, the harder it is for the analysis on the program.

Sign abstraction, in the above example, is only a simple form of interpretation; there is a whole range of abstract domains suitable for many levels of precision of abstraction. A simple abstract domain is the *box domain* (or interval domain). It consists of a set of intervals assigned to each variable at a given point of programs. For instance, an abstract state assigns the value $v(x)$ to variable x an interval $[L, H]$ where $L \leq v(x) \leq H$. The *octagon domain* [Min01] constraints any pair of variables with an upper bound and a lower bound. For example, given any two variables x, y, octagon abstraction provides this set of constraints $L \leq \pm x \pm y \leq H$. Octagon abstraction is more precise than interval abstraction due to the relation between any pair of variables; however, its complexity is quadratic compared to linear of box abstraction. At a higher level of precision, convex polyhedral approximation [CH78] uses a finite set of linear inequalities (with no restriction on their coefficients) of all variables which are satisfied by their concrete values in the program. This domain can capture a closed interpretation of the concrete semantics of a program but requires polynomial time to compute. Abstract domains can also be refined or combined to form hybrid domains where the new domains are able to express some of the concrete properties which are hard to strictly specify in the original domains.

In general, any techniques based on abstract interpretation are *sound*, which means no conclusion derived from the abstract semantics is wrong relative to the program's concrete semantics. However, they may discover *false positive*, i.e., a violation of abstract semantics on abstracted model may not be a violation in the concrete semantics of the actual program. For this reason, abstract interpretation is often useful for proving correctness of programs rather than finding bugs.

There is also a hybrid approach of abstract interpretation and explicit-state model-checking, called explicit-value analysis [BL13]. Roughly speaking, in the analysis, a state in a program execution is tracked in two levels: (1) one group of variables is computed precisely on precise domain, e.g., Binary Decision Diagram (BDD) [Bry86]; and (2) while the other group is associated with abstract domains. Therefore, this approach has the ability to balance the level of precision and computability for abstraction.

2.2.3 Predicate Abstraction

Predicate Abstraction [CU98, GS97] is another technique based on abstraction. However, as opposed to abstract interpretation, predicate abstraction abstracts programs' states by only keeping track of certain predicates on the state, instead of using abstract domains. By this mechanism, the original data variables are replaced by Boolean variables to represent predicates in the abstract program. According to the construction, the abstract program is guaranteed an over-approximation of the original program. Therefore, predicate abstraction may produce considerable reductions in the state space, and can prove correctness. Nonetheless, when the analysis of the abstract program issues a counterexample, it may not be a concrete (genuine) counterexample. Here, refinement needs to be performed in order to adjust the set of predicates in a way that eliminates this spurious counterexample.

The abstraction refinement process is automated in a paradigm called Counterexample-guide Abstraction Refinement (CEGAR) [CGJ$^+$03, CGJ$^+$00, DD01]. More specifically, CEGAR starts with an abstract model and checks if an error path can be found on the program by using that model. If no error path is found, the analysis terminates, and reports that the program is correct. Otherwise, the error path is checked for feasibility, which means whether the path is executable according to the concrete semantics of the program. If the path is *feasible*, the analysis also terminates, reporting the violation of the property, together with the feasible error path as witness. In the negative case, which means that the path is found *infeasible*, this path is then used to automatically refine the current abstraction model. This process keeps iterating until no more error path is found (i.e., program is correct) or a feasible path is reported (i.e., program contains bugs).

Predicate abstraction, therefore, is a sound and complete technique (i.e., prove absence of error or give genuine counterexample). There are several verification tools developed based on predicate abstraction [BR02, BHJM07, BK11, CKSY05, PR11, GPR11]. Nevertheless, one drawback of predicate abstraction lies in the abstraction refinement process. The predicate refinement, while discarding spurious counterexample, may not eventually converge; or in other words, this process may run indefinitely long and can be intractable.

2.3 Concurrent Verification via Sequentialization

Sequentialization is basically a context bound technique for the verification of concurrent programs. It translates a multi-threaded program into a non-deterministic sequential program that simulates all possible schedules up to a given context switch bound. The goal is to reuse verification tools originally developed for sequential programs to analyse concurrent programs.

Sequentialization was originally proposed by Qadeer and Wu [QW04]. They transformed a concurrent program into a sequential one that simulates all executions of the original program with at most two context switches. More specifically, the first schema simply schedules the threads in a way whereby all threads share a unique call stack. The call stack, at each step, can be split into contiguous parts, where each part corresponds to the whole stack of an executed thread. This schema, however, only allows a small number of maximum context switches that can be considered (e.g., for two threads only two context switches can be simulated).

Eager Sequentialization. Lal and Reps (LR) generalised the above concept to arbitrary context bounds [LR09]. In their paper, LR schema defines a transformation from concurrent Boolean programs into sequential Boolean programs with a fixed number of threads and a parameterised number of round-robin schedules. In LR sequentialization, the sequential program simulates the threads in the concurrent program in a fixed order, according to round-robin schedules. Moreover, all threads are simulated until completion in each round; and each round holds its own copy of the shared global memory.

The sequential program starts with the simulation of the first thread in the first round. This thread begins its execution by "eagerly" guessing (non-deterministically) the initial values of all memory copies as well as the context-switch points. At each guessed context-switch point, the thread switches over to the memory copy for the next round, and it will be simulated in this manner until its terminate point. The subsequent threads in the round are simulated similarly, except that they work with the values of the shared memory copies left by their respective predecessors in the current round. Once the simulation of the last thread finishes, an auxiliary checker is injected to prune away all initial guesses that do not correspond to feasible computations (i.e., a feasible computation is where all the values guessed for one round match the values computed at the end of the previous round). This checker thus requires a second set of memory copies (note that the local variables of each thread can be safely discarded as each thread runs to completion); and the global memory copies serve as interfaces between the threads. LR schema is often regarded as *eager sequentialization*, because the non-determinism of data which induce the exploration of unreachable states are pruned away only at the end of simulation.

The first LR schema was only designed for programs where threads are only created at the beginning of the execution. This problem was then overcome in delay bounded sequentialization [EQR11], where thread creation can be handled by transforming threads into function calls; and these function calls are simulated right at the point their corresponding threads are spawned. Similarly, LR sequentialization was further extended to allow modelling of unbounded, dynamic thread creation [BEP11, LMP12], or complex dynamically linked data structures allocated on the heap [ABQ11]. Moreover, LR has been also applied in several tools [CGS11, LQR09, Qad11, LQL12, FIP13a, FIP13b], due to its relative ease of implementation.

LR can also be extended alternatively by looking at the number of memory accesses (i.e., read and write operations on global/shared variables) done by threads. The idea is to use an explicit representation of the write operations in a sequence that, for each write, contains the identification of writing thread, the variable or synchronisation primitive, and the written value; in other words, this schema models an unwound (bounded) memory, or memory unwinding (MU) representation of concurrent programs on the corresponding sequential programs. Each thread is also translated in a function where write/read accesses over the global shared variables are replaced by operations over the unwound memory. The simulation starts with "eagerly" guesses of the MU; after that, each thread is simulated similarly to LR where all context switches are implicitly simulated on the MU in rely-guarantee manner [Jon83]. MU sequentialization schema is implemented in MU-CSeq tool [TIF$^+$14], and later augmented with "grained" tuning feature [TIF$^+$15a], and "individual memory location" [TNI$^+$16b]. Recently, MU schema has been extended further to handle weak memory models in modern CPU architectures [TNF$^+$17].

Lazy Sequentialization. Even though *eager sequentialization* can provide a neat way to reason about concurrent programs, an eagerly grasp of the whole state space from the beginning of the simulation can impose a huge burden on the checkers who have to prune away spurious computations in the end. Indeed, the set of reachable states of a concurrent program may be much smaller than the whole state space; therefore, techniques that explore only feasible paths, or *lazy* exploration, are thus more desirable [LR09]. *Lazy sequentialization* was first applied in the verification of concurrent Boolean programs [LMP09a] as an alternative approach to a fixed-point algorithm in La Torre et al. [LMP09b]. The schema was later extended to concurrent Boolean programs with unbounded threads [LMP10, LMP12].

The lazy sequentialization schema in [LMP09a] (LMP) also simulates the original concurrent program in bounded round-robin schedule and keeps the copies of the shared memory, similar to LR. However, instead of guessing the memory like LR, the copies are computed on-the-fly for each thread as the simulation proceeds round-by-round. LMP, in this way, explores only the reachable states of the concurrent programs. Nonetheless, this schema requires the recomputation of thread-local states when a thread resumes its computation from the previous context, as the call-stack and the program counter of a thread are not stored at context-switches. Fortunately, this recomputation is not an actual problem for tools that can compute function summaries, in particular, LMP for concurrent Boolean programs, which reuses the summaries computed in the previous iterations. However, the recomputation may be a serious drawback for the application of LMP in Bounded Model Checking [GHR10].

Lazy Sequentialization schema for BMC. Recently, the above problem of recomputation has been addressed in a fine-tuning LMP schema for BMC [ITF$^+$14a]. The corresponding schema is implemented in Lazy-CSeq tool [ITF$^+$14b, INF$^+$15, ITF$^+$14a], which is an effective and scalable tool for finding bugs in concurrent C programs with POSIX thread library. We briefly outline the lazy sequentialization encoding as follows.

```
mutex m; int c=0;

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

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

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

Figure 2.4: A Producer/Consumer program.

Assume that a concurrent program P consists of $n+1$ functions $f_0, \ldots, f_n$, where f_0 denotes the `main` function, and that P creates at most n threads respectively with start functions $f_1, \ldots, f_n$. Moreover, each function f_i does not contain loops. Note that these assumptions can easily be enforced by bounding the programs in BMC fashion and cloning the start functions, if necessary (*bounded multi-threaded program*). Since each start function is thus associated with at most one thread, we can identify threads and (start) functions.

Consider a bounded multi-threaded program P as described above. In the analysis of bounded round-robin computations, A number of rounds K and an arbitrary schedule ρ is fixed by permuting the functions $f_0, \ldots, f_n$ that form the starting program. Thus, the lazy sequentialization of P yields a sequential program P' such that P fails an assertion in K rounds if and only if P' fails the same assertion. P' is composed of a new function `main` and a thread simulation function T_i for each thread f_i in P. The lazy sequentialization of the Producer/Consumer program given in Figure 2.4 generated by Lazy-CSeq (with two loop unwindings) is the code shown in Figure 2.5. In the figure, the code injected by Lazy-CSeq is shown in gray while the original code is shown in black.

It is observable that the sequential verification of P' relies on stubs provided by Lazy-CSeq. P' thus uses a slightly modified version of the POSIX thread library. For example, the `thread_create` stub takes an additional argument (in gray) for the (statically known) id of the calling thread; see Inverso et al. [ITF+14a] for details.

The new `main` of P' is a driver that calls, in the order given by ρ, the functions T_i for K complete rounds. For each thread it maintains the label at which the context switch

```
bool active[T]={1,0,0,0,0};
int cs,ct,pc[T],size[T]={5,8,8,2,2};
#define G(L) assume(cs>=L);
#define J(A,B) if(pc[ct]>A||A>=cs) goto B;
mutex m; int c=0;

 void P1(void *b) {
  0:J(0,1) static int tmp;
             tmp=(*b);
  1:J(1,2) mutex_lock(&m);
  2:J(2,3) if(c>0)
  3:J(3,4)   c++;
            else { G(4)
  4:J(4,5)  c=0;
            if(!(tmp>0)) goto _l1;
  5:J(5,6)  c++; tmp--;
            if(!(tmp>0)) goto _l1;
  6:J(6,7)  c++; tmp--;
            assume(!(tmp>0));
            _l1:  G(7);
            } G(7)
  7:J(7,8) mutex_unlock(&m);
            goto _P0; _P0:  G(8)
  8:       return;
  }

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

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

 void C2() {...}
```

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

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

Figure 2.5: Lazy-CSeq sequentialized code of the Consumer/Producer program in Figure 2.4.

was simulated in the previous round and where the computation must thus resume in the current round. Moreover, before each call to T_i, the label at which the control will context-switch out is non-deterministically guessed.

Each T_i is essentially f_i with few lines of injected control code and with labels to denote the relevant context-switch points in the original code. When executed, each T_i jumps (in multiple hops) to the saved position in the code and then restarts its execution until the label of the next context switch is reached. This is achieved by the J-macro. Context-switching at branching statements requires some extra care; see Inverso et al. [ITF+14a] for details. The local variables are also made persistent (i.e., `static`) such that there is no need to recompute them when resuming suspended executions.

Some additional data structures and variables are also used to control the context-switching in and out of threads as described above. The data structures are parameterised over $\texttt{T} \leq n$ which denotes the maximal number of threads activated in P executions. Auxiliary variables are also used to keep track of the active threads (`active`), the arguments passed in each thread creation (we omitted it in our example since the considered thread functions have no arguments), the largest label used in each T_i (`size`), the current label of each T_i (`pc`), and, for the currently executed thread, its index (`ct`) and the context-switch point guessed in the main driver before calling the thread (`cs`).

Note that the control code that is injected in the translation is designed such that each T_i reads but does not write any of the additional data structures. This data is updated only in the main driver and in the portions of code simulating the API functions concerning thread creation and termination. This has the advantage of introducing fewer dependencies between the injected code and the original code, which typically leads to a better performance of the backend tool (e.g., for BMC backends this results in smaller formulas).

Lazy Sequentialization for WMMs. Lazy sequentialization can be extended to handle complex weak memory models on modern computer architectures. The first lazy sequentialization for multi-threaded programs for the TSO and PSO memory models is proposed by Tomasco et al. [TNI+16a]. More specifically, this schema, while following precisely the lazy sequentialization schema [ITF+14a], replaces all access to shared memory items (i.e., reads from and writes to shared memory locations, and synchronisation primitives like lock and unlock) by explicit calls to *API operations* over a *shared memory abstraction* (SMA). For example, if `x` and `y` are two shared scalar variables, then the statement $\texttt{x} = \texttt{y} + \texttt{x} + \texttt{3}$ is translated into $\texttt{write}(\texttt{x}, \texttt{read}(\texttt{y}) + \texttt{read}(\texttt{x}) + \texttt{3})$.

The SMA can be seen as an abstract data type (ADT) that encapsulates the semantics of the underlying WMM and allows it to be implemented under the simpler SC model. This abstract data type also isolates the (weak) memory model from the remaining concurrency aspects, and allows the reuse of existing (lazy) sequentialization techniques and tools for SC. The approach shares some similarities with the axiomatic representation of memory models [SW11, AKT13] but the fundamental difference is that it works at the code level—in effect, the very idea of sequentialization to WMMs themselves.

The efficiency of the sequentialization depends on how to implement the SMA to leverage the backend verification tool. In the work by Tomasco et al. [TNI+16a], efficient TSO and PSO implementations of the SMA targeting BMC tools are also given. These implementations are carefully designed to optimise some parameters that lead, in combination with a lazy sequentialization targeting CBMC [CKL04], to efficient SAT encodings, thanks to the introduction of the *Temporal Circular Doubly Linked List* data structure.

2.4 CSeq Sequentialization Framework

The CSeq framework is built on ideas from CSeq tool [FIP13a]; and later improved and fully re-engineered [INF+15]. It now provides support for quickly prototyping new sequentialization-based verification tools. To date, the framework has been used to implement Lazy-CSeq [INF+15, ITF+14b, ITF+14a], MU-CSeq [TIF+14, TIF+15a, TNI+16b] and UL-CSeq [NFLP15, NFLP16a] tools. Moreover, all the tools presented in Chapters 3, 4, and 5 are built or extended on this framework.

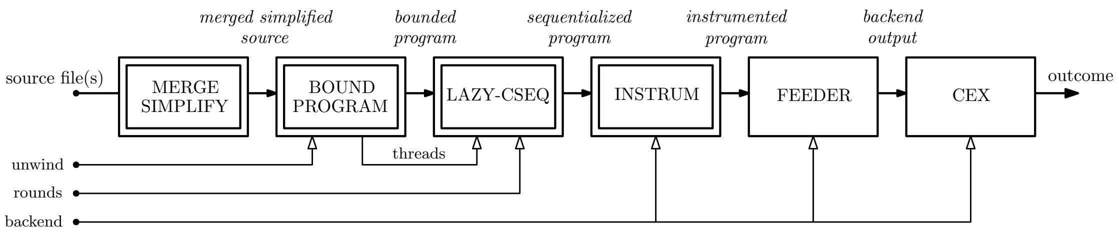


Figure 2.6: Configuration sequence of Lazy-CSeq. Double-framed boxes denote modules composed of multiple submodules.

The framework comprises several modules that are either *translators* that implement source-to-source transformations of C programs, or *wrappers* that work on generic strings and are used for general-purpose tasks that do not produce source code. Each tool within CSeq is identified by a *configuration* that corresponds to a sequence of translators followed by a sequence of wrappers; for example, Figure 2.6 sketches the configuration for Lazy-CSeq.

A verification tool takes as input the file containing the source code of the concurrent C program to analyse and the list of verification parameters. For Lazy-CSeq, the verification parameters are the number of *rounds*, the *unwinding depth* and the acronym of the backend tool. The input parameters are passed to the appropriate modules; additionally the first module takes as input also the input source file and then the output of each module is fetched as input to the following module. The output of the last module in the sequence is the analysis outcome.

The first translator is always a *merger*: the input source code is merged with external sources pulled in by the `#include` directives[2]. The last translator is typically an *instrumenter*, which instruments the output according to the backend tool (as explained below). The purpose of the wrappers is to interact with the backend tool and interpret its answer at the end of the analysis; in particular, there is a `cex` module that is responsible for tracking back the counterexample generated by the backend tool on the input source code, and thus output the counterexample.

[2]Include directive, `https://en.wikipedia.org/wiki/Include_directive`

Translators run in two steps: (1) the input code is parsed in order to build the abstract syntax tree (AST)[3], the symbol table, and other data structures; (2) the AST is recursively traversed and unparsed back into a string that corresponds to the output C code. This mechanism is built on top of `pycparser`[4], a parser for C99 that uses `PLY`[5], an implementation of Lex-Yacc [AJ74, LS90] for Python; and it is implemented by conveniently overriding `pycparser`'s AST-based pretty-printer, so that the output code is transformed while visiting the AST. In particular, the transformation is made on-the-fly by directly changing the output generated by AST subtree visits rather than altering the structure of the AST itself. Other source-to-source translation tool [BPM04] uses rewrite rules instead. String-based source transformations are in contrast more intuitive and require a less steep learning curve, and combined with Python's flexibility, it is relatively easy to implement complex code transformations quickly. String-based rewriting is also used in the ROSE framework [QSPK01].

The CSeq framework also provides a *line-mapping* functionality that is independent from the specific translation performed and is a useful support for the counterexample generation. The idea is to keep track of the location in the source code where each line of the output was translated from. During the generation of the output, translators automatically create maps from output to input, in a similar way to how the C Preprocessor (CPP) uses line control information when merging multiple source files, to keep track of which line comes from which source file. However, rather than inserting explicit `#line` directives in the source code (like C preprocessor does[6]) the information is stored as a table which maps output lines back to input lines (note that each input line may generate several output lines, for instance after unfolding a loop). At the end of the last translation, it is possible to track line numbers back to the output of the first module. For the first module (merger), since there might be multiple input files (due to the `#include` directives), output line numbers are mapped to pairs of the form (*linenumber, filename*).

Instrumenting the code for a specific backend is in itself a quite simple standalone transformation undertaken by the instrumentation module. It consists in replacing the primitives for handling non-determinism (that are backend-independent and potentially injected at any point by any module) with backend-specific statements. This involves three kinds of statements: (1) variable assignment statements to non-deterministic values using `nondet_int`, `nondet_long`, etc., (2) restrictions of non-determinism using `assume`, and (3) explicit condition checks (e.g., for reachability) using `assert`. This requires a simple renaming of the function calls, or inserting ad-hoc functions definition, depending on whether or not the desired verification backend intrinsic models all of the above. The size of a backend integration is therefore usually short (e.g., less than 10 lines); however,

[3]Abstract syntax tree, `https://en.wikipedia.org/wiki/Abstract_syntax_tree`
[4]`pycparser`, `https://github.com/eliben/pycparser`
[5]`PLY`, `http://www.dabeaz.com/ply/`
[6]Preprocessor output, `https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html`

the CBMC default backend exploits its bitvector feature to optimise the representation of the program counters and is thus more complicated (please refer to Chapter 3 for other uses of the bitvector feature).

Chapter 3

Lazy Sequentialization and Interval Analysis

In this chapter, we present how we use abstract interpretation to minimise the representation of the concurrent program's state (shared global and thread-local) variables. More specifically, we run the Frama-C abstract interpretation tool over the programs constructed by Lazy-CSeq to compute over-approximating intervals for all (original) state variables and then exploit CBMC's bitvector support to reduce the number of bits required to represent these in the sequentialized program. We have implemented this approach in the tool named Lazy-CSeq+ABS and demonstrate the effectiveness of this approach. In particular, we show that it leads to large performance gains for very hard verification problems.

The content of this chapter is largely based on our published work [NIF+17, NFLP17].

3.1 Introduction

Sequentialization has proven itself as one of the most effective symbolic techniques for concurrent program verification, evidenced for example by the fact that most concurrency medals in the recent SV-COMP program verification competitions were won by various sequentialization-based tools [TIF+14, ITF+14b, TIF+15a, TNI+16b]. It is based on the idea of translating concurrent programs into non-deterministic sequential programs that (under certain assumptions) behave equivalently, so that the different interleavings do not need to be treated explicitly during verification and, consequently, sequential program verification methods can be reused. Eager sequentialization approaches [LR09, FIP13b, TIF+15b] guess the different values of the shared memory before the verification and then simulate (under this guess) each thread in turn. They can thus explore infeasible computations that need to be pruned away afterwards, which requires a second copy of the shared memory, and so increases the state space. Lazy sequentialization approaches [LMP09a] instead guess the context switch points and recompute the memory contents, and thus explore only feasible computations. They also preserve the sequential ordering of the interleaved thread executions and thus the local invariants of the original program. Lazy approaches, such as Lazy-CSeq [INF$^+$15, ITF$^+$14b, ITF$^+$14a], are thus typically more efficient than eager approaches.

Lazy-CSeq is implemented as a source-to-source transformation in the CSeq framework [FIP13a, INF$^+$15] (see Section 2.4): it reads a multi-threaded C99 program that uses the POSIX thread library [ISO09], applies the translation sketched in Section 3.2 and described in more detail by Inverso et al. [ITF$^+$14a] and outputs the resulting non-deterministic sequential C program. This allows us to use any off-the-shelf sequential verification tool for C as backend, although we have achieved the best results with CBMC [CKL04].

Lazy-CSeq's translation is carefully designed to introduce very small memory overheads and very few sources of non-determinism, so that it produces simple formulas. It also aggressively exploits the structure of bounded programs and works well with backends based on bounded model checking. It is very effective in practice, and scales well to larger and harder problems. Currently, Lazy-CSeq is the only tool able to find bugs in the two hardest known concurrency benchmarks, `safestack` [Vyu10] and `eliminationstack` [HSY04]. However, for such hard benchmarks the computational effort remains high; in particular, the analysis requires several hours on a standard machine.

A detailed analysis of these benchmarks shows that a large fraction of the overall effort is not spent on finding the right interleavings that expose the bugs, but on finding the right values of the original (concurrent) programs' shared global and individual thread-local variables. We found that this is caused by the unnecessarily large number of propositional variables (reflecting the default bit-widths of the variables in C) that CBMC uses. In an experiment, we manually reduced this to the minimum required to find the bug (three bits in the case of `safestack`), which leads to a 20x speed-up [TNI$^+$16a]. This clearly indicates the potential benefits of such a reduction.

Contributions In this chapter, we describe an automated method based on abstract interpretation to reduce the size of the concurrent programs' shared global and thread-local state variables. More specifically, we run the Frama-C abstract interpretation tool [CCM09] over the sequentialized programs constructed by Lazy-CSeq to compute over-approximating intervals for these variables. We use the intervals to minimise the representation of the (original) state variables, exploiting CBMC's bitvector support to reduce the number of bits required to represent these in the sequentialized program, and, hence, ultimately in the formula fed into the SAT solver. Note that this approach relies on two crucial aspects of Lazy-CSeq's design. On the theoretical side, we rely on the fact that lazy sequentializations only explore feasible computations to infer "useful" invariants that actually speed up the verification; our approach would not work with eager sequentializations because they leave the original state variables unconstrained, leading to invariants that are too weak. On the practical side, we rely on the source-to-source approach implemented in Lazy-CSeq, in order to re-use an existing abstract interpretation tool.

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

Organisation of the chapter. In the next section, we give a description of our approach. Section 3.3 gives details on our implementation and Section 3.4 presents the results of our experimental evaluation. Section 3.5 draws comparisons with related work, and Section 3.6 concludes.

3.2 Verification approach

In this section we illustrate the verification approach we propose in this chapter. We refer to multi-threaded programs (see Section 2.1) and recall context-bounded analysis before we give some details on the two pillars of our approach: the lazy sequentialization performed by the tool Lazy-CSeq [ITF+14a] and the value analysis performed by the tool Frama-C [CCM09].

3.2.1 The general schema

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

1. We compute a sequential program that preserves the reachable states of the input program up to a given number of thread context-switches (*sequentialization*).
2. We compute the bounds on the values that the variables can store along any computation of the sequential program (*value analysis*).

```
bool active[T]={1,0,0,0,0};
int cs,ct,pc[T],size[T]={5,8,8,2,2};
#define G(L) assume(cs>=L);
#define J(A,B) if(pc[ct]>A||A>=cs) goto B;
mutex m; bitvector[4] c=0;

 void P1(void *b) {
  0:J(0,1) static bitvector[4] tmp;
                tmp=(*b);
  1:J(1,2) mutex_lock(&m);
  2:J(2,3) if(c>0)
  3:J(3,4)   c++;
           else { G(4)
  4:J(4,5)   c=0;
             if(!(tmp>0)) goto _l1;
  5:J(5,6)   c++; tmp--;
             if(!(tmp>0)) goto _l1;
  6:J(6,7)   c++; tmp--;
             assume(!(tmp>0));
             _l1:  G(7);
             } G(7)
  7:J(7,8) mutex_unlock(&m);
             goto _P0; _P0:  G(8)
  8:         return;
  }

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

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

 void C2() {...}
```

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

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

Figure 3.1: Lazy-CSeq sequentialized code of the Consumer/Producer program modified according to the value analysis by Frama-C.

3. We transform the sequentialized program by changing the program variables of numerical type (i.e., `integer` and `double`) to bitvector types of sizes determined by the results of the value analysis (*model refinement*).

4. We verify the resulting sequential program (*verification*).

In sequentializations the control non-determinism of the original program is replaced by data non-determinism and thread invocations are replaced by function calls. Lazy sequentialization methods also preserve the sequential ordering of the interleaved thread executions, and thus also the local invariants of the original program. This property ensures that the value analysis can produce good over-approximations of the variable ranges (i.e., tight intervals). We instantiate our approach with the lazy sequentialization implemented in Lazy-CSeq (see Section 2.3), and the value analysis given by Frama-C.

3.2.2 Value analysis

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

The value analysis of Frama-C [CCM09] is a plug-in based on abstract interpretation and is capable of handling C programs with pointers, arrays, structs, and type casts. Abstract interpretation links the set of all possible executions of a program (*concrete semantics*) to a more coarse-grained semantics (*abstract semantics*). Frama-C explores symbolic execution of the program, translating all operations into the abstract semantics. For the soundness of the approach, any transformation in the concrete semantics must have an abstract counterpart that captures all possible outcomes of the concrete operation. Thus, when several execution paths are possible, e.g., when analysing an if-statement, all branches need to be explored and then at the point where the branches join together, e.g., after the if statement, the union of the results along each branch is taken. For-loops require additional care, since value analysis is not guaranteed to terminate. However, this feature is not used in our approach as the output of Lazy-CSeq does not contain loops (bounded program).

As an example, consider the sequentializazion of the Producer/Consumer program in Figure 2.4 generated by Lazy-CSeq in the sequentialized code shown in Figure 3.1 (see Section 2.3 for more details). On this program, Frama-C computes for the integer shared variable `c` and the integer local variable `tmp` of producer threads the interval of values $[-2, 5]$. Thus, in the verification analysis we can safely reduce the size of these integer variables to 4 bits (one bit is for the sign) instead of the standard 32 bits used for the type `int`. Therefore, we can transform the sequentialized program accordingly by replacing the type `int` in the declaration of these variables with the bitvector type.

3.3 Implementation

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

The architecture of Lazy-CSeq+ABS is shown in Figure 3.2. We now briefly illustrate the architecture of Lazy-CSeq (shown in Figure 3.2 in blue, see Section 2.4), and then incrementally describe how we have extended it.

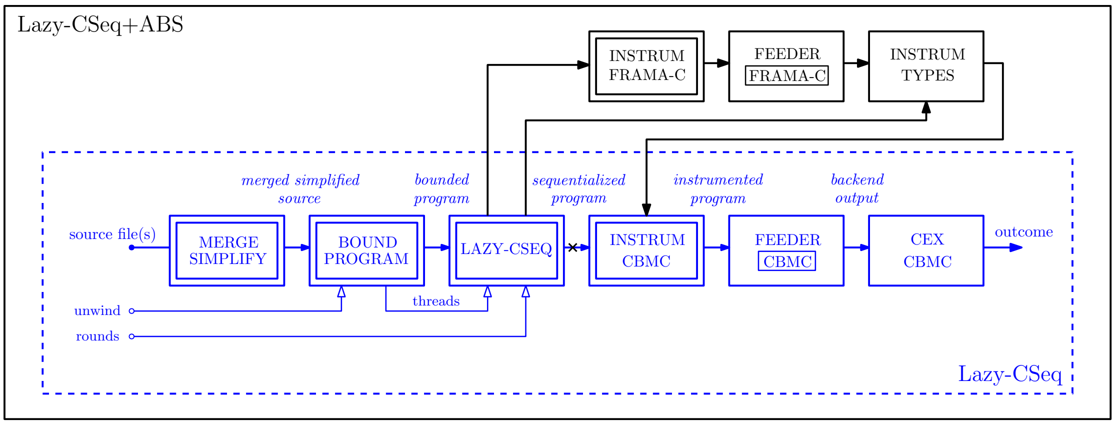


Figure 3.2: Lazy-CSeq+ABS architecture.

Lazy-CSeq consists of a chain of modules:

- a module that preprocesses the source files merging them into a single file;
- a module that simplifies the syntax;
- a module for unrolling loops and inlining functions to produce a bounded program;
- a module that implements the Lazy-CSeq sequentialization [ITF+14a] which produces a backend-independent sequentialized file;
- a module to instrument the sequentialized file for a specific backend (in our case, CBMC);
- two wrappers, one for backend invocation (FEEDER), and another one that generates counterexamples (CEX).

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

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

3.4 Experiments

In this section we report on a large number of experiments where we compare Lazy-CSeq v1.0 and Lazy-CSeq+ABS with the aim of demonstrating the effectiveness of the approach proposed in this chapter. The results of this empirical study show that Lazy-CSeq+ABS is substantially more efficient on complex benchmarks, i.e., larger programs that contain rare bugs. Furthermore, for simple benchmarks, which Lazy-CSeq v1.0 already solves quickly, the overhead of running Frama-C is often negligible.

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

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

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

SV-COMP'16 benchmarks

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

For these benchmarks the experiments are split in two parts: Table 3.1 reports on the experiments for the unsafe benchmarks and Table 3.2 on those for the safe ones. Each row of these two tables summarises the experiments by grouping them into sub-categories. For each sub-category, we report the number of files and the total number of lines of code in that sub-category. The tables also gather the results of the experiments performed using Lazy-CSeq v1.0 and Lazy-CSeq+ABS on these benchmarks. For the

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

CBMC backend analysis, we indicate with *time* the average time in seconds, *mem* the average memory peak usage expressed in MB, and with *#vars* and *#clauses* the average number of variables and clauses of the CNF formula produced by CBMC. Furthermore, only for Lazy-CSeq+ABS, the column *Frama-C* denotes the average time in seconds taken by Frama-C for the value analysis.

			Lazy-CSeq				Lazy-CSeq+Abs					
			CBMC				CBMC				Frama-C	Total
Subcategory	#files	LOC	time	memory	#vars	#clauses	time	memory	#vars	#clauses	time	time
pthread	17	4085	34.69	84.91	89317	336250	18.04	66.79	47961	184287	5.47	23.51
pthread-atomic	2	204	1.66	33.29	9131	29186	1.80	46.22	6259	17936	0.89	2.69
pthread-ext	8	780	6.54	358.42	647840	2654905	4.46	83.12	89718	423391	1.05	5.50
pthread-lit	3	123	1.93	38.33	9993	31206	1.94	49.29	5882	16421	1.20	3.14
pthread-wmm	754	236496	2.01	31.38	2427	5668	2.19	46.08	2402	5578	0.92	3.12

Table 3.1: Experiments on SV-COMP'16 unsafe benchmarks

			Lazy-CSeq				Lazy-CSeq+Abs					
			CBMC				CBMC				Frama-C	Total
Subcategory	#files	LOC	time	memory	#vars	#clauses	time	memory	#vars	#clauses	time	time
pthread	15	1285	172.44	1124.41	1732068	7270420	98.59	945.27	1424912	6004425	8.43	107.02
pthread-atomic	9	1136	2.73	37.92	18947	67709	2.91	47.66	16611	58334	2.01	4.92
pthread-ext	45	3683	71.69	876.75	1660452	6949976	49.35	552.60	937204	4036919	2.19	51.55
pthread-lit	8	432	5.81	43.70	15207	57356	4.87	51.58	11094	42161	0.98	5.86
pthread-wmm	144	29282	1.62	31.45	3154	9420	1.62	45.72	3065	9084	0.89	2.52

Table 3.2: Experiments on SV-COMP'16 safe benchmarks

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

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

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

Complex benchmarks

We now report on the experiments for the three unsafe benchmarks that present a non-trivial challenge for bug-finding tools. These benchmarks consist of non-blocking algorithms for shared data-structures. It is hardly surprising that lock-free programming is an important source of truly concurrently complex benchmarks. In fact, the focus there is to minimise the amount of synchronisation for performance optimisation thus generating a large amount of non-determinism due to interleaving. Here we demonstrate that Lazy-CSeq is very effective in spotting these rare bugs and that Lazy-CSeq+Abs allows us to amplify its effectiveness both in terms of verification time and memory peak usage.

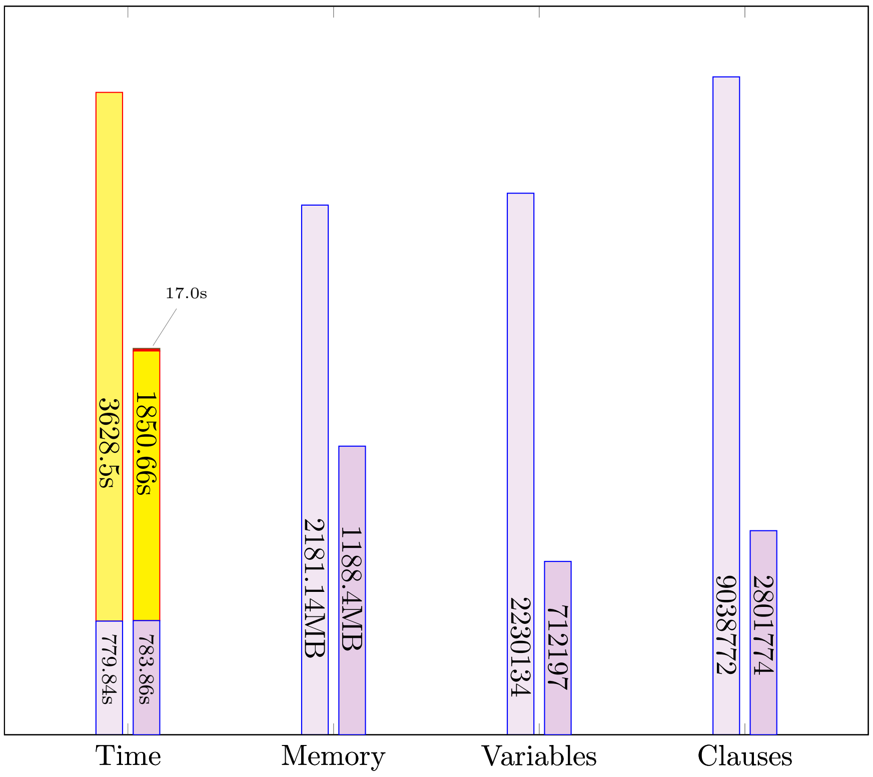


Figure 3.3: Experiment on `safestack` benchmark.

`safestack` benchmark. This is a real-world benchmark implementing a lock-free stack designed for weak-memory models. It was posted to the CHESS forum by Dmitry

Vyukov[2]. It is unique in the sense that it contains a very rare bug that requires at least three threads and five context-switches to be exposed when running under the SC semantics. In the verification literature, it was shown that real-world bugs require at most three context-switches to manifest themselves [23]. `safestack`, for this reason, presents a non-trivial challenge for concurrency testing and symbolic tools. Lazy-CSeq is the only tool we are aware of that can automatically find these bugs: it requires about 1h:13m:28s to find one of them and has a memory peak of 2.18 GB (by setting the minimal parameters to expose the bug to 4 rounds of computation and 3 loop-unwinding). Lazy-CSeq+ABS, with the same parameters, requires 44m:11s time, where 17s is the time required for the value analysis by Frama-C, which leads to a 1.7x speed-up. Also, it uses only 1.19 GB of memory, i.e., roughly half of the memory required by Lazy-CSeq. All this is illustrated in Figure 3.3 where we also report on the number of variables and clauses of the produced CNF formulas.

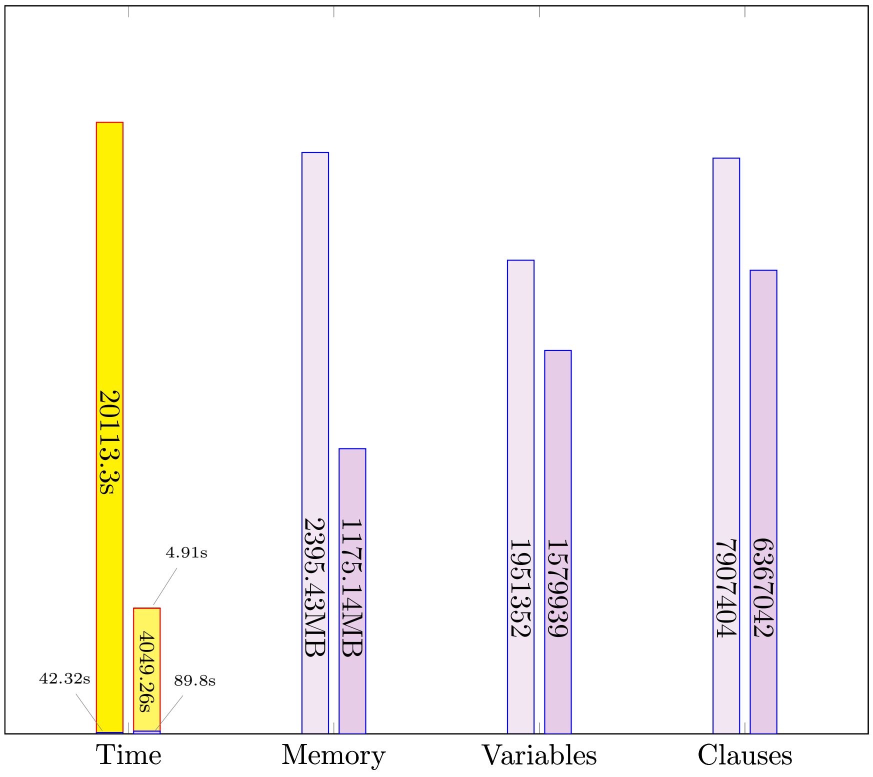


Figure 3.4: Experiment on `eliminationstack` benchmark.

`eliminationstack` benchmark. This is a C implementation of Hendler et al.'s Elimination Stack [HSY04] that follows the original pseudocode presentation. It augments Treiber's stack with a "collision array", used when an optimistic push or pop detects a conflicting operation; the collision array pairs together concurrent push and pop operations to "eliminate" them without affecting the underlying data structure.

[2]https://social.msdn.microsoft.com/Forums/en-US/91c1971c-519f-4ad2-816d-149e6b2fd916/bug-with-a-context-switch-bound-5?forum=chess

This implementation is incorrect if memory is freed in pop operations. In particular, if memory is freed only during the “elimination” phase, then exhibiting a violation (an instance of the infamous ABA problem) requires a seven thread client where three push operations are concurrently executed with four pops. To witness the violation, the implementation is annotated with several assertions that manipulate counters as described by Bouajjani et al. [BEEH15]. Lazy-CSeq is the only tool we are aware of that can automatically find bugs in this benchmark and requires almost 5h:35m:13s time and 2.39 GB of memory to find a bug. Lazy-CSeq+ABS, with the same parameters, requires 1h:07m:29s time, where 4.9s is the time required for the value analysis by Frama-C, which leads to a 5x speed-up. As for the memory usage, it uses only half of the memory required by Lazy-CSeq—namely 1.17 GB. All this is illustrated in Figure 3.4 where we also report on the number of variables and clauses of the produced CNF formulas.

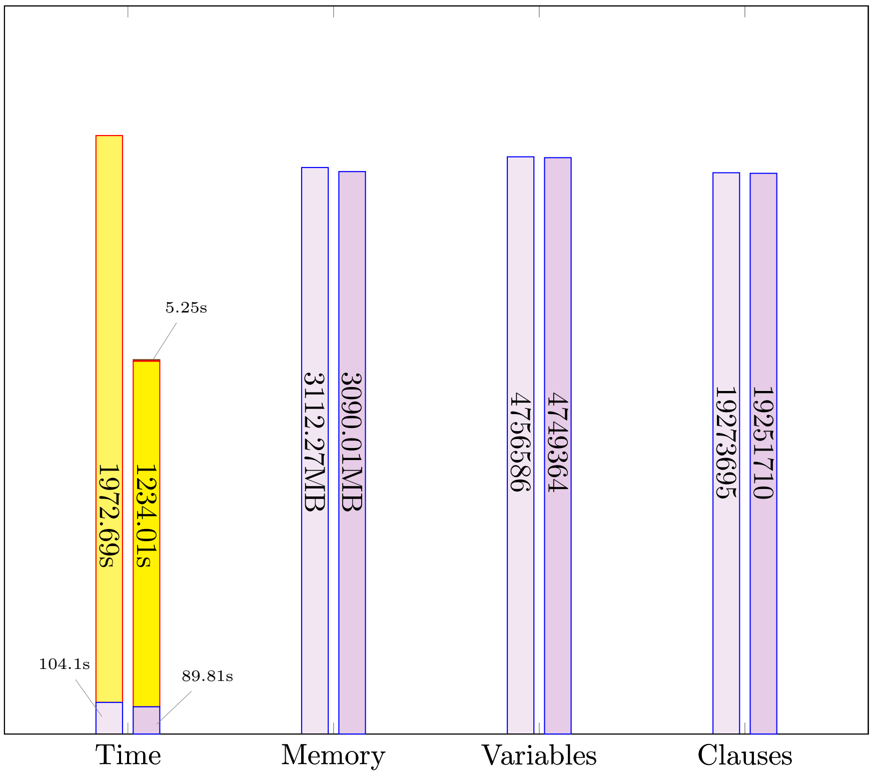


Figure 3.5: Experiment on DCAS benchmark.

DCAS benchmark. This is a non-blocking algorithm for two-sided queues presented by Agesen et al. [ADF+00]. This algorithm has a subtle bug that was discovered in an attempt to prove its correctness with the help of the PVS theorem prover. The discovery of the bug took several months of human effort. Although the bug has been automatically discovered using the model checker SPIN (see [Hol14] and http://spinroot.com/dcas/), a generalised version of the benchmark remains a challenge for explicit exploration approach. In fact, after 138h of CPU-time (using 1000 cores) and an exploration of 10^{11} states the error was still undetected[3] [Hol16]. Here, we have translated this benchmark from Promela to C99 with POSIX thread library considering a more complex version that has 10 threads while the version proposed by Holzmann [Hol16] only considers 8 threads. Lazy-CSeq can detect the bug within 1972.69 seconds and with a memory peak usage of 3112.27MB. Instead, Lazy-CSeq+Abs takes only 1234.01 seconds with a memory peak of 3090.01MB. All this is illustrated in Figure 3.5 where we also report on the number of variables and clauses of the produced CNF formulas.

3.5 Related Work

The idea of sequentialization was originally proposed by Qadeer and Wu [QW04]. The first schema for an arbitrary but bounded number of context switches was given by Lal and Reps [LR09]. Since then, several algorithms and implementations have been developed (see [FIP13a, LQL12, CGS11, LMP09a, LMP09b, TIF$^+$15b]). *Lazy* sequentialization schemas have played an important role in the development of efficient tools. The first such sequentialization was given by La Torre et al. [LMP09a] for bounded context switching and extended to unboundedly many threads in the works of the same authors [LMP10, LMP12]. These schemas require frequent recomputations and are not suitable to be used in combination with bounded model checking (see [GHR10]). Lazy-CSeq [ITF$^+$14a] avoids such recomputations and achieves efficiency by handling context-switches with a very lightweight and decentralised control code. Lazy-CSeq has been recently extended to handle relaxed memory models [TNI$^+$16a] and to prove correctness [NFLP16a].

Abstract interpretation [CC77a] is a widely used static analysis technique which has been scaled up to large industrial systems [CCF$^+$09]. However, since the abstraction functions typically over-approximate the values a program variable can take on, abstract interpretation is prone to false alarms, and considerable effort went into designing suitable abstractions (e.g., [OV15, Ven12]).

An alternative approach combines abstract interpretation with a post-processing phase based on a more precise analysis to either confirm or filter out warnings. Post et al. [PSKG08] describe a semi-automatic process in which they use CBMC repeatedly on larger and larger code slices around potential error locations identified by Polyspace.[4] They report a reduction of false alarms by 25% to 75%, depending on the amount of manual intervention. Chebaro et al. [CKGJ11, CCK$^+$14] describe the SANTE tool, which uses dynamic symbolic execution or concolic testing to try and construct concrete test inputs that confirm the warnings. The main difference to our work is that such approaches use abstract interpretation only to “guide” the more precise post-processing

[3] `http://spinroot.com/dcas/`
[4] `https://www.mathworks.com/products/polyspace.html`

phase towards possible error locations but do not inject information from the abstractions into the post-processing in the same way as we have done in our work.

Wu et al. [WCM+15] also combine sequentialization and abstract interpretation, but in a different context and with different goals. More specifically, they consider interrupt-driven programs (IPDs) for which they devise a specific lazy sequentialization schema; they then run a specialised abstract interpretation, which takes into account some properties of the IPDs such as schedulability, in order to prove the absence of some numerical run-time errors. In contrast, we consider general C programs over the more general POSIX thread library, and use a generic sequentialization schema but a simpler abstract interpretation. However, the main difference is that we use the abstract interpretation only to produce hints for a more precise analysis (i.e., BMC), and not to produce the ultimate analysis result.

3.6 Conclusions

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

Our approach is easy to implement and effective because of the confluence of four different strands. First, we use a source-to-source transformation tool for the sequentialization. This makes it easy to re-use an off-the-shelf tool (i.e., Frama-C) for the interval analysis. Second, we use a backend verification tool (i.e., CBMC) that can effectively exploit the information provided by Frama-C, by means of a specialised bitvector type. Third, we are using a lazy sequentialization, which ensures that the interval analysis can compute tight intervals; our approach would not work with an eager sequentialization where the state variables remain unconstrained. Fourth, the interval analysis strikes the right balance between analysis efforts and results—that is, it runs fast enough, and the computed intervals are tight enough, so that the overheads are easily recouped, and we actually increase the overall performance. Other, more elaborate, abstract interpretations have in fact proven to be counter-productive.

Chapter 4

A Pragmatic Verification Approach for Concurrent Programs

Concurrency poses a major challenge for program verification, but it can also offer an opportunity to scale when subproblems can be analysed in parallel. In this chapter, we exploit this opportunity and use a parametrisable code-to-code translation to generate a set of simpler program instances, each capturing a reduced set of the original program's interleavings. These instances can then be checked independently in parallel. Our approach does not depend on the tool that is chosen for the final analysis, is compatible with weak memory models, and amplifies the effectiveness of existing tools, making them find bugs faster and with fewer resources. We use Lazy-CSeq as an off-the-shelf final verifier to experimentally demonstrate that our approach is able to find bugs in the hardest known concurrency benchmarks in a matter of minutes where other dynamic and static tools fail to conclude.

The content of this chapter is largely based on our technical report [NFLP16b].

4.1 Introduction

Developing correct, scalable, and efficient concurrent programs is a complex and difficult task, due to the large number of possible concurrent executions that must be considered. Modern multi-core processors and weak memory models make this task even harder, as they introduce additional executions that confound the developers' reasoning. Due to these complex interactions, concurrent programs often contain bugs that are difficult to find, reproduce, and fix.

Existing automatic bug-finding techniques and tools are not effective when facing concurrent programs. They struggle particularly with programs that contain rare concurrency bugs, i.e., programs where only a few, specific interleavings violate the specification. For techniques that analyse executions explicitly, finding rare bugs is like looking for a needle in a haystack. For techniques that analyse all executions collectively using symbolic representations, finding rare bugs is also challenging due to the large amount of memory required for the analysis. As a result, we currently do not have techniques and tools that can reliably find such rare bugs.

Although concurrency is clearly a problem for reasoning about programs, it also offers a chance to scale up verification, as suggested by Holzmann et al. [HJG11]: *"...to scale applications of logic model checking to larger problem sizes then, we must be able to leverage the availability of potentially large numbers of processors that run at [...] relatively low speed."*

Different approaches have been tried out to achieve this leveraging, with varying degrees of success. In (truly) *distributed algorithms*, multiple processors are running the same algorithm jointly on the same problem and periodically exchange information. However, verification using distributed model checking techniques (e.g., [SD97, OU09]) has had limited success because they need to share too much information, leading to high communication overheads and contention.

In *strategy competition*, multiple processors are running different variants of the same underlying algorithm independently (i.e., without exchanging information) on the same problem; the first variant that produces a definitive answer (i.e., counter-example or proof) "wins" and aborts the others. This exploits the fact that complex search procedures such as model checkers [HJG08, HJG11], SAT solvers [XHHL08], or first-order theorem provers [Sch96, WL99, SS99] have many control parameters and strategies that can be used to explore different parts of the search space. Holzmann et al. have applied this idea in *swarm verification* [HJG08, HJG11] to scale up model checking based on explicit state space exploration. More specifically, their approach is to spawn a large number of instances (the "swarm") of the SPIN model checker, each with different parameters and search strategies; each instance runs an incomplete search, but in aggregate the swarm substantially outperforms an exhaustive search.

Contributions. In this chapter, we propose a different approach called *task competition*: we run the *same* algorithm on multiple processors, again in competition without information exchange, but now on *different* and *easier* verification tasks derived from the original problem. Specifically, each task captures a subset of the program's interleavings under analysis, in a way that each of such interleavings is captured by at least one of these tasks. Thus, for programs with rare concurrency bugs, most tasks do not contain a bug. However, in tasks that do contain such bugs, the bugs are generally more frequent (i.e., manifest in a higher fraction of the interleavings) than in the original program.

Consequently, bugs can be found faster and with fewer resources, because the individual tasks are simpler and can be analysed in parallel, each with a shorter time-out and a smaller memory.

We develop and evaluate this approach under a bounded context-switch analysis where only interleavings with up to k context-switches (for a given k) are explored. This choice is justified by an empirical study which shows that most of the concurrency bugs manifest themselves within a small number of context-switches [MQ07]. We use a code-to-code translation to derive the tasks as variants of the original program by splitting the code of each thread into fragments (*tiles*) and allowing context-switches only in some of them. By selecting k tiles in all possible ways, we thus ensure the coverage of all interleavings up to k context-switches.

Our approach offers a number of advantages. First, since it is using code-to-code translations, it is "agnostic" of the underlying verification techniques and tools: existing bug-finding tools can be reused as is, and while we have achieved very good results using bounded model checking (BMC) techniques (in particular Lazy-CSeq [INF+15, ITF+14b, ITF+14a]), it can also be used with other symbolic analysis techniques, explicit state space exploration techniques, or even testing.

Second, our approach amplifies the effectiveness of existing bug-finding tools. We empirically demonstrate that it is particularly effective for symbolic methods: it reduces the memory consumption and runtime of each individual verification task, and also leads to a considerable reduction in time for the global verification. More specifically, we demonstrate a substantial reduction in the wall-clock times required to find the bug in some very difficult problems: from 8-12 hours using a single instance of Lazy-CSeq, the only tool capable of finding the underlying bugs, down to 15-30 minutes using Lazy-CSeq on a modest number (5-50) of processors. Looking at this from an opposite perspective, our approach enables existing tools to find rare bugs that were previously out of their reach.

Third, our approach is also oblivious of the assumed memory model and therefore works for WMMs, as long as the underlying analysis tool supports their semantics. Moreover, our experiments demonstrate that the approach is also effective for reducing the additional verification complexity introduced by relaxed semantics.

Finally, our approach is tuneable. The verification complexity of each of the instances generally depends on the underlying analysis tool and number of interleavings captured by each instance. Our technique allows us to control the number of interleavings. From our experience, instances with roughly the same number of interleavings have similar verification times. We empirically learn the number of interleavings per instance that the underlying tool can handle and then generate all the instances to capture all interleavings according to a fixed schema. However, it can happen that the number of instances to generate can be extremely high. In that case we have shown empirically that, even when accounting for only a few randomly selected ones, we are still able to find rare bugs. In our experiments we demonstrate that we only have to consider a few instances (out of millions or billions) to find bugs with high probability.

In summary, in this chapter we make three main contributions. First, we propose a new swarm verification approach for the analysis of concurrent programs that is based on a code-to-code translation and leverages the power of sequential verification engines. Second, we implement the approach as an extension to Lazy-CSeq. Third, we report the results of an evaluation of our approach on the two hardest known concurrency benchmarks, `safestack` [Vyu10] and `eliminationstack` [HSY04] for three different memory models (SC, TSO, PSO).

Organisation of the chapter. In the next section, we give a high-level overview of our approach. Sections 4.3 and 4.4 describe our code-to-code translation. Section 4.5 gives details on our implementation and Section 4.6 presents the results of our experimental evaluation. Section 4.7 compares with related work, and Section 4.8 concludes.

4.2 Approach

We consider a multi-threaded program where threads communicate through shared memory, for example, a C program that uses the POSIX threads library for concurrency (see Section 2.1). As in bounded model checking, we first flatten the program by inlining functions and unrolling loops up to a given bound. The resulting *bounded* program, say P, consists of a finite number of threads; the control in each thread can only move down in the code. The goal of the analysis is to find an assertion violation of P that may occur through an execution that involves at most k context-switches (for a given k). Let k be a small natural number denoting the maximum number of context-switches to consider along an execution and T be the set of P's threads. We denote with $\mathcal{I}_k(P)$ the set of all executions that P can exhibit with at most k context-switches.

4.2.1 Splitting Computations with Tilings

Our goal is to define a code-to-code translation for P, parameterised over k, that generates a set of simpler program variants, each capturing a subset of P's executions, and such that each of P's executions involving at most k context-switches is captured by at least one of them. The resulting variants can then be checked independently in parallel.

We construct these variants by building on the notion of tiling. A *tiling of a thread* $t \in T$ is a partition of t's statements. Each element of a tiling is called *tile*. For example, consider Figure 4.1. The program has two threads with respectively seven (A, ..., G)

```
thread0{          thread1{
#1 { A;             H; }
   { B;             I; } #4
#2 { C;             J; } #5
   { D;             K; }
   { E;             L; } #6
#3 { F;           }
   { G;
}
```

Figure 4.1: Tiling example

and five (H, ..., L) statements, and the tilings of each thread are marked with the braces. A *tiling of a program* P is a set of threads' tilings, one for each thread of P.

Let $\Theta_P = \{\Theta_t\}_{t\in T}$ be a tiling of P. A *z-selection* over Θ_P is a set $\{\theta_t\}_{t\in T}$ where $\theta_t \subseteq \Theta_t$ contains exactly z tiles for each thread $t \in T$.

We build the variants as follows. For a given tiling Θ_P of P and any of its z-selections $\vartheta = \{\theta_t\}_{t\in T}$, we construct a program variant P_ϑ obtained from P by instrumenting it in a way that each thread t can only be pre-empted at statements belonging to the tiles of θ_t and at any other blocking statement of t (this last is to allow an execution to continue when a statement is blocked and there are other threads that can execute). For example, consider again Figure 4.1. If we take the 1-selection ϑ corresponding to selecting tiles #1 and #4, the executions of the corresponding variant P_ϑ are of the form uvw where: (1) u is any interleaving of A-B and H-I; (2) if u ends with B, then $v = C$-...-G and $w = J$-K-L; (3) if u ends with I, then $v = J$-K-L and $w = C$-...-G. For example, A-B-H-...-L-C-...-G and A-H-I-B-C-...-G-J-K-L denote possible executions of P_ϑ.

We observe that the set of executions over all the variants P_ϑ, for ϑ being a $\lceil\frac{k}{2}\rceil$-selection over Θ_P, that contain at most k context-switches is exactly the set $\mathcal{I}_k(P)$. In fact, every execution of a variant P_ϑ of P is also an execution of P since P_ϑ, by construction, is the same as P, except that we forbid context-switches to occur at some points. Vice-versa, along any execution $\pi \in \mathcal{I}_k(P)$ clearly we context-switch out of each thread at most $\lceil\frac{k}{2}\rceil$ times, thus it suffices to select $\lceil\frac{k}{2}\rceil$ tiles per thread to capture π. Therefore, π is an execution of some variant P_ϑ, for a $\lceil\frac{k}{2}\rceil$-selection ϑ.

4.2.2 Tile Selection versus Random Selection

We compare the splitting strategy based on the notion of tiles and a splitting strategy based on a direct random selection of the points where threads can be pre-empted. In the following, we refer to the first strategy as TS (*tiling selection*) and the second one as IS (*individual program-counter selection*). We compare the two strategies on the number of instances that they need to generate in order to ensure the coverage of all the runs where, in each thread, a pre-emption may occur at most r times (*bounded context-switching analysis*). We show that the number of possible instances generated with IS is larger than the one with TS, by an exponential factor in the number of the selected points.

For the ease of presentation we adopt some simplifications. We assume that all threads have the same number of statements (we recall that the splitting is applied to bounded programs; thus each statement of the considered program is executed at most once in a run). Further, we assume that the points where a pre-emption is allowed are evenly split among all threads.

Therefore, we make use of the following parameters in our analysis:

- n is the number of threads;
- k is the number of statements per each thread;
- r is the maximal number of pre-emptions allowed in each thread;
- m is the number of points (program counters) for each thread that are selected to allow pre-emption; thus, $r \leq m \leq k$;
- s is the size of tiles, i.e., the number of statements in each tile.

Since in TS, for covering all runs that pre-empt each thread at least r time, we need to pick at most r tiles per thread; we also assume that $m = r \cdot s$. We finally assume that k is a multiple of m and thus s.

We denote with χ_{TS} the number of all instances that are generated by TS under the above assumptions and with χ_{IS} the one by IS. By a simple counting, it can be easily verified that:

$$\chi_{TS} = (T_{TS})^n, \text{ where } T_{TS} = \frac{\frac{k}{s}!}{r!\,\left(\frac{k}{s} - r\right)!} \tag{4.1}$$

$$\chi_{IS} = (T_{IS})^n, \text{ where } T_{IS} = \frac{k!}{m!\,(k-m)!} \tag{4.2}$$

Denote with $\Gamma = \frac{T_{IS}}{T_{TS}}$, from equations 4.1 and 4.2, we get:

$$\Gamma = \frac{k!\; r!\; \left(\frac{k}{s} - r\right)!}{m!\; (k-m)!\; \frac{k}{s}!} \tag{4.3}$$

From

$$\frac{k!}{(k-m)!} = k\;(k-1)\;\ldots(k-m+1) \tag{4.4}$$

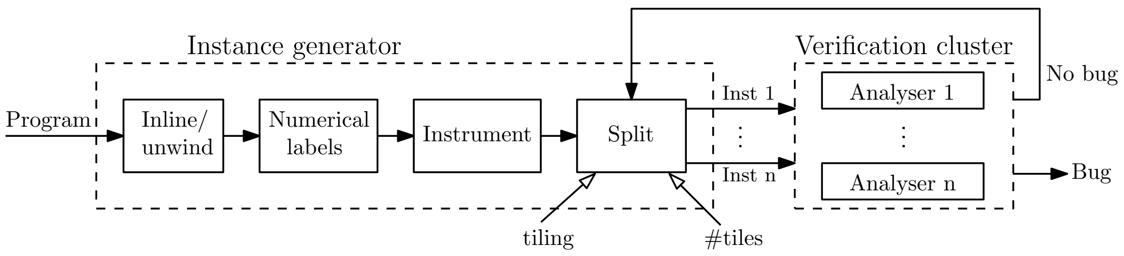


Figure 4.2: Verification approach.

$$\frac{\left(\frac{k}{s}-r\right)!}{\frac{k}{s}!} = \frac{1}{\frac{k}{s}\ \left(\frac{k}{s}-1\right)\ \ldots \left(\frac{k}{s}-r+1\right)} \tag{4.5}$$

$$m! = \frac{1}{m\ (m-1)\ldots(r+1)\ r!} \qquad \text{(recall } m \geq r\text{)} \tag{4.6}$$

we get

$$\Gamma = \frac{k\ (k-1)\ \ldots (k-m+1)\ r!}{m\ (m-1)\ldots(r+1)\ r!\ \frac{k}{s}\ \left(\frac{k}{s}-1\right)\ \ldots \left(\frac{k}{s}-r+1\right)} \tag{4.7}$$

By simplifying the $r!$ factor and splitting the fraction as the product of fractions taking a term from the numerator and a term from the denominator in the order they appear in the above formula, we get:

$$\Gamma = \frac{k}{m} \cdot \frac{(k-1)}{(m-1)} \cdot \ldots \cdot \frac{(k-m+r+1)}{(r+1)} \ \cdot\ \frac{(k-m+r)}{\frac{k}{s}} \cdot \ldots \cdot \frac{(k-m+1)}{\left(\frac{k}{s}-r+1\right)} \tag{4.8}$$

Now denote $\alpha_i = \frac{(k-i)}{(m-i)}$ for $i = 0, \ldots, m-r-1$ and $\beta_i = \frac{(k-m+r-i)}{\frac{k}{s}-i}$ for $i = 0, \ldots, r-1$. Let $\alpha(x) = \frac{(k-x)}{(m-x)}$ and $\beta(x) = \frac{(k-m+r-x)}{\frac{k}{s}-s}$ be the corresponding functions over a continuous domain.

We observe that $\frac{d}{dx}\alpha(x) = \frac{k-m}{(m-x)^2}$ and $\frac{d}{dx}\alpha(x) > 0$ for $x \neq m$. Therefore, $\frac{k}{m} = \alpha_0 < \alpha_1 < \ldots < \alpha_{m-r-1}$.

Also, $\frac{d}{dx}\beta(x) = \frac{s\ (s-1)\ (k-m)}{(k-s\,x)^2}$ and $\frac{d}{dx}\beta(x) > 0$ for $x \neq \frac{k}{s}$. Moreover, $\frac{(k-m+r)}{\frac{k}{s}} = \frac{s\ (k-m+r)}{k}$. Therefore, $\frac{s\ (k-m+r)}{k} = \beta_0 < \beta_1 < \ldots < \beta_{r-1}$.

Since $\frac{k}{m} > 1$, we get that $\alpha_i > 1$ for $i = 0, \ldots, m-r-1$. From $\frac{s\ (k-m+r)}{k} = s - \frac{m}{k}(s-1)$, assuming $s > 1$ (non trivial tiles), also $\beta_i > 1$ for $i = 0, \ldots, r-1$. Thus, a constant $c \geq \min\{\frac{k}{m}, s - \frac{m}{k}(s-1)\} > 1$ exists such that $\Gamma = c^m$, and hence $\chi_{IS} = c^{m\,n} \cdot \chi_{TS}$ that shows our claim. $\square$

4.2.3 Overall approach

Our verification approach works in two phases. We first generate P's variants according to any selection of an input tiling. We then search for bugs in each of the resulting program variants (typically in parallel) using an analyser such as a testing tool or a model checker. The analysis phase can be stopped as soon as we find a bug in any of the generated program variants. The overall scheme of our approach is shown in Figure 4.2; in the following, we sketch the two phases in turn.

Instance Generation. The first phase is composed of a chain of code-to-code transformations of the input multi-threaded program.

The first module transforms this program into a bounded multi-threaded program P that is syntactically guaranteed to terminate after a bounded number of transitions, by applying standard BMC program transformations [CKY03] such as inlining the functions and unwinding the loops (up to a given bound). P thus has a different function associated with each thread; we refer to these as *thread functions*.

The second module injects into P numerical labels at each *visible* statement of the thread functions (i.e., at the beginning and end of the function, before each access to the shared memory, and before each call to a thread synchronisation primitive). The labels start at zero in each thread function and increase consecutively in statement order; we assume that any other label of the program is non-numerical. This labelling simplifies the code injected by the third module for the tile selection.

The third module instruments the code with guarded commands that at each numerical label enable/disable context-switch points and statement reordering. This control-flow code is used by the next module to capture the tile selection in the code. The detailed description of the translation by this module will be given in Section 4.4.

The fourth and last module generates the variants of P according to any z-selection of the input tiling Θ_P (where z is the value of the input parameter #tiles). This is done by triggering the guards injected by the previous module.

Note that the number of different program variants that we can generate this way is finite, but can be large. Therefore, we consider a randomised version of this module along with a new input parameter, the number n of instances to be generated. The n instances are generated by *randomly* choosing the z-selections. This, also introduces a *loop* in our verification approach: we repeat the random generation of n new variants until either we find a bug or we have already generated all the variants.

Verification Cluster. Since the generated problem instances can be solved independently, we can achieve in our scheme a high diversification and parallelism of the analysis. In fact, we can solve each instance on a separate core and possibly using a different tool for concurrent program verification.

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

Figure 4.3: Revised syntax of multi-threaded programs.

4.3 Programming and Execution Models

Our implementation can handle the full C language (see Section 4.5), but we describe our approach for multi-threaded programs in a simple imperative language (see Section 2.1). This features dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronisation. Thread communication is implemented via shared memory and modelled by global variables. All threads share the same address space: they can write to or read from global (*shared*) variables of the program to communicate with each other. We assume that each statement is atomic. This is not a severe restriction, as it is always possible to decompose a statement into a sequence of statements, each involving at most one shared variable [Mül06].

Syntax. The syntax of multi-threaded programs is defined by the grammar shown in Figure 4.3. Terminal symbols are set in typewriter font. Notation $\langle n\ \texttt{t}\rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals `t`; x denotes a local variable, y a shared variable, t a thread variable and p a procedure name. All variables involved in a sequential statement are local. We assume expressions e to be local variables, integer constants, which can be combined using mathematical operators. Boolean expressions b can be `true` or `false`, or Boolean variables, which can be combined using standard Boolean operations.

A *multi-threaded* program consists of a list of *global* variable declarations (i.e., *shared* variables), followed by a list of procedures. Each procedure has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement. A statement is either a simple statement or a compound statement, i.e., a sequence of statements enclosed in braces. A simple statement is either a labelled simple statement, or a sequential statement, or a concurrent statement.

A *sequential statement* can be an `assume`- or `assert`-statement, an assignment, a call to a procedure that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional statement, a `while`-loop, or a jump to a label. Local variables are considered uninitialised right after their declaration, which means that they can take any value from their domains. Therefore, until not explicitly set by an assignment statement, they can non-deterministically assume any value allowed by their type. We also use the symbol `*` to denote the expression that non-deterministically evaluates to any possible value, for example, with x`:=*` we mean that x is assigned with any possible value of its type domain.

A *concurrent statement* can be a concurrent assignment, a call to a thread routine, such as a thread creation, a join, or a mutex operation (i.e., init, lock, unlock, and destroy). A concurrent assignment assigns a shared (resp. local) variable to a local (resp. shared) one. Unlike local variables, global variables are always assumed to be initialised to a default value. For the sake of simplicity, we assume that the default value always `0` regardless of the variable type. A thread creation statement $t :=$ `create` $p(e_1, \ldots, e_n)$ spawns a new thread from procedure p with expressions $e_1, \ldots, e_n$ as arguments. A thread join statement, `join` t, pauses the current thread until the thread identified by t terminates its execution, i.e., after the thread has executed its last statement. Lock and unlock statements respectively acquire and release a mutex. If the mutex is already acquired, the lock operation is blocking for the thread, i.e., the thread is suspended until the mutex is released and can then be acquired.

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

Semantics. As common, a program configuration is a tuple of configurations of each thread that has been created and has not yet terminated, along with a valuation of the global variables. A thread configuration consists of a *stack* which stores the history of positions at which calls were made, along with valuations for local variables, and the top of the stack contains the local and global valuations, and a pointer to the current statement being executed.

The behavioural semantics of a program P is obtained by interleaving the behaviours of its threads. At the beginning of any computation only the main thread is *ready* and *running*. At any point of a computation, only one of the ready threads is *running*. A

$$
\begin{array}{rcl}
\left[\!\!\left[\begin{array}{l}(dec\texttt{;})^* \ (type\ p_i\ \texttt{(}\langle dec\texttt{,}\rangle^*\texttt{)}\\ \texttt{\{}(dec\texttt{;})^* stm\texttt{\}})_{i=0,\dots,n-1}\end{array}\right]\!\!\right] & \overset{\text{def}}{=} & \begin{array}{l} \texttt{bool yields[n][h]} = \{\{a^0_0, a^0_1, ..., a^0_{h-1}\}, \dots, \\ \qquad\qquad \{a^{n-1}_0, a^{n-1}_1, ..., a^{n-1}_{h-1}\}\}; \\ (dec\texttt{;})^* \ (type\ p_i\ \texttt{(}\langle dec\texttt{,}\rangle^*\texttt{)} \\ \qquad \texttt{\{}(dec\texttt{;})^*[\![stm]\!]_i\texttt{\}})_{i=0,\dots,n-1}\end{array} \\[2ex]
[\![stm]\!]_i & \overset{\text{def}}{=} & [\![sstm]\!]_i \mid \texttt{\{}\langle [\![stm]\!]_i\texttt{;}\rangle^*\texttt{\}} \\[2ex]
[\![sstm]\!]_i & \overset{\text{def}}{=} & seq \mid conc \mid [\![l\colon sstm]\!]_i \\[2ex]
[\![l\colon sstm]\!]_i & \overset{\text{def}}{=} & \begin{cases} l\colon \texttt{if(yields[}i\texttt{][}l\texttt{] \&\& *) yield;} & \\ \quad [\![sstm]\!]_i\texttt{;} & \text{if } l \text{ is numerical} \\ & \\ l\colon [\![sstm]\!]_i & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4.4: Formal description of the code-to-code translation by module *Instrument*.

step is either the execution of a step of the running thread or a *context-switch* that non-deterministically replaces the running thread with one of the ready ones that thus becomes the running thread at the next step. A thread will no longer be available when its execution is terminated, i.e., there are no more steps that it can take.

4.4 Code-to-code translation

In this section, we give a formal description of the code instrumentation done by module *Instrument* in Figure 4.2.

In order to enable/disable context-switches in the code, for the class of programs that form the output of this module we assume a semantics in the style of *pre-emptive asynchronous programs* with non-deterministic scheduler. In particular, we augment the concurrent statements of the syntax from Figure 4.3 with a `yield`-statement, i.e., we add the rule *conc* ::= `yield`, and restrict the context-switches to occur only if explicitly requested via a `yield`-statement (which thus causes the control to return to the non-deterministic scheduler). Moreover, we modify the semantics of `lock`- and `join`-statements such that a thread terminates (instead of pausing) when attempting to acquire a mutex already taken or waiting for the termination of another thread. In the following, we will refer to this class of programs as *extended pre-emptive asynchronous programs* (EPA programs, for short).

It is simple to show that given a multi-threaded program P under the syntax and semantics of Section 4.3 we can easily obtain an equivalent program P' under syntax and semantics sketched above by simply inserting a `yield`-statement guarded by a non-deterministic guess in front of each statement of P. Moreover, as we are interested only in reachability of program counters or assertion failure checking, such as in standard bug-finding analysis, it is sufficient to account only for context-switches that occur at visible statements (i.e., the concurrent statements and each thread's first and last statements).

We recall that we assume that all the labels of the original multi-threaded program must be non-numerical and that after the code-to-code translation by modules *Inline/unwind* and *Numerical labels*, we get that: (1) the code of each thread is all contained within the same procedure, i.e., there are no procedure calls, and there are no loops; and (2) the visible statements are all labelled with a numerical label such that in each thread code labels start from `0` and increase by 1 according to the statement order.

In module *Instrument*, we thus rewrite the code by inserting a guarded `yield`-statement after each numerical label. Guards are triggered by input Boolean parameters. In particular, we use a_l^i to activate the `yield`-statement at the numerical label l of thread i. These parameters are assigned to a Boolean array `yields`.

After the instrumentation, the portion of code l`:`*sstm* of thread i, where l is a numerical label, is thus:

```
    if(yields[i][l] && *) yield; sstm;
```

The rest of the code stays unchanged.

We formally give our code-to-code translation in Figure 4.4 as rewrite rules over the syntax of programs. In the figure, we have denoted with `n` the number of threads and with `h` the maximum number of numerical labels over all threads.

We observe that whenever `yields[`i`][`l`]` holds, by the choice operator `*`, the `yield`-statement is non-deterministically executed or not. This can be used to select in the code the points where context-switches can happen. Thus, the following module *Split* (see Figure 4.2) will assign the array `yields` accordingly to a valid selection for the input tiling.

4.5 Implementation

We have implemented the verification approach illustrated in Figure 4.2 to analyse concurrent C programs that use the concurrency library POSIX threads. We optimise the tilings by taking into account only statements at which context-switches can occur which correspond to numerical labels. For the instance generation we use *uniform window tilings*, i.e., tilings where all tiles have the same number of numerical labels except for the last one that can have fewer, and all tiles correspond to contiguous portions of a thread code. For example, the tiling from Figure 4.1 is not uniform though tiles cover contiguous portions of code.

The *Instance generator* is written as an independent piece of software that takes as input: (1) a multi-threaded program P with assertions, (2) the unwinding bound, (3) the size t of each tile (i.e., the number of numerical labels), (4) the number s of tiles to select from each thread, and (5) the number n of randomly chosen instances to generate. The pool of (bounded) EPA programs generated by the *Instance generator* is then verified on a cluster of computers with a modified version of the symbolic verification tool Lazy-CSeq [INF+15].

Below we provide more details on our implementation of the *Instance generator*, the verification tool, and the cluster.

Instance generator

Our tool, named VeriSmart ("Verification-Smart"), builds upon the CSeq framework [INF+15], and is composed of a chain of software modules that matches the chain of modules of the *Instance generator* from Figure 4.2. We recall that CSeq is a framework that comprises several software modules implementing standard source-to-source transformations of C programs. We re-use CSeq modules to implement the modules *Inline/unwind* and *Numerical labels*. For module *Instrument*, we have realised a new software module that implements the code-to-code translation detailed in Section 4.4. It is written in Python and uses the AST built by `pycparser` on the fetched program to implement the rewriting rules of Figure 4.4. The last module *Split* is also written in Python. It generates each instance by randomly selecting s tiles per thread by setting to true the corresponding entries of the array `yields`. This module also takes an additional parameter that allows to bound the number of generated instances. In our setting we manually allocate the resulting instances to several verification units that are analysed independently.

The code of VeriSmart is publicly available at `http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html` and can be used, in combination with different bug-finding analysis tools, for experimenting with the verification approach proposed in this chapter.

Backend verification tool

Lazy-CSeq [INF+15] is a symbolic bug-finding tool for multi-threaded programs based on sequentialization and bounded model checking. The multi-threaded input program is translated into a corresponding sequential program up to a given number of rounds of execution, where in each round each thread is executed exactly once according to a fixed ordering. The resulting sequential program is then verified using existing verification tools for sequential programs.

We modify Lazy-CSeq to account for the syntax and the semantics of EPA programs. Essentially, we extend the parsing for handling also the `yield`-statement and then adjust the code-to-code translation such that in the resulting sequential C program the context-switches are now simulated only at the `yield`-statements according to the semantics of the EPA programs. In the following, we refer to this modified version of Lazy-CSeq as EPA-Lazy-CSeq.

In our experiments we use the C bounded model checker CBMC [AKT13] as sequential backend for Lazy-CSeq and EPA-Lazy-CSeq. CBMC encodes symbolically the multiple execution paths of the input program, which is then checked by a SAT/SMT solver. If the formula is satisfiable there is a definite execution path that leads to an assertion violation. In this case the SAT/SMT solver returns the values of the variables along this execution path. From these values, which in particular include the input values, a test can be constructed and executed in order to debug the reason for the assertion violation. If the formula is unsatisfiable then there is no execution that violates any of the assertions (up to the considered depth). The formula is generated by symbolically executing the program while encoding the control flow structure into additional Boolean variables. The formula is linear in the size of the program, but implicitly encodes a potentially exponential number of execution paths. Hence, unlike path-wise enumeration approaches [GKS05, CDE08] this approach avoids explicitly enumerating a potentially exponential number of execution paths and maximises the exploitation of today's optimised SAT solvers.

Lazy-CSeq was initially developed for C program running under the SC semantics. It has been recently extended to handle WMM semantics such as TSO and PSO [TNI+16a]. Our modification are also compatible with these extensions. In our experiments we only use EPA-Lazy-CSeq for the analysis carried out on the cluster. Our choice has been motivated by the effectiveness of Lazy-CSeq to cope with complex benchmarks containing rare bugs.

4.6 Experiments

Here we report on a large number of experiments that we have conducted to demonstrate the effectiveness of the VeriSmart approach.

4.6.1 Benchmarks

Our first effort was to identify suitable benchmarks that present a non-trivial challenge for concurrency bug-finding tools. We discarded all concurrency benchmarks used in the 2016 Software Verification Competition because all of them are easy to verify (at

SV-COMP'16 three tools, including an explicit state model checker, were able to find with a timeout of 900 seconds bugs in all benchmarks deemed to be buggy).

We have instead considered several concurrency benchmarks from the literature, including the SCT Benchmarks [TDB16][1]. From these, we have discarded all benchmarks that Lazy-CSeq can solve in less than 900 seconds and are thus "too simple" to benefit substantially from the VeriSmart approach; this includes several benchmarks that are traditionally considered to be hard, e.g., DCAS [DFG+00] or the work stealing queue [TDB16]. We have further discarded all parametric benchmarks whose complexity comes simply from increasing the number of threads (e.g., `CS.reorder_`n`_bad` and `CS.twostage_`n`_bad` [TDB16], where n is the number of threads), or the size of the data structures (e.g., the work stealing queue), since their bugs can already be exposed with smaller instances, and they do thus not reflect realistic bug-finding scenarios[2]. This leaves us with two benchmarks that both come from the domain of concurrent data structures[3].

`eliminationstack` is a C implementation of Hendler et al.'s Elimination Stack [HSY04] that follows the original pseudocode presentation. It augments Treiber's stack with a "collision array", used when an optimistic push or pop detects a conflicting operation; the collision array pairs together concurrent push and pop operations to "eliminate" them without affecting the underlying data structure. This implementation is incorrect if memory is freed in pop operations. In particular, if memory is freed only during the "elimination" phase, then exhibiting a violation (an instance of the infamous ABA problem) requires a seven thread client with three push operations concurrent with four pops. To witness the violation, the implementation is annotated with several assertions that manipulate counters as described in [BEEH15]. Lazy-CSeq is the only tool we are aware of that can automatically find bugs in this benchmark and requires almost 13 hours and 4 GB of memory to find a bug.

`safestack` is a real world benchmark implementing a lock-free stack designed for weak-memory models. It was posted to the CHESS forum by Dmitry Vyukov [Vyu10]. This benchmark is unique in the sense that it contains a very rare bug that requires at least three threads and five context-switches to get exposed when running under the SC semantics, whereas it requires only four context-switches when running either under TSO or PSO. In the verification literature, it was shown that real-world bugs require at most three context-switches to manifest themselves [MQ07]. `safestack`, for this reason,

[1]https://sites.google.com/site/sctbenchmarks/

[2]Note that Lazy-CSeq can generally handle these benchmarks quite well. For example, for `CS.reorder_`n`_bad` and `CS.twostage_`n`_bad`, Lazy-CSeq finds bugs in both benchmarks for $n = 400$ in 15 minutes, whereas testing struggles to find a bug for $n = 20$ and $n = 100$, respectively [TDB16]. For the work stealing queue, Lazy-CSeq finds the bug for queue size 16 and 9 threads in less than 3000 seconds, whereas the method in [TDB16] can only handle a queue size of 4 and 3 threads.

[3]It is hardly surprising that lock-free programming is an important source of truly *concurrently complex* benchmarks since the focus there is to *minimise* the amount of synchronisation for performance optimisation thus generating a large amount of non-determinism due to interleaving.

`eliminationstack`-SC (unwind=1, rounds=2, thread=8, visible point=52)

COMPREHENSIVE VERIFICATION		
	Time	Memory
One schedule	80.8	661.1
All schedules	46764	4203.9

VERISMART: 2 tiles per thread

#1: tile size 12, t_max 1.5hrs			#2: tile size 14, t_max 2hrs			#3: tile size 18, t_max 3hrs		
Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
Min	34.9	945.2	Min	39.7	979.84	Min	37.1	999.8
Max	4753.6	1199.1	Max	7195.2	1281.3	Max	10762.0	1785.5
Average	1116.3	1017.8	Average	2169.5	1096.3	Average	3162.41	1156.91
instances with bug: 38.33%			instances with bug: 61.38%			instances with bug: 69.01%		

`safestack`-SC (unwind=3, rounds=4, thread=4, visible point=152)

COMPREHENSIVE VERIFICATION		
	Time	Memory
One schedule	55.4	700.1
All schedules	24139	6632.4

VERISMART: 4 tiles per thread

#1: tile size 11, t_max 1hr			#2: tile size 14, t_max 1hr			#3: tile size 20, t_max 4hrs		
Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
Min	195.6	774.5	Min	574.8	846.6	Min	313.0	850.3
Max	2662.6	1265.7	Max	3521.8	1450.4	Max	10315.8	3830.8
Average	1172.2	928.8	Average	1851.1	1147.3	Average	2167.5	1230.1
instances with bug: 1.26%			instances with bug: 2.14%			instances with bug: 10.20%		

`safestack`-PSO (unwind=3, rounds=3, thread=4, visible point=152)

COMPREHENSIVE VERIFICATION		
	Time	Memory
One schedule	272.5	2651.8
All schedules	4777	3708.4

VERISMART: 3 tiles per thread

#1: tile size 12, t_max 1.5hrs			#2: tile size 16, t_max 2.25hrs			#3: tile size 20, t_max 3hrs		
Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
Min	898.1	2795.9	Min	593.5	2862.6	Min	1083.8	2910.1
Max	5348.0	3280.7	Max	8083.1	3942.2	Max	10771.7	3784.1
Average	2929.8	2872.9	Average	4607.1	3015.4	Average	5176.9	3073.8
instances with bug: 7.63%			instances with bug: 16.73%			instances with bug: 26.85%		

Figure 4.5: VERISMART experiments: each experiment is carried out using 8,000 instances chosen randomly. The verification is done using Lazy-CSeq 1.0 for all schedules, and EPA-Lazy-CSeq for the program variants. Time is given in seconds and memory in MB. Data refers only to tasks where a bug was found.

presents a non-trivial challenge for concurrency testing and symbolic tools. Lazy-CSeq is the only tool we are aware of that can automatically find these bugs: it requires almost 7 hours to find these bugs and consumes more than 6 GB of memory[4]. We give the code of `safestack` and the shortest counterexample that we found in Appendix 6.2.

[4]The tool Relacy [TDB16] can find a bug in a modified version of `safestack` where an explicit `pthread_yield` call has been added to help the search. In our experiments, on the plain `safestack` benchmark used here, Relacy was not able to detect a bug within one million iterations.

4.6.2 Experimental Set-Up

In our experimental evaluation we compare VeriSmart against Lazy-CSeq v1.0[5]. For VeriSmart, we use EPA-Lazy-CSeq for solving the individual generated tasks. Lazy-CSeq v1.0 instead is used to solve the original problem comprehensively (i.e., analysing all interleavings in one attempt).

In both cases, we use the minimum number of unwindings and rounds of computation that are required to expose the bug in the original program; the exact values for these parameters are reported in Figure 4.5 for each considered benchmark.

For VeriSmart, we use a uniform window tiling with two different tile sizes (see Figure 4.5 again) to evaluate the effect of tiling. For each benchmark, we select as many tiles per thread as the number of rounds required to expose the bug, that as argued in Section 4.2 ensures that the bug can (in principle) be found. We then generate 8,000 tasks by randomly selecting the tiles. This allows us to estimate the probability p of picking a buggy program variant with a confidence level of 0.99, that in turn gives the expected number of variants we need to select randomly in order to discover a bug. We use this to determine the probability of finding a buggy instance when randomly selecting n of them, that is $1-(1-p)^n$.

We also estimate for each considered benchmark and setting, how the expected bug-finding time varies as the number of cores increases. For a pull of n tasks, the verification time is taken as the minimum time to find a bug over all of them. We plot the expected verification time for values of n up to 2,000 cores (see Figure 4.6).

We carried out our experiments on a cluster with 750 compute nodes equipped with dual 2.6 GHz Intel Sandybridge processors. Each compute node has 16 CPUs with 64 GB of physical shared memory running 64-bit linux 3.0.6. On each CPU of a node we run EPA-Lazy-CSeq over a single verification task produced by the instance generator, with the timeout given in Figure 4.5.

4.6.3 Experimental Results

Figure 4.5 and 4.6 summarise the results of our experiments except for `safestack`-TSO that will be reported in the text.

The results for the comprehensive verification using Lazy-CSeq are given in the upper smaller tables of Figure 4.5 (under "All schedules") and for `safestack`-TSO finding a bug requires 11,005 seconds and 4.3 GB of memory. As expected, the results confirm that both benchmarks are very hard; less expectedly, they also show that WMMs actually make it easier to find bugs, as both runtime and memory requirements go down. This

[5]CSeq framework, `http://users.ecs.soton.ac.uk/gp4/cseq`

is a consequence of both the lower number of rounds required to expose the bug, and the higher number of buggy executions that the WMMs allow.

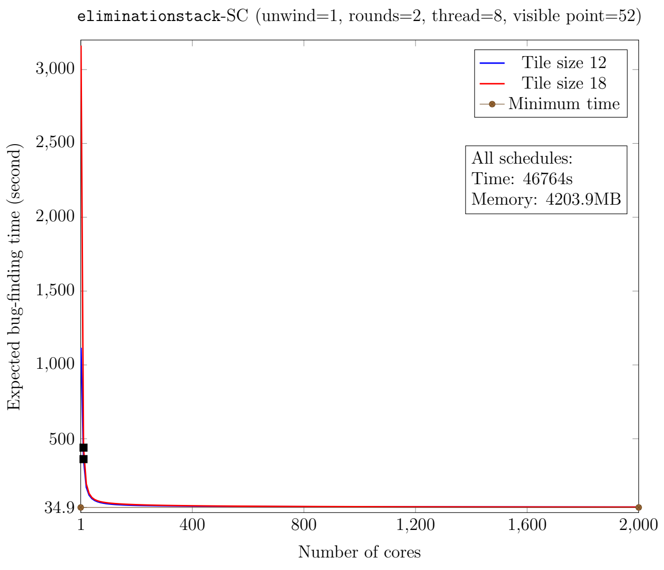
We also tried to measure the "sequential complexity" of the benchmarks (i.e., the difficulty to expose the bug given one fixed buggy schedule) because this gives us an estimated lower bound for the analysis of each of the independent tasks generated by VeriSmart. We therefore ran Lazy-CSeq with the schedule representation accordingly pre-set, rather then letting it search for a buggy schedule. The results are shown in Figure 4.5 under "One schedule" and for `safestack`-TSO we get time 243.5 seconds and memory 2.6 GB. Overall they show that for SC most of the complexity indeed comes from the huge number of schedules that need to be considered. The analysis of a single schedule is approximately 500x faster and requires an order of magnitude less memory. This indicates the potentially huge benefits of the VeriSmart approach. For WMMs, the analysis of a single schedule is still faster (20x–50x) and still requires less memory (30% reduction), but the effects are much less pronounced. This reflects the fact that WMMs introduce a lot of non-determinism even if the thread schedule is fixed.

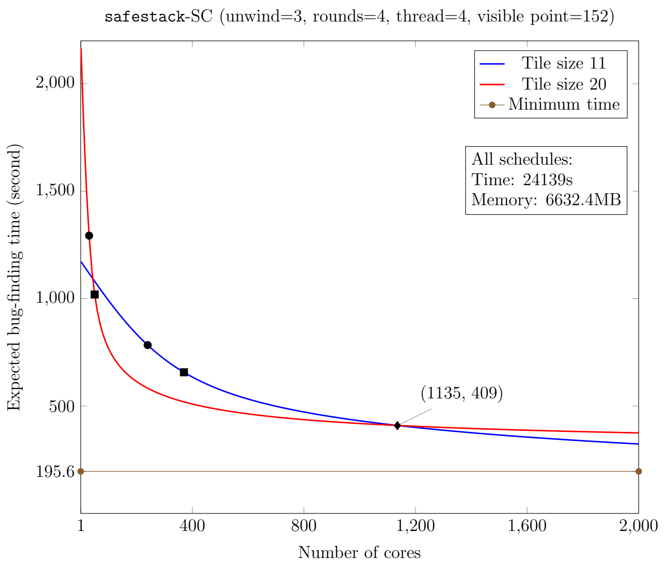
The main part of Figure 4.5 shows the VeriSmart results for the different tile sizes. We focus on the SC benchmarks first, and defer the discussion of WMM benchmarks to below.

For SC, we see the best-case resource consumption required to expose a bug in any of the generated instances drops roughly in line with the estimates from the "One schedule" experiment discussed above. In other words, if we were able to pick the right instances, we could expose the bugs 100x to 1000x faster, with only 10% to 25% of the memory. Obviously, a similar argument could be made in the case of an explicit schedule exploration, but in our case the relative numbers of buggy instances are much more favourable. Moreover, they can be improved even further, by increasing the tile sizes. These increased odds come at modest costs: average times to expose the bugs increase 2x–3x, while average memory consumption remains roughly stable.

Taken together, this means that we only need to analyse a small number (less than 100, and in many cases less than 10) of relatively simple (average times to find the bugs between 20 minutes and one hour) problems to find a bug with probability approximating 1 (see also Figure 4.6).

For `safestack`-PSO the results appear at first less impressive. While the best-case bug-finding times still represent a roughly 5x speed-up, the average times are closer to (and in some times even exceed) the comprehensive analysis times. However, the high fraction of buggy instances (roughly 5%–25%) allows us to play the numbers game to achieve a good overall performance. If we run a moderate number (say 50) of tasks in parallel, we will with high probability (since the distribution of the bug-finding times exhibits a log-normal shape) also come across one of the "faster" tasks, which will abort





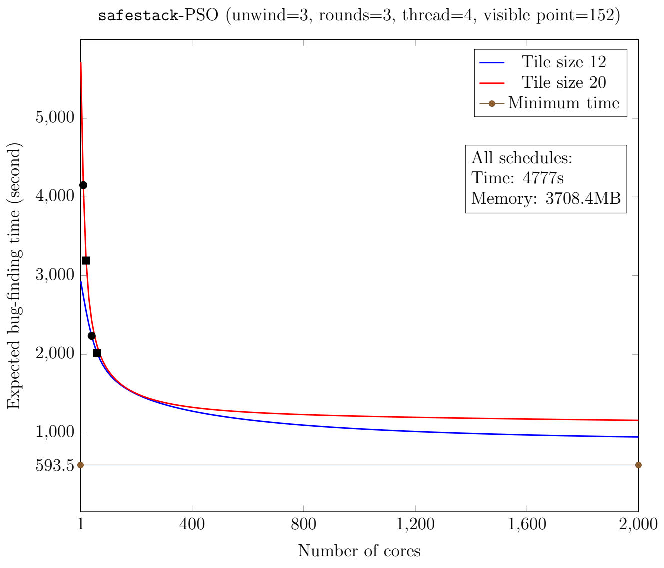


Figure 4.6: Expected bug-finding time varying over the number of cores. A round (resp. square) point marks the expected time corresponding to the number of cores that are needed to find a bug with probability 0.95 (resp. 0.99).

the remaining “slower” tasks, giving us wall-clock speed-ups roughly similar to the best cases.

Finally, for `safestack`-TSO none of the 24,000 tasks generated with three selected tiles exposes a bug within the given timeouts (despite the fact that this already represents 52,000 hours CPU-time). Since the experimental set-up means we are only looking for bugs that occur only under TSO but not under SC, we can clearly see that the TSO-only bug is extremely rare. In such cases, VeriSmart is unable to leverage the effectiveness of the underlying analysis tool, and suffers from the same explosion of the search spaces as other sampling-based methods such as testing or explicit state model checking. Here, symbolic methods that analyse all behaviours simultaneously have the upper hand, as demonstrated by Lazy-CSeq’s ability to expose the TSO-only bug.

Since the generated problem instances are completely independent and can be analysed in parallel, VeriSmart is indeed a very effective verification approach. Figure 4.6 shows that with moderate resources (5–50 cores) we can get a noticeable speed-up on the expected bug-finding time. In particular, for `eliminationstack`-SC the expected time is roughly the same as the minimum time already for few cores and reaches a more than 1000x speed-up compared to the all-schedule comprehensive verification time.

In general, Figure 4.6 shows that the expected bug-finding time converges faster to the minimum time when the probability of finding a buggy program variant is higher. Since larger tiles would ensure higher probability of finding a buggy variant but at the same time could affect the verification time, determining the right tile size seems to be a crucial aspect for maximizing the benefits of our approach.

4.7 Related work

There is a wide range of approaches proposed in the literature on automatic analysis of concurrent programs. Here we briefly describe the related work and compare it to the work presented in this chapter.

Parallel verification Attempts to parallelise verification by partitioning the problem and distributing the workload have been implemented in explicit-state model checking [SD97, BBC05] and SAT solving [OU09]. With the rise of multi-core processors, techniques that exploit shared memory for communication have been proposed [BBR07, HB07]. However, these approaches suffer from the overhead introduced by exchanging information between the instances. Approaches that run several tools with different strategies and heuristics in parallel on the unpartitioned problem have been more successful. Such *portfolio* approaches have been implemented in automated theorem provers [Sch96, WL99, SS99] and SAT/SMT solvers [XHHL08, WHdM09].

Our approach leverages so-called *swarm verification* (SV), as promoted by Holzmann et al for explicit-state model checking [HJG08, HJG11, Hol16]. In SV computing instances do not collaborate directly in finding a solution, but solve independent subproblems that cover the original problem. We lift this idea to symbolic model checking through sequentialization.

Sequentialization Reducing the analysis of a concurrent program to the analysis of sequential programs was first proposed by Qadeer and Wu [QW04]. They transform a concurrent program into a sequential one that simulates all executions of the original program with at most two context-switches. Lal and Reps [LR09] generalised the concept to arbitrary context bounds. In our experiments, we used Lazy-CSeq [INF+15, ITF+14b, ITF+14a], which implements a sequentialization as a code-to-code translation that is efficiently analysable by sequential bounded model checking tools such as CBMC [CKL04]. Musuvathi and Qadeer [MQ07] propose an algorithm for iteratively relaxing the context bound. This is orthogonal to our approach: in our experiments we fixed the maximum number of context-switches and analysed tilings of three different sizes. We could consider their algorithm as a starting point to automatically find good parameter values for our tilings.

Concolic testing Concolic testing [GKS05] combines symbolic aspects with concrete inputs. Namely, it runs the program over an input vector with both concrete and symbolic values, and uses SMT solvers to compute new input vectors that systematically explore the branches of the program. Farzan et al [FHRV13] extend this idea to concurrent programs and call it (con)2colic testing. They use the notion of thread interference scenario, which is a representation of a set of bounded interferences [RFH12] among the threads, which define the scheduling constraints for a concurrent program run. These interference scenarios are then explored in a systematic way by generating a schedule and input vectors that conform with the scenario. (con)2colic testing analyses the program sequentially and accumulates information about explored scenarios in a data structure, whereas our approach is capable of analysing tilings *independently* and can thus be parallelised at large scale. Moreover, (con)2colic testing requires to modify the core of a concolic testing tool in order to handle concurrent programs and thus it cannot flexibly leverage the increasing power of existing sequential checkers.

Testing Automated testing tools such as CHESS [MQB+08] have been highly successful for finding concurrency bugs in large code bases because of their ability to handle code independently of its sequential complexity. CHESS controls the scheduler and explores all possible interleavings giving priority to schedules with few context-switches. Nonetheless, the success of testing depends on the proportion of schedules that lead to a bug w.r.t. the total number of schedules, as shown by a recent empirical study [TDB14, TDB16] on testing of concurrent programs. Preemption sealing [BBC$^+$10] consists of inhibiting preemptions in some program modules which corresponds in our approach to choose a tiling where tiles exactly correspond to program modules. This strategy was aimed to tolerating errors for finding more ones and compositional testing of layered concurrent systems. The uniform tiling we implement in this chapter is irrespective of the structure of the program and looks more appropriate for an exhaustive bug-finding search up to a given number of context-switches. There are also differences in the implementation of the two techniques, we do not seal portions of code with scope functions but rather we implement tiles statically, that in general makes the underlying BMC analysis simpler. Other testing tools try to mutate observed interleavings to find bugs (e.g., [ZLO$^+$11, RIKG12]). Our approach can be seen as a way to tune concurrency verification between concrete testing and fully symbolic verification.

4.8 Conclusions

We present a swarm verification approach to finding bugs in concurrent programs. We perform a code-to-code translation that constructs program variants by placing tiles over the threads, and thus reducing non-determinism by allowing context-switches to occur only in a selected subset of tiles and inhibiting statement reordering in other selected ones. The set of possible program variants defined by a tiling covers all possible interleavings of the concurrent programs. We can analyse these program variants in parallel on a cluster using any off-the-shelf backend tool. We implement the approach building on the CSeq framework and use Lazy-CSeq with CBMC backend for the final analysis. We experimentally show that we can find bugs in very hard concurrency benchmarks, `eliminationstack` and `safestack`, under three different memory models (SC, TSO, PSO) by analysing only a modest number of randomly picked program variants. In comparison with analysing the original program, our approach reduces time and memory footprint of the backend analysis tool when launched on a program variant. In summary, we are able to reduce the wall clock time to find “Heisenbugs” by at least two orders of magnitude on the hardest known concurrency benchmarks.

Chapter 5

Lazy Sequentialization of Unbounded Concurrent Programs

In this chapter, we describe and evaluate a new lazy sequentialization translation that does not unwind loops and thus allows us to analyse unbounded computations, even with an unbounded number of context switches. In connection with an appropriate sequential backend verification tool, it can thus also be used for the safety verification of concurrent programs, rather than just for bug-finding. The main technical novelty of our translation is the simulation of the thread resumption in a way that does not use `goto`s and thus does not require that each statement is executed at most once. We have implemented this translation in the UL-CSeq tool for C99 programs that use the POSIX thread library. We evaluate UL-CSeq on several benchmarks, using different sequential verification backends on the sequentialized program, and show that it is more effective than previous approaches in proving the correctness of the safe benchmarks, and still remains competitive with state-of-the-art approaches for finding bugs in the unsafe benchmarks.

The content of this chapter is largely based on our published work [NFLP15, NFLP16a].

5.1 Introduction

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

An alternative approach is to translate the concurrent program into a non-deterministic sequential program that *simulates* the original program, and then to reuse an existing sequential program verification tool as a black-box backend to verify this simulation program. This approach is also known as *sequentialization* [QW04, LR09, LMP09a]. It has been used successfully both for bug-finding purposes [CGS11, ITF$^+$14a, TIF$^+$15b] and for the verification of reachability properties [GM11, LMP10, LMP12]. Its main advantage is that it separates the concurrency aspects from the rest of the verification tool design and implementation. This has several benefits. First, it simplifies the concurrency handling, which can be reduced to one (usually simple) source-to-source translation. Second, it also makes it thus also easier to experiment with different concurrency handling techniques; for example, we have already implemented a number of different translations (such as [FIP13a, ITF$^+$14a, TIF$^+$15b]) within our CSeq framework [INF$^+$15]. Third, it makes it easier to integrate different sequential backends. Finally, it reduces the overall development effort, because the sequential program aspects and tools can be reused.

The most widely used sequentialization (implemented in Corral [LQL12], SMACK [RE14], and LR-CSeq [FIP13a]) by Lal and Reps [LR09] uses additional copies of the shared variables for the simulation and guesses their values (*eager* sequentialization). This makes the schema unsuitable to be extended for proof finding: it can handle only a bounded number of context switches, and the unconstrained variable guesses lead to over-approximations that are too coarse and make proofs infeasible in practice. *Lazy* sequentializations [LMP09a], on the other hand, do not over-approximate the data; thus they maintain the concurrent program's invariants and simulate only feasible computations. They are therefore, in principle, more amenable to be extended for correctness proofs although efficient implementations exist only for bounded programs [LMP10, LMP12].

Contributions. We develop and implement a lazy sequentialization that can handle programs with unbounded loops and an unbounded number of context switches, and is therefore suitable for program verification (both for correctness and bug-finding). The main technical novelty of our translation is the simulation of the thread resumption in a way that does not require that each statement is executed at most once and (unlike Lazy-CSeq [ITF$^+$14a, INF$^+$15, ITF$^+$14b]) does not rely on `goto`s to reposition the execution. Instead, we maintain a single scalar variable that determines whether the simulation needs to skip over a statement or needs to execute it.

Our first contribution in this chapter is the description of the corresponding source-to-source translation in Section 5.2. As a second contribution, we have implemented this sequentialization in the UL-CSeq tool (within our CSeq framework) for C99 programs that use the POSIX thread library (see Section 5.3). We have evaluated, as a third contribution, UL-CSeq on a large set of benchmarks from the literature and the concurrency category of the Software Verification Competition, using different sequential verification backends on the sequentialized program. We empirically demonstrate, also in Section 5.3, that our approach is surprisingly efficient in proving the correctness of the safe benchmarks and improves on existing techniques that are specifically developed for concurrent programs. Furthermore, we show that our solution is competitive with state-of-the-art approaches for finding bugs in the unsafe benchmarks. We present related work in Section 5.4 and conclude in Section 5.5.

5.2 Unlimited Lazy Sequentialization Schema

In this section we present a code-to-code translation from a multi-threaded program P (see Section 2.1) to a sequential program P^{seq} that simulates all executions of P.

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

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

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

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

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

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

Code-to-code translation

We now describe our translation in a top-down fashion and convey an informal correctness argument as we go along. The entire translation is formally described by the recursive code-to-code translation function $[\![\cdot]\!]$ defined by the rewrite rules given in Figure 5.1. Rule 1 gives the outer structure of P^{seq}: it adds the declarations of the global auxiliary variables, replaces each thread function f_i with the corresponding simulation function f_i^{seq}, adds the code stubs for the thread routines, and then adds the main function. The remaining rules give the transformation for all statement types in our grammar; we return to this in the description of the translation of each thread function f_i into the corresponding simulation function f_i^{seq}.

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

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

– `bool` $\texttt{created}_i$ tracks whether the thread with identifier i has ever been created. Initially, only $\texttt{created}_0$ is set to `true` since f_0^{seq} simulates the `main` function of P.

– `int` $\texttt{pc}_i$ stores the numerical label of the last context switch point for thread i. All the variables $\texttt{pc}_i$ are initialised to `0`, which is the numerical label of the first statement of all thread functions.

1.	$[\![\ (dec;)^*\ \ (\ \texttt{void } f_i\,\texttt{()}\ \ \texttt{\{}(dec;)^*\, stm\texttt{\}}\ )_{i=0,\dots,n}\ ]\!]$	$\overset{\text{def}}{=}$	$\texttt{bool created}_0\texttt{=1,created}_1\texttt{,...,created}_n\texttt{;}$ $\texttt{int s, pc}_0\texttt{,...,pc}_n\texttt{;}$ $(dec;)^*\ \big(\texttt{void } f_i^{seq}\,\texttt{()\{(static } dec;)^*[\![stm]\!]_i\texttt{\}}\big)_{i=0,\dots,n}$ $\texttt{seq\_create(int t, int arg)\{...\}}$ $\texttt{seq\_join(int t)\{...\}}$ $\texttt{seq\_init(int m)\{...\}}\ \ \texttt{seq\_destroy(int m)\{...\}}$ $\texttt{seq\_lock(int m)\{...\}}\ \ \texttt{seq\_unlock(int m)\{...\}}$ $\texttt{main()\{...\}}$
2.	$[\![stm]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{CONTR}(l)\ l\texttt{:}\ [\![seq]\!]_i \mid \texttt{CONTR}(l)\ l\texttt{:}\ \texttt{EXEC}([\![con]\!]_i) \mid \texttt{\{}\langle[\![stm]\!]_i\texttt{;}\rangle^*\texttt{\}}$
3.	$[\![seq]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{EXEC(assume(}b\texttt{))} \mid \texttt{EXEC(assert(}b\texttt{))} \mid \texttt{EXEC(}x\texttt{=}e\texttt{)} \mid$ $\texttt{EXEC(return }e\texttt{)} \mid [\![\texttt{if(}b\texttt{) } stm\ [\texttt{else } stm]]\!]_i \mid$ $[\![\texttt{while(}b\texttt{) do } stm]\!]_i \mid \texttt{EXEC(goto }l\texttt{)}$
4.	$[\![con]\!]_i$	$\overset{\text{def}}{=}$	$x\texttt{=}y \mid y\texttt{=}x \mid [\![t := \texttt{create } f_j()]\!]_i \mid [\![\texttt{join } t]\!]_i$ $\mid [\![\texttt{init } m]\!]_i \mid [\![\texttt{lock } m]\!]_i \mid [\![\texttt{unlock } m]\!]_i \mid [\![\texttt{destroy } m]\!]_i$
5.	$\left[\!\!\left[\begin{array}{l}\texttt{if (}b\texttt{) \{ } \dots\ l_1 : stm_1\texttt{\}}\\ \texttt{[else \{ } \dots\ l_2 : stm_2\texttt{\}]}\end{array}\right]\!\!\right]_i$	$\overset{\text{def}}{=}$	$\texttt{if ((s==RESUME \&\& pc}_i\texttt{<=}l_1\texttt{) \textbar\textbar{} (s==EXECUTE \&\& }b\texttt{))}$ $\quad [\![\texttt{\{}\dots\ l_1 : stm\texttt{\}}]\!]_i$ $\texttt{else if ((s==RESUME \&\& pc}_i\texttt{<=}l_2\texttt{) \textbar\textbar{} (s==EXECUTE))}$ $\quad [\![\texttt{\{}\dots\ l_2 : stm\texttt{\}}]\!]_i\texttt{;}$
6.	$[\![\texttt{while (}b\texttt{) do \{ } \dots\ l_1 : stm\texttt{\}}]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{while(\ \ (s == RESUME \&\& pc}_i\texttt{ <= l}_1\texttt{)}$ $\quad\texttt{\textbar\textbar{} (s == EXECUTE \&\& }b\texttt{)) do}$ $\quad [\![\texttt{\{}\dots\ l_1 : stm\texttt{\}}]\!]_i\texttt{;}$
7.	$[\![t := \texttt{create } f_j()]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{\{}\ t := j;\ \texttt{seq\_create}(e, j)\ \texttt{\}}$
8.	$[\![\texttt{join } t]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{seq\_join}(t)$
9.	$[\![\texttt{init } m]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{seq\_init}(m)$
10.	$[\![\texttt{lock } m]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{seq\_lock}(m)$
11.	$[\![\texttt{unlock } m]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{seq\_unlock}(m)$
12.	$[\![\texttt{destroy } m]\!]_i$	$\overset{\text{def}}{=}$	$\texttt{seq\_destroy}(m)$
	$\texttt{CONTR}(l)$	$\overset{\text{def}}{=}$	$\texttt{if(s == RESUME \&\& pc}_i\texttt{ == }l\texttt{) s = EXECUTE;}$ $\texttt{if(s == EXECUTE \&\& *) \{ pc}_i\texttt{=}l\texttt{; s=SUSPEND;\}}$
	$\texttt{EXEC}(x)$	$\overset{\text{def}}{=}$	$\texttt{if(s == EXECUTE) \{}x\texttt{; \};}$

Figure 5.1: Rewriting rules for the lazy sequentialization.

– `int s` tracks the simulation mode as described above. It can only assume the values `RESUME`, `EXECUTE`, or `SUSPEND`.

Main driver. The new `main` of P^{seq} (see Figure 5.2) consists of an infinite loop that calls at each iteration the thread functions of the active threads.

Thread simulation functions. Each function f_i representing a thread in P is translated into the thread simulation function f_i^{seq} in P^{seq} as follows. First, the local variables of f_i are declared as `static` in f_i^{seq} to make them *persistent* between consecutive invocations of f_i^{seq}. Then, $[\![\cdot]\!]_i$ is applied recursively to the statements in the body of f_i^{seq} (see Rule 1 of Figure 5.1).

For each statement we inject a few lines of code that implement the control of the simulation, i.e., make decisions on mode transitions in the simulation and, depending

```
int main(void){
   while(true) do {
      s = RESUME; /* set mode to RESUME before thread simulation */
      f_0();                   /* main thread simulation */

      s = RESUME;
      if (created_1) f_1();    /* simulation of thread with id 1 */
      ...
      s = RESUME;
      if (created_n) f_n();   /* simulation of thread with id n */
   }
}
```

Figure 5.2: The main function of P^{seq}.

on the current mode, execute or skip the guarded statement. Specifically, every original statement is preceded by the code of the macro CONTR defined in Figure 5.1 that takes as input the label l of the statement (see Rule 2). The injected code allows us to set the mode to EXECUTE if the simulation is in RESUME mode and the old context switch point is reached. After that, if the simulation is in EXECUTE mode, it can non-deterministically transit into SUSPEND, and if so the label l is stored into $\texttt{pc}_i$. Note that, to skip the execution of a thread in a round, we need first to switch from RESUME to EXECUTE and then to SUSPEND before the simulation of the original statement. Furthermore, except for if- and while-statements, all the other statements are guarded by an if-statement injected by the macro EXEC that prevents their simulation unless the mode of the simulation is EXECUTE.

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

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

Figure 5.3: Translation of thread P from Figure 2.3.

Figure 5.3 shows the thread simulation function resulting from sequentializing the thread *P* shown in Figure 2.3 (see Section 2.1).

Simulation of the thread routines. For each thread routine we provide a verification stub, i.e., a simple standard C function that replaces the original implementation for verification purposes. The verification stubs are identical to those used by Lazy-CSeq.

Below, we informally describe how they work; full details are given in Inverso et al.'s work [ITF+14a]. In `seq_create` we simply set the thread's `created` flag. Note that we do not need to store the thread start function, as the `main` driver calls all thread simulation functions explicitly and `seq_create` uses an additional integer argument that serves as thread identifier that is statically determined in the call.

According to the semantics of the `join`-statement, a thread executing `join t` should be blocked until thread `t` is terminated (i.e., the corresponding `pc` variable is set to `LAST_LABEL` that is a statically defined constant larger than any other label in P). We choose to not implement in P^{seq} any notion of blocking or unblocking a thread; instead `seq_join` uses an `assume`-statement with the condition `pc_t == LAST_LABEL` to prune away any simulation that corresponds to a blocking join. We can then see that this pruning does not alter the thread reachability properties of the original program. Assume that the joining thread `t` terminates after the execution of `join t`. The invoking thread should be unblocked then but the simulation has already been pruned. However, this execution can be captured by another simulation in which a context switch is simulated right before the execution of this `join`-statement, and the invoking thread is scheduled to run only after `t` has terminated, hence avoiding the pruning as above.

For mutexes we need to know whether they are free or already destroyed, or which thread holds them otherwise. For this, in the corresponding functions, we use two constants, `FREE` and `DESTROY`. On initialising or destroying a mutex we assign it the appropriate constant. In `seq_lock`, we assert that the mutex is not destroyed and then check whether it is free before assigning it the index of the thread that has invoked the function. As in the case of the `join`-statement we block the simulation if the lock is held by another thread. In `seq_unlock`, we first assert that the lock is held by the invoking thread and then set it to `FREE`. We also support reentrant mutexes[1].

Correctness. The correctness of our construction is quite straightforward.

For the *completeness*, assume any non-empty execution ρ of P that creates at most N threads. Let $\rho = \rho_0 \dots \rho_k$ be split into maximal execution contexts (i.e., each ρ_i is non-empty and has statements only from one thread and ρ_i and ρ_{i+1} are from different threads). Clearly, ρ_0 is a context of the main thread of P that is the only one existing in the beginning. P^{seq} starts the execution from the driver `main` and then calls f_0^{seq} (i.e., the simulation function of the main thread of P). At the first injected control code, since `s` evaluates to `RESUME` and $\texttt{pc}_0$ evaluates to 0 (since `s` is always set to `RESUME` in the driver before calling a simulation function and all the $\texttt{pc}_i$s are initialised to 0), and since we do not context-switch yet, `s` is updated to `EXECUTE` and the original statement of P is executed (see Figure 5.1). The simulation of the remaining statements in ρ_0 is done similarly. On context-switching from ρ_0 to ρ_1, at the second `if`-statement of the macro `CONTR` injected to control the first statement in ρ_1, since we are in the `EXECUTE`

[1] https://en.wikipedia.org/wiki/Reentrant_mutex

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

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

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

Theorem 5.1. *A concurrent program P violates an assertion in at least one of its executions with at most N thread creations if and only if P^{seq} violates the same assertion.*

5.3 Implementation and Experiments

Implementation

We have implemented in UL-CSeq v0.2[2] the schema discussed in Section 5.2 as a code-to-code transformation for sequentially-consistent concurrent C99 programs with POSIX thread library. This implementation is slightly optimised compared to the version that participated (using the CPAchecker backend) in SV-COMP'16 (v0.1) [NFLP15].

UL-CSeq is implemented as a chain of modules within the CSeq framework [FIP13a, FIP13b] (see Section 2.4). The sequentialized program is obtained from the original program through transformations, which (1) insert boilerplate code for simulating the POSIX thread library; (2) unwind any loops that create threads; (3) create multiple

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

copies of the thread start functions, and inline all other function calls; (4) implement the translation rules, as shown in Figure 5.1; and (5) insert code for the main driver, and finalise the translation by adding backend-specific instrumentation.

Experiments

We experimentally evaluated the capabilities and performance of our UL-CSeq implementation (as sketched above) for both verification and bug-finding purposes. We mainly used the benchmark set from the Concurrency category of SV-COMP'16 [Bey16]. These are widespread benchmarks, and many state-of-the-art analysis tools have been trained on them. They offer a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms. In addition, we also used two smaller benchmark collections from the literature [WKO13, GM11]. For all benchmarks we unwound thread-creating loops twice. Since we executed the verification and the bug-finding experiments on different machines and benchmark subsets, we report on them separately.

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

Experimental Setup. For the verification experiments, we used the 221 safe benchmarks from the SV-COMP collection as well as the 13 safe benchmarks proposed by Watcher et al. [WKO13] and Garg and Madhusudan [GM11]. The total size of the benchmarks was approximately 37K lines of code. We ran the experiments on a large compute cluster of Xeon E5-2670 2.6GHz processors with 16GB of memory each, running a Linux operating system with 64-bit kernel 2.6.32. We set a 15GB memory limit and a 900s timeout for the analysis of each benchmark. We used SeaHorn [GKKN15] (v0.1.0),[3] an LLVM-based [LA04] framework for verification of safety properties of programs using Horn Clause solvers; Ultimate Automizer [HCD+13] (version SV-COMP'16),[4] an automata-based software model checker that is implemented in the Ultimate software analysis framework; CPAchecker (v1.4 with predicate abstraction),[5] a tool for configurable software verification that supports a wide range of techniques, including predicate

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

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

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

			UL-CSeq +															
			SeaHorn				Automizer				CPAchecker				VVT			
sub-category	files	l.o.c.	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time
pthread	15	1285	3	2	10	67.3	3	2	10	390.8	2	3	10	204.9	5	3	7	247.3
pthread-atomic	9	1136	6	1	2	167.9	3	1	5	456.7	5	0	4	352.6	5	0	4	171.8
pthread-ext	45	3679	27	0	18	199.1	12	2	31	226.5	15	0	30	214.6	16	5	24	179.7
pthread-lit	8	427	3	0	5	23.3	1	0	7	544.9	3	0	5	164.1	3	2	3	79.8
pthread-wmm	144	29426	144	0	0	32.5	60	0	84	421.6	26	0	118	271.3	141	0	3	275.3
[WKO13]	7	542	5	0	2	51.1	3	1	3	238.6	4	0	3	244.7	4	1	2	133.1
[GM11]	6	290	6	0	0	5.7	5	0	1	181.8	5	0	1	44.9	6	0	0	17.2
Totals	234	36785	194	3	37	59.9	87	6	141	376.2	60	3	171	235.7	180	11	43	248.2

			Impara				Satabs				Threader				VVT			
sub-category	files	l.o.c.	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time	pass	fail	t.o.	time
pthread	15	1285	5	2	8	12.2	3	8	4	308.7	6	8	1	128.4	5	1	9	7.3
pthread-atomic	9	1136	5	0	4	61.8	4	3	2	1.3	7	0	2	24.4	7	1	1	143.7
pthread-ext	45	3679	30	0	15	8.7	15	13	17	34.6	36	1	8	104.8	38	1	6	66.2
pthread-lit	8	427	2	0	6	0.4	2	5	1	8.1	0	7	1	N/A	5	1	2	7.3
pthread-wmm	144	29426	24	0	120	9.0	100	22	22	312.2	0	144	0	N/A	130	0	14	222.2
[WKO13]	7	542	6	0	1	0.5	4	1	2	1.0	5	1	1	27.5	4	3	0	154.7
[GM11]	6	290	5	1	0	2.7	6	0	0	0.8	3	3	0	58.2	3	3	0	8.8
Totals	234	36785	77	3	154	11.2	134	52	48	244.0	57	164	13	88.2	192	10	30	172.6

Table 5.1: Performance comparison of different verification tools on safe benchmarks: UL-CSeq with different sequential backends (top); other tools with built-in concurrency handling (bottom). Each row corresponds to a sub-category of the SV-COMP'16 benchmarks, or to one of the benchmark sets from the literature; we report the number of files and the total number of lines of code. *pass* denotes the number of correctly verified safe benchmarks (i.e., proofs found), *fail* is the number of benchmarks where the tool found a spurious error or crashed (including running out of memory), *t.o.* is the number of benchmarks on which the tool exceeded the given time limit, and *time* is the average proof time (i.e., excluding failed attempts).

abstraction, and shape and value analysis; Impara (v0.2),[6] a tool that implements an algorithm that combines a symbolic form of partial-order reduction and lazy abstraction with interpolants for concurrent programs; Satabs (v3.2),[7] a verification tool based on predicate abstraction; and Threader (version SV-COMP'14),[8] a tool that uses compositional reasoning with regards to the thread structure of concurrent programs based on abstraction refinement. VVT (version SV-COMP'16),[9] a tool that can both verify programs using IC3 and predicate abstraction, also can find bugs using bounded model checking. We ran each tool with its default configuration.

Results. Table 5.1 summarises the results. It demonstrates that our approach is (with suitable backends) surprisingly effective: using SeaHorn, we can prove 194 out of the 234 benchmarks, and just edge out victory over VVT, the best-performing tool with built-in concurrency handling. However, note that UL-CSeq's performance varies widely with the applied backend, and using Automizer or CPAchecker produces noticeably worse results. Proof times are difficult to compare in aggregate but, overall, UL-CSeq's proof times are within the range of the other tools, indicating that the sequentialization does

[6] `http://www.cprover.org/concurrent-impact/impara-linux64-0.2.tgz`
[7] `http://www.cprover.org/satabs/download/satabs-3-2-linux-32.tgz`
[8] `https://www7.in.tum.de/tools/threader/threader.tgz`
[9] `http://vvt.forsyte.at/releases/vvt-svcomp.tar.xz`

not introduce too much complexity. This is further corroborated by the fact that the combination of UL-CSeq and VVT (which finds 180 proofs) is only slightly weaker than VVT relying on its built-in concurrency handling (which finds 192 proofs).

Bug-finding. Here, we used UL-CSeq in combination with CBMC as the sequential backend, and compared it with four different bug-finding tools, Lazy-CSeq, CBMC, CIVL, and SMACK. All four are (ultimately) based on bounded model checking, and have performed very well in the recent SV-COMP verification competitions: both Lazy-CSeq and CIVL scored full marks. Note that the verifiers we used in the experiments described in the previous section performed noticeably worse.

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

We used CBMC [CKL04] (v5.4)[10] both as sequential backend (for UL-CSeq and Lazy-CSeq) and stand-alone bug-finding tool. It is a mature SAT-based bounded software model checker that uses a partial-order approach [AKT13] to handle concurrent programs. We further used Lazy-CSeq [ITF$^+$14a] (v1.0),[11] a lazy sequentialization for bounded programs; CIVL [ZEL$^+$16] (v1.5),[12] a framework that uses a combination of explicit model checking and symbolic execution for verification; and SMACK [RE14] (v1.5.2),[13] a bounded software model checker that verifies programs up to a given bound on loop iterations and recursion depth. For all tools we used as loop unwinding and round bounds the (same) minimum values necessary to find all bugs in the given sub-category.

Results. Table 5.2 summarises the results. We can see that our *proof*-oriented sequentialization does not actually impact negatively on our tool's *bug-finding* performance. UL-CSeq solves 781 of the 784 benchmarks, only three fewer than Lazy-CSeq (whose sequentialization specifically exploits the structure of bounded programs) or CIVL, and more than SMACK. Analysis times are comparable across all tools, with the exception of the noticeably slower SMACK. These results indicate that unwinding and lazy sequentialization can effectively be applied in either order.

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

[10] `http://www.cprover.org/cbmc/download/cbmc-5-4-linux-64.tgz`
[11] `http://users.ecs.soton.ac.uk/gp4/cseq/files/lazy-cseq-cav14-1.0.tar.gz`
[12] `http://vsl.cis.udel.edu/lib/sw/civl/1.5/svcomp16/CIVL-1.5_2739_svcomp16.tgz`
[13] `http://soarlab.org/smack/smack-1.5.2-64.tgz`

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

Table 5.2: Performance comparison of different tools on the unsafe instances of the SV-COMP'16 *Concurrency category*. Each row corresponds to a sub-category of the SV-COMP'16 benchmarks; we report the number of files and the total number of lines of code. *pass* now denotes the number of correctly identified unsafe benchmarks (i.e., counterexamples found), *t.o.* is the number of benchmarks on which the tool exceeded the given time limit, and *time* is the average time to find a bug. None of the tools reported any spurious counterexample.

5.4 Related Work

There is a wide range of approaches to verify concurrent programs. However, here we focus on more closely related sequentialization approaches. The idea of sequentialization was originally proposed by Qadeer and Wu [QW04]. The first schema for an arbitrary but bounded number of context switches was given by Lal and Reps [LR09]. Since then, several algorithms and implementations have been developed (see [FIP13a, LQL12, CGS11, LMP09a, LMP09b]).

Lazy sequentialization schemas have played an important role in the development of efficient tools. Their main feature is that they do not guess the original program's data but just its schedules and so induce less non-determinism and often simpler verification conditions. They also only explore reachable states of the original program, thus preserving the local invariants. This last property makes them suitable for static analysis [LR09]. The first such sequentialization was given by La Torre et al. [LMP09a] for bounded context switching and extended to unboundedly many threads in the works of the same authors [LMP10, LMP12]. These schemas avoid the cross-product of the local states (since only one thread is tracked at any time of a computation) but require their recomputation at each context-switch. This is a major drawback when such a sequentialization is used in combination with bounded model checking (see [GHR10]). The schema Lazy-CSeq [ITF$^+$14a] avoids such recomputations by flattening the programs and making the locals persistent, and achieves efficiency by handling context-switches with a very lightweight and decentralised control code.

All sequentializations mentioned above yield under-approximations of the multi-threaded programs and thus (except for [LMP10] that gives a sufficient condition to test completeness of the reached state space) are designed mainly for bug-finding. The new lazy sequentialization that we have designed in this chapter is similar in spirit to Lazy-CSeq in that it injects lightweight control code to reposition the program counter on simulating a thread resumption but the injected control code itself is completely different. The main limitation of Lazy-CSeq's approach is that it assumes that each thread program counter uniquely identifies its local state (which can be guaranteed for loop-free bounded programs), whereas our approach can handle a wider class of programs. First, we do not unwind loops and thus we allow for an exact simulation of unbounded loops. Second, we do not bound the number of context-switches in any explored computation. Our experiments show that the new control code is almost as effective as the goto-based control code used in Lazy-CSeq when using UL-CSeq with a bounded model checking backend, and performs very well when used to prove correctness of programs.

The only sequentialization that can be used to prove correctness of multi-threaded programs is that proposed by Garg and Madhusudan [GM11], but its approach is quite different from ours. It is closely related to the rely-guarantee style proofs and its aim is to avoid the cross-product of the thread-local states. Only the valuation of some local variables of the other threads (forming the abstraction for the assume-guarantee relation) is retained when simulating a thread. For this, frequent recomputations of the thread local states are required (in particular, whenever a context switch needs to be simulated in the construction of the rely-guarantee relations) which introduces control non-determinism and recursive function calls even if the original program does not contain any recursive calls. Moreover, the resulting sequentialization yields an over-approximation of the original program and thus cannot be used for bug-finding.

5.5 Conclusions

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

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

Chapter 6

Conclusions

6.1 Summary of Work

In this thesis, we have presented our comprehensive work for the analysis of real-world concurrent software in both finding bugs and proving absence of errors, by targeting the largely representative category of multi-threaded C programs with POSIX thread library.

We have shown that Lazy-CSeq sequentialization can be augmented by minimising the sizes of the individual states (which are determined by the concurrent program's shared global and thread-local variables) using abstract interpretation. We have used interval analysis in Frama-C on the produced sequentialized programs from Lazy-CSeq to determine the domains of each variable. Then, these intervals are used to minimise the representation of programs' variables, exploiting the backend's bitvector support, to ultimately reduce the formula fed into the SAT solver. We have implemented this approach on top of the CSeq framework in Lazy-CSeq+ABS tool, and demonstrated the effectiveness of this approach; that it leads to large performance gains compared to Lazy-CSeq for very hard verification problems.

We have demonstrated a swarm verification approach, called "task competition", for finding rare concurrency bugs in concurrent programs. We use code-to-code translation that constructs program variants by placing tiles over the threads, thus reducing non-determinism by allowing context switches to occur only within a selected subset of tiles and inhibiting statement reordering in other selected ones. These program variants can be analysed in parallel on any multi-core platform using any off-the-shelf backend tool. We have implemented the approach in VeriSmart tool via the CSeq framework. Empirical evaluation has shown that VeriSmart can find rare bugs in very hard concurrency benchmarks, by analysing only a modest number of randomly picked program variants, with considerably less resource for the backend analysis tool for each program variant when compared to analysing the original program. Moreover, the wall-clock time to find “Heisenbugs” on those benchmarks is also considerably reduced.

We have presented a novel sequentialization for multi-threaded programs that does not need to bound the number of context-switches or to unwind the loops, i.e., unbounded concurrent programs. In combination with suitable sequential verification tools, this schema can be used both to prove concurrent programs safe or to find bugs. We have implemented this sequentialization in UL-CSeq, on top of the CSeq framework, and provided support for several backends. We have also conducted a large set of experiments which show that UL-CSeq is very competitive for proving correctness, and performs almost as efficiently as the best-performing tools for bug-finding.

6.2 Future Directions

In our experiments on the swarm verification approach, we use a BMC backend, which is very efficient in instances that actually contain an interleaving that exposes the bug, but which may be very slow on the other bug-free instances. We postulate that abstract interpretation has the potential for quickly discharging bug-free instances. This is indeed corroborated by our preliminary experiments where we have used standard abstractions available in abstract interpreters such as ConcurInterProc [Jea09]. The experiments have shown that, with abstract interpretation, we can deem bug-free a significant number of instances several times faster than BMC. Therefore, as the next step in this direction, it is worth investigating how to make sound approximation work effectively on proving safe instances.

Effective analysis tools based on sound approximation can also play a significant role in building verification approaches for finding bugs that are extremely rare. We intend to tackle the problem of finding such extremely rare bugs by combining tools based on sound approximations with a recursive verification approach in the style of a divide-and-conquer algorithm. At each level of the recursion, we split the problem using the tiling as shown in Chapter 4. Then, on each instance, we run two tools in parallel: a bug finder and bug-free prover. We also give a timeout to halt them. Now, as soon as one of the two tools succeeds, we either report the bug or discard the instance (bottom of the recursion). The method recurs on all the instances where we reach the timeout. Thus, this method has the potential to take advantage of the best available technologies for finding bugs, such as those based on BMC and testing, and for proving absence of bugs, such as abstract interpretation, enhanced with our swarm verification approach.

Other further possible directions could extend our existing implementations to handle more verification properties, rather than reachability. It is desirable to study how to design effective sequentialization schemas for various concurrency problems, such as deadlock, race condition, liveness, and, notably, linearizability[1]. Moreover, as the current trend in concurrency is about lock-free data structure design, developing a fine-tuning approach for the analysis of this class of programs is also worth exploring. Another interesting direction is about the verification of concurrent programs on weak memory models. Although work has been done for TSO and PSO, verification of more relaxed models, such as ARM or POWER, remains open, due to the extreme level of concurrency in these memory models. Effective verification of concurrent programs on these can also bring a big impact, as POWER/ARM is the main memory model of ARM CPU, which powers billion of mobile devices and Internet of things[2] (IoT) systems.

[1] `Linearizability,https://en.wikipedia.org/wiki/Linearizability`

[2] Internet of things, `https://en.wikipedia.org/wiki/Internet_of_things`

References

[ABP11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting rid of store-buffers in TSO analysis. In Gopalakrishnan and Qadeer [GQ11], pages 99–115.

[ABP14] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Context-bounded analysis of TSO systems. In Saddek Bensalem, Yassine Lakhnech, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*, volume 8415 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2014.

[ABQ11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads. *Logical Methods in Computer Science*, 7(4), 2011.

[ADF$^+$00] Ole Agesen, David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. Dcas-based concurrent deques. In *SPAA*, pages 137–146, 2000.

[AJ74] Alfred V. Aho and Stephen C. Johnson. LR parsing. *ACM Comput. Surv.*, 6(2):99–124, 1974.

[AKNP14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 508–524. Springer, 2014.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pages 141–157, 2013.

[AS06] Mohammad Awedh and Fabio Somenzi. Automatic invariant strengthening to prove properties in bounded model checking. In Ellen Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 1073–1076. ACM, 2006.

[BB14] Armin Biere and Roderick Bloem, editors. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[BBC05] Jiri Barnat, Lubos Brim, and Ivana Cerná. Cluster-based LTL model checking of large systems. In *FMCO*, pages 259–279, 2005.

[BBC$^+$10] Thomas Ball, Sebastian Burckhardt, Katherine E. Coons, Madanlal Musuvathi, and Shaz Qadeer. Preemption sealing for efficient concurrency testing. In *TACAS*, pages 420–434, 2010.

[BBdH$^+$09] Thomas Ball, Sebastian Burckhardt, Jonathan de Halleux, Madanlal Musuvathi, and Shaz Qadeer. Deconstructing concurrency heisenbugs. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 403–404. IEEE, 2009.

[BBR07] Jiri Barnat, Lubos Brim, and Petr Rockai. Scalable multi-core LTL model-checking. In *SPIN*, pages 187–203, 2007.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BEEH15] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable refinement checking for concurrent objects. In *POPL*, pages 651–662. ACM, 2015.

[BEP11] Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato. On sequentializing concurrent programs. In Eran Yahav, editor, *SAS*, volume 6887, pages 129–145, 2011.

[Bey16] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In Chechik and Raskin [CR16], pages 887–904.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and

Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

[Bie09] Armin Biere. Bounded Model Checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.

[BK04] Jason Baumgartner and Andreas Kuehlmann. Enhanced diameter bounding via structural. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 36–41. IEEE Computer Society, 2004.

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Gopalakrishnan and Qadeer [GQ11], pages 184–190.

[BKA02] Jason Baumgartner, Andreas Kuehlmann, and Jacob A. Abraham. Property checking via structural analysis. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2002.

[BL13] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[BM08] Aaron R. Bradley and Zohar Manna. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20(4-5):379–405, 2008.

[BOW12] Daniel Bundala, Joël Ouaknine, and James Worrell. On the magnitude of completeness thresholds in bounded model checking. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 155–164. IEEE Computer Society, 2012.

[BPM04] Ira D. Baxter, Christopher W. Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004), 23-28 May*

2004, Edinburgh, United Kingdom, pages 625–634. IEEE Computer Society, 2004.

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. *SIGPLAN Not*, page 2002, 2002.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BT15] Christel Baier and Cesare Tinelli, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*. Springer, 2015.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. In *Language Design for Reliable Software*, pages 77–94, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In Alfred V Aho, Stephen N Zilles, and Barry K Rosen, editors, *Conference Record of the Sixth Annual {ACM} Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. {ACM} Press, 1979.

[CC14] Patrick Cousot and Radhia Cousot. Abstract interpretation: past, present and future. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, page 2. ACM, 2014.

[CCF$^+$05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Shmuel

Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CCF$^+$09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[CCK$^+$14] Omar Chebaro, Pascal Cuoq, Nikolai Kosmatov, Bruno Marre, Anne Pacalet, Nicky Williams, and Boris Yakobowski. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng.*, 21(1):107–143, 2014.

[CCM09] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. In *SPAA*, pages 123–124. IEEE Computer Society, 2009.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[CF11] Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *ICSE*, pages 331–340. ACM, 2011.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 25–35. ACM Press, 1989.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E Allen Emerson and A Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGS11] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Time-bounded analysis of real-time systems. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 72–80. FMCAD Inc., 2011.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V Aho, Stephen N Zilles, and Thomas G Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. {ACM} Press, 1978.

[CKGJ11] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In Martin Gogolla and Burkhart Wolff, editors, *TAP*, volume 6706 of *Lecture Notes in Computer Science*, pages 78–83. Springer, 2011.

[CKK$^+$12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988, pages 168–176, 2004.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[CKOS05] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005.

[CKSY05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: sat-based predicate abstraction for ANSI-C. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software,* *ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[CKY03] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *In STOC*, pages 151–158. ACM, 1971.

[CR16] Marsha Chechik and Jean-François Raskin, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*. Springer, 2016.

[CU98] Michael Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1998.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 51–58. IEEE Computer Society, 2001.

[DFG$^+$00] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent deques. In *DISC*, volume 1914 of *LNCS*, pages 59–73. Springer, 2000.

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 351–368, 2011.

[DHRR04] Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.

[DKR10] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for* *the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2010.

[EQR11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded Scheduling. In *POPL*, pages 411–422, 2011.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[FHRV13] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *ESEC/FSE*, pages 37–47. ACM, 2013.

[FIP13a] Bernd Fischer, Omar Inverso, and Gennaro Parlato. CSeq: A Concurrency Pre-processor for Sequential C Verification Tools. In *ASE*, pages 710–713, 2013.

[FIP13b] Bernd Fischer, Omar Inverso, and Gennaro Parlato. Cseq: A sequentialization tool for C - (competition contribution). In *TACAS*, pages 616–618, 2013.

[FMNP14] Anna Lisa Ferrara, P. Madhusudan, Truc L. Nguyen, and Gennaro Parlato. Vac - verifier of administrative role-based access control policies. In Biere and Bloem [BB14], pages 184–191.

[FMP12] Anna Lisa Ferrara, P. Madhusudan, and Gennaro Parlato. Security analysis of role-based access control through program verification. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 113–125. IEEE Computer Society, 2012.

[FMP13] Anna Lisa Ferrara, P. Madhusudan, and Gennaro Parlato. Policy analysis for self-administrated role-based access control. In Piterman and Smolka [PS13], pages 432–447.

[FSLN17] Anna Lisa Ferrara, Anna Squicciarini, Cong Liao, and Truc L. Nguyen. Toward group-based user-attribute policies in azure-like access control systems. In Giovanni Livraga, editor, *Data and Applications Security and Privacy XXX - 31th Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017. Proceedings*, Lecture Notes in Computer Science. Springer, 2017.

[GHR10] Naghmeh Ghafari, Alan J. Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: An empirical evaluation. In *Model*

Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings, pages 227–244, 2010.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *CAV*, pages 343–361, 2015.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[GM11] Pranav Garg and P. Madhusudan. Compositionality entails sequentializability. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2011.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 174–186, 1997.

[GPR11] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 331–344. ACM, 2011.

[GQ11] Ganesh Gopalakrishnan and Shaz Qadeer, editors. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*. Springer, 2011.

[GS97] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[HB07] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

[HCD$^+$13] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. Ultimate Automizer with SMTInterpol - (competition contribution). In Piterman and Smolka [PS13], pages 641–643.

[HJG08] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *ASE*, pages 1–6, 2008.

[HJG11] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Trans. Software Eng.*, 37(6):845–857, 2011.

[Hol14] Gerard J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, 2014.

[Hol16] Gerard J. Holzmann. Cloud-based verification of concurrent software. In *VMCAI*, volume 9583 of *LNCS*, pages 311–327. Springer, 2016.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, pages 206–215. ACM, 2004.

[INF$^+$15] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 807–812, 2015.

[ISO09] ISO/IEC. *Information technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, ISO/IEC/IEEE 9945:2009.* ISO, 2009.

[ITF$^+$14a] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In Biere and Bloem [BB14], pages 585–602.

[ITF$^+$14b] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq: A lazy sequentialization tool for C - (competition contribution). In *TACAS*, pages 398–401, 2014.

[Jea09] Bertrand Jeannet. Relational interprocedural verification of concurrent programs. In *SEFM*, pages 83–92. IEEE Computer Society, 2009.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[KOS$^+$11] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear completeness thresholds for bounded model checking. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 557–572, 2011.

[Kro06] Daniel Kroening. Computing over-approximations with bounded model checking. *Electr. Notes Theor. Comput. Sci.*, 144(1):79–92, 2006.

[KS03] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[Lam79] Leslie Lamport. On the proof of correctness of a calendar program. *Commun. ACM*, 22(10):554–556, 1979.

[LMP09a] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, pages 477–492, 2009.

[LMP09b] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. In *PLDI*, pages 211–222, 2009.

[LMP10] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Model-checking parameterized concurrent programs using linear interfaces. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 629–644. Springer, 2010.

[LMP12] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Sequentializing parameterized programs. In Sebastian S. Bauer and Jean-Baptiste Raclet, editors, *FIT*, volume 87, pages 34–47, 2012.

[LQ13] Akash Lal and Shaz Qadeer. Reachability modulo theories. In Parosh Aziz Abdulla and Igor Potapov, editors, *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*, volume 8169 of *Lecture Notes in Computer Science*, pages 23–44. Springer, 2013.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In Madhusudan and Seshia [MS12], pages 427–443.

[LQR09] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using smt solvers. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 509–524. Springer, 2009.

[LR09] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[LS90] M. E. Lesk and E. Schmidt. Unix vol. ii. chapter Lex&Mdash;a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[McK17] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a). *CoRR*, abs/1701.00854, 2017.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *VSTTE*, volume 7152, pages 146–161, 2012.

[Min01] Antoine Miné. The octagon abstract domain. In *WCRE*, page 310, 2001.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 446–455, 2007.

[MQB$^+$08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.

[MS03] Maher N. Mneimneh and Karem A. Sakallah. Sat-based sequential depth computation. In Hiroto Yasuura, editor, *Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ASP-DAC '03, Kitakyushu, Japan, January 21-24, 2003*, pages 87–92. ACM, 2003.

[MS12] P. Madhusudan and Sanjit A. Seshia, editors. *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*. Springer, 2012.

[Mül06] Markus Müller-Olm. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*, volume 3800 of *LNCS*. Springer, 2006.

[NFLP15] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Unbounded Lazy-CSeq: A lazy sequentialization tool for C programs with unbounded context switches - (competition contribution). In Baier and Tinelli [BT15], pages 461–463.

[NFLP16a] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *ATVA*, volume 9938 of *Lecture Notes in Computer Science*, pages 174–191, 2016.

[NFLP16b] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Verismart: A pragmatic verification approach for concurrent programs. November 2016. University of Southampton.

[NFLP17] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Concurrent program verification with lazy sequentialization and interval analysis. In Amr El Abbadi and Benoît Garbinato, editors, *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, volume 10299 of *Lecture Notes in Computer Science*, pages 255–271, 2017.

[NIF$^+$17] Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq 2.0: Combining lazy sequentialization with abstract interpretation - (competition contribution). In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 375–379, 2017.

[OU09] Kei Ohmura and Kazunori Ueda. c-sat: A parallel SAT solver for clusters. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 524–537. Springer, 2009.

[OV15] Mendes Oulamara and Arnaud J. Venet. Abstract interpretation with higher-dimensional ellipsoids and conic extrapolation. In Daniel Kroening and Corina S. Pasareanu, editors, *CAV*, volume 9206 of *Lecture Notes in Computer Science*, pages 415–430. Springer, 2015.

[PR11] Andreas Podelski and Andrey Rybalchenko. Transition invariants and transition predicate abstraction for program termination. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 3–10. Springer, 2011.

[PS13] Nir Piterman and Scott A. Smolka, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013.*

Proceedings, volume 7795 of *Lecture Notes in Computer Science*. Springer, 2013.

[PSKG08] Hendrik Post, Carsten Sinz, Alexander Kaiser, and Thomas Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *ASE*, pages 188–197. IEEE Computer Society, 2008.

[Qad11] Shaz Qadeer. Poirot - a concurrency sleuth. In Shengchao Qin and Zongyan Qiu, editors, *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, page 15. Springer, 2011.

[QSPK01] Daniel J. Quinlan, Markus Schordan, Bobby Philip, and Markus Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In Henry G. Dietz, editor, *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1-3, 2001. Revised Papers*, volume 2624 of *Lecture Notes in Computer Science*, pages 383–394. Springer, 2001.

[QW04] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[RE14] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In *CAV*, pages 106–113, 2014.

[RFH12] Niloofar Razavi, Azadeh Farzan, and Andreas Holzer. Bounded-interference sequentialization for testing concurrent programs. In *ISoLA*, volume 7609 of *LNCS*, pages 372–387. Springer, 2012.

[RIKG12] Niloofar Razavi, Franjo Ivancic, Vineet Kahlon, and Aarti Gupta. Concurrent test generation using concolic multi-trace analysis. In *APLAS*, volume 7705 of *LNCS*, pages 239–255. Springer, 2012.

[Sch96] Johann Schumann. Sicotheo: Simple competitive parallel theorem provers. In *CADE*, volume 1104 of *LNCS*, pages 240–244. Springer, 1996.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the mur*phi* verifier. In *CAV*, pages 256–278, 1997.

[SS99] Geoff Sutcliffe and Darryl Seyfang. Smart selective competition parallelism ATP. In *Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference*, pages 341–345. AAAI Press, 1999.

[SSA$^+$11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 175–186. ACM, 2011.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A Hunt Jr. and Steven D Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[SW11] Nishant Sinha and Chao Wang. On interference abstractions. In *POPL*, pages 423–434, 2011.

[TDB14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: an empirical study. In *PPoPP*, pages 15–28, 2014.

[TDB16] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *TOPC*, 2(4):23, 2016.

[TIF+14] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Mu-cseq: Sequentialization of C programs by shared memory unwindings - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 402–404. Springer, 2014.

[TIF+15a] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Mu-cseq 0.3: Sequentialization by read-implicit and coarse-grained memory unwindings - (competition contribution). In Baier and Tinelli [BT15], pages 436–438.

[TIF+15b] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Verifying concurrent programs by memory unwinding. In Baier and Tinelli [BT15], pages 551–565.

[TNF+17] Ermenegildo Tomasco, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Using shared memory abstractions to design eager sequentializations for weak memory models. In *Proceedings of the 15th International Conference on Software Engineering and Formal Methods, SEFM 2017, Trento, Italy, Sep 04-08*, 2017.

[TNI+16a] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *FMCAD*, pages 193–200, 2016.

[TNI+16b] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Mu-cseq 0.4: Individual memory location unwindings - (competition contribution). In Chechik and Raskin [CR16], pages 938–941.

[UAS+12] Emre Uzun, Vijayalakshmi Atluri, Shamik Sural, Jaideep Vaidya, Gennaro Parlato, Anna Lisa Ferrara, and Parthasarathy Madhusudan. Analyzing temporal role based access control models. In Vijay Atluri, Jaideep Vaidya, Axel Kern, and Murat Kantarcioglu, editors, *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 177–186. ACM, 2012.

[UAV+14] Emre Uzun, Vijayalakshmi Atluri, Jaideep Vaidya, Shamik Sural, Anna Lisa Ferrara, Gennaro Parlato, and P. Madhusudan. Security analysis for temporal role based access control. *Journal of Computer Security*, 22(6):961–996, 2014.

[Ven12] Arnaud Venet. The gauge domain: Scalable analysis of linear inequality invariants. In Madhusudan and Seshia [MS12], pages 139–154.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[Vyu10] Dmitry Vyukov. Bug with a context switch bound 5, 2010.

[WCM+15] Xueguang Wu, Liqian Chen, Antoine Miné, Wei Dong, and Ji Wang. Numerical static analysis of interrupt-driven programs via sequentialization. In *EMSOFT*, pages 55–64, 2015.

[WHdM09] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. A concurrent portfolio approach to SMT solving. In *CAV*, pages 715–720, 2009.

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multithreaded software with impact. In *FMCAD*, pages 210–217. IEEE, 2013.

[WL99] Andreas Wolf and Reinhold Letz. Strategy parallelism in automated theorem proving. *IJPRAI*, 13(2):219–245, 1999.

[XHHL08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.

[ZEL+16] Manchun Zheng, John G. Edenhofner, Ziqing Luo, Mitchell J. Gerrard, Michael S. Rogers, Matthew B. Dwyer, and Stephen F. Siegel. CIVL: applying a general concurrency verification framework to C/pthreads programs (competition contribution). In Chechik and Raskin [CR16], pages 908–911.

[ZLO$^+$11] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas W. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, pages 251–264, 2011.

safestack benchmark

The code of safestack is shown in Figure 2, whereas Figure 1 shows the transformations of the stack data structure performed by the counterexample reported by Lazy-CSeq (using the SC semantics). The leftmost number is the number of elements stack.count on the stack. The three elements of stack.array are visualised next. The value of the element pointed by stack.head is in a circle whereas the other two elements are in a rectangular box. next points to the element below on the stack. If there is nothing below or an element is unused its value is -1 in the data structure. In this case no arrow is displayed. We highlight the values that are written by the threads in bold numbers. The three rightmost columns show the functions called by the three threads. Context switches are indicated by horizontal lines. We denote functions (except thread) by reactangular boxes. Pop(0), for instance, means that function Pop() is called and element 0 is popped. Note that the executions of these functions, and hence the boxes, are interrupted by context switches. The numbers indicate the line numbers in Figure 2 with read and write operations performed by the segment of the function executed. The assertion failure finally occurs in line 60 in thread 2 because both thread 0 and 2 have popped element 1 and then simultaneously try to write a value into element 1 and push it. This faulty behaviour is enabled by thread 1 whose attempt to pop element 0 is interrupted three times, which causes both other threads, 0 and 2, see element 1 on top of the stack.

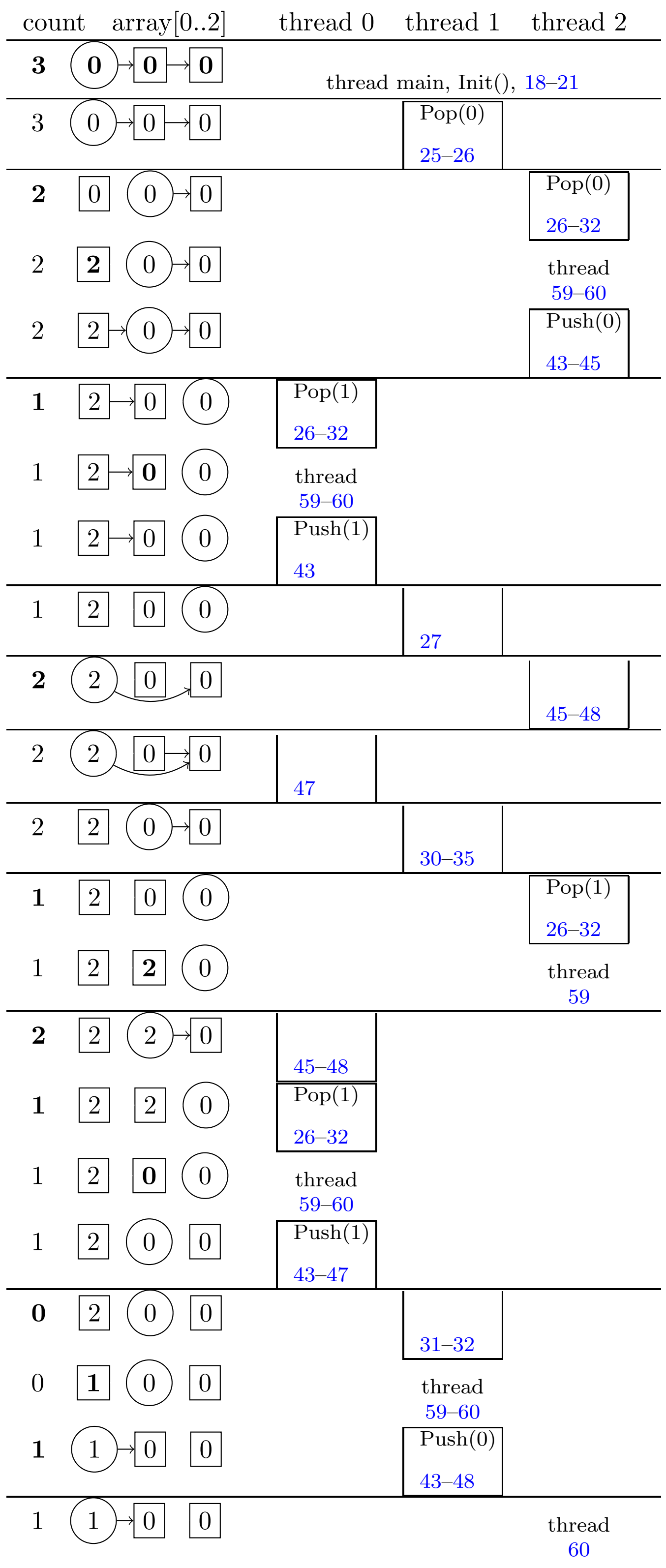


Figure 1: `safestack` counterexample

```
#define NUM_THREADS 3

typedef struct SafeStackItem {
  volatile int Value;
  int Next;
} SafeStackItem;

typedef struct SafeStack {
  SafeStackItem array[3];
  int head;
  int count;
} SafeStack;

pthread_t threads[NUM_THREADS];
SafeStack stack;

void Init(int pushCount) {
  atomic_store(&stack.count, pushCount);
  atomic_store(&stack.head, 0);
  for (int i = 0; i < pushCount - 1; i++) atomic_store(&stack.array[i].Next, i + 1);
  atomic_store(&stack.array[pushCount - 1].Next, -1);
}

int Pop(void) {
  while (atomic_load(&stack.count) > 1) {
    int head1 = atomic_load(&stack.head);
    int next1 = atomic_exchange(&stack.array[head1].Next, -1);
    if (next1 >= 0) {
      int head2 = head1;
      if (atomic_compare_and_exchange(&stack.head, &head2, next1)) {
        atomic_fetch_sub(&stack.count, 1);
        return head1;
      }
      else {
        atomic_exchange(&stack.array[head1].Next, next1);
      }
    }
  }
  return -1;
}

void Push(int index) {
  int head1 = atomic_load(&stack.head);
  do {
    atomic_store(&stack.array[index].Next, head1);
  }
  while (!(atomic_compare_and_exchange(&stack.head, &head1, index)));
  atomic_fetch_add(&stack.count, 1);
}

void* thread(void* arg) {
  int idx = (int)(size_t)arg;
  for (size_t i = 0; i < 2; i++) {
    int elem;
    for (;;) {
      elem = Pop();
      if (elem >= 0) break;
    }
    stack.array[elem].Value = idx;
    assert(stack.array[elem].Value == idx);
    Push(elem);
  }
  return NULL;
}

int main(void) {
  Init(NUM_THREADS);
  for (int i = 0; i < NUM_THREADS; ++i) pthread_create(&threads[i], NULL, thread, (void*)i);
  for (int i = 0; i < NUM_THREADS; ++i) pthread_join(threads[i], NULL);
  return 0;
}
```

Figure 2: `safestack` benchmark